ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

SCHOOL OF ENGINEERING AND ARCHITECTURE

MASTER'S DEGREE
IN
TELECOMMUNICATIONS ENGINEERING

AUTOSCALING IN KUBERNETES: FROM NATIVE RESOURCE-BASED SCALING TO EVENT-DRIVEN SCALING WITH KEDA

Master Thesis
in
Principles, Models and Applications for Distributed Systems

Supervisor
Prof. DAVIDE BORSATTI

Candidate
MATTEO PANNIA

Co-supervisor
Dr. FEDERICO TONINI

SESSION I
ACADEMIC YEAR 2025/2026

"The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise."

— Edsger W. Dijkstra

Contents

Abstract **xv**

1 Kubernetes Fundamentals **1**
1.1 Introduction 1
1.2 Kubernetes Cluster Architecture 2
1.2.1 Control Plane Components 3
1.2.2 Worker Nodes 5
1.3 Kubernetes Networking 6
1.4 Declarative Management Paradigm 10
1.5 Deployment Lifecycle 11
1.6 Namespaces, Pods and Services 13
1.6.1 Services 15
1.7 Chapter Summary 19

2 Autoscaling Mechanisms in Kubernetes **21**
2.1 Introduction 21
2.2 Native Kubernetes Autoscaling Mechanisms 22
2.3 Limitations and Event-Driven Scaling 24
2.4 Chapter Summary 25

3 KEDA: Event-Driven Autoscaling for Kubernetes **27**
3.1 Introduction 27
3.2 KEDA Architecture Overview 29
3.3 KEDA Core Components 31
3.4 KEDA Components Overview 37
3.5 Chapter Summary 38

4 Native CPU-Based Autoscaling in Kubernetes 39
4.1 Introduction . . . 39
4.2 Native Autoscaling Components . . . 40
4.3 Architecture and Metrics Collection Workflow . . . 42
4.4 HPA Decision Logic and Scaling Execution . . . 46
4.5 Numerical Examples and Scaling Interpretation . . . 51
4.6 Limitations of Native CPU-Based Autoscaling . . . 53
4.7 Alternative CPU-Based Scaling: AverageValue Mode . . . 55
4.8 Chapter Summary . . . 57

5 Experimental Evaluation of Native CPU-Based Autoscaling 59
5.1 Experimental Objective . . . 59
5.2 Metrics Server Setup and Verification . . . 60
5.3 Resource Metrics Observation . . . 62
5.4 Application Deployment . . . 68
5.5 Service Resource Creation . . . 71
5.6 HPA Manifest Configuration . . . 73
5.7 Initial System Steady State Verification . . . 77
5.8 Load Generation . . . 79
5.9 Scaling Behavior Analysis . . . 81
5.10 Chapter Summary . . . 83

6 KEDA Integration with Native CPU-Based Autoscaling 85
6.1 Introduction . . . 85
6.2 Architectural Considerations . . . 86
6.3 ScaledObject-Based CPU Autoscaling . . . 88
6.4 Workflow . . . 90
6.5 Metrics Server Requirement . . . 92
6.6 Chapter Summary . . . 93

7 Experimental Evaluation of KEDA Integration with Native CPU-Based Autoscaling 95
7.1 Experimental Objective . . . 95
7.2 Prerequisites and KEDA Installation . . . 97
7.3 Application Resources . . . 98
7.4 ScaledObject Configuration . . . 99
7.5 Automatic HPA Generation and Verification . . . 102
7.6 Load Generation and Monitoring . . . 104

7.7 Scaling Behavior Analysis . . . 105
7.8 Critical Analysis . . . 107
7.9 Chapter Summary . . . 109

8 Time-Based Autoscaling with KEDA: Cron Scaler 111
8.1 Introduction . . . 111
8.2 Cron Scaler Overview . . . 112
8.3 Time-Based Autoscaling Workflow . . . 113
8.3.1 Automatic HPA Creation and External Metrics Binding 115
8.3.2 Cron Scaler Execution . . . 116
8.3.3 Metrics Exposure Through the KEDA Metrics Adapter . 116
8.3.4 HPA Query and Scaling Decision . . . 117
8.3.5 Scaling Execution . . . 118
8.4 Runtime Interaction Scenarios . . . 119
8.4.1 Scaling Down to Zero . . . 119
8.4.2 Scale from Zero to an Active Window . . . 120
8.4.3 Scaling Between Non-Zero Replica Counts . . . 120
8.5 Gradual Scaling and Stabilization Mechanisms . . . 122
8.5.1 Gradual Scaling in Native Kubernetes . . . 123
8.5.2 Configuring HPA Scaling Behavior . . . 123
8.5.3 Interpretation of Scaling Policies . . . 127
8.5.4 Default Kubernetes Scaling Behavior . . . 129
8.5.5 Stabilization Window . . . 130
8.5.6 Gradual Scaling with KEDA . . . 134
8.5.7 Important Limitation: When Gradual Scaling Does Not Apply . . . 137
8.5.8 Difference Between Stabilization Window and Cooldown Period . . . 138
8.5.9 Cooldown Period in Event-Driven Scaling: Queue-Based Example . . . 139
8.5.10 Cooldown Period in Time-Based (Cron) Scaling . . . 141
8.6 Chapter Summary . . . 143

9 Experimental Evaluation of Time-Based Autoscaling with KEDA Cron Scaler 145
9.1 Experimental Objective . . . 145
9.2 Experimental Setup . . . 146

9.3 Deployment Configuration 146
9.4 ScaledObject Configuration 148
9.4.1 Default HPA Scaling Policies 151
9.4.2 Cron Trigger Configuration 152
9.4.3 Interpretation of the Time Window in the Cron Scaler Configuration 152
9.5 Resource Deployment 155
9.6 Monitoring the Experiment 156
9.7 Experimental Results 157
9.7.1 Observed Behavior 157
9.7.2 Metric Interpretation 158
9.7.3 Scale from Zero 159
9.7.4 Scale-Up Performed by the Horizontal Pod Autoscaler . 160
9.7.5 Exit from the Active Time Window 162
9.7.6 Stabilization Window Analysis 163
9.7.7 Scale-to-Zero Performed by KEDA 166
9.8 Chapter Summary 167

10 Application-Level Metrics and Monitoring: Prometheus and Grafana 169
10.1 Introduction 169
10.2 Prometheus Overview 170
10.3 Prometheus Exporters 170
10.4 Prometheus Integration with Kubernetes Autoscaling 171
10.5 Chapter Summary 171

11 Application-Level Metrics Autoscaling Based on Prometheus without KEDA 173
11.1 Introduction 173
11.2 Architecture Overview 173
11.3 Application Metrics Exposure 175
11.3.1 NGINX Metrics Exposure: The `/stub_status` Endpoint 176
11.3.2 Raw Output of the `/stub_status` Endpoint 177
11.3.3 Interpretation of the Exposed Fields 177
11.3.4 Role of the NGINX Prometheus Exporter 178
11.3.5 Prometheus Metrics Exposed by the Exporter 178
11.3.6 Exporter Deployment Models 179

11.3.7 Exporter Configuration in Kubernetes Deployment . . . 179
11.3.8 Exporter Output (`/metrics` Endpoint) 181
11.4 Prometheus-Based Autoscaling Workflow 182
11.4.1 Step 1 – Metrics Collection by Prometheus 182
11.4.2 Step 2 – Metric Aggregation and Querying 182
11.4.3 Step 3 – Prometheus Adapter Integration 183
11.4.4 Step 4 – API Aggregation Layer 184
11.4.5 Step 5 – HPA Metric Retrieval 184
11.4.6 Step 6 – Scaling Decision 185
11.4.7 Step 7 – Scaling Formula 185
11.4.8 Step 8 – Deployment Update 186
11.5 Key Observations 186
11.6 Chapter Summary 188

12 Experimental Evaluation: Prometheus-Based Autoscaling without KEDA 189
12.1 Experimental Objective 189
12.2 Prerequisites 190
12.3 Prometheus Installation 191
12.4 Prometheus Stack Components 192
12.5 NGINX and Exporter Deployment 197
12.5.1 NGINX Configuration with `/stub_status` 197
12.5.2 Exporter Access to the NGINX Metrics Endpoint 199
12.5.3 NGINX Location Matching Behavior 199
12.5.4 Deployment with NGINX and Exporter Sidecar 200
12.5.5 Service for Prometheus Scraping and Load Generation . 202
12.5.6 Service Networking and Pod Discovery 204
12.6 Prometheus Scraping Configuration 204
12.6.1 ServiceMonitor Configuration 205
12.6.2 Target Discovery Issue and Resolution 205
12.6.3 Port Name Resolution 206
12.6.4 Deployment of Monitoring Resources 207
12.6.5 Namespace Consistency Considerations 207
12.7 Verification of Metrics Collection in Prometheus 208
12.7.1 Verification of the Exporter Endpoint 208
12.7.2 Verification of Registered Prometheus Targets 209
12.7.3 Verification Through the Prometheus Targets API 210

12.7.4 Accessing the Prometheus Web UI 211
12.7.5 Verification of Active Targets 211
12.7.6 Querying and Verifying Autoscaling Metrics 212
12.8 Prometheus Adapter Installation and Configuration 215
12.8.1 Adding the Helm Repository 215
12.8.2 Custom Adapter Configuration 216
12.8.3 Installing the Prometheus Adapter 217
12.8.4 Verification of the Adapter Deployment 217
12.8.5 Verification of the Prometheus Server Service 218
12.8.6 Verification of the Adapter Deployment Configuration . . 218
12.8.7 Correcting an Incorrect Prometheus URL 218
12.9 Verification of the External Metrics API 219
12.9.1 Verification of the API Registration 219
12.9.2 Verification of Available External Metrics 220
12.9.3 Validation of the Exposed External Metric 220
12.9.4 Pull-Based Behaviour of the Prometheus Adapter 221
12.9.5 Interpretation of the Observed Metric Value 222
12.10 Horizontal Pod Autoscaler Configuration 223
12.10.1 Horizontal Pod Autoscaler Manifest 223
12.10.2 Description of the Configuration Fields 224
12.10.3 Deploying the Horizontal Pod Autoscaler 227
12.10.4 Monitoring the Horizontal Pod Autoscaler 227
12.11 Experimental Load Generation and Autoscaling Validation . . . 229
12.11.1 Generating the Workload 229
12.11.2 Observing the Scale-Out Process 230
12.11.3 Terminating the Workload 231
12.11.4 Observing the Scale-In Process 232
12.12 Key Observations and Conclusions 233
12.13 Chapter Summary 233

13 NGINX Autoscaling with KEDA and Prometheus Metrics 235
13.1 Introduction 235
13.2 Common Monitoring Components 236
13.3 KEDA Integration and Architectural Changes 237
13.3.1 ScaledObject Creation and Validation 237
13.3.2 KEDA Controller and Metric Polling 238
13.3.3 Metric Evaluation and Transformation 239

13.3.4 Metric Exposure Through KEDA Metrics Adapter 239
13.4 Automatic HPA Generation 243
13.5 Scaling Decision 243
13.6 Chapter Summary 244

14 Experimental Evaluation of KEDA-Based Prometheus Autoscaling (NGINX Use Case) 247
14.1 Experimental Objective 247
14.2 Reusing the Existing Monitoring Infrastructure 248
14.3 ScaledObject Configuration 250
14.3.1 Scaling Parameters 252
14.3.2 Advanced HPA Behaviour Configuration 253
14.3.3 Prometheus Trigger Configuration 255
14.4 ScaledObject Deployment and Verification 259
14.5 Load Generation and Autoscaling Validation 261
14.6 Chapter Summary 262

15 Monitoring KEDA Autoscaling Metrics through Prometheus and Grafana 265
15.1 Introduction 265
15.2 Enabling Prometheus Monitoring for KEDA Components 266
15.3 Verification of KEDA Metrics Collection 269
15.3.1 Verification via the Prometheus HTTP API 271
15.4 Visualization through Grafana 272
15.5 Chapter Summary 273

16 RabbitMQ-Based Event-Driven Autoscaling with KEDA 275
16.1 Introduction 275
16.2 RabbitMQ Architecture and Message Flow 276
16.3 RabbitMQ Scaler and Autoscaling Workflow 280
16.4 Chapter Summary 283

17 Experimental Evaluation of RabbitMQ-Based Autoscaling with KEDA 285
17.1 Experimental Objective 285
17.2 Consumer and Producer Implementation 289
17.3 KEDA Configuration 301
17.4 Experimental Evaluation 309

17.4.1 Additional Remarks on Cooldown Behavior and Transient Workload Spikes . . . 320
17.5 Chapter Summary . . . 322

18 RabbitMQ Native Prometheus Exporter and KEDA Autoscaling 325
18.1 Experimental Objective . . . 325
18.2 RabbitMQ Prometheus Integration . . . 326
18.3 Experimental Implementation . . . 328
18.3.1 RabbitMQ Deployment and Service Modifications . . . 328
18.3.2 ServiceMonitor Configuration . . . 330
18.3.3 Validation . . . 331
18.3.4 Prometheus-based ScaledObject Configuration . . . 332
18.4 Experimental Evaluation . . . 333
18.5 Chapter Summary . . . 336

19 Comparative Evaluation of Static and Event-Driven Scaling Strategies in a RabbitMQ Scenario 339
19.1 Experimental Objective . . . 339
19.2 Experimental Architecture . . . 340
19.3 Producer Redesign . . . 341
19.4 Producer Deployment and Service Exposure . . . 347
19.5 Controlled Workload Triggering . . . 351
19.6 Metrics and Evaluation Methodology . . . 352
19.6.1 Time-Series Panels . . . 353
19.6.2 Bar Gauge Panels . . . 355
19.6.3 Computation Methodology . . . 358
19.6.4 Summary of Dashboard Metrics . . . 360
19.7 Reference Estimation of Idle Consumer CPU Usage . . . 361
19.7.1 Time-Series Analysis of Idle Consumer CPU Usage . . . 363
19.7.2 Average CPU Usage of Individual Idle Consumers . . . 364
19.7.3 Reference CPU Usage of a Generic Idle Consumer . . . 366
19.8 Static Underprovisioned Scenario . . . 369
19.8.1 Scenario Description . . . 369
19.8.2 Summary Metrics . . . 370
19.8.3 Message Rate Analysis . . . 372
19.8.4 Queue State and Backlog Analysis . . . 373

19.8.5 Consumer State Analysis . . . 375
19.8.6 Idle CPU Usage Estimation . . . 376
19.8.7 Engineering Discussion . . . 377
19.9 Static Overprovisioned Scenario . . . 378
19.9.1 Scenario Description . . . 378
19.9.2 Summary Metrics . . . 379
19.9.3 Message Rate Analysis . . . 381
19.9.4 Queue State and Backlog Analysis . . . 382
19.9.5 Consumer State Analysis . . . 384
19.9.6 Idle CPU Usage Estimation . . . 385
19.9.7 Engineering Discussion . . . 386
19.10KEDA Autoscaling Configuration . . . 387
19.11KEDA Scale-to-Zero Scenario . . . 391
19.11.1 Scenario Description . . . 391
19.11.2 HPA Timeline and Autoscaling Behaviour . . . 391
19.11.3 Summary Metrics . . . 394
19.11.4 Message Rate Analysis . . . 396
19.11.5 Queue State and Backlog Analysis . . . 397
19.11.6 Consumer State Analysis . . . 399
19.11.7 Idle CPU Usage Estimation . . . 401
19.11.8 Engineering Discussion . . . 402
19.12KEDA Warm-Start Scenario . . . 403
19.12.1 HPA Timeline and Autoscaling Behaviour . . . 404
19.12.2 Summary Metrics . . . 406
19.12.3 Message Rate Analysis . . . 408
19.12.4 Queue State and Backlog Analysis . . . 410
19.12.5 Consumer State Analysis . . . 412
19.12.6 Idle CPU Usage Estimation . . . 413
19.12.7 Engineering Analysis . . . 414
19.13Final Comparison and Trade-off Analysis . . . 415
19.14Chapter Summary . . . 419

20 Conclusions and Future Work **421**
20.1 Conclusions . . . 421
20.2 Future Work . . . 423

Acknowledgements **425**

A Grafana Dashboard PromQL Queries **429**
A.1 Dashboard Overview 429
A.2 Consumer Efficiency Summary 430
A.2.1 Peak Active Consumers 430
A.2.2 Average Active Consumers 431
A.2.3 Average Busy Consumers 431
A.2.4 Average Idle Consumers 432
A.3 Queue and Processing Summary 432
A.3.1 Load Duration 433
A.3.2 Peak Ready Messages 433
A.3.3 Backlog Duration 433
A.3.4 Peak Busy Consumers 433
A.4 Message Rates 434
A.4.1 Incoming Rate 434
A.4.2 Delivery Rate 434
A.4.3 Acknowledgement Rate 434
A.5 Queue State and Backlog 435
A.5.1 Total Queue Messages 435
A.5.2 Unacknowledged Queue Messages 435
A.5.3 Ready Queue Messages 435
A.6 Consumer State 436
A.6.1 Active Consumers 436
A.6.2 Busy Consumers 436
A.6.3 Idle Consumers 436

Author's Notes **437**

Abstract

Containerized applications and microservice-based architectures have become the dominant paradigms for modern cloud-native systems. In such environments, efficiently adapting computational resources to workload variations is essential to guarantee both service performance and resource utilization efficiency. Kubernetes has emerged as the de facto standard platform for container orchestration, providing automated deployment, workload management and native autoscaling capabilities. This thesis investigates autoscaling mechanisms in Kubernetes environments, with a particular focus on Kubernetes Event-Driven Autoscaling (KEDA), an open-source framework that extends native Kubernetes scaling capabilities by enabling scaling decisions based on external events and application-specific metrics. After introducing the fundamental concepts of Kubernetes and its native autoscaling solutions, the architecture and operating principles of KEDA are analyzed. A set of practical demonstrations is first presented to illustrate the behavior of native Kubernetes autoscaling and selected KEDA scalers. The proposed approach is then evaluated through a more comprehensive case study based on a RabbitMQ workload and Prometheus metrics collected from a Kubernetes cluster. The experimental results highlight the trade-offs between static resource provisioning and event-driven autoscaling, showing how KEDA can dynamically scale the number of application instances in response to workload conditions while improving resource utilization and maintaining application responsiveness. These results demonstrate the effectiveness of KEDA as a flexible and efficient solution for autoscaling cloud-native applications in dynamic cloud-native environments.

Chapter 1

Kubernetes Fundamentals

1.1 Introduction

The widespread adoption of cloud-native applications has significantly transformed the way modern software systems are designed and deployed. Traditional monolithic applications are increasingly being replaced by microservice-based architectures, where individual functionalities are implemented as independent services that can be developed, deployed and updated independently. This evolution has been enabled by containerization technologies, which provide lightweight and portable execution environments capable of packaging applications together with their dependencies. Compared to traditional virtual machines, containers offer reduced overhead, faster startup times and more efficient resource utilization, making them particularly suitable for modern cloud infrastructures. While containers simplify application deployment, managing large numbers of distributed containerized workloads introduces significant operational challenges. Tasks such as application deployment, monitoring, fault recovery, load balancing and resource management become increasingly complex as infrastructures grow in size and complexity. To address these challenges, container orchestration platforms have been introduced. Among them, Kubernetes has emerged as the de facto standard for orchestrating cloud-native applications. Originally developed by Google and currently maintained by the Cloud Native Computing Foundation (CNCF), Kubernetes provides a unified platform for deploying and managing containerized workloads across clusters of physical or virtual machines. It automates several operational tasks, including workload scheduling, service discovery, load balancing, fault recovery and

application scaling. Furthermore, Kubernetes adopts a declarative management model in which users specify the desired state of the system, while the platform continuously monitors the infrastructure and performs the actions required to maintain the requested configuration. Beyond traditional cloud computing environments, Kubernetes has also become increasingly relevant in the telecommunications sector. The transition from dedicated hardware appliances to software-based network functions has introduced the need for platforms capable of managing large numbers of distributed services while ensuring high availability, flexibility and efficient resource utilization. Through features such as automated deployment, self-healing capabilities and dynamic scaling, Kubernetes enables the efficient execution of cloud-native network functions and distributed applications across both centralized cloud infrastructures and geographically distributed edge environments. Consequently, it is now considered a key enabling technology for modern distributed systems, including cloud platforms, 5G networks, Multi-access Edge Computing (MEC) infrastructures, Internet of Things (IoT) ecosystems and other latency-sensitive services. Thanks to its flexibility, extensibility and rich ecosystem, Kubernetes has become one of the most widely adopted platforms for managing distributed applications and constitutes the technological foundation upon which the autoscaling mechanisms discussed throughout this thesis are built. In order to better understand the autoscaling mechanisms analyzed in the following chapters, the next section introduces the architecture of a Kubernetes cluster and its main components.

1.2 Kubernetes Cluster Architecture

A Kubernetes cluster consists of a collection of interconnected machines cooperating to execute containerized applications. These machines can be implemented as physical servers deployed within a data center or as virtual machines hosted by cloud providers. Regardless of the underlying infrastructure, Kubernetes abstracts the available computational resources and exposes them as a unified platform for deploying and managing applications. From a logical perspective, a Kubernetes cluster is composed of two main categories of nodes:

- `Control Plane Nodes`
- `Worker Nodes`

The Control Plane contains the components responsible for cluster management and decision-making, whereas Worker Nodes provide the computational resources required to execute application workloads.

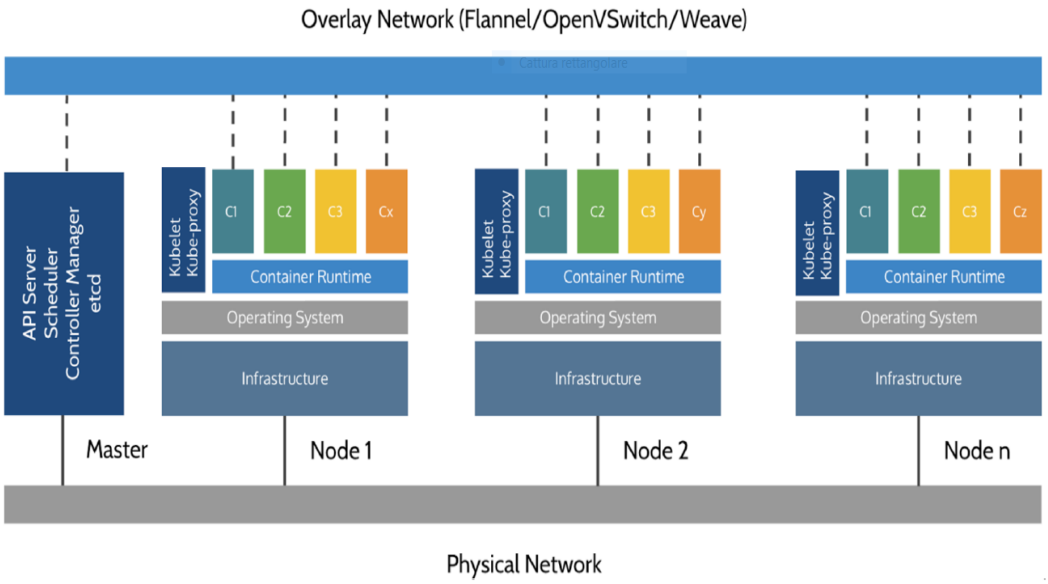


Figure 1.1: High-level architecture of a Kubernetes cluster showing the Control Plane, Worker Nodes and the networking infrastructure enabling communication among cluster components [1].

As illustrated in Figure 1.1, the Control Plane represents the management layer of the cluster, while application workloads are executed within Pods running on Worker Nodes. Communication between the various cluster components is coordinated through the Kubernetes API, which acts as the central interaction point for both internal and external entities.

1.2.1 Control Plane Components

The Control Plane represents the logical core of Kubernetes and is responsible for maintaining the overall state of the cluster. It continuously monitors the system and ensures that deployed applications operate according to the configuration specified by users. The Control Plane is composed of several services working together to manage cluster operations. The most important components are the API Server, the Scheduler, the Controller Manager and

the etcd database. Together, these components coordinate the deployment and management of workloads running on Worker Nodes.

API Server
The API Server is the central entry point of Kubernetes and acts as the communication hub between all cluster components. Every operation performed within the cluster ultimately passes through the API Server. Users interact with the API Server through tools such as `kubectl`, web dashboards or third-party applications. Similarly, internal components including controllers, schedulers and kubelets communicate through the API Server. Whenever a request is received, the API Server validates it, verifies its correctness and updates the corresponding resource definitions stored within the cluster database. Because all cluster operations transit through the API Server, it can be considered the single source of interaction with Kubernetes.

etcd
`etcd` is a distributed key-value database responsible for storing all cluster information. It contains resource definitions, configuration parameters and the current state of Kubernetes objects. Every resource created within Kubernetes is ultimately persisted in `etcd`. Consequently, `etcd` represents the authoritative source of truth for the entire cluster. The reliability of the Control Plane strongly depends on the availability of `etcd`, since a failure of this component may compromise the ability of Kubernetes to maintain and recover the desired cluster state.

Scheduler
The Scheduler is responsible for assigning Pods to Worker Nodes. When a new Pod is created, it initially remains in a `Pending` state because no execution node has yet been selected. The Scheduler continuously monitors the cluster for unscheduled Pods and determines the most appropriate node on which they should run. The scheduling decision is based on the available resources and scheduling constraints defined within the cluster. Once a suitable node has been identified, the Scheduler updates the Pod definition by associating it with the selected Worker Node.

Controller Manager
The Controller Manager executes a collection of controllers responsible for continuously monitoring cluster resources and ensuring that the actual state matches the desired state specified by users. Each controller continuously

observes the cluster state and performs corrective actions whenever inconsistencies are detected. This reconciliation mechanism represents a fundamental building block of Kubernetes and will be discussed in greater detail in the following sections.

1.2.2 Worker Nodes

Worker Nodes provide the computational resources required to execute containerized workloads. Depending on the deployment environment, Worker Nodes may correspond to physical servers located within a data center or virtual machines instantiated by cloud providers. Each Worker Node hosts application Pods and includes several components required for workload execution and communication with the Control Plane. Together, these components ensure that applications are correctly deployed, executed and monitored while continuously reporting their execution state back to the Control Plane. The most important components are kubelet, kube-proxy and the container runtime.

kubelet
The kubelet is an agent running on every Worker Node and serves as the primary connection between the node and the Control Plane. Its main responsibility is to ensure that the containers assigned to the node are correctly running. The kubelet continuously communicates with the API Server, receives workload definitions and verifies that the actual execution state matches the expected configuration. If a container terminates unexpectedly, the kubelet detects the failure and reports the updated status to the Control Plane. In addition, the kubelet interacts with the container runtime to create, start and terminate containers according to the workload specifications received from the API Server. It also periodically reports node health information and resource usage, allowing the Control Plane to continuously monitor the status of the cluster.

Container Runtime
The container runtime is the software component responsible for executing containers. Common container runtimes include:

- `containerd`
- `CRI-O`

- Docker Engine (historically)

Kubernetes interacts with the runtime through the Container Runtime Interface (CRI), which abstracts the underlying implementation and allows different runtimes to be used interchangeably. When instructed by the kubelet, the runtime downloads container images, creates containers and manages their execution lifecycle. The runtime is also responsible for interacting with the underlying operating system to provide process isolation, resource management and access to the networking and storage resources required by each container.

kube-proxy
The kube-proxy component manages network communication inside the cluster. Its primary responsibility is implementing Kubernetes Services by configuring the networking rules required to forward traffic toward the appropriate Pods. Depending on the cluster configuration, kube-proxy typically relies on technologies such as `iptables` or `IPVS` to implement packet forwarding and load balancing functionalities. Although kube-proxy plays a fundamental role in cluster networking, it should not be confused with Container Network Interface (CNI) plugins, which are responsible for implementing the cluster networking infrastructure itself.

1.3 Kubernetes Networking

Kubernetes provides a networking model that enables communication among all components deployed within the cluster. One of its fundamental design principles is the adoption of a flat network architecture, in which every Pod receives its own unique IP address and can communicate directly with other Pods regardless of the node on which they are running. As a consequence, the Pod rather than the node becomes the fundamental network endpoint within the cluster. This abstraction simplifies the development of distributed applications, since services can communicate without requiring knowledge of the underlying infrastructure. Since Pods may be distributed across multiple Worker Nodes, Kubernetes relies on Container Network Interface (CNI) plugins to provide network connectivity throughout the cluster. The Container Network Interface (CNI) is not itself responsible for packet forwarding, but rather defines a standardized interface through which different networking implementations can be integrated into Kubernetes. CNI plugins are responsible for allocating Pod IP addresses, configuring virtual network interfaces

and establishing connectivity both within the same node and across different Worker Nodes. Several implementations are available, including Calico, Flannel, Cilium and Weave Net, each providing different networking capabilities and performance characteristics. As illustrated in Figure 1.1, this connectivity is commonly achieved through an overlay network built on top of the physical infrastructure network. While the physical network provides connectivity between nodes, the overlay network creates a unified logical address space that enables Pod-to-Pod communication across the entire cluster. Conceptually, the relationship between the physical infrastructure and the Kubernetes networking model is summarized in Figure 1.2:

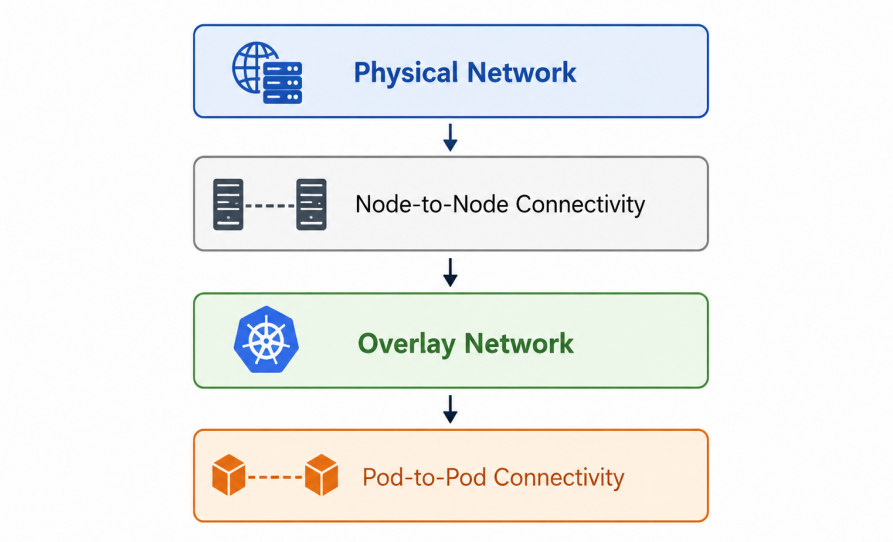


Figure 1.2: Relationship between the physical infrastructure and the logical networking model adopted by Kubernetes.

The overlay network uses the physical network only as a transport mechanism. From the application perspective, this abstraction completely hides the underlying physical topology. Developers do not need to know whether two Pods are running on the same Worker Node or on different nodes, since Kubernetes guarantees transparent communication through the networking layer. This greatly simplifies the development of distributed applications and microservice-based architectures. When a Pod needs to communicate with another Pod located on a different node, the communication is encapsulated and transported across

the underlying infrastructure. This networking model allows applications to communicate as if they were connected to the same flat IP network, independently of their physical location. Furthermore, it provides the foundation for higher-level abstractions such as Kubernetes Services, which offer stable communication endpoints despite the dynamic nature of Pod lifecycles.

Communication Model

Kubernetes is composed of multiple distributed components that must continuously exchange information to coordinate cluster operations. To simplify these interactions, Kubernetes adopts a centralized communication model in which the API Server acts as the primary communication hub. Both external users and internal cluster components interact with the API Server, which provides a consistent interface for accessing and managing cluster resources. This centralized communication model considerably simplifies the overall system architecture because individual Kubernetes components do not communicate directly with one another. Instead, every operation involving cluster resources is mediated by the API Server, which validates requests, stores the desired state in etcd and propagates the corresponding updates to the remaining Control Plane components. The communication model adopted by Kubernetes is summarized in Figure 1.3: User requests generated through tools such as `kubectl` are processed by the API Server, while components such as controllers, schedulers and kubelets use it to exchange information and monitor resource state. External interactions with Kubernetes typically rely on REST APIs and JSON representations of resources. Internally, however, Kubernetes frequently uses Protocol Buffers (protobuf), a compact binary serialization format that reduces communication overhead and improves scalability. This architecture decouples the various cluster components and enables Kubernetes to coordinate workload deployment, monitoring and management through a single control point. Together with the networking model discussed previously, this communication architecture enables Kubernetes to coordinate distributed workloads while maintaining a clear separation between application logic and infrastructure management.

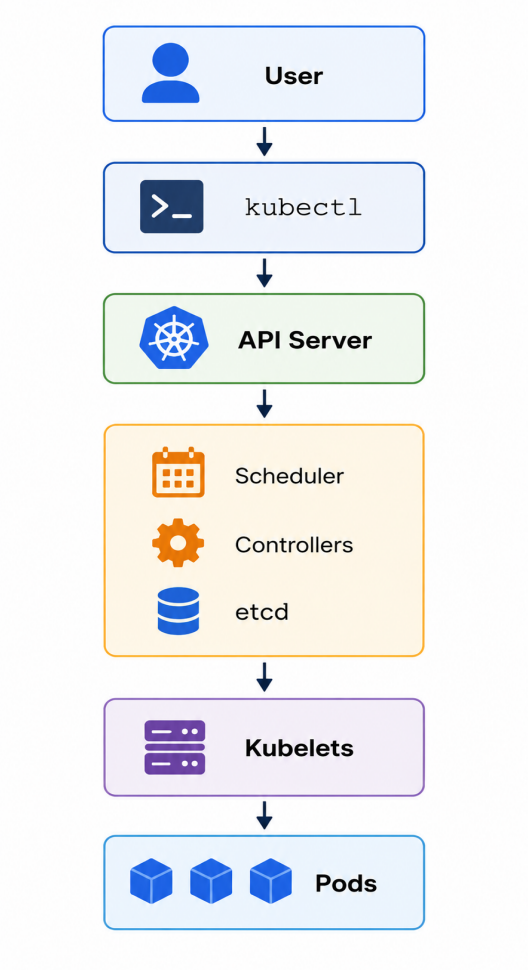


Figure 1.3: Simplified communication model in Kubernetes.

Kubernetes Resources

One of the fundamental concepts of Kubernetes is that every managed object is represented as a resource. Applications, nodes, services, configuration objects and many other entities are exposed through the Kubernetes API using a common resource-based model. Resources constitute the building blocks of a Kubernetes cluster and provide a standardized way to describe the desired configuration of applications and infrastructure components. Users interact with these resources through the Kubernetes API, typically by defining their desired state in YAML manifests. This resource-oriented approach is closely

related to the declarative management model adopted by Kubernetes. Rather than manually executing operations, users define the desired configuration of a resource, while Kubernetes continuously monitors the cluster and performs the actions required to maintain the requested state. Kubernetes provides a large number of resource types, each serving a specific purpose. Among the most commonly used are Nodes, Pods, Deployments, Services, ConfigMaps and Namespaces. These resources play a central role in application deployment and management and will be discussed in greater detail in the following sections.

1.4 Declarative Management Paradigm

One of the key characteristics of Kubernetes is its declarative management model. Unlike traditional imperative approaches, where users explicitly specify the sequence of operations to be executed, Kubernetes allows users to describe the desired state of the system while delegating the implementation details to the platform.

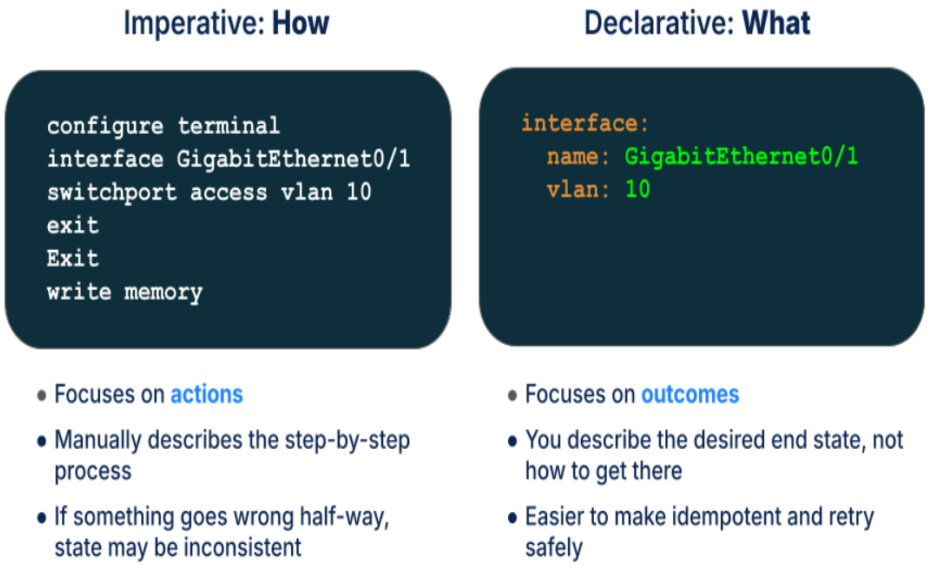


Figure 1.4: Conceptual comparison between imperative and declarative management approaches. In the imperative model, the sequence of operations is explicitly specified, whereas in the declarative model only the desired final state is defined [2].

As illustrated in Figure 1.4, the declarative approach focuses on describing the desired outcome rather than the sequence of actions required to achieve it. For example, rather than manually creating and managing individual application instances, a user can simply specify that a given application should run with a certain number of replicas. Kubernetes then determines the actions required to achieve and maintain that configuration. At first glance, the declarative approach may appear simpler because users only need to describe the desired outcome rather than the sequence of operations required to achieve it. However, this apparent simplicity is made possible by the additional intelligence implemented within Kubernetes itself. Components such as controllers, schedulers and node agents continuously cooperate to determine how the requested configuration should be achieved and maintained over time. Consequently, the operational complexity is not eliminated but rather shifted from the user to the orchestration platform. The declarative model is based on the distinction between the desired state and the current state of the cluster. The desired state represents the target configuration specified by the user through Kubernetes resources, whereas the current state reflects the actual condition of the running system. To ensure consistency between these two states, Kubernetes continuously performs a reconciliation process. Cluster components monitor the current state and compare it with the desired configuration stored in the cluster. Whenever a discrepancy is detected, corrective actions are automatically performed in order to restore the requested state. This approach enables important capabilities such as self-healing and automated resource management. For example, if an application instance unexpectedly terminates, Kubernetes can automatically recreate it to restore the desired configuration. Another important consequence of the declarative model is idempotency. Applying the same resource definition multiple times produces the same final result without generating unintended side effects. This property greatly simplifies infrastructure automation and represents one of the main advantages of declarative system management.

1.5 Deployment Lifecycle

The concepts introduced in the previous sections can be better understood by examining how Kubernetes deploys and manages an application. A typical deployment workflow highlights the interaction among the main cluster compo-

nents and illustrates how the declarative paradigm is implemented in practice. Applications are commonly deployed through a `Deployment` resource, which describes the desired state of an application, including the container image to be executed and the number of replicas that should be running. When a user submits a Deployment manifest using the `kubectl apply` command, the request is sent to the API Server, which validates the resource definition and stores it in `etcd`. At this stage, the Deployment becomes part of the desired state of the cluster. The Deployment Controller continuously monitors the resources stored in the cluster and detects the creation of the new Deployment. In response, it creates a `ReplicaSet` resource responsible for maintaining the desired number of application replicas. The ReplicaSet subsequently creates the corresponding Pod objects. These Pods are initially stored by the API Server and remain in the `Pending` state because no Worker Node has yet been assigned. The Scheduler continuously monitors the API Server for unscheduled Pods and detects the newly created Pod in the `Pending` state. It then selects the most suitable Worker Node according to the available resources and scheduling constraints. Once a node has been chosen, the Scheduler updates the Pod definition by associating it with the selected Worker Node. The kubelet running on the selected Worker Node detects the new assignment through the API Server and requests the container runtime to start the corresponding workload. If necessary, the container runtime first downloads the required container image and then creates and starts the container. During this phase, the kubelet continuously updates the Pod status through the API Server. After being assigned to a node, the Pod transitions from the `Pending` state to the `Creating` state while the container image is downloaded and the application is initialized. Once the container has been successfully started and all initialization operations have completed, the Pod reaches the `Running` state and becomes available to serve application requests. The deployment process does not terminate once the application is running. Kubernetes continuously monitors the state of deployed resources in order to ensure consistency with the desired configuration. If a Pod unexpectedly crashes or becomes unavailable, the ReplicaSet detects that the current number of running replicas no longer matches the desired state and automatically creates a replacement Pod. Through this mechanism, Kubernetes provides self-healing capabilities and can recover from many common failures without requiring manual intervention. The same reconciliation process is also applied when the user modifies the Deployment definition. For example, if the number of replicas is increased,

Kubernetes detects the change and creates additional Pods until the new target configuration is reached. Conversely, if the replica count is reduced, the excess Pods are terminated. In both cases, the system automatically converges towards the newly specified desired state. This workflow clearly illustrates how Kubernetes combines declarative resource definitions, continuous monitoring and automated reconciliation to manage application lifecycles. The same principles will later be exploited by autoscaling mechanisms such as the Horizontal Pod Autoscaler (HPA) and KEDA, which dynamically modify resource definitions according to workload conditions.

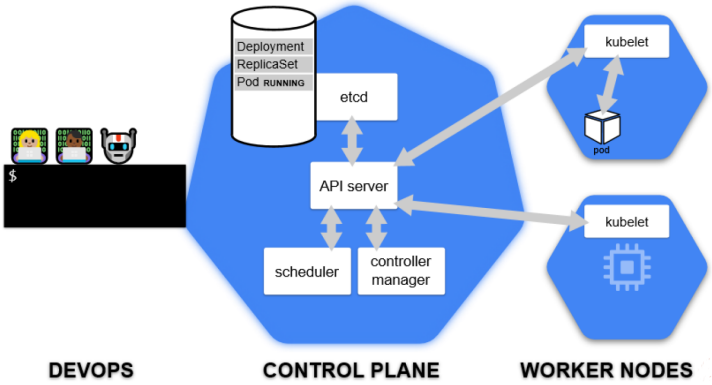


Figure 1.5: Interaction among the main Kubernetes components during the deployment of an application, from the submission of a Deployment resource to the execution of the corresponding Pods on Worker Nodes [3].

1.6 Namespaces, Pods and Services

Namespaces, Pods and Services are among the most frequently used Kubernetes resources and constitute the foundation upon which application deployment and communication are built. Together, they provide mechanisms for organizing resources, executing workloads and enabling network connectivity among distributed application components.

Namespaces

Namespaces provide a mechanism for logically partitioning cluster resources. They allow multiple applications, teams or environments to coexist within

the same Kubernetes cluster while remaining logically separated. Resources belonging to different namespaces can be managed independently, simplifying administration and reducing complexity in large deployments. Typical examples include the use of dedicated namespaces for development, testing, monitoring and production environments. Although namespaces do not provide complete isolation, they represent an effective mechanism for organizing Kubernetes resources and improving cluster management.

Pods

The Pod is the smallest deployable unit in Kubernetes and represents the basic execution entity of the platform. A Pod encapsulates one or more containers that are always scheduled and executed together on the same Worker Node. While many Pods contain a single application container, Kubernetes also supports multi-container Pods in which multiple tightly coupled containers cooperate to provide a service. Containers belonging to the same Pod share the same network namespace and therefore the same IP address. As a result, they can communicate directly through the localhost interface without requiring additional networking mechanisms. This design is commonly adopted in sidecar architectures, where auxiliary containers provide functionalities such as monitoring, logging or proxying for the main application. Since Pods are ephemeral resources, they may be recreated during their lifecycle and therefore their IP addresses cannot be considered permanent communication endpoints. For this reason, Kubernetes introduces additional abstractions that provide stable network connectivity independently of individual Pod instances.

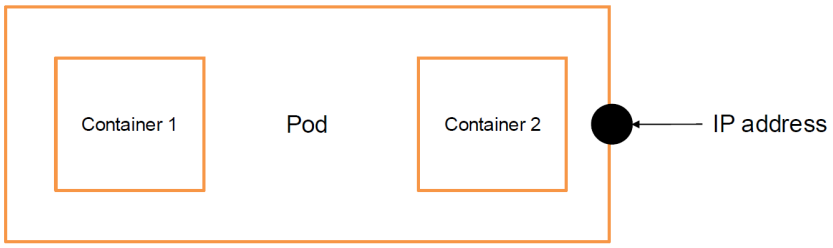


Figure 1.6: Structure of a Kubernetes Pod containing one or more containers sharing the same network namespace and IP address [3].

1.6.1 Services

To provide stable communication endpoints despite the dynamic nature of Pods, Kubernetes introduces the Service abstraction. A Service groups a set of Pods and exposes them through a stable virtual endpoint, identified by a virtual IP address, that remains available even when individual Pods are created, removed or replaced. Applications therefore communicate with Services rather than directly with actual Pod IP addresses. Services are implemented through kube-proxy, which automatically configures the networking rules required to forward traffic towards the appropriate backend Pods. Kubernetes supports multiple Service types depending on the desired accessibility requirements. The most common are ClusterIP and NodePort.

ClusterIP
ClusterIP is the default Service type and provides connectivity exclusively within the Kubernetes cluster. When a ClusterIP Service is created, Kubernetes assigns a virtual IP address, commonly referred to as ClusterIP, and automatically creates a DNS entry associated with the Service name. Applications running inside the cluster can therefore access the Service either through its DNS name or through its virtual IP address. From the application's perspective, the Service behaves as a stable endpoint. Internally, however, traffic is forwarded towards one of the backend Pods associated with the Service. In addition to providing a stable endpoint, the Service also acts as a load-balancing abstraction. Requests directed to the Service are distributed among the available backend Pods, allowing traffic to be shared across multiple application replicas and improving scalability and fault tolerance. The internal communication process associated with a ClusterIP Service is summarized in Figure 1.7:

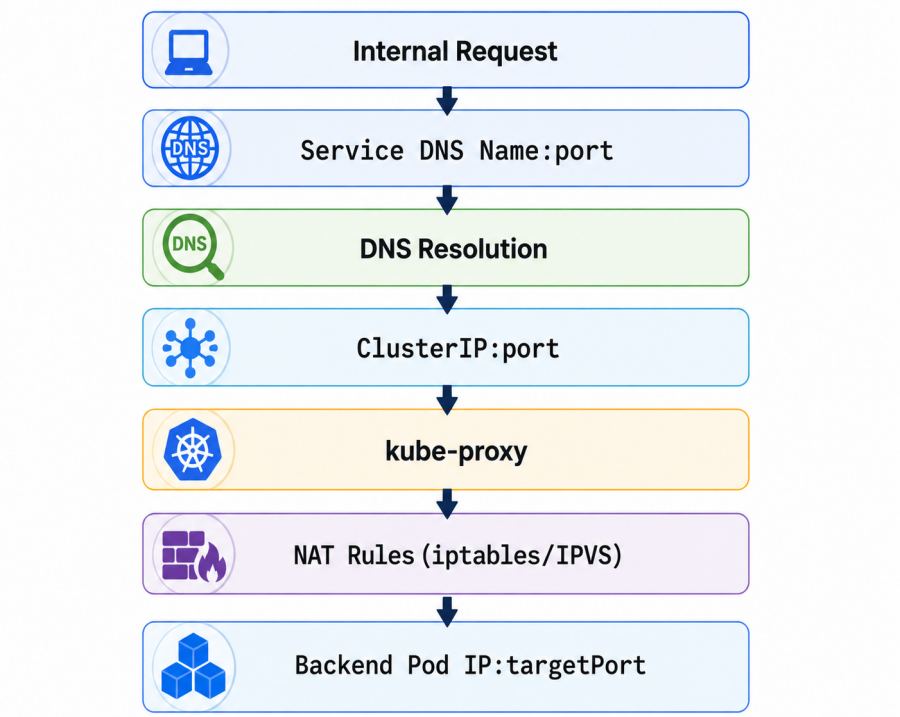


Figure 1.7: Internal traffic flow for a ClusterIP Service.

In this workflow, DNS resolution maps the Service name to its virtual IP address. Once traffic reaches the Service, kube-proxy applies automatically generated forwarding rules that redirect the request towards one of the available backend Pods. ClusterIP Services are commonly used to enable communication among microservices and internal application components.

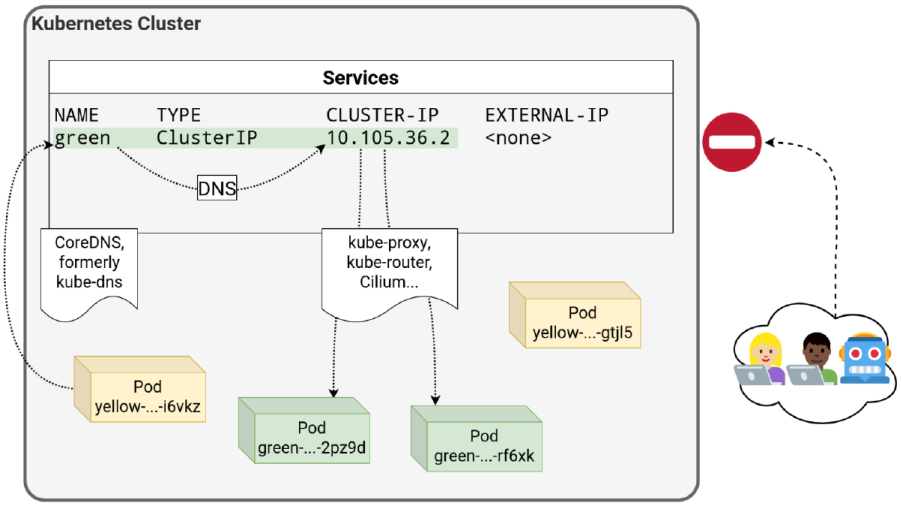


Figure 1.8: Internal Service exposure through a virtual IP address and DNS name. Traffic is forwarded by kube-proxy towards the backend Pods associated with the Service [3].

NodePort

NodePort extends the functionality of ClusterIP by allowing external access to a Service. When a NodePort Service is created, Kubernetes automatically provisions all the mechanisms associated with a ClusterIP Service, including the virtual IP address and DNS entry, and additionally exposes the Service through a dedicated port on every Worker Node. External clients can therefore reach the application by contacting the IP address of any Worker Node together with the configured NodePort value. Importantly, the selected Worker Node does not necessarily need to host one of the application Pods. Since kube-proxy configures the forwarding rules on every node, requests received by any Worker Node can be transparently redirected to backend Pods running elsewhere in the cluster. A NodePort Service may involve three different port numbers:

- `nodePort`, used for external access through the Worker Nodes interfaces
- `port`, representing the Service port exposed inside the cluster
- `targetPort`, corresponding to the port used by the application running inside the Pod

The external communication workflow associated with a NodePort Service is summarized in Figure 1.9:

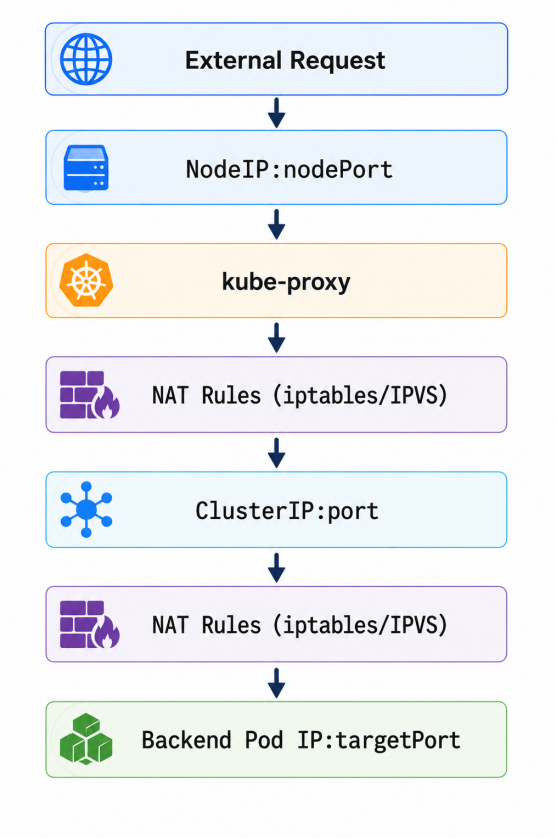


Figure 1.9: External traffic flow for a NodePort Service.

Unlike internal Service access, external requests do not require DNS resolution and directly reach one of the Worker Nodes interfaces through the configured NodePort. kube-proxy then applies the necessary forwarding rules to route traffic towards the Service and subsequently to one of the backend Pods. Internally, NodePort continues to rely on the ClusterIP mechanism. Consequently, the Service remains accessible from within the cluster through its virtual IP address and DNS name while simultaneously being reachable from external networks. Unless explicitly restricted through node-level configuration, a NodePort Service is exposed on all network interfaces of every Worker Node.

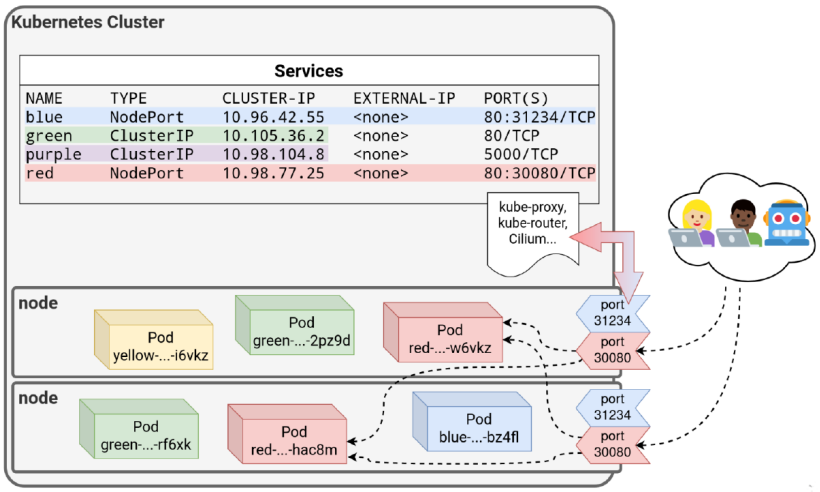


Figure 1.10: External Service exposure through a port opened on every Worker Node. Incoming traffic is forwarded to the corresponding ClusterIP Service and then routed to one of the backend Pods [3].

LoadBalancer
In cloud environments, Services can also be exposed through the LoadBalancer type. In this case, Kubernetes interacts with the underlying infrastructure to provision an external load balancer and associate it with the Service. Compared to NodePort, this approach provides a dedicated external entry point and avoids the need for clients to directly contact Worker Node addresses. For this reason, LoadBalancer Services are generally preferred in production cloud environments, while NodePort is more commonly used in testing, development and on-premise deployments. This approach simplifies application exposure to external users and is widely adopted in managed cloud environments. However, it depends on the capabilities provided by the underlying infrastructure and is therefore less commonly used in local or on-premise Kubernetes deployments.

1.7 Chapter Summary

This chapter introduced the fundamental concepts of Kubernetes that constitute the foundation for the experimental activities presented throughout

this thesis. After motivating the adoption of Kubernetes in modern cloud-native and distributed environments, the chapter described the overall cluster architecture, highlighting the roles of the Control Plane and Worker Nodes as well as the interactions among the main system components responsible for cluster management. The discussion then focused on the Kubernetes networking model, including the flat Pod network, the role of Container Network Interface (CNI) plugins, the overlay networking approach and the centralized communication model coordinated by the API Server. Subsequently, the resource-oriented design adopted by Kubernetes was presented together with the declarative management paradigm, illustrating how the desired state of applications is continuously enforced through the reconciliation process. Finally, the chapter introduced the Kubernetes resources most relevant to the remainder of this work, including Pods, Deployments and Services, describing how they are used to deploy, manage and expose containerized applications inside a cluster. These concepts provide the theoretical foundation required to understand the native autoscaling mechanisms offered by Kubernetes and the event-driven extensions introduced by KEDA. Building upon these foundations, the next chapter analyzes the autoscaling mechanisms available in Kubernetes, discussing their operating principles, capabilities and limitations. Particular attention is then devoted to KEDA, which extends the native Kubernetes autoscaling model and represents the core focus of this thesis.

Chapter 2

Autoscaling Mechanisms in Kubernetes

2.1 Introduction

Kubernetes provides built-in mechanisms to automatically adapt both application workloads and the underlying infrastructure to varying traffic demands. By design, Kubernetes is capable of intelligently scaling applications, deployed as Pods, as well as the cluster itself, ensuring that system performance and resource utilization remain optimal under dynamic conditions. This adaptability is essential in modern cloud-native environments, where workloads can fluctuate significantly over time. Without automated scaling mechanisms, administrators would be required to manually provision or release computational resources in response to workload variations, leading either to resource overprovisioning or to performance degradation during peak demand. Kubernetes supports both horizontal scaling, which consists of changing the number of instances of a resource, and vertical scaling, which consists of adjusting the resource allocation of existing instances. Furthermore, it can dynamically adapt the number of Worker Nodes in the cluster to meet application requirements. To achieve these goals, Kubernetes provides three main autoscaling mechanisms: the Horizontal Pod Autoscaler (HPA), the Vertical Pod Autoscaler (VPA) and the Cluster Autoscaler (CA). Each of these operates at a different level of abstraction and targets a specific aspect of the system.

2.2 Native Kubernetes Autoscaling Mechanisms

Cluster Autoscaler
The Cluster Autoscaler (CA) is responsible for scaling the infrastructure layer, i.e., the nodes of the Kubernetes cluster. Nodes are typically virtual machines that host the Pods scheduled by Kubernetes. The CA dynamically adjusts the number of nodes based on the cluster scheduling needs. When the Kubernetes Scheduler is unable to place new Pods due to insufficient available resources, such as CPU or memory, those Pods enter a `Pending` state. In this scenario, the Cluster Autoscaler provisions additional nodes to accommodate the required workload. Conversely, when nodes are underutilized, the CA can remove them to reduce resource waste. It is important to note that the Cluster Autoscaler primarily performs horizontal scaling of nodes by adding or removing virtual machines, while vertical resizing of nodes is generally managed externally by the underlying cloud provider infrastructure.

Vertical Pod Autoscaler
In contrast, both the Horizontal Pod Autoscaler and the Vertical Pod Autoscaler operate at the application level, focusing on workloads deployed through Pods rather than on the cluster infrastructure itself. The Vertical Pod Autoscaler (VPA) performs vertical scaling by dynamically adjusting the resource requests and limits, mainly CPU and memory, assigned to Pods. Unlike HPA, which increases or decreases the number of replicas, VPA attempts to adapt the resources assigned to existing Pods. This approach is particularly useful for stateful applications or workloads that are difficult to scale horizontally, as it allows a single Pod to handle increased load by allocating more resources. However, VPA may require Pod restarts to apply new resource configurations, which can impact availability if not carefully managed.

Horizontal Pod Autoscaler
The Horizontal Pod Autoscaler (HPA) adjusts the number of Pod replicas based on observed metrics. This represents the primary mechanism for horizontal scaling at the application level. By default, HPA relies on resource-based metrics such as CPU and memory utilization. These metrics are typically collected and exposed through the Kubernetes Metrics Server. However, HPA can also be extended to use custom and external metrics through additional components capable of exposing such metrics to the Kubernetes API Server. The HPA continuously monitors these metrics and increases or decreases the

number of Pod replicas to maintain the desired performance level. Due to its flexibility and responsiveness, HPA is the most commonly used autoscaling mechanism and serves as the foundation for advanced event-driven scaling solutions such as KEDA.

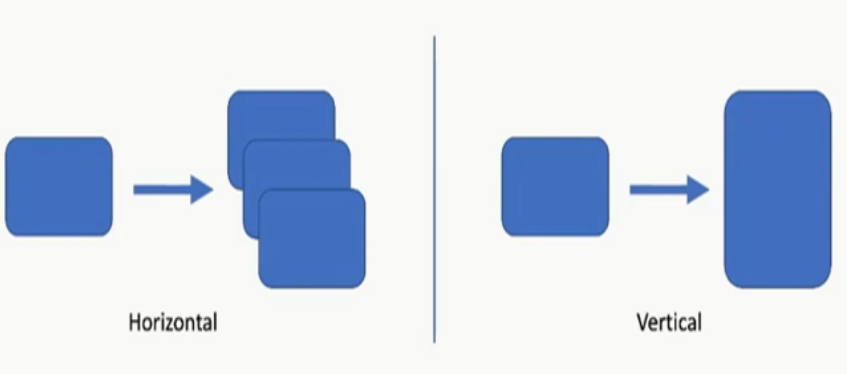


Figure 2.1: Comparison between horizontal and vertical scaling in Kubernetes environments [4].

Figure 2.1 compares the two fundamental scaling approaches adopted in Kubernetes. Horizontal scaling increases the number of application instances or replicas, allowing the workload to be distributed across multiple components. In contrast, vertical scaling increases the computational resources allocated to an existing instance, such as CPU and memory, without changing the number of running replicas.

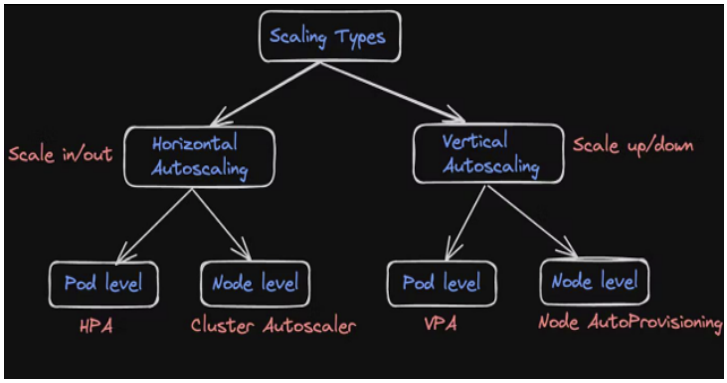


Figure 2.2: Overview of the main Kubernetes autoscaling mechanisms and their interactions [5].

Figure 2.2 illustrates how the three main Kubernetes autoscaling mechanisms operate at different layers of the system. The Horizontal Pod Autoscaler (HPA) and the Vertical Pod Autoscaler (VPA) act at the application level by respectively adjusting the number of Pod replicas and the computational resources allocated to individual workloads. In contrast, the Cluster Autoscaler (CA) operates at the infrastructure level by dynamically increasing or decreasing the number of Worker Nodes available in the cluster. The figure also highlights the role of the Kubernetes Scheduler, which assigns Pods to nodes according to the available resources. Whenever a Pod cannot be scheduled because no suitable node is available, it remains in the *Pending* state, a condition that may trigger the intervention of the Cluster Autoscaler to provision additional cluster capacity.

Terminology and Scaling Operations
In the context of Kubernetes, the terms scale-out and scale-in are typically associated with horizontal scaling, whereas scale-up and scale-down are commonly used to describe vertical scaling. However, both in technical literature and in practical usage, these terms are often used interchangeably. Since this thesis focuses primarily on horizontal autoscaling mechanisms, both terminologies may occasionally be used to indicate an increase or decrease in the number of application instances.

2.3 Limitations and Event-Driven Scaling

Despite their flexibility, native Kubernetes autoscaling mechanisms exhibit some limitations when dealing with highly dynamic event-driven workloads. In particular, the Horizontal Pod Autoscaler operates based on periodically collected metrics and does not natively provide direct integration with external event sources such as message queues, streaming platforms or custom application-level signals. Although it is possible to extend HPA through custom and external metrics, this often introduces additional complexity in terms of configuration, deployment and maintenance. Moreover, traditional metric-based scaling may not be sufficiently reactive or expressive for workloads characterized by asynchronous events, such as queue-based processing systems. In these scenarios, scaling decisions should often depend on application-specific indicators such as queue length, backlog size or incoming request rate, rather than CPU or memory utilization alone. These limitations highlight the need for

a more flexible and event-driven autoscaling approach, which is precisely the problem addressed by KEDA. KEDA extends Kubernetes autoscaling capabilities beyond traditional resource-based metrics while maintaining full compatibility with native Kubernetes mechanisms. This makes it possible to scale applications according to external events and application-specific metrics, providing a more suitable approach for workloads based on queues, streams and other asynchronous systems. The next chapter introduces KEDA and describes how it enables event-driven autoscaling in Kubernetes environments.

2.4 Chapter Summary

This chapter presented the main autoscaling mechanisms provided by Kubernetes, namely the Cluster Autoscaler (CA), the Vertical Pod Autoscaler (VPA) and the Horizontal Pod Autoscaler (HPA). Their operating principles, objectives and areas of application were analyzed, highlighting how each mechanism addresses a different aspect of resource management within a Kubernetes cluster. Particular attention was devoted to the Horizontal Pod Autoscaler, which dynamically adjusts the number of Pod replicas according to workload metrics and constitutes the foundation upon which the autoscaling solutions investigated in this thesis are built. The chapter also discussed the advantages and limitations of traditional metric-based autoscaling approaches. In particular, it highlighted how relying exclusively on resource utilization metrics, such as CPU or memory consumption, may be insufficient for applications whose workload is driven by external events or asynchronous message streams. These limitations motivate the adoption of event-driven autoscaling solutions capable of reacting to application-specific metrics rather than infrastructure-level resource usage. Building upon these considerations, the next chapter introduces Kubernetes Event-Driven Autoscaling (KEDA), describing its architecture, operating principles and integration with the native Kubernetes autoscaling framework.

Chapter 3

KEDA: Event-Driven Autoscaling for Kubernetes

3.1 Introduction

KEDA, which stands for Kubernetes Event-Driven Autoscaling, is an open-source lightweight component designed to extend the native autoscaling capabilities of Kubernetes. Its primary goal is to enable event-driven scaling of containerized applications by allowing scaling decisions to be driven by external events rather than solely by resource utilization metrics. KEDA is particularly useful in modern cloud-native architectures, where applications are often driven by asynchronous events rather than continuous resource utilization. While Kubernetes natively supports autoscaling based on CPU and memory consumption, it lacks built-in support for directly reacting to external event sources such as message queues, streaming systems or custom application-level signals. KEDA addresses this limitation by allowing applications to scale dynamically based on a wide variety of external triggers. One of the most significant advantages commonly associated with KEDA is its ability to scale workloads down to zero replicas when no triggering events are present. This capability is not natively supported by the standard Horizontal Pod Autoscaler and can significantly reduce resource consumption in workloads characterized by intermittent activity. When new events arrive, KEDA can automatically reactivate the application by scaling it back up. The conditions under which scale-to-zero is supported will be discussed in more detail in the following sections. At the core of KEDA's architecture lies the concept of triggers, which

represent sources of events that can influence scaling decisions. Each trigger is associated with a specific scaler, which is responsible for connecting to an external system and retrieving relevant metrics or event data. KEDA supports a rich and continuously evolving ecosystem of scalers, covering a wide range of technologies such as message brokers (e.g., Kafka and RabbitMQ), cloud services, monitoring systems such as Prometheus and many others. From an architectural perspective, KEDA integrates seamlessly with Kubernetes as a lightweight add-on. It is deployed as a set of containers running as Pods within the cluster, similarly to other Kubernetes components, and does not require invasive changes to the existing Kubernetes infrastructure. This design makes KEDA easy to install, manage and extend. It can be deployed using package managers such as Helm or by applying standard Kubernetes manifests, ensuring flexibility across different environments and deployment strategies.

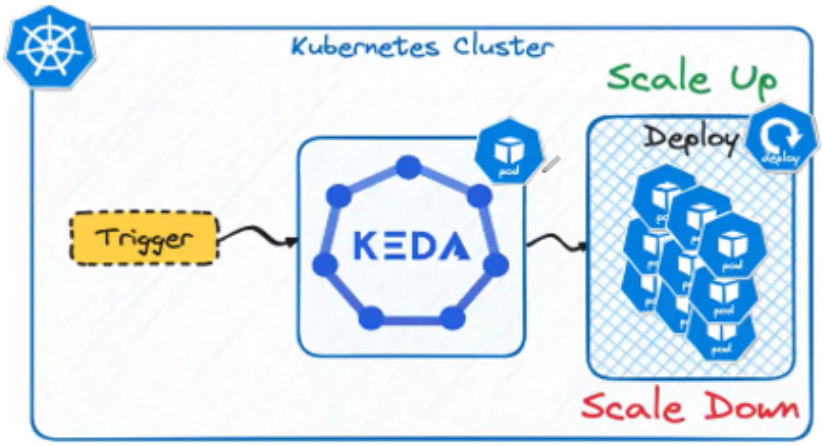


Figure 3.1: High-level overview of KEDA event-driven autoscaling. External event sources are monitored through KEDA scalers, which translate event information into scaling decisions. KEDA interacts with Kubernetes and the Horizontal Pod Autoscaler to dynamically adjust the number of application replicas according to workload demand [6].

As illustrated in Figure 3.1, KEDA acts as an intermediary layer between external event sources and Kubernetes autoscaling mechanisms. Incoming events are evaluated through the associated scalers and translated into scaling actions affecting the target workload. It is important to highlight that KEDA focuses exclusively on horizontal scaling at the Pod level, leveraging and extending

the capabilities of the Horizontal Pod Autoscaler. It does not provide vertical scaling functionalities, which remain the responsibility of mechanisms such as the Vertical Pod Autoscaler (VPA). Internally, KEDA integrates with the Horizontal Pod Autoscaler by automatically creating and managing HPA resources whenever scaling decisions involve one or more running replicas, while directly handling scale-to-zero transitions when required. Another key strength of KEDA is its event-driven nature, which distinguishes it from traditional autoscaling approaches. Instead of relying solely on periodically sampled resource metrics, KEDA enables scaling decisions based on real-time external signals. For example, an application can scale according to the number of messages waiting in a cloud-based queue service, even if that service resides outside the Kubernetes cluster. Similarly, KEDA can leverage metrics exposed by a Prometheus instance running within the cluster or custom metrics provided by user-defined applications. This flexibility allows developers to design highly responsive and efficient systems tailored to specific workload characteristics. Furthermore, KEDA integrates smoothly with existing Kubernetes metric-based scaling. Rather than replacing native autoscaling mechanisms, it extends them through a unified framework supporting both resource-based and event-driven scaling approaches. KEDA is compatible with modern Kubernetes distributions, making it a widely adoptable solution across different cloud providers and on-premise environments. The following sections provide a deeper analysis of KEDA's internal architecture, deployment model and scaling workflow, including the role of key resources such as the ScaledObject and their interaction with the Kubernetes control plane.

3.2 KEDA Architecture Overview

Once KEDA is installed in a Kubernetes cluster as a lightweight add-on, it introduces a set of internal components that operate behind the scenes to enable event-driven autoscaling. From an architectural perspective, KEDA is composed of four main components: the KEDA Operator (also referred to as the KEDA Controller), the KEDA Metrics Adapter, the Scalers and the Admission Webhook. These components are fully integrated within the native Kubernetes architecture and interact seamlessly with existing control plane elements. Importantly, KEDA does not replace Kubernetes autoscaling mechanisms but rather extends them. The actual scaling operation is still per-

formed by the Horizontal Pod Autoscaler (HPA), a native Kubernetes resource. KEDA enhances HPA by enabling it to consume external, event-driven metrics. This design highlights how KEDA augments Kubernetes without introducing architectural conflicts, acting as a non-intrusive extension that significantly broadens the range of supported scaling strategies. The interaction among the various KEDA components and the native Kubernetes control plane is illustrated in Figure 3.2:

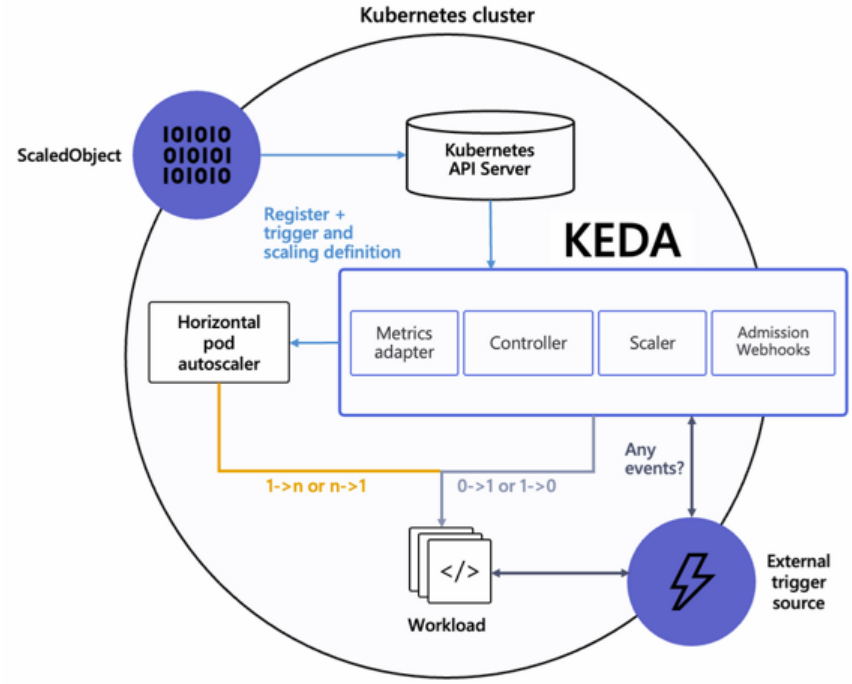


Figure 3.2: KEDA integration within the Kubernetes architecture. KEDA components interact with external event sources, the Kubernetes control plane and the Horizontal Pod Autoscaler to enable event-driven autoscaling while preserving compatibility with native Kubernetes mechanisms [7].

As illustrated in Figure 3.2, KEDA is deployed within the cluster as a collection of internal components including the Operator, Metrics Adapter, Scalers and Admission Webhook. These components interact both with Kubernetes resources and with external event sources. A key element of the architecture is the connection between KEDA and external triggers. These triggers can originate from a wide variety of systems such as message brokers (e.g., Kafka and

RabbitMQ), cloud queue services, monitoring platforms such as Prometheus or custom application-level metrics. Each trigger is associated with a specific scaler, which is responsible for collecting and interpreting information from the corresponding event source. The scaler itself does not directly perform scaling actions. Instead, it retrieves event information and transforms it into numerical metrics that can be consumed by the autoscaling system. These metrics are then propagated through the KEDA architecture and ultimately used to determine the desired number of replicas. KEDA communicates with the Horizontal Pod Autoscaler, which remains responsible for performing scaling operations involving one or more running replicas. While the HPA natively operates in the range from one to many replicas, KEDA extends this capability by enabling scale-to-zero behavior. In particular, KEDA directly manages transitions between zero and one replica, allowing workloads to completely stop when idle and automatically reactivate when new events are detected. Both KEDA and HPA ultimately influence the target workload by adjusting the number of running Pods. However, KEDA introduces an additional event-driven control layer that enables workloads to react directly to external signals, improving resource efficiency in scenarios characterized by intermittent, bursty or asynchronous traffic patterns. The architecture described above provides the foundation for understanding the role of the individual KEDA components, which will be analyzed in detail in the following sections.

3.3 KEDA Core Components

KEDA Operator and Scalers

The Scalers represent one of the fundamental building blocks of KEDA. A scaler can be defined as a software component responsible for connecting to an external event source, interpreting incoming data and producing a numerical metric that can be used for autoscaling decisions. Each scaler is specifically designed to interact with a particular type of system, such as Kafka, RabbitMQ, Prometheus or cloud-based services. KEDA currently provides a rich ecosystem of built-in scalers supporting a wide range of event sources. From an operational perspective, a scaler performs two main tasks:

- Periodically polling an external event source at a configurable interval
- Transforming the observed event into a numerical metric

The polling frequency is controlled through the `pollingInterval` field of the `ScaledObject` resource and is set to 30 seconds by default. During each polling cycle, the scaler queries the external system and extracts a relevant numerical value, such as queue length, request rate or consumer lag. This value represents the metric that will drive the scaling process. In essence, a scaler acts as a bridge between the external world and Kubernetes, converting asynchronous events into structured numerical signals that can be processed by the autoscaling framework. The KEDA Operator represents the core orchestration component of the entire architecture. It acts as the control entity responsible for managing all KEDA-specific resources and coordinating the autoscaling workflow. One of the most important aspects of KEDA installation is the introduction of a Kubernetes Custom Resource Definition (CRD) called `ScaledObject`. This resource extends the Kubernetes API and allows users to define event-driven autoscaling configurations as if they were native Kubernetes resources. When a user deploys a `ScaledObject`, the KEDA Operator performs several actions:

1. Retrieves the configuration from the Kubernetes API Server.
2. Initializes the required scaler or scalers.
3. Creates and manages a corresponding Horizontal Pod Autoscaler linked to the workload referenced through the `scaleTargetRef` field.

Additionally, the Operator coordinates the polling process and ensures that metrics generated by the scalers are correctly propagated within the KEDA architecture. This design demonstrates how KEDA integrates seamlessly with Kubernetes. Although the `ScaledObject` resource is introduced by KEDA, it is managed through the native Kubernetes API, reinforcing the idea that KEDA extends rather than replaces native Kubernetes functionalities. This architectural choice allows KEDA to support a large number of trigger types while maintaining a relatively small operational footprint within the cluster.

Important Note on Scaler Implementations
A common misconception is that each supported scaler corresponds to a dedicated Kubernetes component deployed within the cluster. In reality, KEDA does not create separate Pods or Deployments for individual scalers. Instead, scaler implementations are embedded directly within the KEDA Operator itself. When a `ScaledObject` references a specific trigger type, the Operator

dynamically instantiates the corresponding scaler implementation and uses it to retrieve metrics from the external event source. Consequently, regardless of the number of supported scaler types available in KEDA, the cluster does not require a dedicated runtime component for each scaler. This design simplifies deployment, reduces resource consumption and contributes to the lightweight nature of the KEDA architecture. This architectural choice allows KEDA to support a large number of trigger types while maintaining a relatively small operational footprint within the cluster.

KEDA Metrics Adapter

The KEDA Metrics Adapter acts as the bridge between KEDA-generated metrics and the Kubernetes Horizontal Pod Autoscaler. While scalers continuously retrieve metrics from external event sources, the HPA can only consume metrics exposed through Kubernetes metric APIs. To solve this integration challenge, KEDA deploys a dedicated Metrics Adapter implementing the Kubernetes External Metrics API. The Metrics Adapter exposes metrics obtained through the KEDA Operator and the associated scalers, making them available through the Kubernetes External Metrics API. The HPA can then retrieve these metrics through the standard Kubernetes metrics interfaces and use them to compute the desired number of replicas. From an architectural perspective, the Metrics Adapter enables seamless integration between external event sources and the native Kubernetes autoscaling framework without requiring modifications to the HPA itself. Its role can be summarized as follows:

1. Receive metrics produced by KEDA scalers
2. Expose them through the Kubernetes External Metrics API
3. Allow the HPA to consume external event-driven metrics
4. Enable scaling decisions based on non-native Kubernetes metrics

Through this mechanism, KEDA allows the native Kubernetes autoscaling framework to react to information originating outside the cluster while preserving full compatibility with existing HPA functionality. The complete interaction among the main KEDA components involved in the scaling process is summarized in Figure 3.3:

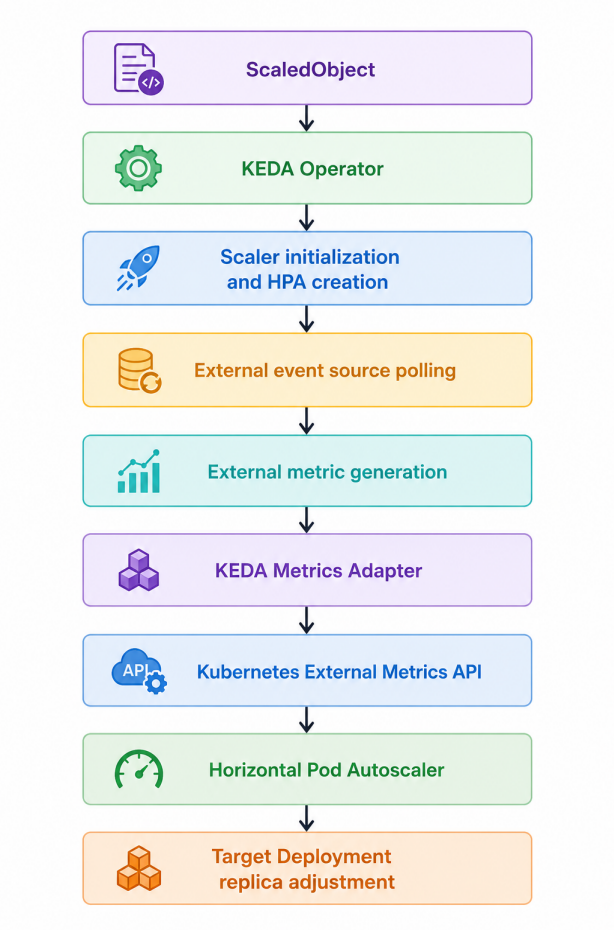


Figure 3.3: Simplified KEDA scaling workflow. The ScaledObject defines the autoscaling configuration, the KEDA Operator initializes the required scalers, the Metrics Adapter exposes external metrics through the Kubernetes External Metrics API, and the Horizontal Pod Autoscaler adjusts the number of replicas of the target workload.

As shown in Figure 3.3, KEDA acts as an intermediary layer between external event sources and the native Kubernetes autoscaling framework.

KEDA Admission Webhook

The KEDA Admission Webhook is responsible for validating KEDA-specific resources before they are persisted within the Kubernetes cluster. Its primary objective is to prevent invalid or inconsistent autoscaling configurations from

being accepted by the system. From a Kubernetes perspective, the Admission Webhook is integrated into the standard admission control process. Whenever a user attempts to create or modify a KEDA resource, such as a `ScaledObject`, the request is intercepted by the Webhook before being stored by the API Server. The Webhook then performs a series of validation checks and either accepts or rejects the operation. By detecting configuration issues early, the Admission Webhook helps improve system reliability and prevents situations that could otherwise lead to unpredictable autoscaling behavior.

Syntactic and Semantic Validation

The validation process performed by the Admission Webhook can be divided into two categories: Syntactic validation and Semantic validation. Syntactic validation focuses on verifying that the submitted resource conforms to the expected schema. This includes checks such as the presence of mandatory fields, the correctness of field types and the overall structure of the resource definition. Semantic validation, on the other hand, verifies that the configuration is logically consistent and compatible with KEDA operational requirements. These checks go beyond simple schema validation and ensure that the requested autoscaling configuration can be correctly interpreted and executed by the platform. While syntactic validation guarantees that a resource is formally correct, semantic validation ensures that it is also meaningful from an operational perspective. Syntactic validation is typically performed by verifying that resource definitions comply with the expected schema and data types defined by the Custom Resource Definition (CRD). Semantic validation, however, requires a deeper understanding of KEDA operational constraints and therefore cannot be fully enforced through schema definitions alone. For this reason, the Admission Webhook plays a crucial role in ensuring that user-provided configurations are not only syntactically correct but also operationally valid.

Examples of Semantic Validation Rules

Several validation rules implemented by the Admission Webhook focus on preventing conflicting autoscaling configurations. One important example concerns the relationship between workloads and `ScaledObject` resources. KEDA does not allow multiple `ScaledObject` resources to independently control the same target workload. If such a configuration were accepted, different autoscaling policies could simultaneously attempt to modify the number of replicas of the same Deployment, potentially leading to inconsistent behavior. Conversely, KEDA fully supports the presence of multiple triggers within a single

`ScaledObject`. In this case, each trigger is evaluated independently and generates its own desired replica count. KEDA then selects the highest recommendation and applies it as the final scaling decision. This approach enables a workload to react to multiple event sources simultaneously while avoiding conflicts between independent autoscaling resources.

Integration with Kubernetes Admission Control
The Admission Webhook is implemented through the Kubernetes admission control framework and is registered through a ValidatingWebhookConfiguration resource. Whenever a matching resource creation or update request reaches the Kubernetes API Server, the request is forwarded to the KEDA Admission Webhook for validation. Depending on the outcome of the validation process, the request is either accepted or rejected before being persisted within the cluster state. This mechanism allows KEDA to enforce autoscaling constraints while remaining fully integrated with native Kubernetes resource management workflows. The validation process occurs before the resource is persisted in the Kubernetes API Server. As a consequence, invalid configurations can be rejected before they become part of the cluster state, reducing the risk of runtime errors and simplifying troubleshooting activities.

Constraints and Supported Configurations
The Admission Webhook also enforces several operational constraints that define the supported KEDA configuration model. As previously discussed, a workload can be associated with only one `ScaledObject`. This restriction prevents multiple autoscaling resources from competing for control of the same Deployment. Within a single `ScaledObject`, however, multiple triggers can coexist. This configuration is commonly adopted when autoscaling decisions must consider multiple event sources simultaneously. In such scenarios, each trigger independently computes a desired replica count and KEDA selects the highest recommendation as the final scaling target. The rationale behind these restrictions is to avoid ambiguous scaling decisions. Allowing multiple independent `ScaledObject` resources to control the same workload could lead to conflicting scaling recommendations and unpredictable autoscaling behavior. By enforcing a single `ScaledObject` per workload while supporting multiple triggers within the same resource, KEDA provides a flexible yet deterministic scaling model. Another important aspect concerns the behavior of the Webhook itself in case of failures. Through the Kubernetes `failurePolicy` setting, administrators can determine how the API Server should behave if

the Webhook becomes unavailable. Depending on the selected policy, requests can either be rejected or allowed to proceed despite the temporary inability to perform validation. These constraints contribute to maintaining consistency and predictability within the KEDA autoscaling framework.

3.4 KEDA Components Overview

The previous sections introduced the main components that constitute the KEDA architecture and described their respective roles within the event-driven autoscaling workflow. Table 3.1 summarizes the responsibilities and deployment characteristics of the most important KEDA components.

Component	**Deployment Model**	**Runs As**	**Main Responsibility**
KEDA Operator	Kubernetes Deployment	Pod	Manages ScaledObjects, initializes scalers and creates or updates Horizontal Pod Autoscalers.
Scalers	Integrated within the Operator	Internal modules	Connect to external event sources and generate metrics used for scaling decisions.
Metrics Adapter	Kubernetes Deployment	Pod	Exposes KEDA-generated metrics through the Kubernetes External Metrics API.
Admission Webhook	Kubernetes Deployment	Pod	Validates KEDA resources before they are persisted in the Kubernetes API Server.

Table 3.1: Summary of the main KEDA components and their responsibilities.

Additional Remarks

The architecture presented throughout this chapter highlights how KEDA extends Kubernetes autoscaling capabilities without replacing native platform mechanisms. Rather than introducing a completely independent scaling framework, KEDA builds upon existing Kubernetes components, particularly the

Horizontal Pod Autoscaler, and enriches them with support for event-driven metrics originating from external systems. This design choice enables KEDA to maintain strong integration with Kubernetes while providing significantly greater flexibility for workloads characterized by asynchronous events, intermittent traffic patterns and external dependencies. Through components such as the Operator, the Metrics Adapter and the Admission Webhook, KEDA introduces a complete framework capable of transforming external events into autoscaling decisions while preserving the declarative and extensible nature of Kubernetes. The concepts introduced in this chapter provide the foundation required to understand the practical demonstrations and experimental activities presented in the following chapters.

3.5 Chapter Summary

This chapter introduced KEDA and presented its architecture, operating principles and core components. After motivating the need for event-driven autoscaling beyond the limitations of traditional resource-based approaches, the chapter described how KEDA extends the native Kubernetes autoscaling framework while maintaining full compatibility with its control plane and reconciliation mechanisms. The chapter analyzed the roles of the KEDA Operator, Scalers, Metrics Adapter and Admission Webhook, as well as the purpose of the `ScaledObject` resource and its integration with native Kubernetes autoscaling mechanisms. Particular attention was devoted to the interaction between KEDA and the Horizontal Pod Autoscaler, highlighting how KEDA periodically collects workload-specific metrics from external event sources, exposes them through the Kubernetes External Metrics API and delegates replica computation to the HPA. Finally, the chapter discussed the scale-to-zero capability introduced by KEDA, emphasizing how it enables Kubernetes workloads to be completely deactivated in the absence of events and rapidly reactivated when new workload is detected. These capabilities significantly extend the native autoscaling model of Kubernetes and provide the foundation for the experimental demonstrations presented in the remainder of this thesis. Building upon these concepts, the next chapter focuses on practical autoscaling demonstrations and introduces the experimental environment used throughout the rest of this work.

Chapter 4

Native CPU-Based Autoscaling in Kubernetes

4.1 Introduction

Before introducing KEDA-based scaling mechanisms, it is essential to thoroughly understand how native CPU-based autoscaling is implemented in Kubernetes. In particular, this chapter focuses on the Horizontal Pod Autoscaler (HPA) operating in CPU utilization mode and provides a detailed step-by-step description of the entire autoscaling workflow. Kubernetes implements autoscaling through a distributed architecture involving multiple components operating at different levels of the cluster. Resource consumption is continuously monitored at container level, processed through intermediate components and ultimately used by the Horizontal Pod Autoscaler to determine the appropriate number of pod replicas. Understanding this workflow is fundamental for two reasons. First, it provides insight into how Kubernetes performs resource-based autoscaling using native mechanisms. Second, it establishes the conceptual foundation required to understand how KEDA extends Kubernetes autoscaling capabilities through event-driven metrics, which will be discussed in subsequent chapters. The following sections analyze the components involved in native CPU-based autoscaling and describe the complete metric collection and scaling process from the generation of raw CPU usage measurements to the final scaling decisions performed by the HPA.

4.2 Native Autoscaling Components

The CPU-based autoscaling process in Kubernetes involves multiple components distributed across the cluster. Each component performs a specific task within the metric collection and scaling pipeline.

Kubelet
The kubelet is a fundamental Kubernetes component running on every Worker Node. It is responsible for managing containers on the node and exposing resource usage metrics to other components of the Kubernetes ecosystem.

cAdvisor (Container Advisor)
cAdvisor is integrated within the kubelet and continuously monitors resource consumption of running containers. In particular, it collects CPU and memory usage information by reading resource statistics from Linux cgroups.

Container Runtime
The Container Runtime, such as containerd, is responsible for executing containers on each node and interacting with the underlying operating system.

Metrics Server
The Metrics Server is a cluster-level component deployed as a Pod within the Kubernetes cluster. Its primary responsibility is collecting resource usage metrics from all nodes and making them available to autoscaling mechanisms.

Kubernetes API Server
The Kubernetes API Server acts as the central entry point of the control plane and exposes resource usage metrics through aggregated APIs.

Horizontal Pod Autoscaler (HPA)
The Horizontal Pod Autoscaler is the native Kubernetes resource responsible for dynamically adjusting the number of Pod replicas according to observed metrics.

Deployment
The Deployment represents the workload whose replica count is modified during scaling operations. The HPA interacts directly with the Deployment resource by updating its `replicas` field.

From an architectural perspective, each Worker Node contains:

- One kubelet

- One container runtime
- Multiple running containers belonging to different Pods

Together, these components form the foundation of the native Kubernetes autoscaling pipeline. The interaction among the main components involved in native CPU-based autoscaling is summarized in Figure 4.1:

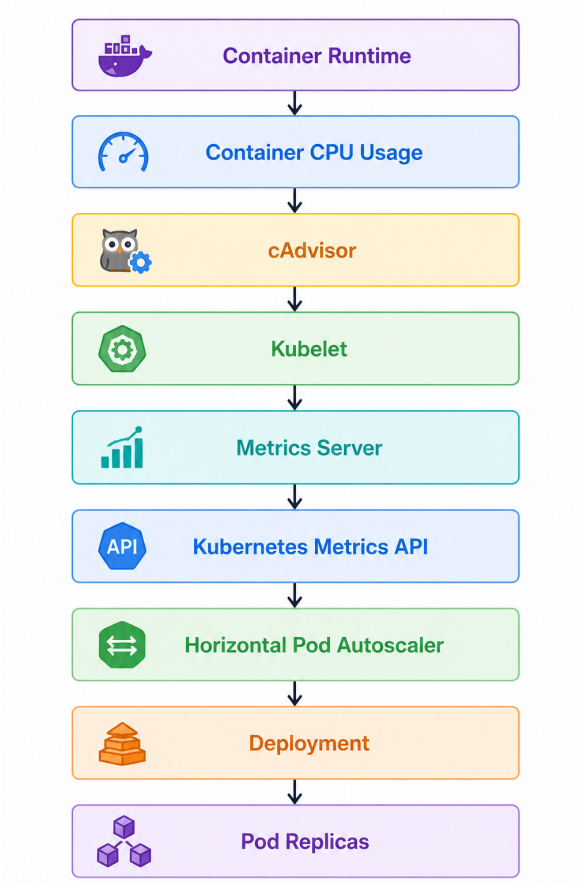


Figure 4.1: Simplified workflow of native CPU-based autoscaling in Kubernetes.

Figure 4.1 illustrates how CPU utilization metrics are progressively propagated through the native Kubernetes autoscaling pipeline. Resource usage is initially collected at the container level by cAdvisor and made available to the

kubelet, which exposes these metrics to the Metrics Server. The Metrics Server aggregates the collected information and publishes it through the Kubernetes Metrics API, allowing the Horizontal Pod Autoscaler (HPA) to retrieve the required metrics and dynamically adjust the number of replicas of the target Deployment.

4.3 Architecture and Metrics Collection Workflow

For the sake of simplicity, we initially assume that each Pod contains a single container. This assumption allows the autoscaling workflow to be introduced more clearly before extending the discussion to multi-container Pods. The native CPU-based autoscaling process can be viewed as a pipeline in which resource usage metrics are generated, collected, processed and ultimately used by the Horizontal Pod Autoscaler to determine the desired number of replicas. The following sections describe each stage of this workflow in detail.

Step 1 – Raw CPU Usage Generation
When a container is executed on a Worker Node, it consumes CPU resources of the underlying virtual machine. At the operating system level, CPU usage is not directly represented as a percentage but rather as cumulative CPU time. This metric is typically measured in nanoseconds and represents the total amount of CPU time consumed by the container since its creation. Being cumulative and monotonically increasing, this value does not provide an instantaneous view of resource utilization. Consequently, it cannot be directly used for autoscaling purposes and is therefore considered a raw metric. The objective of the subsequent stages of the autoscaling pipeline is to transform this raw CPU measurement into a meaningful metric that can be consumed by the Horizontal Pod Autoscaler.

Step 2 – Monitoring via cAdvisor and Kubelet
The cAdvisor component, embedded within the kubelet, continuously monitors all containers running on the node by collecting resource statistics from Linux cgroups. For each container, cAdvisor gathers:

- Cumulative CPU usage
- Memory consumption

- Metadata such as Pod name, namespace and container identifiers

These measurements are updated at a relatively high frequency, typically around one second, and are exposed through kubelet interfaces such as the Summary API and the `/metrics/resource` endpoint. The kubelet therefore acts as an intermediary component, exposing raw container-level metrics to higher-level services operating within the Kubernetes cluster.

Step 3 – Metrics Collection by Metrics Server
The Metrics Server is deployed as a Pod within the cluster and periodically queries all kubelets to retrieve resource usage information. By default, the Metrics Server polls kubelets approximately every 15 seconds, although this interval can be modified through the `--metric-resolution` parameter. At this stage, the Metrics Server collects cumulative CPU values for every container running across all nodes of the cluster. These values still correspond to raw cumulative measurements and therefore require further processing before becoming suitable for autoscaling decisions. The interactions among the components involved in the metric collection process are summarized in Figure 4.2:

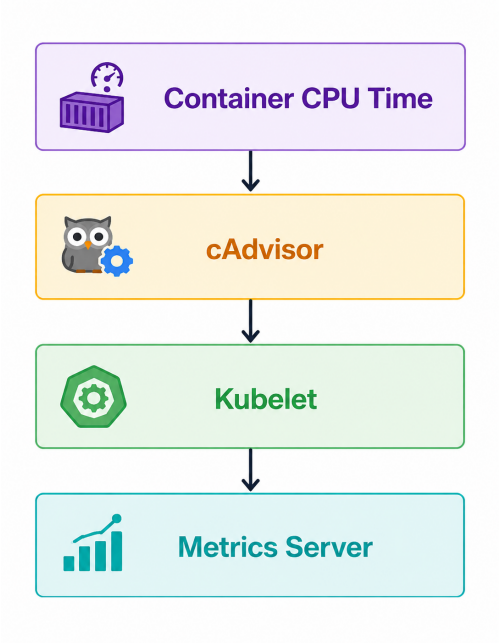


Figure 4.2: Simplified metric collection workflow.

Figure 4.2 illustrates the initial stage of the native Kubernetes autoscaling pipeline. Raw CPU usage generated by running containers is continuously monitored by cAdvisor, exposed through the kubelet and subsequently collected by the Metrics Server. The collected metrics are then made available to other Kubernetes components responsible for implementing autoscaling decisions, such as the Horizontal Pod Autoscaler.

Step 4 – CPU Usage Normalization (Rate Computation)
Since cumulative CPU values are not directly meaningful for autoscaling, the Metrics Server performs a normalization process. The objective is to compute the rate of CPU consumption over time by evaluating the variation between two consecutive measurements. The resulting metric is defined as:

$$\text{Current CPU Usage} = \frac{\Delta\text{CPU Time}}{\Delta t} \text{ [cores]}$$

where:

- ΔCPU Time represents the difference between two consecutive cumulative CPU measurements
- Δt represents the elapsed time between the two observations

The resulting value corresponds to the average CPU consumption during the observation interval and is expressed in CPU cores or, more commonly, in millicores (mCPU), where:

$$1 \text{ CPU Core} = 1000 \text{ mCPU}$$

This normalized metric provides a meaningful representation of the computational resources effectively consumed by the container during the measurement interval.

Step 5 – Aggregation from Container to Pod Level
A key challenge arises from the fact that resource monitoring occurs at container level, whereas the Horizontal Pod Autoscaler operates at Pod level. If a Pod contains multiple containers, the Metrics Server aggregates the CPU usage values of all containers belonging to the same Pod. The resulting Pod-level CPU usage is computed as:

$$\text{Current CPU Usage}_{pod} = \sum_{i=1}^{n} \text{Current CPU Usage}_i \text{ [cores]}$$

where n represents the number of containers within the Pod. The resulting metric is expressed in mCPU and represents the overall CPU consumption of the entire Pod. If the Pod contains only one container, no aggregation is required. The Metrics Server is able to correctly associate containers with Pods thanks to metadata provided by the kubelet, including Pod names, namespaces and container identifiers.

Step 6 – Exposure through the Kubernetes Metrics API
After normalization and aggregation, the Metrics Server exposes the resulting metrics through the Kubernetes Metrics API (`metrics.k8s.io`). This integration is achieved through the Kubernetes API Aggregation Layer, which allows external services such as the Metrics Server to extend the Kubernetes API transparently. At this point, the metrics are:

- Normalized
- Aggregated at Pod level
- Accessible cluster-wide

The metrics are now ready to be consumed by the Horizontal Pod Autoscaler during the scaling decision process.

Temporal Dynamics of the System
It is important to highlight that different components of the autoscaling pipeline operate at different time scales. Typical values are:

- cAdvisor sampling frequency: Approximately 1 second
- kubelet update frequency: Approximately 1 second
- Metrics Server polling interval: Approximately 15 seconds
- HPA synchronization interval: Approximately 15 seconds (default value)

These time intervals directly influence the responsiveness of the autoscaling system and determine how quickly workload variations can be detected and translated into scaling actions.

4.4 HPA Decision Logic and Scaling Execution

After the Metrics Server has exposed normalized CPU usage metrics at Pod level through the Kubernetes Metrics API, the final stage of the autoscaling workflow is handled by the Horizontal Pod Autoscaler (HPA). The HPA is responsible for retrieving metrics, computing CPU utilization, determining the desired number of replicas and ultimately triggering scaling operations on the target workload. The following sections describe this process in detail.

Step 7 – Metrics Retrieval via the API Aggregation Layer
The HPA periodically queries the Kubernetes API Server to retrieve resource usage metrics. By default, the HPA executes its control loop every 15 seconds. This interval can be modified through the Kubernetes Controller Manager parameter:

```
1 --horizontal-pod-autoscaler-sync-period
```

An important architectural aspect is that the HPA does not communicate directly with the Metrics Server. Instead, the following workflow takes place:

1. The HPA sends a request to the Kubernetes API Server
2. The API Server forwards the request through the API Aggregation Layer
3. The Metrics Server returns the requested metrics
4. The API Server forwards the response back to the HPA

This mechanism allows external services such as the Metrics Server to extend the Kubernetes API transparently while preserving a unified API interface. The interaction between the HPA and the Metrics Server through the API Aggregation Layer is summarized in Figure4.3:

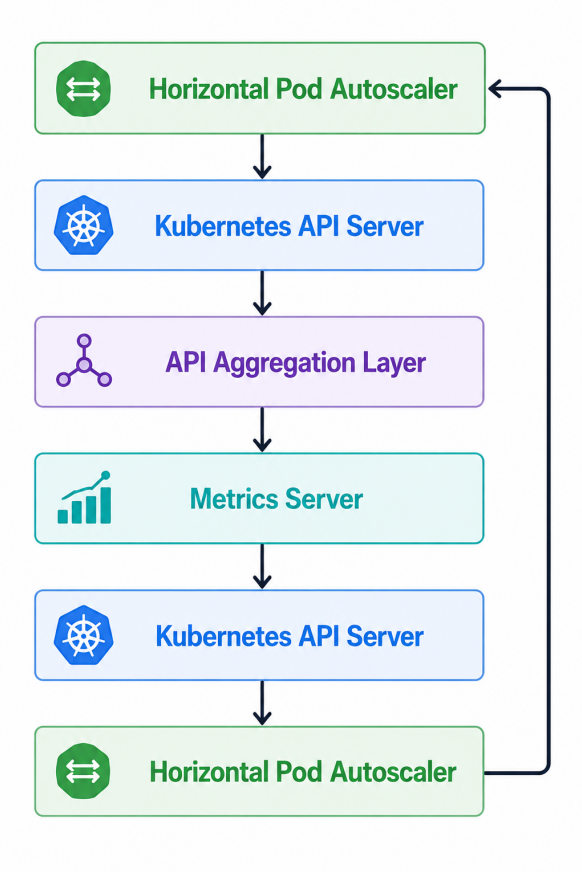


Figure 4.3: Simplified interaction between the HPA and the Metrics Server through the Kubernetes API Aggregation Layer.

From an architectural perspective, the API Aggregation Layer acts as a proxy mechanism within the Kubernetes API Server. External services such as the Metrics Server are registered through Kubernetes resources called `APIService`, which specify both the API group being extended and the location of the corresponding service within the cluster. As a result:

- The Metrics Server behaves as an extension API server
- Clients such as the HPA are unaware of its existence as a separate component
- All communication is mediated by the Kubernetes API Server

For CPU-based autoscaling, the HPA retrieves metrics through the `metrics.k8s.io` API and only requests information associated with the Pods belonging to the target Deployment specified through the `scaleTargetRef` field.

Step 8 – CPU Utilization Computation
Once the HPA receives the CPU usage values for each Pod, expressed in mCPU, it computes CPU utilization. For a generic Pod i, utilization is defined as:

$$\text{Utilization}_i = \frac{\text{Current CPU Usage}_i}{\text{CPU Request}_{pod,i}} \times 100 \ [\%]$$

where:

- Current CPU Usage$_i$ represents the current CPU consumption of the Pod
- CPU Request$_{pod,i}$ represents the CPU request associated with that Pod

The resulting value is expressed as a percentage.

Why CPU Requests Are Critical
The CPU request parameter plays a fundamental role within Kubernetes. It is defined in the Deployment manifest as:

```
resources:
  requests:
    cpu: ...
```

and serves two distinct purposes:

1. Scheduling: The Kubernetes Scheduler uses CPU requests to determine whether a node has sufficient resources to host a Pod
2. Autoscaling: The HPA uses CPU requests as the denominator in the utilization formula.

Without CPU requests, CPU utilization cannot be computed correctly, making utilization-based autoscaling impossible.

Multi-container Pods
If a Pod contains multiple containers, the Pod CPU request is computed as the sum of the CPU requests of all containers:

$$\text{CPU Request}_{pod} = \sum_{i=1}^{k} \text{CPU Request}_i \ [\text{cores}]$$

where k denotes the number of containers within the Pod.

Step 9 – Average Utilization Across Pods
When multiple replicas of the same Pod are running, the HPA computes utilization individually for each replica and then calculates the average utilization across all Pods. The resulting value is:

$$\text{Average Utilization} = \frac{1}{m} \sum_{i=1}^{m} \text{Utilization}_i \ [\%]$$

where m represents the number of Pod replicas.

Why HPA Uses the Average
The HPA relies on average utilization rather than maximum or minimum utilization values. This design choice helps:

- Avoid overreacting to isolated outliers
- Improve stability of scaling decisions
- Provide a global view of workload distribution across replicas

For example, a temporary CPU spike affecting a single Pod should not necessarily trigger scaling if the remaining replicas remain largely underutilized. By using the average, the HPA makes more balanced and robust decisions.

Example Table:

Pod	Current CPU (mCPU)	CPU Request (mCPU)	Utilization
A	200	400	50%
B	300	400	75%
C	100	400	25%

Table 4.1: Example of CPU utilization computation for multiple Pod replicas.

The average utilization is therefore:

$$\text{Average Utilization} = \frac{50 + 75 + 25}{3} = 50\%$$

Step 10 – Comparison with Target Utilization
The HPA compares the observed average utilization with a target value specified in the HPA manifest. A typical configuration is:

```
metrics:
- type: Resource
  resource:
    name: cpu
    target:
      type: Utilization
      averageUtilization: 50
```

The `averageUtilization` parameter represents the desired CPU utilization percentage. The scaling logic is straightforward:

- If Average Utilization > Target Utilization → Scale Out.
- If Average Utilization < Target Utilization → Scale In.

Step 11 – Desired Replicas Calculation
The desired number of replicas is computed according to the following formula:

$$\text{Desired Replicas} = \left\lceil \text{Current Replicas} \times \frac{\text{Average Utilization}}{\text{Target Utilization}} \right\rceil$$

where $\lceil \cdot \rceil$ denotes the ceiling function, meaning that the result is always rounded up to the nearest integer. The resulting value is constrained by two configuration parameters:

- `minReplicas`
- `maxReplicas`

For example:

```
minReplicas: 1
maxReplicas: 10
```

An important limitation of native HPA is that the minimum allowed value for `minReplicas` is 1. Consequently, native CPU-based autoscaling cannot scale workloads down to zero replicas.

Step 12 – Scaling Execution
Once the desired number of replicas has been computed, the HPA updates the `replicas` field of the target Deployment. This modification is submitted to the Kubernetes API Server, which triggers the Kubernetes reconciliation process. As a consequence:

- New Pods are created during scale-out operations
- Existing Pods are terminated during scale-in operations

The cluster then gradually converges toward the newly requested state.

4.5 Numerical Examples and Scaling Interpretation

To better understand the behavior of the Horizontal Pod Autoscaler, it is useful to analyze a few numerical examples based on the desired replicas formula introduced in the previous section.

Example 1 – Scale Out
Consider the following configuration:

- Current replicas: 4
- Average utilization: 80%
- Target utilization: 50%

Applying the HPA formula:

$$\text{Desired Replicas} = \left\lceil 4 \times \frac{80}{50} \right\rceil = \lceil 6.4 \rceil = 7$$

In this case, the observed utilization is above the target value, meaning that the system is under-provisioned. Each Pod is handling more workload than expected, and the HPA increases the number of replicas from 4 to 7.

Example 2 – Scale In
Consider now the opposite situation:

- Current replicas: 4
- Average utilization: 25%
- Target utilization: 50%

Applying the same formula:

$$\text{Desired Replicas} = \left\lceil 4 \times \frac{25}{50} \right\rceil = \lceil 2 \rceil = 2$$

In this case, the observed utilization is below the target value, meaning that the system is over-provisioned. The HPA therefore reduces the number of replicas from 4 to 2.

Scaling Philosophy
The fundamental objective of the HPA is to drive the system toward the target utilization value. When the average utilization is above the target, the current number of replicas is insufficient and each Pod is handling too much workload. The HPA performs a scale-out operation, increasing the number of replicas and distributing the workload across a larger number of Pods. As a result, the average CPU utilization per Pod decreases and moves closer to the target value. Conversely, when the average utilization is below the target, the system is over-provisioned and Pods are underutilized. The HPA performs a scale-in operation, reducing the number of replicas and increasing the workload handled by each remaining Pod. This raises the average utilization toward the target value. This feedback mechanism enables Kubernetes to dynamically adapt to workload variations while maintaining efficient resource usage.

Observation on CPU Request Indexing
In the definition of CPU utilization for a generic Pod, the utilization formula is often expressed using an index i:

$$\text{Utilization}_i = \frac{\text{Current CPU Usage}_i}{\text{CPU Request}_i} \times 100 \ [\%]$$

At first glance, the presence of the index i in the denominator may appear unnecessary. Under normal operating conditions, the Pod specification is defined once within the Deployment manifest and all replicas are generated from the same template. Consequently, every Pod replica is typically assigned the same CPU request value. During autoscaling operations, Kubernetes only modifies the number of replicas while preserving the internal Pod specification, including container images, resource requests and resource limits. As a result, in standard scenarios:

$$\text{CPU Request}_i = \text{CPU Request}$$

for all Pod replicas, and therefore the only quantity that varies among Pods is the observed CPU usage. Under these conditions, differences in utilization are entirely determined by differences in runtime CPU consumption.

Why the Index is Still Used
Although CPU requests are typically identical across all replicas, the indexed formulation represents the most general case and remains valid even in non-uniform deployment scenarios. Situations in which different replicas may temporarily have different CPU requests include:

- Rolling updates, during which old and new Pod versions coexist
- Configuration changes involving resource requests
- Transitional states in which multiple generations of the same workload are simultaneously present in the cluster

In such scenarios, the CPU request may vary across replicas and must therefore be represented as:

$$\text{CPU Request}_i$$

rather than as a single constant value. The indexed formulation of the utilization equation represents the most general and mathematically correct representation of the autoscaling process. However, in steady-state conditions characterized by a stable Deployment configuration, CPU requests are typically identical across all replicas. Under these circumstances, utilization differences between Pods are entirely driven by variations in their runtime CPU consumption rather than by differences in their resource requests.

4.6 Limitations of Native CPU-Based Autoscaling

Although CPU-based autoscaling is one of the most widely adopted scaling mechanisms in Kubernetes, it presents several limitations that may reduce its effectiveness in certain application scenarios.

Lag in Reaction Time
CPU utilization is an indirect consequence of workload execution. As a result, the HPA can react only after the workload has already started consuming

computational resources. For example, if a large number of requests suddenly arrive or a queue rapidly accumulates messages, CPU utilization will increase only once Pods begin processing the incoming workload. Consequently, scaling actions are inherently reactive rather than proactive.

Limited Suitability for Event-Driven Workloads
CPU utilization is not always the most representative indicator of application demand. In many event-driven systems, metrics such as queue length, consumer lag, request rate or backlog size provide a more accurate representation of the actual workload. Since native CPU-based autoscaling relies exclusively on resource consumption, it may fail to react optimally in these scenarios.

Dependence on CPU Requests
Utilization-based scaling relies heavily on correctly configured CPU requests. As discussed previously, CPU utilization is computed as the ratio between observed CPU consumption and the requested CPU resources. Consequently, inaccurate CPU request values may lead to incorrect utilization estimates and therefore to suboptimal scaling decisions. Overestimated requests may reduce the measured utilization and delay scaling actions, whereas underestimated requests may artificially increase utilization and trigger unnecessary scale-out operations.

Lack of Application-Level Semantics
CPU utilization represents a low-level infrastructure metric and does not capture the semantic meaning of application behavior. Two applications may exhibit identical CPU utilization while experiencing completely different operational conditions. For example, an application processing a large backlog of pending messages may report the same CPU usage as an idle application performing background activities. As a consequence, CPU-based autoscaling cannot directly account for application-specific information that may be more relevant for scaling decisions.

These limitations motivate the adoption of more advanced autoscaling approaches capable of reacting directly to external events and application-level metrics. KEDA addresses these challenges by extending Kubernetes autoscaling beyond traditional resource-based signals and enabling scaling decisions driven by event sources such as message queues, monitoring systems and custom application metrics.

4.7 Alternative CPU-Based Scaling: AverageValue Mode

So far, the discussion has focused on the most commonly adopted CPU-based autoscaling strategy in Kubernetes, namely utilization-based scaling. In this mode, scaling decisions are driven by CPU utilization expressed as a percentage of the requested resources. However, Kubernetes also supports an alternative CPU-based scaling approach known as *AverageValue* scaling. In this mode, scaling decisions are based directly on CPU consumption rather than on CPU utilization percentages.

Configuration in the HPA Manifest
AverageValue scaling is configured within the HPA manifest as follows:

```
metrics:
- type: Resource
  resource:
    name: cpu
    target:
      type: AverageValue
      averageValue: 200m
```

Key Differences from Utilization-Based Scaling
Unlike utilization-based scaling, where CPU usage is normalized with respect to CPU requests, AverageValue scaling operates directly on CPU usage measurements. In particular:

- The target value is expressed in millicores (mCPU)
- The HPA retrieves CPU usage values directly from the Metrics Server
- No utilization computation is required
- CPU requests are not involved in the scaling formula

The overall autoscaling workflow remains largely unchanged, with the exception that the HPA directly compares observed CPU consumption against a target CPU consumption value.

Average CPU Usage Computation

The HPA computes the average CPU consumption across all Pod replicas according to:

$$\text{Average CPU Usage} = \frac{1}{m}\sum_{i=1}^{m}\text{Current CPU Usage}_i \text{ [mCPU]}$$

where:

- m represents the number of Pod replicas
- Current CPU Usage$_i$ represents the CPU consumption of Pod i

The resulting value is then compared directly with the configured `averageValue` target.

Scaling Logic
The scaling logic remains conceptually similar to utilization-based scaling:

- If Average CPU Usage¿Target AverageValue, the HPA performs a scale-out operation
- If Average CPU Usage¡Target AverageValue, the HPA performs a scale-in operation

The difference lies in the fact that the comparison is performed directly on CPU usage values rather than on utilization percentages.

Comparison with Utilization Mode
The main differences between the two CPU-based scaling modes are summarized in Table 4.2:

Aspect	**Utilization Mode**	**AverageValue Mode**
Metric used	CPU utilization (%)	CPU usage (mCPU)
Depends on CPU requests	Yes	No
Requires normalization	Yes (Usage / Request)	No
Target parameter	`averageUtilization`	`averageValue`
Scaling decision based on	Percentage utilization	Absolute CPU consumption

Table 4.2: Comparison between the two CPU-based autoscaling modes supported by the Horizontal Pod Autoscaler.

Important Observation on CPU Requests
Although CPU requests are not directly involved in scaling decisions when using the AverageValue mode, they must still be specified within the Pod definition. This requirement stems from the role CPU requests play within the Kubernetes scheduling process. The Scheduler relies on requested resources to determine whether a node has sufficient capacity to host a Pod and to decide where workloads should be placed within the cluster. Consequently, AverageValue scaling removes the dependency between autoscaling decisions and CPU requests, but it does not eliminate the need to define resource requests within the workload specification. CPU requests therefore remain essential for correct cluster operation, even when they are not directly used by the HPA scaling algorithm.

4.8 Chapter Summary

This chapter presented the complete workflow of native CPU-based autoscaling in Kubernetes, from the generation of raw CPU usage measurements to the scaling decisions performed by the Horizontal Pod Autoscaler. The roles of the kubelet, cAdvisor, Metrics Server, Kubernetes API Server and HPA were analyzed in detail, together with the metric collection process, CPU utilization computation and replica calculation algorithm. Particular attention was devoted to the mathematical foundations of utilization-based autoscaling and to the interaction between the HPA and the Kubernetes control plane. The chapter also illustrated the complete metrics collection pipeline, highlighting how CPU usage measurements are progressively propagated through the Kubernetes monitoring infrastructure before being consumed by the HPA. Furthermore, the different target metric configurations supported by the Horizontal Pod Autoscaler were discussed, emphasizing the differences between the *Utilization* and *AverageValue* modes and the scenarios in which each approach is most appropriate. Finally, the limitations of native CPU-based autoscaling were analyzed, showing how resource-based metrics alone may not accurately represent the workload of event-driven applications or asynchronous processing systems. These concepts provide the theoretical foundation required to understand the practical demonstration presented in the next chapter, where the behavior of the Horizontal Pod Autoscaler will be analyzed experimentally.

Chapter 5

Experimental Evaluation of Native CPU-Based Autoscaling

5.1 Experimental Objective

The objective of this experiment is to validate the behavior of the Kubernetes Horizontal Pod Autoscaler (HPA) when configured in CPU utilization mode. In particular, the experiment demonstrates how Kubernetes automatically adapts the number of running application instances according to variations in CPU resource consumption. By generating a controlled workload and monitoring the resulting scaling actions, it is possible to observe how the HPA continuously compares the measured CPU utilization with the configured target value and adjusts the number of replicas accordingly. More specifically, the experiment aims to:

- Generate a controlled CPU workload
- Observe scale-out and scale-in operations
- Analyze how CPU utilization evolves as replicas are added or removed
- Correlate the observed behavior with the theoretical HPA model presented in the previous chapter

This experiment therefore represents the practical validation of the native CPU-based autoscaling workflow previously discussed. The experimental setup involves several Kubernetes resources, each contributing to a specific stage of

the native autoscaling workflow. Their respective roles are summarized in Table 5.1:

Kubernetes Resource	**Role in the Experiment**
Metrics Server	Collects CPU and memory metrics from kubelets and exposes them through the Kubernetes Metrics API, enabling CPU-based autoscaling.
Deployment	Deploys the NGINX application and represents the workload whose number of replicas is managed by the HPA.
Service	Provides a stable ClusterIP endpoint and automatically distributes incoming requests among all available Pod replicas.
Horizontal Pod Autoscaler	Continuously monitors CPU utilization and automatically adjusts the number of Deployment replicas according to the configured target utilization.
Load Generator	Generates a controlled CPU workload by continuously sending HTTP requests to the application through the Kubernetes Service.

Table 5.1: Main Kubernetes resources involved in the native CPU-based autoscaling experiment and their respective roles.

5.2 Metrics Server Setup and Verification

Metrics Server Requirement
Before proceeding with the experiment, it is essential to verify that the Metrics Server is correctly installed and operational. As discussed in the previous chapter, the Horizontal Pod Autoscaler relies on resource consumption metrics to make scaling decisions. These metrics are collected from kubelets running on cluster nodes and exposed through the Kubernetes Metrics API by the Metrics Server. Consequently, the Metrics Server represents a mandatory component for CPU-based autoscaling. In a standard Kubernetes cluster, the core control-plane components typically include:

- Kubernetes API Server

- Scheduler
- Controller Manager
- Kubelet (running on each Worker Node)

Depending on the Kubernetes distribution, the Metrics Server may or may not be installed by default. Therefore, its presence should always be verified before performing autoscaling experiments. If the Metrics Server is not already installed, it can be deployed using:

```
kubectl apply -f https://github.com/kubernetes-sigs/metrics-server/
releases/latest/download/components.yaml
```

Once installed, the Metrics Server:

- Runs as a Pod managed by a Deployment
- Is typically deployed inside the `kube-system` namespace
- Exposes resource metrics through the Kubernetes Metrics API (`metrics.k8s.io`).

Metrics Server Verification
Before testing the Horizontal Pod Autoscaler, it is necessary to verify that the Metrics Server is correctly installed and operational. The first verification consists of checking whether the Deployment exists inside the `kube-system` namespace:

```
kubectl get deployment metrics-server -n kube-system
```

If the Deployment exists and reports the desired number of available replicas, the Metrics Server is correctly deployed. A second verification consists of executing:

```
kubectl top nodes
```

If CPU and memory statistics are displayed without errors, the Metrics Server is correctly collecting and exposing resource metrics. Conversely, if the Metrics Server is missing or unavailable, commands such as:

```
kubectl top nodes
kubectl top pods
```

typically return:

```
error: Metrics API not available
```

indicating that no resource metrics are currently available within the cluster.

5.3 Resource Metrics Observation

Role of the `kubectl top` Command
The `kubectl top` command provides a simplified real-time view of resource consumption for both cluster nodes and application Pods. However, it does not directly measure CPU or memory usage. Instead, it acts as a client for the Kubernetes Metrics API, querying the following endpoints:

```
/apis/metrics.k8s.io/v1beta1/nodes
/apis/metrics.k8s.io/v1beta1/pods
```

The overall workflow is therefore:

- The Metrics Server collects resource statistics from kubelets
- It processes and normalizes the collected measurements
- It publishes aggregated metrics through the Kubernetes Metrics API
- The `kubectl top` command retrieves these metrics from the API Server

The interaction among the components involved in metric observation is summarized in Figure 5.1:

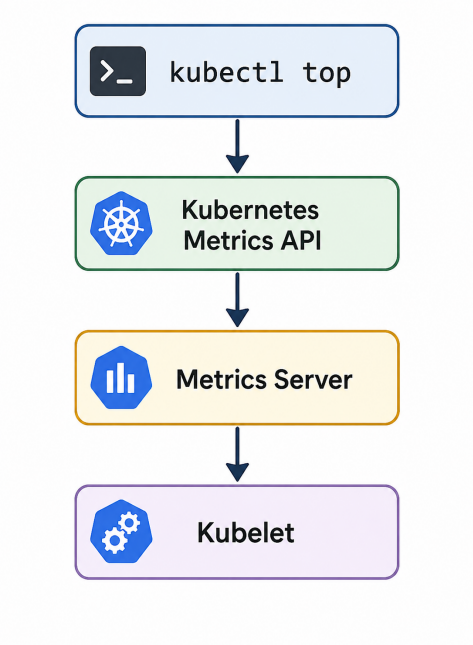


Figure 5.1: Workflow followed by the `kubectl top` command to retrieve resource usage metrics from the Kubernetes cluster.

Node-Level Monitoring

Resource usage at node level can be inspected through the following command:

```
kubectl top nodes
```

A typical output is shown below:

NAME	CPU(cores)	CPU%	MEMORY(bytes)	MEMORY%
node-1	250m	12%	1024Mi	45%

Table 5.2: Example of node resource utilization reported by the Metrics Server.

This command provides an overview of the current resource utilization of each Worker Node in the cluster. In particular, it reports:

- Current CPU consumption of the node
- CPU utilization percentage relative to the node capacity

- Current memory consumption
- Memory utilization percentage relative to the node capacity

Although node-level metrics are not directly used by the Horizontal Pod Autoscaler, they provide valuable information about the overall resource usage of the cluster and help verify that sufficient computational resources are available during autoscaling experiments.

Pod-Level Monitoring
The resource consumption of individual application Pods can instead be monitored using:

```
kubectl top pods
```

A typical output is:

NAME	CPU(cores)	MEMORY(bytes)
cpu-demo-abc	150m	80Mi
cpu-demo-def	200m	90Mi

Table 5.3: Example of Pod resource utilization reported by the Metrics Server.

This command reports:

- CPU consumption of each running Pod
- Memory consumption of each running Pod

CPU usage is expressed in millicores (mCPU), whereas memory usage is reported in mebibytes (MiB). For CPU-based autoscaling experiments, the values displayed by `kubectl top pods` provide a convenient approximation of the metrics observed by the Horizontal Pod Autoscaler through the Kubernetes Metrics API.

Accessing Raw Metrics
Besides the simplified output provided by `kubectl top`, resource metrics can also be accessed directly through the Kubernetes Metrics API. Node-level metrics can be retrieved using:

```
kubectl get --raw "/apis/metrics.k8s.io/v1beta1/nodes" | jq
```

Similarly, Pod-level metrics can be obtained through:

```
kubectl get --raw "/apis/metrics.k8s.io/v1beta1/pods" | jq
```

Unlike the output of `kubectl top`, the raw Metrics API exposes additional information regarding the metric collection process, including:

- Collection timestamp
- Observation window
- Raw CPU usage values
- Raw memory usage values

A typical response is shown below:

```
{
  "timestamp": "2026-06-06T08:29:40Z",
  "window": "10.027s",
  "usage": {
    "cpu": "188291612n",
    "memory": "4185636Ki"
  }
}
```

Listing 5.1: Example of a node metrics response returned by the Kubernetes Metrics API.

Interpretation of CPU Values

An important difference between the raw Metrics API and the output of `kubectl top` concerns the representation of CPU consumption. The Kubernetes Metrics API expresses CPU usage in nanocores (`n`), whereas `kubectl top` automatically converts these values into millicores (mCPU) to improve readability. For example:

$$188291612\,n = 188.29\,mCPU \approx 0.188\,\text{CPU cores}$$

This conversion explains why the values displayed by `kubectl top` differ from those returned by the raw Metrics API while representing exactly the same CPU consumption.

Meaning of the Observation Window

Another important field exposed by the Metrics API is the observation window:

```
"window": "10.027s"
```

This value should not be confused with either the polling interval of the Metrics Server or the synchronization period of the Horizontal Pod Autoscaler. Instead, it represents the actual time interval used to compute CPU usage from the cumulative CPU counters collected by the kubelet. As discussed in the previous chapter, CPU usage is computed according to:

$$\text{CPU Usage} = \frac{\Delta\text{CPU Time}}{\Delta\text{Time}} \ [\text{cores}]$$

where:

- ΔCPU Time represents the variation of the cumulative CPU counters
- ΔTime corresponds to the observation window reported by the Metrics API

Since the observation interval is determined by the actual sampling process, its duration may vary slightly between consecutive measurements and is therefore not necessarily equal to the Metrics Server polling interval.

Relationship with the Horizontal Pod Autoscaler
The Horizontal Pod Autoscaler retrieves resource metrics through the same Kubernetes Metrics API used by the `kubectl top` command. Consequently:

- `kubectl top pods` provides visibility into the CPU metrics available to the HPA
- The HPA retrieves these metrics through the Kubernetes Metrics API
- The HPA computes CPU utilization relative to the configured CPU requests
- Scaling decisions are derived from the resulting utilization values

Therefore,

$$\texttt{kubectl top} \approx \text{View of the metrics available to the HPA}$$

although the Horizontal Pod Autoscaler performs additional utilization computations before making scaling decisions.

Practical Use During the Experiment
Throughout the experimental evaluation, the following commands are executed simultaneously:

```
kubectl get hpa -w
kubectl top pods
```

Executing both commands in parallel makes it possible to:

- Monitor CPU consumption in real time
- Observe the CPU utilization reported by the HPA
- Correlate metric variations with autoscaling decisions
- Understand how changes in the number of replicas influence the average CPU utilization

Key Experimental Insight
By combining the information provided by:

- `kubectl top pods`, which exposes the CPU metrics collected by the Metrics Server
- `kubectl get hpa -w`, which continuously reports the HPA status and scaling decisions

it becomes possible to experimentally verify the native Kubernetes autoscaling mechanism. More specifically, the experiment demonstrates that:

- Scaling decisions are based on the average CPU utilization across all running Pods
- Increasing the number of replicas distributes the workload among multiple instances
- The CPU utilization of each individual Pod decreases as new replicas become available
- The system progressively converges toward the configured target utilization

5.4 Application Deployment

The application used throughout this experiment consists of a simple NGINX Deployment configured with a single replica and explicit CPU resource requirements. The complete Deployment manifest used throughout the experiment is reported in Listing 5.2:

File: nginx-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: cpu-demo
spec:
  replicas: 1
  selector:
    matchLabels:
      app: cpu-demo
  template:
    metadata:
      labels:
        app: cpu-demo
    spec:
      containers:
      - name: nginx
        image: nginx
        resources:
          requests:
            cpu: "100m"
          limits:
            cpu: "500m"
        ports:
        - containerPort: 80
```

Listing 5.2: Deployment manifest used in the native CPU-based autoscaling experiment.

Internal Workflow

Once the Deployment manifest is submitted to the Kubernetes API Server, the standard Kubernetes reconciliation process takes place. The overall workflow can be summarized as follows:

- The manifest is submitted to the Kubernetes API Server
- The desired state is stored inside etcd
- The Deployment Controller creates the corresponding ReplicaSet
- The ReplicaSet creates the requested Pod
- The Scheduler assigns the Pod to a Worker Node
- The kubelet starts the NGINX container on the selected node

This sequence corresponds exactly to the Deployment lifecycle introduced in Chapter 1.

Most Important Fields:

```
1 replicas: 1
```

The Deployment initially creates a single Pod replica. Any subsequent modification of the replica count is performed automatically by the Horizontal Pod Autoscaler during the experiment.

```
1 image: nginx
```

A standard NGINX container image is used as the application under test. Although NGINX is a lightweight web server, it provides a simple HTTP endpoint that can easily be stressed through synthetic traffic generation, making it particularly suitable for demonstrating CPU-based autoscaling.

```
1 resources.requests.cpu: "100m"
```

This parameter represents the most important configuration value for the entire experiment. The CPU request specifies the amount of CPU resources guaranteed to the container by the Kubernetes Scheduler:

$$100m = 0.1 \text{ CPU core}$$

As discussed in the previous chapter, the Horizontal Pod Autoscaler computes CPU utilization relative to this value. For example:

$$\frac{50m}{100m} \times 100 = 50\%$$

Similarly:

$$\frac{80m}{100m} \times 100 = 80\%$$

Consequently, autoscaling decisions are not based on absolute CPU consumption but rather on CPU utilization relative to the configured CPU request.

```
resources.limits.cpu: "500m"
```

The CPU limit specifies the maximum amount of CPU resources that the container is allowed to consume:

$$500m = 0.5 \text{ CPU core}$$

If the application attempts to exceed this threshold, Kubernetes throttles CPU usage. Although the CPU limit is not directly involved in the HPA utilization calculation, it prevents individual containers from consuming excessive computational resources during load-generation experiments.

```
containerPort: 80
```

The container exposes the standard HTTP port used by NGINX. This endpoint will receive the requests generated during the experiment and therefore represents the entry point through which CPU utilization will be increased.

Deployment Creation

The Deployment is created using:

```
kubectl apply -f nginx-deployment.yaml
```

Deployment Verification

After applying the manifest, the Deployment can be verified through:

```
kubectl get pods
```

This command allows monitoring the Pod lifecycle as it transitions through states such as `Pending`, `Running` and `Ready`. A successful output confirms that the Deployment has correctly instantiated a single running replica of the `cpu-demo` application. This verification also confirms that the application is ready for the subsequent autoscaling experiment.

5.5 Service Resource Creation

Once the application Deployment has been created, a Kubernetes Service is required to provide a stable network endpoint through which the application can be accessed during the experiment. As discussed in Chapter 1, Pods are ephemeral resources whose IP addresses may change over time. A Service provides a permanent virtual IP address and DNS name, allowing clients to communicate with the application independently of the currently running Pod replicas. The complete Service manifest used throughout the experiment is reported in Listing 5.3:

File: nginx-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: cpu-demo-service
spec:
  selector:
    app: cpu-demo
  ports:
  - port: 80
    targetPort: 80
```

Listing 5.3: ClusterIP Service manifest used to expose the application within the Kubernetes cluster.

Most Important Fields:

```
selector
```

The selector associates the Service with the Pods created by the Deployment through their labels. In this experiment, all Pods labeled with:

```
app: cpu-demo
```

automatically become endpoints of the Service. As the Horizontal Pod Autoscaler creates or removes Pod replicas, the Service dynamically updates its list of endpoints without requiring any manual intervention.

```
port and targetPort
```

The Service exposes port 80 and forwards incoming HTTP requests to port 80 of each backend Pod. Consequently, clients communicate only with the Service, while Kubernetes transparently forwards requests to one of the available Pod replicas.

Role in the Experiment
Since no Service type is explicitly specified, Kubernetes creates a ClusterIP Service, which is the default Service type. The Service is assigned a virtual IP address that is reachable from within the cluster and acts as a stable entry point for the application. Although the Service does not directly participate in autoscaling decisions, it plays a fundamental role during the experiment. The load generator sends HTTP requests to the Service rather than to individual Pods. Kubernetes then distributes the incoming requests among all available Pod replicas matching the Service selector. Consequently, the Service effectively acts as an internal load balancer for all NGINX replicas managed by the Deployment. This behavior becomes particularly important during autoscaling:

1. Initially, all requests are handled by a single Pod
2. As the HPA creates additional replicas, the Service automatically starts distributing traffic across multiple Pods
3. The workload becomes shared among replicas
4. The average CPU utilization per Pod decreases
5. The HPA progressively converges toward the configured target utilization

This interaction between the Service and the HPA is fundamental to understanding horizontal autoscaling. While the HPA continuously evaluates the average CPU utilization across all replicas, the Service ensures that incoming traffic is evenly distributed among them, allowing the workload to be balanced as the number of Pods increases.

Service Creation
The Service is created through:

```
kubectl apply -f nginx-service.yaml
```

Service Verification
After applying the Service manifest, the following command is used to verify that the Service has been correctly created within the cluster:

```
kubectl get svc
```

This command lists all Services in the default namespace, including the newly created `cpu-demo-service`. A successful output confirms that:

- The Service resource exists
- A ClusterIP address has been assigned
- The Service is ready to route incoming requests to the Pods selected through its label selector

This verification is essential to ensure that the application is reachable from within the cluster and that traffic will be correctly distributed across all running Pod replicas during the autoscaling experiment.

5.6 HPA Manifest Configuration

After deploying the application and exposing it through a Service, the Horizontal Pod Autoscaler can be configured. The HPA continuously monitors CPU utilization and automatically adjusts the number of Pod replicas according to the target utilization specified by the user. The complete HPA manifest used in this experiment is reported in Listing 5.4:

File: hpa.yaml

```
1  apiVersion: autoscaling/v2
2  kind: HorizontalPodAutoscaler
3  metadata:
4    name: cpu-demo-hpa
5  spec:
6    scaleTargetRef:
7      apiVersion: apps/v1
8      kind: Deployment
9      name: cpu-demo
10   minReplicas: 1
11   maxReplicas: 10
```

```
12  behavior:
13    scaleUp:
14      stabilizationWindowSeconds: 0
15    scaleDown:
16      stabilizationWindowSeconds: 60
17  metrics:
18  - type: Resource
19    resource:
20      name: cpu
21      target:
22        type: Utilization
23        averageUtilization: 50
```

Listing 5.4: Horizontal Pod Autoscaler manifest configured in CPU utilization mode.

The configuration presented in Listing 5.4 corresponds to the theoretical concepts introduced in Chapter 4. In particular, the HPA monitors CPU utilization, maintains at least one running replica, limits the maximum number of replicas to ten, and targets an average CPU utilization of 50%.

Most Important Fields:

```
1 scaleTargetRef}
```

The `scaleTargetRef` field identifies the Kubernetes workload managed by the Horizontal Pod Autoscaler. In this experiment, the HPA continuously monitors the CPU utilization of the Pods created by the `cpu-demo` Deployment and automatically updates its replica count whenever scaling actions are required.

```
1 minReplicas
```

The minimum number of replicas is set to one. As discussed in the previous chapter, native CPU-based autoscaling cannot reduce the number of replicas below one because the HPA requires at least one running Pod to continuously collect CPU utilization metrics.

```
1 maxReplicas
```

The maximum number of replicas is limited to ten. This upper bound prevents uncontrolled resource consumption while still providing sufficient scaling capacity for the workload generated during the experiment.

```
1 metrics
```

The `metrics` section specifies which metric is monitored by the Horizontal Pod Autoscaler to make scaling decisions. In this experiment, the HPA relies on native CPU resource metrics collected by the Metrics Server. More specifically, CPU utilization is evaluated relative to the configured CPU requests of the Pods, and the resulting average utilization is continuously compared with the configured target value.

```
1 averageUtilization
```

The target CPU utilization is configured to 50%. Consequently, the Horizontal Pod Autoscaler continuously attempts to maintain the average CPU utilization of all running Pods close to this target value. Whenever the observed utilization exceeds 50%, additional Pod replicas are created. Conversely, when the utilization falls below the target value, the HPA progressively removes unnecessary replicas while respecting the configured scaling behavior.

Practical Interpretation
The configured target utilization can be directly related to the CPU request specified in the Deployment manifest. Since each Pod requests:

$$100m = 0.1 \text{ CPU core}$$

and the target utilization is set to 50%, the HPA attempts to maintain an average CPU consumption of approximately:

$$100m \times 50\% = 50m$$

per Pod. Consequently, whenever the average CPU consumption significantly exceeds approximately 50 mCPU per Pod, the HPA considers the workload to be under-provisioned and initiates a scale-out operation. Conversely, when the average CPU consumption falls below this threshold, the HPA progressively removes unnecessary replicas while respecting the configured scaling behavior. This relationship provides a practical interpretation of the configured utilization target and helps explain the scaling decisions observed during the experiment.

HPA Creation
The Horizontal Pod Autoscaler is created using:

```
kubectl apply -f hpa.yaml
```

HPA Verification
After applying the HPA manifest, the following command is used to verify that the Horizontal Pod Autoscaler has been correctly created:

```
kubectl get hpa
```

Successful output reports:

NAME	TARGETS	MINPODS	MAXPODS	REPLICAS
cpu-demo-hpa	45%/50%	1	10	1

Table 5.4: Example of the Horizontal Pod Autoscaler status reported by `kubectl get hpa`.

The meaning of each column is summarized below:

- **NAME**: Name of the Horizontal Pod Autoscaler resource
- **REFERENCE**: Workload managed by the HPA.
- **TARGETS**: Current CPU utilization compared with the configured target utilization.
- **MINPODS**: Minimum allowed number of replicas.
- **MAXPODS**: Maximum allowed number of replicas.
- **REPLICAS**: Current number of running replicas.
- **AGE**: Elapsed time since the HPA resource was created.

At this stage, the Horizontal Pod Autoscaler is fully configured and continuously monitors the CPU utilization of the application Pods. Since no workload has yet been generated, the Deployment remains at its minimum number of replicas while the HPA waits for CPU utilization to exceed the configured target before triggering scale-out operations.

5.7 Initial System Steady State Verification

Before generating any workload, it is important to verify that the application is running under normal operating conditions and that all Kubernetes resources have been correctly deployed. At this stage, the Deployment, Service and Horizontal Pod Autoscaler have already been created, but no requests are being sent to the application. Consequently, the CPU utilization is expected to remain close to zero and the HPA should maintain the minimum number of configured replicas. This initial verification establishes the baseline of the experiment and provides a reference point against which the subsequent scaling behavior can be evaluated.

Monitoring Commands
During this phase, the following commands are executed in parallel:

```
kubectl get hpa -w
```

and

```
kubectl top pods
```

The first command continuously monitors the state of the Horizontal Pod Autoscaler, while the second reports the CPU and memory consumption of the running Pods. Executing both commands simultaneously allows the CPU utilization observed by the HPA to be correlated with the actual resource consumption of the application.

Initial HPA Status
A typical output of the HPA before generating any workload is shown below:

NAME	TARGETS	MINPODS	MAXPODS	REPLICAS
cpu-demo-hpa	0%/50%	1	10	1

Table 5.5: Example of the Horizontal Pod Autoscaler status under low CPU utilization.

This output confirms that the current CPU utilization is well below the configured target utilization of 50%. Since no workload is present, the HPA maintains the minimum number of replicas specified by the `minReplicas` parameter.

Why "unknown" May Appear
Immediately after creating the Horizontal Pod Autoscaler, the `TARGETS` column may temporarily display:

```
unknown/50%
```

Listing 5.5: Example of the HPA target value when metrics are temporarily unavailable.

The HPA can compute CPU utilization only after the Metrics Server has collected and exposed at least one valid set of resource metrics through the Kubernetes Metrics API. Once the Metrics Server completes its first metric collection cycle and the HPA retrieves the corresponding CPU measurements, the `unknown` value is automatically replaced by the actual CPU utilization. If the `TARGETS` field appears as `unknown`, it typically indicates that the Metrics Server is not yet providing data or is not properly configured.

Initial Pod Resource Usage
The resource consumption of the running Pod can be observed through:

```
kubectl top pods
```

A typical output is:

NAME	CPU(cores)	MEMORY(bytes)
cpu-demo-xxxxxxxxxx	1m	4Mi

Table 5.6: Example of Pod resource utilization under low CPU load.

The observed CPU usage is extremely low because the NGINX application is running in an idle state and is not processing client requests. Consequently, the CPU utilization computed by the HPA also remains close to zero, confirming that no scaling action is required.

Interpretation
This verification confirms that the experimental environment has been correctly configured. More specifically:

- The Deployment is running correctly
- The Service is available

- The Metrics Server is collecting resource metrics
- The Horizontal Pod Autoscaler is successfully retrieving CPU utilization values
- No scaling actions are triggered while the application remains idle

Establishing this initial steady-state condition is essential before introducing the synthetic workload used to evaluate the native CPU-based autoscaling mechanism.

5.8 Load Generation

After verifying that the application is running under steady-state conditions, a controlled CPU workload is generated to trigger the Horizontal Pod Autoscaler. The workload is produced by deploying a dedicated Pod that continuously sends HTTP requests to the application through the ClusterIP Service created in the previous section. This approach allows the generated traffic to be automatically distributed among all available Pod replicas as the HPA performs scale-out operations. Consequently, the experiment can demonstrate how CPU utilization evolves and how the Horizontal Pod Autoscaler reacts to changing workload conditions. The complete manifest used to deploy the load generator is reported in Listing 5.6.

File: load-generator.yaml

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: load-generator
5  spec:
6    containers:
7    - name: busybox
8      image: busybox
9      command: ["/bin/sh"]
10     args:
11       - -c
12       - |
13         while true; do
14           wget -q -O- http://cpu-demo-service:80;
```

```
15          done
```

Listing 5.6: Load generator Pod used to produce a controlled CPU workload.

Most Important Fields:

```
1 image
```

The load generator uses a lightweight container image capable of executing simple shell commands. Its only purpose is to continuously generate HTTP requests toward the NGINX application during the experiment, without introducing unnecessary computational overhead.

```
1 command
```

The container executes a shell script immediately after startup. Rather than serving an application, the Pod continuously generates HTTP requests directed to the Kubernetes Service associated with the NGINX Deployment.

Infinite Loop
An infinite loop is used to continuously repeat the request generation process. This ensures that the workload remains constant throughout the experiment, allowing the CPU utilization of the application Pods to progressively increase until the HPA performs the corresponding scaling actions.

HTTP Requests
Each iteration sends an HTTP request to the ClusterIP Service rather than directly to a specific Pod. This choice is fundamental because the Service automatically distributes incoming requests across all available replicas created by the Horizontal Pod Autoscaler. As a consequence, the generated workload follows the scaling process naturally, without requiring any modification to the load generator itself.

Sleep Interval
A short delay is introduced between consecutive requests to avoid generating an unrealistically aggressive workload. The sleep interval provides a controlled and repeatable traffic pattern, making the autoscaling behavior easier to observe and analyze.

Load Generator Deployment
The load generator is deployed using:

```
kubectl apply -f load-generator.yaml
```

Load Generator Verification
The successful creation of the load generator Pod can be verified through:

```
kubectl get pods
```

A successful output confirms that the load generator Pod is running together with the NGINX application Pod.

Role in the Experiment
The load generator represents the component responsible for initiating the autoscaling process. Unlike the Horizontal Pod Autoscaler, which only reacts to changes in CPU utilization, the load generator actively creates the workload that produces those CPU variations. During the experiment, the generated HTTP requests are first handled by a single application Pod. As CPU utilization increases beyond the configured target, the HPA creates additional replicas. The Kubernetes Service then automatically distributes incoming requests among the newly created Pods, reducing the average CPU utilization of each replica. This interaction between the load generator, the Service and the Horizontal Pod Autoscaler demonstrates the complete native autoscaling workflow introduced in the previous chapters. The next section analyzes the scaling behavior observed after the workload is generated, correlating CPU utilization, replica count and HPA decisions throughout the experiment.

5.9 Scaling Behavior Analysis

The deployment of the load generator marks the beginning of the autoscaling experiment. From this point onward, the Horizontal Pod Autoscaler continuously monitors the CPU utilization of the application Pods and compares the observed value with the configured target utilization of 50%. As the generated workload progressively increases the CPU consumption of the NGINX application, the HPA evaluates whether additional replicas are required to restore the desired utilization level. Throughout the experiment, the scaling behavior is monitored using the same commands introduced in the previous section, allowing CPU consumption and replica count to be correlated in real time.

Monitoring During the Experiment
Throughout the experiment, the following commands are executed simultaneously:

```
kubectl get hpa -w
```

and

```
kubectl top pods
```

The first command continuously reports the current status of the Horizontal Pod Autoscaler, while the second provides the CPU and memory consumption of each running Pod. Observing both outputs simultaneously makes it possible to directly correlate CPU utilization with the scaling decisions performed by the HPA.

Initial Scale-Out
Immediately after the load generator starts producing HTTP requests, the CPU utilization of the application Pod begins to increase. Once the observed utilization exceeds the configured target of 50%, the Horizontal Pod Autoscaler computes a larger desired number of replicas according to the scaling algorithm presented in the previous chapter. Initially, only one application Pod is available. Consequently, all incoming requests are processed by a single replica, resulting in a rapid increase in CPU utilization.

Progressive Replica Creation
As additional Pod replicas become available, the Kubernetes Service automatically distributes the incoming HTTP requests among all running instances. Instead of concentrating the workload on a single Pod, the requests are progressively balanced across multiple replicas. Consequently, the average CPU utilization per Pod decreases, allowing the Horizontal Pod Autoscaler to progressively converge toward the configured target utilization.

Interpretation
The observed behavior perfectly matches the theoretical operation of the Horizontal Pod Autoscaler described in Chapter 4. As the CPU utilization increases above the configured threshold, the HPA creates additional replicas. The Kubernetes Service then distributes incoming requests among the newly created Pods, reducing the average CPU utilization of each individual replica. Rather than attempting to minimize CPU utilization, the HPA continuously seeks an equilibrium in which the average utilization remains close to the configured target value.

Workload Removal

Once the scale-out behavior has been successfully observed, the synthetic workload is removed by deleting the load generator Pod. The load generator is removed using:

```
kubectl delete -f load-generator.yaml
```

Alternatively, the Pod can be deleted directly using its name:

```
kubectl delete pod load-generator
```

After the load generator is removed, no additional HTTP requests are sent to the application Service. Consequently, the CPU utilization of all application Pods rapidly decreases.

Scale-In Behavior
Although CPU utilization immediately drops after removing the workload, the Horizontal Pod Autoscaler does not instantly reduce the number of replicas. Instead, Kubernetes waits for the configured scale-down stabilization window before initiating replica removal. This behavior prevents rapid oscillations in the number of replicas caused by temporary workload fluctuations. Only after the stabilization window expires does the HPA progressively reduce the number of replicas until the Deployment returns to its minimum replica count. This behavior shows the fundamental operating principle of Kubernetes native autoscaling: The system continuously adjusts the number of replicas in order to maintain the average CPU utilization close to the desired target value. As workload increases, additional replicas are created; when workload decreases, unnecessary replicas are removed, allowing resources to be used more efficiently.

Final Observation
At the end of the experiment, the system returns to the same steady-state condition observed before workload generation. The application again consists of a single running Pod, CPU utilization returns close to zero and the Horizontal Pod Autoscaler remains ready to react to future workload variations.

5.10 Chapter Summary

This chapter experimentally validated the native CPU-based autoscaling workflow introduced in Chapter 4. Starting from a single NGINX Pod, a controlled

workload was generated through a dedicated load generator. The resulting increase in CPU utilization triggered the Horizontal Pod Autoscaler, which progressively increased the number of application replicas. As new Pods became available, the Kubernetes Service automatically distributed incoming requests among them, reducing the average CPU utilization and allowing the HPA to converge toward the configured target utilization. After removing the synthetic workload, CPU utilization rapidly decreased. However, the HPA delayed the scale-in operation according to the configured stabilization window before progressively returning the Deployment to its minimum replica count. Overall, the experiment confirmed the theoretical behavior of native CPU-based autoscaling discussed in the previous chapter and demonstrated the interaction among the Metrics Server, the Kubernetes API Server, the Horizontal Pod Autoscaler, the Deployment and the ClusterIP Service. The following chapter extends this analysis by introducing KEDA-based CPU autoscaling and comparing its behavior with the native Kubernetes implementation.

Chapter 6

KEDA Integration with Native CPU-Based Autoscaling

6.1 Introduction

In this chapter, the CPU-based autoscaling experiment previously implemented using the native Kubernetes Horizontal Pod Autoscaler (HPA) is revisited by integrating KEDA into the autoscaling workflow. At first glance, this scenario may appear unusual. CPU-based autoscaling is already fully supported by Kubernetes through the Metrics Server and the Horizontal Pod Autoscaler without requiring any additional components. Furthermore, CPU utilization is an internal resource metric rather than an external event source, making this example fundamentally different from the event-driven use cases for which KEDA was originally designed. Nevertheless, this scenario provides an excellent opportunity to illustrate how KEDA integrates with existing Kubernetes autoscaling mechanisms without replacing them. Rather than introducing an alternative autoscaling pipeline, KEDA leverages the native Kubernetes infrastructure while simplifying its configuration through the ScaledObject abstraction. The main objectives of this chapter are:

- Demonstrate that KEDA can also be adopted for traditional resource-based autoscaling scenarios.
- Show that KEDA does not replace the native Kubernetes autoscaling pipeline.

- Highlight that existing Kubernetes components remain fully operational and continue performing their original responsibilities.
- Illustrate how KEDA provides a unified abstraction layer through the ScaledObject resource, simplifying autoscaling configuration and management.

This example therefore serves as a bridge between native Kubernetes autoscaling and the event-driven autoscaling scenarios discussed in the following chapters.

6.2 Architectural Considerations

One of the most common misconceptions regarding KEDA is that it introduces a dedicated CPU metrics collection mechanism similar to the architecture adopted by its event-driven scalers. This is not the case. Unlike RabbitMQ, Kafka, Prometheus, Cron or other event-driven scalers, CPU-based autoscaling in KEDA does not rely on a KEDA-specific metrics pipeline. Instead, the underlying architecture remains essentially identical to the native Kubernetes implementation. The overall CPU metrics pipeline is summarized in Figure 6.1:

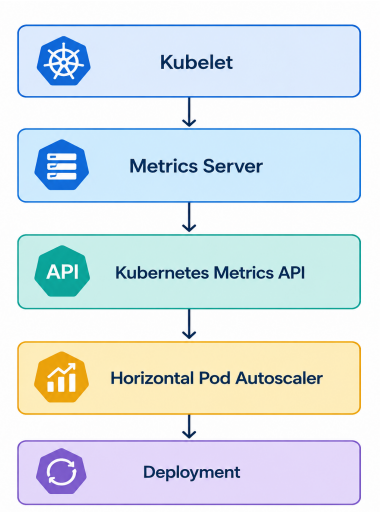


Figure 6.1: Native CPU metrics pipeline preserved when CPU-based autoscaling is configured through KEDA.

As illustrated in Figure 6.1, all CPU resource metrics continue to flow through the standard Kubernetes metrics pipeline. KEDA does not introduce an alternative mechanism for collecting CPU utilization, but instead relies entirely on the native Kubernetes infrastructure.

Role of the Metrics Server
The Metrics Server continues to perform exactly the same responsibilities as in the native Kubernetes implementation. More specifically, it:

- Collects CPU and memory resource consumption metrics from the kubelets running on the cluster nodes
- Aggregates and normalizes the collected resource usage information
- Exposes CPU and memory metrics through the Kubernetes Metrics API

Consequently, no KEDA component is involved in the collection of CPU resource metrics.

Role of the Horizontal Pod Autoscaler
Similarly, the Horizontal Pod Autoscaler continues to execute the same autoscaling algorithm discussed in the previous chapters. In particular, the HPA:

- Retrieves CPU metrics through the Kubernetes API Server
- Computes CPU utilization relative to the configured resource requests
- Calculates the desired number of replicas
- Performs scale-out and scale-in operations according to the computed utilization

Therefore, even when KEDA is introduced, the Horizontal Pod Autoscaler remains the component responsible for executing the actual scaling decisions.

Important Observation
Consequently, KEDA does not introduce a dedicated CPU scaler responsible for collecting metrics from the Kubernetes nodes. All CPU measurements continue to be provided through the standard Kubernetes Metrics Server infrastructure, while the HPA remains responsible for evaluating CPU utilization and performing scaling decisions. The integration of KEDA therefore does not modify the native CPU metrics pipeline but instead builds upon the existing Kubernetes autoscaling architecture.

6.3 ScaledObject-Based CPU Autoscaling

Although the underlying autoscaling pipeline remains unchanged, the configuration model differs significantly from the native Kubernetes approach. In the native Kubernetes approach, the user explicitly creates and manages a Horizontal Pod Autoscaler resource that references the target Deployment. When KEDA is adopted, the user no longer defines an HPA directly. Instead, the autoscaling configuration is expressed through a `ScaledObject`, which represents the primary resource managed by KEDA. The KEDA Operator continuously monitors the cluster for newly created ScaledObjects and automatically generates the corresponding Horizontal Pod Autoscaler targeting the specified Deployment. Consequently, the user interacts exclusively with the ScaledObject resource, while the lifecycle of the HPA is entirely managed by KEDA. As a result, the autoscaling logic is expressed through a unified KEDA abstraction regardless of the metric source being used. The differences between the native Kubernetes approach and the KEDA-based approach are summarized in Table 6.1:

Native Kubernetes	**KEDA**
User explicitly creates the Horizontal Pod Autoscaler.	User creates a ScaledObject resource.
The HPA lifecycle is manually managed by the user.	The HPA is automatically generated and managed by the KEDA Operator.
Autoscaling configuration is defined directly inside the HPA.	Autoscaling configuration is defined through the ScaledObject abstraction.
CPU metrics are provided by the Metrics Server.	CPU metrics are still provided by the Metrics Server.
The HPA performs the scaling decisions.	The generated HPA continues to perform the scaling decisions.

Table 6.1: Comparison between native Kubernetes CPU autoscaling and KEDA-based CPU autoscaling.

As shown in Table 6.1, the primary difference introduced by KEDA concerns

the autoscaling configuration model rather than the autoscaling algorithm itself. The Metrics Server, the Kubernetes Metrics API and the Horizontal Pod Autoscaler continue to perform exactly the same responsibilities discussed in the previous chapter.

ScaledObject Definition for CPU Autoscaling
KEDA introduces the `ScaledObject` Custom Resource Definition (CRD), which specifies the autoscaling configuration through a unified interface independently of the trigger type. For CPU utilization-based autoscaling, the trigger configuration typically includes the following definition:

Example CPU Trigger Configuration

```
triggers:
- type: cpu
  metricType: Utilization
  metadata:
    value: "50"
```

The specified threshold is automatically translated into the corresponding Horizontal Pod Autoscaler configuration. More specifically, the value configured inside the ScaledObject is mapped to the `averageUtilization` field of the HPA automatically generated by KEDA.

AverageValue Metric Type
KEDA also supports an alternative configuration mode based on the average absolute CPU consumption instead of CPU utilization. In this case, the trigger is configured as follows:

Example CPU Trigger Using AverageValue

```
triggers:
- type: cpu
  metricType: AverageValue
  metadata:
    value: "200"
```

Unlike the previous configuration, the threshold is expressed in millicores (mCPU) rather than as a percentage of the configured CPU requests. The specified value is automatically mapped to the `averageValue` field of the Horizontal Pod Autoscaler generated by KEDA. Consequently, the two trigger configurations correspond directly to the two CPU target types supported by

the native Horizontal Pod Autoscaler while allowing both configurations to be expressed through the same ScaledObject abstraction.

6.4 Workflow

Once the ScaledObject has been created, the overall autoscaling workflow closely resembles the native Kubernetes CPU-based autoscaling process. The main difference lies in the initial configuration phase, where the ScaledObject replaces the manually defined Horizontal Pod Autoscaler. The complete workflow can be summarized as follows:

1. The user applies the ScaledObject manifest
2. The ScaledObject resource is submitted to the Kubernetes API Server
3. Kubernetes validates the resource syntax, while the KEDA Admission Webhook performs semantic validation
4. The KEDA Operator detects the newly created ScaledObject and automatically generates the corresponding Horizontal Pod Autoscaler targeting the specified Deployment
5. Once the HPA has been created, the autoscaling process follows the standard Kubernetes CPU-based workflow

At this point, the native Kubernetes components resume their standard responsibilities:

- The Metrics Server collects CPU resource metrics from the cluster nodes
- The Horizontal Pod Autoscaler retrieves these metrics through the Kubernetes API Server
- CPU utilization is computed relative to the configured resource requests
- The HPA calculates the desired number of replicas and performs the corresponding scale-out or scale-in operations

The complete workflow is summarized in Figure 6.2:

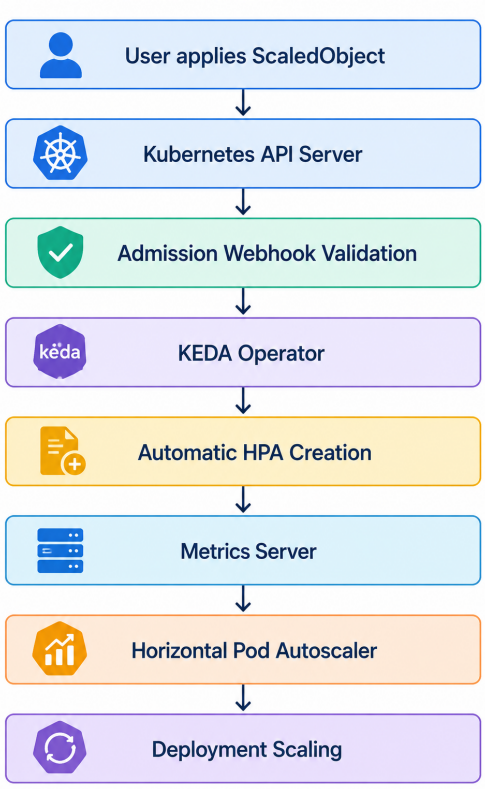


Figure 6.2: Workflow of CPU-based autoscaling using KEDA.

As illustrated in Figure 6.2, KEDA participates only in the initial configuration stage by creating and managing the Horizontal Pod Autoscaler. Once the HPA has been generated, the remaining autoscaling process is identical to the native Kubernetes implementation discussed in the previous chapter.

Important Observation
This behavior represents one of the main differences between CPU-based autoscaling and the event-driven scenarios that will be introduced in the following chapters. For CPU utilization-based autoscaling, KEDA primarily acts as an orchestration and abstraction layer rather than as a metrics provider. Unlike event-driven scalers such as RabbitMQ, Kafka or Prometheus, KEDA does not retrieve CPU metrics directly. Instead, it delegates metric collection to

the native Kubernetes Metrics Server and relies on the automatically generated Horizontal Pod Autoscaler to perform the scaling decisions.

6.5 Metrics Server Requirement

Even when CPU-based autoscaling is configured through KEDA, the Metrics Server remains a mandatory component of the autoscaling architecture. Unlike event-driven scalers, CPU utilization metrics cannot be obtained from external systems and must therefore continue to be collected directly from the Kubernetes nodes. Consequently, the Metrics Server preserves exactly the same responsibilities discussed in the previous chapter:

- It collects CPU and memory resource metrics from the cluster nodes
- It exposes these metrics through the Kubernetes Metrics API
- It provides the resource information consumed by the Horizontal Pod Autoscaler

Therefore, introducing KEDA does not eliminate or replace the Metrics Server with the Metrics Adapter. Instead, the generated Horizontal Pod Autoscaler continues to retrieve CPU metrics exactly as in the native Kubernetes implementation. This requirement clearly distinguishes CPU-based autoscaling from the event-driven scenarios discussed in the following chapters, where KEDA can retrieve metrics directly from external systems without relying on the Kubernetes Metrics Server.

Limitations
Although CPU-based autoscaling is configured through KEDA, this approach still inherits all the intrinsic limitations of traditional resource-based autoscaling. In particular:

- Scaling decisions remain reactive because they are based on observed resource consumption rather than on workload predictions
- CPU utilization is an internally generated resource metric and therefore cannot anticipate future workload variations
- The overall autoscaling process remains dependent on the availability of the Metrics Server

The most significant limitation concerns one of KEDA's most distinctive capabilities: scale-to-zero. Since CPU utilization can only be measured while at least one Pod is running, the system must always maintain a minimum of one active replica. Consequently, CPU-based autoscaling managed through KEDA remains subject to the same minimum replica constraints as the native Horizontal Pod Autoscaler. These limitations highlight that, in this particular scenario, KEDA primarily simplifies the autoscaling configuration rather than extending the capabilities of the underlying autoscaling algorithm. Rather than replacing Kubernetes components, KEDA acts as a lightweight abstraction layer that allows autoscaling to be expressed through the ScaledObject resource while preserving the standard Horizontal Pod Autoscaler workflow. The true value of KEDA becomes evident when moving beyond resource-based autoscaling toward event-driven scenarios, where scaling decisions are driven directly by external systems such as message brokers, monitoring platforms and scheduled events. These use cases are presented in the following chapters.

6.6 Chapter Summary

This chapter introduced the integration of KEDA with native CPU-based autoscaling, demonstrating that KEDA can be adopted even for traditional resource-based scaling scenarios. Unlike event-driven use cases, CPU-based autoscaling does not require a dedicated KEDA metrics pipeline. The Metrics Server, Kubernetes Metrics API and Horizontal Pod Autoscaler continue to perform exactly the same responsibilities as in the native Kubernetes implementation. The primary difference lies in the configuration model: instead of manually defining a Horizontal Pod Autoscaler, the user creates a ScaledObject, from which KEDA automatically generates and manages the corresponding HPA. Although this approach simplifies autoscaling configuration through a unified abstraction, it does not overcome the intrinsic limitations of CPU-based autoscaling, including its reactive nature and the inability to scale workloads down to zero replicas. The following chapter experimentally validates the concepts presented here by implementing the same CPU-based autoscaling scenario using KEDA and comparing its behavior with the native Kubernetes implementation discussed in the previous chapters.

Chapter 7

Experimental Evaluation of KEDA Integration with Native CPU-Based Autoscaling

7.1 Experimental Objective

The objective of this experiment is to reproduce the CPU-based autoscaling scenario previously implemented using native Kubernetes mechanisms, this time leveraging KEDA. Unlike the event-driven use cases explored in the following chapters, this experiment does not introduce a new scaling model. Instead, it demonstrates how KEDA integrates with the existing Kubernetes autoscaling infrastructure while preserving the native CPU-based scaling workflow. More specifically, the experiment aims to:

- Validate the integration between KEDA and the native Kubernetes autoscaling mechanisms
- Demonstrate the automatic creation of the Horizontal Pod Autoscaler through the ScaledObject resource
- Compare the observed behavior with the native Kubernetes implementation presented in the previous chapter
- Highlight the architectural differences between resource-based (CPU/Memory) autoscaling and event-driven autoscaling in KEDA

As discussed in the previous chapter, this experiment primarily validates the integration model adopted by KEDA for CPU-based autoscaling. Since the native Kubernetes metrics pipeline remains unchanged, the observed scaling behavior is expected to closely match that of the native Horizontal Pod Autoscaler while demonstrating the additional abstraction provided by the ScaledObject resource.

The experimental setup involves several Kubernetes resources, including the ScaledObject introduced by KEDA. Their respective roles in the experiment are summarized in Table 7.1:

Kubernetes Resource	**Role in the Experiment**
Metrics Server	Collects CPU resource metrics from the cluster nodes and exposes them through the Kubernetes Metrics API, enabling CPU-based autoscaling.
Deployment	Deploys the NGINX application whose number of Pod replicas is automatically managed during the experiment.
Service	Provides a stable ClusterIP endpoint and automatically distributes incoming requests across all available Pod replicas.
ScaledObject	Defines the autoscaling configuration and instructs KEDA to automatically generate the corresponding Horizontal Pod Autoscaler.
HPA (generated by KEDA)	Executes the actual scaling decisions by continuously monitoring CPU utilization and adjusting the number of Deployment replicas.
Load Generator	Produces the HTTP workload required to trigger autoscaling actions during the experiment.

Table 7.1: Main Kubernetes resources involved in the KEDA-based CPU autoscaling experiment and their respective roles.

7.2 Prerequisites and KEDA Installation

Before starting the experiment, the Kubernetes environment must satisfy a number of prerequisites. More specifically, the following components must already be available:

- A running Kubernetes cluster.
- Metrics Server correctly installed and operational.
- KEDA installed within the cluster.

It is important to note that, unlike the event-driven KEDA scalers introduced in the following chapters, CPU-based autoscaling still relies on the Kubernetes Metrics Server. Consequently, the Metrics Server remains a mandatory component even when KEDA is used.

KEDA Installation
KEDA can be installed using Helm through the following commands:

```
helm repo add kedacore https://kedacore.github.io/charts
helm repo update
helm install keda kedacore/keda \
    --namespace keda \
    --create-namespace
```

Installation Verification
After the installation has completed, the following command can be used to verify that all KEDA components are correctly deployed:

```
kubectl get pods -n keda
```

A successful installation should report the KEDA main components. In particular, the KEDA namespace contains three fundamental components:

- **KEDA Operator**, responsible for monitoring ScaledObjects and automatically generating the corresponding Horizontal Pod Autoscalers
- **KEDA Metrics API Server**, which exposes metrics required by event-driven scalers
- **KEDA Admission Webhooks**, responsible for validating KEDA resources before they are accepted by the Kubernetes API Server

Although the Metrics API Server is deployed as part of the KEDA installation, it does not actively participate in CPU-based autoscaling. As discussed in the previous chapter, CPU metrics continue to be collected through the native Kubernetes Metrics Server, while the KEDA Metrics API Server becomes essential only for event-driven scaling scenarios.

7.3 Application Resources

The `nginx-deployment.yaml`, `nginx-service.yaml` and `load-generator.yaml` descriptor files used throughout this experiment are identical to those presented in the previous native Kubernetes implementation and are therefore not repeated here. Since the integration of KEDA does not modify the application deployment or the networking configuration, these resources preserve exactly the same responsibilities discussed in the previous chapter. More specifically:

- The **Deployment** hosts the NGINX application that will be automatically scaled
- The **Service** provides a stable ClusterIP endpoint and distributes incoming requests across all running Pod replicas
- The **Load Generator** produces the synthetic workload required to trigger autoscaling operations

Consequently, the only component introduced in this experiment is the KEDA `ScaledObject`, which replaces the manually created Horizontal Pod Autoscaler while leaving all other Kubernetes resources unchanged.

Deployment of Application Resources
The application resources are deployed using the same manifests adopted in the previous experiment.

```
kubectl apply -f nginx-deployment.yaml
```

This command creates the NGINX Deployment that will be used as the scaling target throughout the experiment.

```
kubectl apply -f nginx-service.yaml
```

This command creates the ClusterIP Service providing stable access to the application and automatically distributing incoming requests across all Deployment replicas.

Verification
After deploying the application resources, the following command is used to verify that the Deployment has been correctly created:

```
kubectl get pods
```

A successful output confirms that:

- One running Pod associated with the `cpu-demo` Deployment is present.

The Service can be verified through:

```
kubectl get svc
```

A successful output confirms that:

- The `cpu-demo-service` ClusterIP Service has been successfully created.
- The Service is correctly associated with the Pods managed by the Deployment.

After verifying that both resources are operational, the autoscaling configuration can be introduced through the KEDA ScaledObject resource, which represents the only architectural difference compared with the native Kubernetes implementation discussed in the previous chapter.

7.4 ScaledObject Configuration

Unlike the previous experiment, where autoscaling was configured by manually creating a Horizontal Pod Autoscaler, KEDA introduces the `ScaledObject` Custom Resource Definition (CRD) as the primary resource used to define the autoscaling behavior. Once applied, the ScaledObject instructs the KEDA Operator to automatically generate and manage the corresponding Horizontal Pod Autoscaler targeting the specified Deployment. The complete ScaledObject manifest used throughout this experiment is reported in Listing 7.1:

File: scaledobject.yaml

```yaml
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: cpu-demo-scaledobject
spec:
  scaleTargetRef:
    name: cpu-demo
  minReplicaCount: 1
  maxReplicaCount: 10
  advanced:
    horizontalPodAutoscalerConfig:
      behavior:
        scaleUp:
          stabilizationWindowSeconds: 0
        scaleDown:
          stabilizationWindowSeconds: 30
  triggers:
  - type: cpu
    metricType: Utilization
    metadata:
      value: "50"
```

Listing 7.1: ScaledObject manifest used in the KEDA CPU-based autoscaling experiment.

Most Important Fields:

```
scaleTargetRef
```

The `scaleTargetRef` field specifies the Kubernetes Deployment managed by the ScaledObject. In this experiment, the generated Horizontal Pod Autoscaler targets the `cpu-demo` Deployment, allowing KEDA to automatically manage its number of replicas according to the observed CPU utilization.

```
minReplicaCount
```

The minimum number of replicas is configured to one. As discussed in the previous chapter, CPU-based autoscaling cannot scale the application down to zero replicas because CPU utilization can only be measured while at least one Pod is running. Consequently, the minimum replica count remains equal to one, exactly as in the native Kubernetes implementation.

```
maxReplicaCount
```

The maximum number of replicas is limited to ten. This upper bound prevents uncontrolled resource consumption while still providing sufficient scaling capacity for the workload generated during the experiment.

CPU Trigger

The trigger configuration is defined as:

```
triggers:
- type: cpu
  metricType: Utilization
  metadata:
    value: "50"
```

This configuration instructs KEDA to automatically generate a Horizontal Pod Autoscaler targeting an average CPU utilization of 50%. Conceptually, this configuration is equivalent to the following native Horizontal Pod Autoscaler definition:

```
metrics:
- type: Resource
  resource:
    name: cpu
    target:
      type: Utilization
      averageUtilization: 50
```

Therefore, although the configuration is expressed through a ScaledObject, the generated Horizontal Pod Autoscaler ultimately performs exactly the same CPU-based autoscaling operations discussed in the previous chapter.

Important Observation

Unlike the event-driven KEDA scalers discussed in the following chapters, this CPU-based ScaledObject does not explicitly define parameters such as `pollingInterval` or `cooldownPeriod`. These parameters are fundamental in event-driven scenarios because they determine how frequently KEDA evaluates external triggers and how scale-to-zero transitions are managed. However, in CPU-based autoscaling, scaling decisions continue to be performed directly by the native Kubernetes Horizontal Pod Autoscaler using metrics provided by the Metrics Server. Consequently, `pollingInterval` and `cooldownPeriod` do

not play a significant role in this scenario and have intentionally been omitted from the ScaledObject definition, leaving their default values unchanged. Their purpose and impact will be thoroughly analyzed in the following chapters dedicated to event-driven KEDA scalers.

ScaledObject Creation
The ScaledObject is created using:

```
kubectl apply -f scaledobject.yaml
```

7.5 Automatic HPA Generation and Verification

After applying the ScaledObject, KEDA automatically generates the corresponding Horizontal Pod Autoscaler targeting the specified Deployment. Unlike the native Kubernetes implementation discussed in the previous chapter, the user does not explicitly create an HPA resource. Instead, the KEDA Operator detects the newly created ScaledObject and transparently creates and manages the corresponding HPA on behalf of the user. This automatic resource generation represents one of the main advantages of KEDA, allowing autoscaling to be configured through a single abstraction independently of the trigger type.

What Happens Internally
After the ScaledObject is applied, the following sequence of operations takes place:

1. The ScaledObject is submitted to the Kubernetes API Server
2. Kubernetes performs syntactic validation of the resource definition
3. The KEDA Admission Webhook performs semantic validation
4. The KEDA Operator detects the newly created ScaledObject
5. KEDA automatically generates a Horizontal Pod Autoscaler targeting the specified Deployment

Once the HPA has been created, the autoscaling process continues through the standard Kubernetes CPU-based autoscaling workflow.

Important Observation

A key observation is that no dedicated KEDA CPU scaler is responsible for collecting resource metrics from the cluster nodes. Unlike RabbitMQ, Kafka, Cron, Prometheus and the other event-driven scalers discussed in the following chapters, CPU utilization continues to follow the native Kubernetes metrics pipeline. The native CPU metrics pipeline used after the automatic HPA creation is summarized in Figure 7.1:

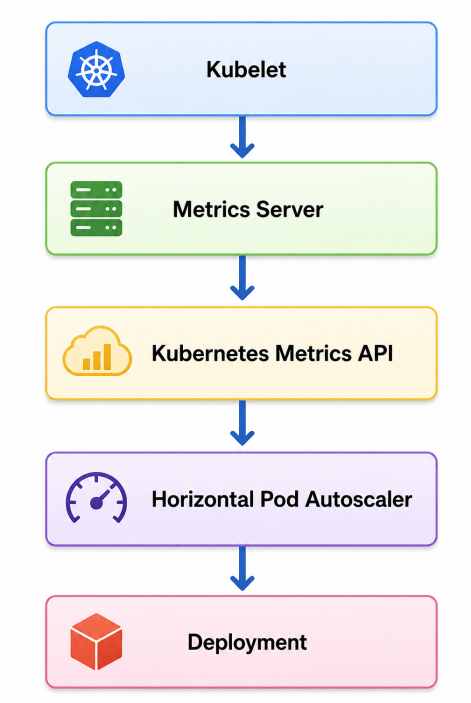


Figure 7.1: Native CPU metrics pipeline used after the HPA is automatically generated by KEDA.

As shown in Figure 7.1, KEDA does not actively participate in the collection of CPU metrics after creating the Horizontal Pod Autoscaler. Instead, CPU utilization continues to be collected by the Metrics Server, exposed through the Kubernetes Metrics API and consumed by the HPA, exactly as in the

native Kubernetes implementation. Consequently, KEDA primarily acts as an abstraction and automation layer rather than as a metrics provider.

Verification
The automatic creation of the Horizontal Pod Autoscaler can be verified through:

```
kubectl get hpa
```

A valid output confirms that:

- An HPA resource has been automatically generated by KEDA
- The generated HPA correctly targets the `cpu-demo` Deployment
- The configured CPU utilization threshold has been successfully propagated from the ScaledObject to the generated HPA

This verification confirms that KEDA has successfully translated the ScaledObject configuration into a fully operational Horizontal Pod Autoscaler without requiring any manual HPA definition.

7.6 Load Generation and Monitoring

Once the Horizontal Pod Autoscaler has been automatically generated and verified, the workload can be introduced to evaluate the CPU-based autoscaling behavior. As in the previous native Kubernetes experiment, workload generation is performed through the same Load Generator Pod. Since neither the application nor the traffic generation mechanism changes, the `load-generator.yaml` manifest introduced in the previous chapter is reused without modifications.

Load Generation
The workload is initiated by deploying the Load Generator Pod:

```
kubectl apply -f load-generator.yaml
```

Once deployed, the Load Generator continuously sends HTTP requests to the ClusterIP Service associated with the NGINX Deployment. Initially, all requests are processed by a single Pod. As CPU utilization increases beyond the configured threshold, the Horizontal Pod Autoscaler automatically creates additional replicas, allowing the Service to progressively distribute the incoming workload across multiple Pods.

Monitoring Commands
Throughout the experiment, the following commands are executed simultaneously:

```
kubectl get hpa -w
```

and

```
kubectl top pods
```

The first command provides real-time visibility into the Horizontal Pod Autoscaler, reporting CPU utilization, replica count and scaling decisions as they evolve during the experiment. The second command displays the current CPU and memory consumption of each running Pod, allowing the observed resource utilization to be correlated with the scaling decisions performed by the HPA.

Observation on Resource Metrics
As discussed in the previous chapters, the resource metrics displayed by the `kubectl top pods` command originate from the Kubernetes Metrics Server. These measurements correspond to the same CPU utilization values consumed by the Horizontal Pod Autoscaler when computing the desired number of replicas. Consequently, the outputs of `kubectl top pods` and `kubectl get hpa -w` provide two complementary views of the same autoscaling process: the first reports the observed resource consumption, while the second shows how the HPA reacts to those measurements by adjusting the number of application replicas. The resulting autoscaling behavior is analyzed in the following section by observing the complete sequence of scale-out and scale-in operations performed after the workload is generated.

7.7 Scaling Behavior Analysis

The autoscaling behavior observed during this experiment closely matches the native Kubernetes implementation presented in the previous chapter. This result is expected because, once the ScaledObject has been translated into a Horizontal Pod Autoscaler, all scaling decisions continue to be performed by the native Kubernetes HPA using CPU metrics collected by the Metrics Server. The following paragraphs describe the complete sequence of events observed during the experiment.

Initial State

Before the workload is generated, the application operates under steady-state conditions. More specifically:

- A single Pod is running
- CPU utilization remains close to zero
- No scaling actions are performed

Consequently, the Horizontal Pod Autoscaler maintains the minimum replica count equal to one, as specified by the ScaledObject configuration.

Load Injection
Once the Load Generator starts producing HTTP requests, the CPU utilization of the application Pod progressively increases. As the generated workload continues to grow, the measured CPU utilization approaches and eventually exceeds the configured target utilization of 50%. At this point, the Horizontal Pod Autoscaler begins evaluating scale-out operations according to the standard Kubernetes autoscaling algorithm discussed in the previous chapters.

Scale-Out Phase
When the observed CPU utilization exceeds the configured target, the Horizontal Pod Autoscaler computes a larger desired replica count and progressively creates additional Pod replicas. As new replicas become available, the Kubernetes Service automatically distributes the incoming HTTP requests across multiple Pods instead of concentrating the workload on a single instance. Consequently, the average CPU utilization of each individual Pod gradually decreases while the overall application throughput increases. This behavior is identical to that observed in the native Kubernetes implementation because the same Horizontal Pod Autoscaler remains responsible for executing all scaling decisions.

Stabilization Phase
After the scale-out operation has completed, the CPU utilization gradually decreases as the workload becomes shared among multiple replicas. The average utilization progressively converges toward the configured target value of 50%, while the number of replicas stabilizes. This behavior confirms the primary objective of CPU-based autoscaling: maintaining resource utilization around a predefined target while continuously adapting the number of application instances to the current workload.

Load Removal
After the scale-out behavior has been successfully observed, the synthetic workload is removed by deleting the Load Generator Pod:

```
kubectl delete pod load-generator
```

Immediately after the workload is removed, the CPU utilization of the application Pods rapidly decreases. The Horizontal Pod Autoscaler detects the reduction in resource consumption through the Metrics Server and prepares to perform the corresponding scale-in operation.

Scale-In Phase
As CPU utilization remains below the configured target threshold, the Horizontal Pod Autoscaler progressively reduces the number of replicas. Eventually, the Deployment returns to its minimum replica count, restoring the same steady-state condition observed before the workload was generated. The observed scale-in process is identical to the native Kubernetes implementation because the same Horizontal Pod Autoscaler remains responsible for all scaling decisions. This experimental observation further confirms that KEDA does not replace the native Kubernetes autoscaling algorithm. Instead, it simplifies its configuration by automatically generating and managing the Horizontal Pod Autoscaler through the ScaledObject resource.

7.8 Critical Analysis

The experimental results confirm that integrating KEDA into a CPU-based autoscaling scenario does not modify the underlying Kubernetes autoscaling workflow.

What KEDA Adds
The experiment highlights several advantages introduced by KEDA:

- Automatic creation and management of the Horizontal Pod Autoscaler
- Unified autoscaling configuration through the ScaledObject resource
- A consistent configuration model that can be applied to both resource-based and event-driven autoscaling scenarios
- Simplified autoscaling management by eliminating the need to manually define and maintain HPA resources

What KEDA Does Not Change
Equally important is understanding which components remain unchanged. Throughout the experiment:

- The Metrics Server continues to collect CPU resource metrics
- CPU utilization is still exposed through the Kubernetes Metrics API
- The Horizontal Pod Autoscaler remains responsible for computing the desired number of replicas
- The autoscaling algorithm remains identical to the native Kubernetes implementation

Therefore, KEDA does not replace the native Kubernetes autoscaling mechanisms. Instead, it orchestrates and automates their configuration through the ScaledObject abstraction. The main differences between the native Kubernetes approach and the KEDA-based approach are summarized in Table 7.2:

Native Kubernetes	**KEDA**
Manual HPA creation	Automatic HPA generation through the ScaledObject
Resource-based autoscaling	Resource-based and event-driven autoscaling
Manual autoscaling configuration	Unified autoscaling configuration

Table 7.2: Comparison between the native Kubernetes approach and the KEDA-based approach.

Observation on Scale-to-Zero
Unlike the event-driven scenarios presented in the following chapters, CPU-based autoscaling managed through KEDA does not support scale-to-zero. CPU utilization can only be measured while at least one Pod is running. Consequently, the minimum replica count remains equal to one, exactly as in the native Kubernetes implementation. Therefore, introducing KEDA does not provide additional scale-to-zero capabilities when CPU or memory utilization is used as the scaling trigger.

7.9 Chapter Summary

This chapter experimentally validated the integration of KEDA with native Kubernetes CPU-based autoscaling. The results demonstrated that KEDA can successfully manage traditional resource-based autoscaling without modifying the underlying Kubernetes metrics pipeline. Rather than replacing the native autoscaling workflow, KEDA automatically generates and manages the corresponding Horizontal Pod Autoscaler while preserving the standard Metrics Server → HPA → Deployment architecture. Consequently, the observed autoscaling behavior remains effectively identical to the native Kubernetes implementation discussed in the previous chapters. The primary benefit introduced by KEDA lies in the unified ScaledObject abstraction, which simplifies autoscaling configuration and provides a common interface for both resource-based and event-driven scaling scenarios. The following chapters build upon this foundation by introducing event-driven autoscaling mechanisms, where KEDA plays a significantly more active role by collecting metrics directly from external systems and enabling capabilities that are not available through native Kubernetes autoscaling alone, such as scale-to-zero.

Chapter 8

Time-Based Autoscaling with KEDA: Cron Scaler

8.1 Introduction

The transition from resource-based autoscaling to event-driven autoscaling represents one of the most significant extensions introduced by KEDA. While the previous chapter demonstrated how KEDA can simplify the configuration of native CPU-based autoscaling, the following sections introduce the first autoscaling strategy that no longer relies on resource utilization metrics. A first step toward event-driven autoscaling is represented by the Cron scaler, which implements a time-based scaling mechanism. Although scaling decisions are not triggered by external event sources such as message brokers or monitoring systems, they are no longer based on CPU or memory utilization. Instead, the autoscaling process is driven exclusively by temporal conditions defined through standard cron expressions. For this reason, the Cron scaler can be considered an intermediate case between traditional metric-driven autoscaling and fully event-driven autoscaling. It preserves the same architectural workflow adopted by the other KEDA scalers while introducing several capabilities that are not available in native Kubernetes autoscaling, most notably scale-to-zero support. This chapter first presents the architecture and workflow of the Cron scaler before introducing the scaling mechanisms that will also be adopted by the more advanced event-driven scenarios discussed in the following chapters.

8.2 Cron Scaler Overview

The Cron scaler evaluates time-based conditions expressed through standard cron expressions in order to determine the desired scaling state of an application. Unlike CPU-based autoscaling, where scaling decisions depend on observed resource utilization, the Cron scaler relies exclusively on temporal conditions. The current time is periodically compared against one or more configured time windows, and scaling decisions are taken according to the result of this evaluation. The cron expressions are interpreted using the system time of the node where the KEDA Controller container is running. When required, the evaluation can be adjusted by specifying an appropriate timezone within the ScaledObject configuration. Consequently, the trigger does not depend on application workload, CPU utilization or memory consumption. Instead, scaling decisions are driven solely by time.

Scale-to-Zero Capability
One of the main advantages of the Cron scaler is that the triggering condition is always available, regardless of whether application Pods are currently running. Unlike CPU-based autoscaling, where metrics can only be collected while at least one Pod is active, temporal conditions exist independently of the workload state. As a consequence, the Cron scaler fully supports scale-to-zero operations. The system can automatically scale from zero to one replica when entering a configured time window and subsequently return to zero replicas when the window expires.

Architectural Workflow
From an architectural perspective, the Cron scaler follows the same workflow adopted by the other KEDA event-driven scalers. The trigger is periodically evaluated by the KEDA Controller. Based on the current time, the scaler generates an external metric representing the desired scaling state. This metric is then exposed to the Horizontal Pod Autoscaler through the KEDA Metrics Adapter and the Kubernetes API Server using the API Aggregation mechanism. Finally, the Horizontal Pod Autoscaler applies the standard Kubernetes scaling algorithm to determine the appropriate number of application replicas.

Desired State Representation
Unlike continuous metrics such as CPU utilization, the metric generated by the Cron scaler does not represent the current workload of the application. Instead, it represents a desired scaling state. When the current time falls

within the configured time window, the scaler enforces a predefined number of replicas. Outside the configured window, the desired replica count is typically reduced to zero, allowing the application to be completely stopped until the next activation period. The main differences between CPU-based autoscaling and Cron-based autoscaling are summarized in Table 8.1:

CPU-Based Autoscaling	**Cron-Based Autoscaling**
Scaling based on CPU or memory utilization.	Scaling based on predefined time windows.
Requires the Kubernetes Metrics Server.	Uses the KEDA Metrics Adapter and External Metrics API.
Metrics are available only while Pods are running.	Trigger conditions are always available.
Scale-to-zero is not supported.	Full scale-to-zero support.
Continuous utilization metrics.	Discrete desired-state metric.

Table 8.1: Comparison between CPU-based and Cron-based autoscaling.

As shown in Table 8.1, the Cron scaler introduces a fundamentally different autoscaling paradigm. Resource utilization is no longer the driving factor behind scaling decisions; instead, scaling actions are determined exclusively by temporal policies. Although the Cron scaler is not triggered by external event sources, it adopts the same architecture used by the remaining KEDA event-driven scalers. For this reason, it provides an ideal transition between native Kubernetes autoscaling and the fully event-driven scenarios presented in the following chapters.

8.3 Time-Based Autoscaling Workflow

The Cron scaler represents an intermediate step between native resource-based autoscaling and fully event-driven autoscaling. Unlike CPU-based autoscaling, which relies on internal resource metrics collected by the Kubernetes Metrics Server, the Cron scaler bases its scaling decisions exclusively on temporal conditions expressed through cron expressions. Consequently, native Kubernetes alone cannot provide this functionality, making KEDA an essential component

of the autoscaling architecture. The complete workflow can be divided into six consecutive phases, illustrated throughout the following sections.

ScaledObject Creation and Validation
The workflow begins with the creation of a `ScaledObject`, which defines the complete autoscaling configuration. For the Cron scaler, the trigger configuration includes:

- The trigger type (`cron`)
- The beginning of the active time window
- The end of the active time window
- The desired number of replicas during the active window

Additional parameters specify the minimum and maximum number of replicas allowed for the target Deployment.

Mapping to the Generated HPA
When the ScaledObject is accepted, KEDA automatically creates the corresponding Horizontal Pod Autoscaler. During this process:

- `maxReplicaCount` is mapped to the HPA `maxReplicas` field
- `minReplicaCount` is mapped to the HPA `minReplicas` field

Important Nuance
Even when the ScaledObject specifies:

```
1 minReplicaCount: 0
```

the generated Horizontal Pod Autoscaler still uses:

```
1 minReplicas: 1
```

This behavior is expected because the native Kubernetes HPA cannot manage workloads with zero running replicas. Consequently, scale-to-zero operations remain under the direct control of KEDA rather than the Horizontal Pod Autoscaler.

Polling Interval
Another important parameter is the `pollingInterval`, whose default value is 15 seconds. It defines how frequently KEDA evaluates the configured trigger.

For the Cron scaler, this means that the current time is periodically compared with the configured time window to determine whether the trigger is active or inactive.

8.3.1 Automatic HPA Creation and External Metrics Binding

Once the ScaledObject has been successfully validated, the KEDA Controller detects the new resource and automatically generates the corresponding Horizontal Pod Autoscaler. Unlike the CPU-based scenario discussed in the previous chapter, scaling is no longer driven by resource metrics collected by the Metrics Server. Instead, the generated HPA retrieves metrics through the Kubernetes External Metrics API exposed by the KEDA Metrics Adapter.

External Metrics Mechanism
In this scenario:

- Scaling is based on external metrics rather than resource metrics.
- Kubernetes accesses these metrics through the External Metrics API.
- KEDA exposes the generated metrics through its Metrics Adapter.
- The Horizontal Pod Autoscaler consumes these metrics exactly as it would consume native metrics.

Metric Type
For the Cron scaler, the generated Horizontal Pod Autoscaler uses the AverageValue metric type. Unlike CPU-based autoscaling, no utilization percentage exists because the Cron scaler produces a desired scaling state rather than a resource utilization metric. Consequently, the generated HPA uses an absolute metric value. In the default configuration, KEDA sets:

```
1 target.averageValue = 1
```

The Horizontal Pod Autoscaler therefore compares the external metric against this unitary target when applying the native Kubernetes scaling algorithm.

8.3.2 Cron Scaler Execution

Once the Horizontal Pod Autoscaler has been generated, the KEDA Controller initializes the Cron scaler associated with the ScaledObject. Unlike resource-based autoscaling, where scaling decisions depend on continuously measured resource utilization, the Cron scaler periodically evaluates the current time against the configured scheduling conditions.

Time Retrieval Mechanism
The Cron scaler executes within the KEDA Controller Pod and retrieves the current system time from the container environment. The evaluated time is typically derived from the clock of the Kubernetes node hosting the KEDA Controller. When required, the retrieved time can be adjusted through the `timezone` parameter defined in the ScaledObject manifest. This mechanism ensures that the configured cron expressions are interpreted consistently regardless of the geographical location of the Kubernetes cluster.

Core Cron Scaler Logic
At every polling interval, the Cron scaler evaluates whether the current time belongs to the configured active time window. Conceptually, its behavior can be summarized as follows:

$$metric_{value} = \begin{cases} desiredReplicas, & \text{if } current_time \in time_window, \\ minReplicaCount, & \text{otherwise.} \end{cases}$$

Unlike CPU utilization, the generated metric does not represent a continuously varying measurement. Instead, it directly encodes the desired scaling state according to the configured scheduling policy.

8.3.3 Metrics Exposure Through the KEDA Metrics Adapter

After computing the desired metric value, the Cron scaler forwards it to the KEDA Metrics Adapter. The primary responsibility of the Metrics Adapter is to expose this external metric to Kubernetes so that it can be consumed by the generated Horizontal Pod Autoscaler.

API Aggregation Mechanism
The metric is exposed through the Kubernetes API Server using the External Metrics API:

```
/apis/external.metrics.k8s.io
```

This functionality relies on the Kubernetes API Aggregation Layer, which allows the API Server to transparently extend its functionality by delegating metric requests to external providers such as the KEDA Metrics Adapter. Consequently, from the perspective of the Horizontal Pod Autoscaler, external metrics become available through the same Kubernetes API infrastructure used for native metrics.

8.3.4 HPA Query and Scaling Decision

Once the external metric has been published, the Horizontal Pod Autoscaler periodically retrieves it through the Kubernetes API Server. By default, the HPA evaluates metrics every 15 seconds, according to the –horizontal-pod-autoscaler-sync-period parameter configured in the Kubernetes Controller Manager.

Data Flow
The complete metric retrieval process follows the sequence:

1. The HPA queries the Kubernetes API Server
2. The API Server forwards the request through the API Aggregation Layer
3. The KEDA Metrics Adapter returns the latest external metric
4. The HPA evaluates the received value

Scaling Formula
The Horizontal Pod Autoscaler computes the desired number of replicas using the standard Kubernetes scaling formula:

$$desiredReplicas = \left\lceil currentReplicas \times \frac{currentMetric}{targetMetric} \right\rceil$$

where:

- *currentReplicas* is the current number of running Pods
- *currentMetric* is the metric provided by the KEDA Metrics Adapter

- *targetMetric* is the configured target value (typically equal to 1).

The resulting value determines the desired number of replicas to be enforced by the Horizontal Pod Autoscaler.

8.3.5 Scaling Execution

Once the desired replica count has been computed, the Horizontal Pod Autoscaler updates the `spec.replicas` field of the target Deployment through the Kubernetes API Server. Kubernetes then performs the remaining operations required to reach the desired state:

- Scheduling newly created Pods onto available cluster nodes
- Starting or terminating Pods as required
- Continuously reconciling the actual cluster state with the desired Deployment specification

The complete workflow is summarized in Figure 8.1:

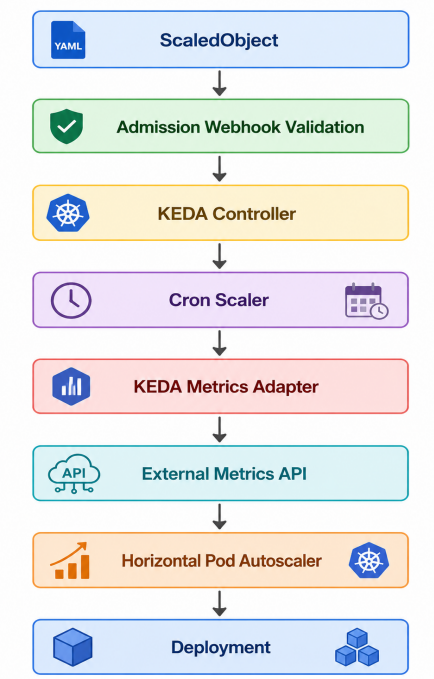


Figure 8.1: Workflow of the KEDA Cron scaler.

As illustrated in Figure 8.1, the Cron scaler introduces a metrics pipeline fundamentally different from CPU-based autoscaling. Instead of relying on resource metrics collected from the Kubernetes nodes, KEDA generates an external metric based exclusively on temporal conditions and exposes it to the Horizontal Pod Autoscaler through the External Metrics API.

8.4 Runtime Interaction Scenarios

To fully understand the behavior of KEDA in time-based autoscaling, it is useful to analyze specific runtime scenarios that highlight the interaction between KEDA components and the native Kubernetes Horizontal Pod Autoscaler. These scenarios show how KEDA extends the native autoscaling capabilities of Kubernetes, especially when the desired scaling behavior involves zero replicas, a condition that cannot be directly handled by the HPA alone.

8.4.1 Scaling Down to Zero

Consider a system initially operating inside a defined time window, where a certain number of replicas, denoted by N_1, is active. When the current time falls outside the configured time window, the Cron scaler detects that the trigger is no longer active. At this point, the scaler determines that the desired number of replicas is zero. This condition represents a fundamental limitation of the native Horizontal Pod Autoscaler. Even if the external metric generated by KEDA indicates that the workload should become inactive, the HPA is constrained by its minimum replica value, which cannot be lower than one. As a consequence, the HPA alone cannot reduce the number of replicas to zero. To overcome this limitation, KEDA directly intervenes by bypassing the HPA and explicitly updating the `replicas` field of the target Deployment through the Kubernetes API Server, setting it to zero. This behavior highlights one of the key advantages of KEDA: the ability to scale workloads down to zero replicas, which is not supported by native Kubernetes autoscaling mechanisms. It is important to note that this transition is not necessarily instantaneous. It can be influenced by the `cooldownPeriod` configured in KEDA and by the HPA scale-down stabilization window, which may introduce a delay before the final scale-to-zero action is completed. In particular, when properly configured, the stabilization window can affect the HPA decision process and lead to a gradual scale-down phase before KEDA performs the final transition to zero replicas.

8.4.2 Scale from Zero to an Active Window

Let us now consider the opposite transition. The system is initially idle, with zero replicas running, because the Deployment has previously been scaled down to zero by KEDA. When the current time enters a new active time window, the Cron scaler detects that the trigger becomes active again. Suppose that the desired number of replicas in this new window is N_2, which may differ from the previous value N_1. This distinction is intentional, since different time windows may be associated with different desired replica configurations. When the system is scaled to zero, the HPA cannot directly operate because no running replica exists. In this phase, KEDA is responsible for reactivating the workload and restoring the autoscaling loop. However, KEDA does not directly enforce N_2 replicas on the Deployment. Instead, it reactivates the scaling process and exposes the corresponding external metric to the HPA, which regains control of the scaling decision process. The HPA then computes the desired number of replicas starting from a non-zero state and progressively applies the configured scale-up policies until the target value N_2 is reached. This results in a scale-out sequence whose intermediate steps depend on the metric values and on the configured HPA scaling policies. For example, Kubernetes default scale-up policies are usually designed to allow rapid growth by selecting the most permissive scaling action between percentage-based and pod-based policies. This behavior highlights the hybrid nature of KEDA-based autoscaling. KEDA is responsible for activating or deactivating the scaling signal, while the HPA remains responsible for computing and applying replica adjustments according to its scaling policies whenever the workload is above zero replicas.

8.4.3 Scaling Between Non-Zero Replica Counts

Finally, consider a scenario where the system transitions from an active window with N_2 replicas to another state where a lower, but still non-zero, number of replicas k is desired, with:

$$0 < k < N_2$$

In this case, the Cron scaler detects the new condition and computes the desired scaling state. Unlike the previous scenarios, the system is not transitioning to or from zero replicas. Therefore, the Horizontal Pod Autoscaler can fully manage the scaling process. A crucial aspect concerns how KEDA communicates the desired scaling behavior to the HPA. The HPA computes

the desired number of replicas using the standard formula:

$$desiredReplicas = \left\lceil currentReplicas \times \frac{currentMetric}{targetMetric} \right\rceil$$

In KEDA-based Cron scaling, the *targetMetric* is typically set to one. Therefore, in order to obtain:

$$desiredReplicas = k$$

the metric consumed by the HPA must encode the ratio between the desired number of replicas and the current number of replicas. More precisely, because the generated HPA uses the `AverageValue` metric type, the HPA internally normalizes the metric value with respect to the current number of replicas:

$$currentMetric = \frac{k}{N_2}$$

This allows the native HPA formula to produce the intended replica count while preserving the standard Kubernetes autoscaling logic.

Numerical Example
Assume the following values:

- $currentReplicas = N_2 = 10$;
- $desiredReplicas = k = 5$;
- $targetMetric = 1$.

The Cron scaler expresses the desired state as:

$$scalerOutput = k = 5$$

Since the HPA uses the `AverageValue` metric type, the effective current metric used in the final scaling computation becomes:

$$currentMetric = \frac{k}{currentReplicas} = \frac{5}{10} = 0.5$$

The HPA therefore computes:

$$desiredReplicas = \left\lceil 10 \times \frac{0.5}{1} \right\rceil = 5$$

which correctly matches the desired scaling logic of the Cron scaler.

Important Architectural Clarification
It is important to emphasize that this normalization is performed by the Horizontal Pod Autoscaler because of the `AverageValue` metric type. The KEDA Metrics Adapter does not modify the metric value produced by the scaler. Its role is limited to exposing the metric through the Kubernetes API Server using the API Aggregation Layer. Once the metric is exposed, the HPA periodically retrieves it, applies its native averaging logic and computes the desired number of replicas using the standard autoscaling formula. In this scenario, scaling is entirely managed by the HPA. The scaling operation is therefore gradual rather than instantaneous, and its rate depends on HPA configuration parameters such as stabilization windows and scaling policies. These parameters can be customized in KEDA through the `advanced.horizontalPodAutoscalerConfig.behavior` section of the ScaledObject manifest.

Final Considerations
The analysis of these three scenarios clearly demonstrates the complementary roles of KEDA and HPA. KEDA directly controls scaling when the system reaches or starts from zero replicas, because this condition cannot be handled by the native HPA alone. Conversely, scaling between non-zero replica counts is managed by the HPA, which uses the external metrics exposed by KEDA and applies its standard scaling algorithm. This interaction enables KEDA to extend Kubernetes autoscaling with time-based and event-driven capabilities while preserving the native HPA control loop whenever possible.

8.5 Gradual Scaling and Stabilization Mechanisms

The previous sections illustrated how KEDA generates and exposes external metrics to the Horizontal Pod Autoscaler. However, the actual scaling operations are not performed instantaneously. Instead, Kubernetes applies a set of mechanisms designed to ensure that replica adjustments occur gradually and remain stable over time. These mechanisms prevent excessive oscillations, avoid abrupt resource allocation changes and improve the overall stability of the autoscaling process. Although these behaviors are implemented by the na-

tive Horizontal Pod Autoscaler, they become particularly relevant in KEDA-based autoscaling because all event-driven scalers ultimately rely on the HPA to execute replica adjustments.

8.5.1 Gradual Scaling in Native Kubernetes

The Horizontal Pod Autoscaler does not necessarily apply the replica count computed by the scaling algorithm immediately. Instead, Kubernetes implements several mechanisms that ensure replica adjustments occur progressively and in a controlled manner. This design choice aims to improve the stability of the autoscaling process by preventing abrupt fluctuations in the number of running Pods caused by temporary metric variations. Without these mechanisms, even small and short-lived changes in the observed metrics could continuously trigger scale-up and scale-down operations, resulting in unnecessary resource consumption and application instability. For this reason, Kubernetes introduces the concept of *gradual scaling*, where scaling decisions are regulated through configurable scaling policies and stabilization mechanisms. These mechanisms determine both how rapidly the number of replicas may change and when a scaling recommendation should actually be applied. Although KEDA introduces new event-driven scaling capabilities, it does not replace these native Kubernetes mechanisms. Once the Horizontal Pod Autoscaler has been created, every replica adjustment continues to be performed according to the standard HPA behavior. Consequently, understanding gradual scaling in native Kubernetes is essential before analyzing how KEDA interacts with these mechanisms in the following sections.

8.5.2 Configuring HPA Scaling Behavior

Although the Horizontal Pod Autoscaler applies built-in mechanisms to avoid abrupt scaling actions, Kubernetes also allows explicit customization of the scaling process through the `behavior` field of the HPA manifest. This field provides fine-grained control over both scale-up and scale-down operations by allowing users to define scaling policies and stabilization windows independently. More specifically, the `behavior` section determines how quickly the HPA is allowed to move from the current number of replicas to the desired number computed by the autoscaling algorithm. Consequently, it does not modify the scaling decision itself, but rather regulates the way in which that

decision is executed over time.

File: hpa.yaml

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: cpu-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: my-app

  minReplicas: 1
  maxReplicas: 10

  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 50

  behavior:
    scaleUp:
      stabilizationWindowSeconds: 0
      policies:
      - type: Percent
        value: 100
        periodSeconds: 15

    scaleDown:
      stabilizationWindowSeconds: 300
      policies:
      - type: Percent
        value: 50
        periodSeconds: 15
```

Listing 8.1: Example of a Horizontal Pod Autoscaler manifest illustrating the configuration of scale-up and scale-down policies.

Most Important Fields Explanation:

- **behavior** Defines the overall scaling behavior of the Horizontal Pod Autoscaler. It contains independent configurations for scale-up and scale-down operations
- **scaleUp** Specifies the rules applied when the HPA needs to increase the number of replicas.
- **scaleDown** Specifies the rules applied when the HPA needs to decrease the number of replicas.
- **stabilizationWindowSeconds** Defines the time interval during which previous scaling recommendations are considered before applying a new scaling decision. Separate stabilization windows can be configured for scale-up and scale-down operations
- **policies** Defines one or more policies limiting how quickly the number of replicas may change during each scaling cycle
- **type** Specifies how the scaling limit is expressed. The available policy types are:
  - `Percent`: Scaling is expressed as a percentage of the current number of replicas
  - `Pods`: Scaling is expressed as an absolute number of Pods
- **value** Defines the maximum variation allowed according to the selected policy type For example:
  - `value:50` with `type:Percent` allows changing at most 50% of the current replicas
  - `value:4` with `type:Pods` allows adding or removing at most four Pods

- **periodSeconds** Defines the time window over which the selected policy is enforced.

Clarification on Scaling Policies Parameters
Within the scaling policies, the `periodSeconds` parameter defines the time interval over which a specific scaling policy is applied. For example, a policy configured as:

```
50% every 15 seconds
```

means that the number of replicas can change by at most 50% of the current replica count during each 15-second interval. If multiple policies are simultaneously defined, the Horizontal Pod Autoscaler evaluates all of them and selects the effective scaling constraint according to the configured `selectPolicy` strategy (e.g., `Max` or `Min`).

Example: Scale-Down Policy
Consider a scale-down policy configured as:

```
50% every 15 seconds
```

Suppose that the current Deployment consists of eight replicas and the HPA determines that the desired state is one replica. The configured policy does not allow the HPA to immediately transition from eight replicas to one. Instead, the replica count is reduced progressively according to the configured policy. Consequently, the scaling process proceeds through intermediate states such as:

$$8 \rightarrow 4 \rightarrow 2 \rightarrow 1$$

depending on the HPA evaluation intervals. This gradual reduction prevents abrupt resource deallocation and contributes to maintaining application stability during scale-in operations.

Example: Scale-Up Policy
A similar reasoning applies to scale-up operations. Consider the following policy:

```
100% every 15 seconds
```

At first glance, a 100% increase might suggest that scaling occurs immediately. However, this interpretation is incorrect. Instead, the policy specifies that the number of replicas may at most double during each 15-second interval. Assume that the Deployment currently runs two replicas and the Horizontal Pod Autoscaler computes a desired state of ten replicas. The scaling process therefore evolves as:

$$2 \rightarrow 4 \rightarrow 8 \rightarrow 10$$

rather than directly jumping from two to ten replicas. The sequence terminates at ten replicas because the HPA manifest specifies:

```
maxReplicas: 10
```

which represents the upper bound for scaling operations. This gradual approach provides a good compromise between responsiveness and stability, allowing Kubernetes to react rapidly to workload increases while avoiding excessive resource allocation during a single reconciliation cycle.

8.5.3 Interpretation of Scaling Policies

The scaling policies defined within the `behavior` section do not specify the desired number of replicas. Instead, they define the maximum amount by which the current replica count is allowed to change during each scaling interval. After the Horizontal Pod Autoscaler computes the desired number of replicas using the standard autoscaling formula, the configured scaling policies determine how quickly that desired state can be reached. Consequently, scaling policies regulate the execution of the scaling decision rather than the scaling decision itself.

Pods Policy

When the policy type is set to `Pods`, the `value` field specifies the maximum absolute number of replicas that may be added or removed during a single scaling interval. For example:

```
type: Pods
 value: 4
```

means that at most four Pods can be created or removed every `periodSeconds` seconds.

Percent Policy

When the policy type is set to `Percent`, the `value` field specifies the maximum percentage variation with respect to the current number of replicas. For example:

```
type: Percent
 value: 100
```

allows the number of replicas to increase or decrease by at most 100% during each scaling interval. Unlike the `Pods` policy, the allowed variation automatically adapts to the current size of the Deployment, making percentage-based policies particularly suitable for large-scale workloads.

The `selectPolicy` Parameter
When multiple scaling policies are simultaneously configured, the Horizontal Pod Autoscaler evaluates each policy independently. The `selectPolicy` parameter determines which policy is ultimately applied. The available options are:

- **Max** The most permissive policy is selected, allowing the largest replica variation during the current scaling interval
- **Min** The most conservative policy is selected, limiting the replica variation according to the smallest allowed change
- **Disabled** Scaling in the corresponding direction is completely disabled

If the parameter is omitted, Kubernetes uses `Max` as the default strategy.

Numerical Example
Assume that the following scale-up policies are simultaneously configured:

- Pods: +4 replicas;
- Percent: +100%.

Suppose that the Deployment currently consists of two replicas. The Pods policy allows increasing the replica count to:

$$2 + 4 = 6$$

whereas the Percent policy allows increasing the replica count to:

$$2 + 100\% = 4$$

If the default `Max` strategy is used, Kubernetes selects the Pods policy because it allows the largest replica increase. Consequently, the Deployment scales directly from:

$$2 \rightarrow 6$$

during the current reconciliation cycle.

8.5.4 Default Kubernetes Scaling Behavior

If no custom scaling behavior is explicitly configured, Kubernetes automatically applies a set of default scaling policies. These defaults are designed to provide a good compromise between responsiveness during workload increases and stability during workload decreases.

Default Scale-Up Behavior
For scale-up operations, Kubernetes combines two policies:

- A `Pods` policy allowing up to four additional replicas every 15 seconds
- A `Percent` policy allowing a 100% increase every 15 seconds

Since the default selection strategy is `Max`, the Horizontal Pod Autoscaler always applies the most aggressive scaling action permitted by the configured policies. This allows applications to rapidly react to sudden workload increases.

Default Scale-Down Behavior
By default, Kubernetes also configures a `Percent` policy equal to 100% for scale-down operations. At first glance, this configuration may suggest that all replicas could be removed immediately during a single reconciliation cycle. However, this is not what normally happens. Unlike scale-up operations, the default scale-down behavior is strongly influenced by the stabilization window, whose default value is 300 seconds. Instead of immediately applying the most recent recommendation, the Horizontal Pod Autoscaler temporarily stores previous recommendations and selects the highest replica count observed within the stabilization window. Consequently, although the default scaling policy itself allows a 100% reduction, the stabilization mechanism makes scale-down operations significantly more conservative and prevents unnecessary oscillations caused by temporary workload fluctuations. The stabilization window is therefore the primary mechanism responsible for the gradual scale-down behavior observed in native Kubernetes autoscaling and is discussed in detail in the following section.

8.5.5 Stabilization Window

While scaling policies regulate the maximum rate at which the number of replicas may change during each reconciliation cycle, they do not determine when a scaling recommendation should actually be applied. To further improve the stability of the autoscaling process, the Horizontal Pod Autoscaler introduces the concept of a *stabilization window*. Rather than immediately applying every newly computed scaling recommendation, the HPA can temporarily retain previous recommendations and evaluate them over a configurable time interval before performing the scaling operation. The primary objective of the stabilization window is to prevent oscillatory behavior caused by temporary workload fluctuations. Short-lived metric variations therefore do not immediately trigger scale-up or scale-down operations, resulting in a more stable autoscaling process.

Default Stabilization Window Values
Kubernetes defines different default stabilization windows for scale-up and scale-down operations.

Scaling Direction	Default Value	Purpose
Scale-Up	0 seconds	Allows immediate reaction to workload increases.
Scale-Down	300 seconds	Prevents unnecessary replica removals caused by temporary workload decreases.

Table 8.2: Default stabilization windows used by the Horizontal Pod Autoscaler.

File: hpa.yaml

```
behavior:
  scaleUp:
    stabilizationWindowSeconds: 0

  scaleDown:
    stabilizationWindowSeconds: 300
```

Listing 8.2: Example of a Horizontal Pod Autoscaler manifest configuring stabilization windows.

Most Important Field:

stabilizationWindowSeconds
Defines the time interval during which previous scaling recommendations are retained before a new scaling decision is applied. Separate stabilization windows can be configured for scale-up and scale-down operations, allowing Kubernetes to react differently to increasing and decreasing workloads.

Scale-Down Stabilization
Scale-down operations adopt the opposite strategy. By default, Kubernetes configures a stabilization window equal to 300 seconds. During this interval, every scale-down recommendation generated by the Horizontal Pod Autoscaler is retained. Rather than selecting the most recent recommendation, Kubernetes preserves the **highest** replica recommendation generated within the stabilization window. This behavior intentionally delays Pod removal and prevents temporary workload reductions from producing unnecessary scale-in operations.

Example: Scale-Down Stabilization Window
Consider a Deployment currently running ten replicas and assume that a scale-down stabilization window of 300 seconds is configured. During this observation interval, the Horizontal Pod Autoscaler continuously evaluates the monitored metrics and computes several scaling recommendations. For simplicity, let us consider only the subset of recommendations reported in Table 8.3:

Time	**Metrics → HPA Decision**
$t_0 - 300\,s$	Suggests 3 replicas
$t_0 - 15\,s$	Suggests 6 replicas
t_0	Suggests 4 replicas

Table 8.3: Example of scale-down recommendations generated during the stabilization window.

The values reported in Table 8.3 represent only a subset of the recommendations generated during the stabilization window. In practice, the HPA continuously evaluates the monitored metrics, typically every 15 seconds, producing multiple recommendations throughout the entire 300-second interval. For the sake of simplicity, this example assumes that all recommendations not explicitly shown correspond to three desired replicas. Without a stabilization

window, the Horizontal Pod Autoscaler would immediately apply every recommendation as it is generated, producing the following sequence:

$$10 \rightarrow 3 \rightarrow 6 \rightarrow 4$$

This behavior may introduce continuous oscillations in the number of running replicas. With a stabilization window of 300 seconds enabled, the HPA instead considers all recommendations generated during the observation interval and selects the highest recommendation:

$$\max(3, 6, 4) = 6$$

Consequently, the Deployment is initially scaled from

$$10 \rightarrow 6$$

rather than directly to three replicas, resulting in a significantly more stable scale-down behavior.

Scale-Up Stabilization
By default, the stabilization window for scale-up operations is equal to zero seconds. Consequently, the Horizontal Pod Autoscaler immediately applies the most recent scaling recommendation whenever the observed metric exceeds the configured threshold. Although the default value is zero, Kubernetes also allows users to configure a non-zero stabilization window for scale-up operations. In this case, the HPA temporarily stores all scale-up recommendations generated during the configured time interval. Unlike scale-down operations, where the highest recommendation is preserved, the HPA selects the **lowest** recommendation observed within the stabilization window. This conservative behavior prevents temporary metric spikes from triggering unnecessarily aggressive scale-up operations.

Example: Scale-Up Stabilization Window
Unlike scale-down operations, Kubernetes does not enable a stabilization window for scale-up by default. For explanatory purposes, let us assume that a stabilization window of 300 seconds has also been configured for scale-up. Suppose that the Deployment currently consists of two running replicas and that the Horizontal Pod Autoscaler generates the recommendations reported in Table 8.4.

Time	**Metrics → HPA Decision**
$t_0 - 300\,s$	Suggests 10 replicas
$t_0 - 15\,s$	Suggests 8 replicas
t_0	Suggests 6 replicas

Table 8.4: Example of scale-up recommendations generated during the stabilization window.

As in the previous example, the values reported in Table 8.4 represent only a subset of the recommendations computed during the stabilization window. In practice, the HPA continuously evaluates metrics throughout the entire observation interval. For simplicity, the omitted recommendations are assumed to correspond to ten desired replicas. In this case, the stabilization mechanism applies the opposite criterion with respect to scale-down operations by selecting the minimum recommendation:

$$\min(10, 8, 6) = 6$$

Consequently, the Horizontal Pod Autoscaler initially scales the Deployment according to the lowest recommendation generated within the stabilization window, avoiding unnecessary reactions to temporary metric spikes.

Important Rule
The stabilization window applies different selection criteria depending on the scaling direction.

- During **scale-up**, if a stabilization window greater than zero is configured, the Horizontal Pod Autoscaler selects the **minimum** recommendation generated within the observation interval.
- During **scale-down**, the Horizontal Pod Autoscaler selects the **maximum** recommendation generated within the stabilization window.

These complementary strategies reduce unnecessary oscillations while ensuring that applications remain responsive to sustained workload variations.

Interaction Between Policies and Stabilization Window
The stabilization window does not modify the replica count computed by the Horizontal Pod Autoscaler. Instead, it determines *when* a scaling recommendation may be applied. Consequently, the stabilization window and the scaling

policies regulate two different aspects of the autoscaling process. For instance, suppose that the stabilization mechanism determines a desired replica count of four, while the Deployment is currently running ten replicas. Further assume that the scale-down policy allows reducing the number of replicas by at most 50% every 15 seconds. In this case, the stabilization window determines *what* scaling decision should be applied (four replicas), whereas the scaling policy determines *how quickly* that decision can be reached. Consequently, the actual scaling process proceeds as follows:

$$10 \rightarrow 5 \rightarrow 4$$

rather than directly scaling from ten to four replicas. This example clearly illustrates that scaling policies limit the maximum replica variation during each reconciliation cycle, whereas the stabilization window determines whether a newly computed recommendation should be immediately accepted or temporarily delayed.

Mechanism	**Role**
Stabilization Window	Determines which scaling recommendation should be applied.
Scaling Policies	Limits the maximum replica variation during each reconciliation cycle.

Table 8.5: Complementary roles of the stabilization window and scaling policies.

8.5.6 Gradual Scaling with KEDA

When KEDA is introduced, the scaling logic is extended but not replaced. As discussed in the previous sections, KEDA still relies on the native Horizontal Pod Autoscaler to perform scaling decisions whenever the number of running replicas is greater than zero. The main difference compared to the native Kubernetes approach lies in the configuration model. Instead of manually creating and managing an HPA resource, the user defines a `ScaledObject`. Based on this resource, KEDA automatically generates and continuously manages the corresponding Horizontal Pod Autoscaler. Consequently, all the scaling mechanisms previously discussed, including scaling policies and stabilization

windows, remain fully applicable. The only difference is that these parameters are now specified inside the `ScaledObject` resource rather than directly in the HPA manifest.

File: scaledobject.yaml

```yaml
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: my-scaledobject
spec:
  scaleTargetRef:
    name: my-app

  minReplicaCount: 0
  maxReplicaCount: 10

  triggers:
  - type: cron
    metadata:
      timezone: UTC
      start: "0 8 * * *"
      end: "0 18 * * *"
      desiredReplicas: "5"

  advanced:
    horizontalPodAutoscalerConfig:
      behavior:
        scaleUp:
          stabilizationWindowSeconds: 0
          policies:
          - type: Percent
            value: 100
            periodSeconds: 15

        scaleDown:
          stabilizationWindowSeconds: 300
          policies:
          - type: Percent
            value: 50
            periodSeconds: 15
```

Listing 8.3: Example of a ScaledObject manifest configuring HPA scaling behavior through KEDA.

Manifest Description
The manifest shown above illustrates how the same scaling behavior previously configured directly within a native Horizontal Pod Autoscaler can be specified inside a KEDA `ScaledObject`. Rather than defining an HPA resource manually, the user configures the desired scaling behavior through the `advanced.horizontalPodAutoscalerConfig` section. KEDA automatically propagates these parameters into the generated Horizontal Pod Autoscaler, preserving the native Kubernetes scaling mechanisms while simplifying autoscaling configuration through a unified abstraction.

Most Important Fields Explanation:

- **advanced** Enables advanced KEDA configuration options that extend the basic ScaledObject specification
- **horizontalPodAutoscalerConfig** Allows the user to customize the configuration of the Horizontal Pod Autoscaler automatically generated by KEDA
- **behavior** Defines the scaling behavior of the generated HPA, including scaling policies and stabilization windows
- **scaleUp** Specifies the rules governing scale-up operations performed by the generated HPA
- **scaleDown** Specifies the rules governing scale-down operations performed by the generated HPA
- `stabilizationWindowSeconds` Defines the stabilization window applied by the generated Horizontal Pod Autoscaler before accepting new scaling recommendations
- **policies** Defines the scaling policies used by the generated HPA to regulate the maximum replica variation during each reconciliation cycle

Mapping Between Native HPA and KEDA

Concept	Native HPA	KEDA
Scaling configuration	`behavior`	`advanced.horizontalPodAutoscalerConfig.behavior`
Scale-up configuration	`scaleUp`	`scaleUp`
Scale-down configuration	`scaleDown`	`scaleDown`
Stabilization window	`stabilizationWindowSeconds`	Same field
Scaling policies	`policies`	Same field

Table 8.6: Correspondence between native HPA behavior configuration and the equivalent KEDA ScaledObject configuration.

As summarized in Table 8.6, KEDA does not redefine the scaling behavior implemented by the Horizontal Pod Autoscaler. Instead, it simply propagates the configuration specified in the `ScaledObject` into the automatically generated HPA. Consequently, all considerations discussed in the previous sections regarding scaling policies, stabilization windows and gradual scaling remain fully applicable when autoscaling is managed through KEDA.

8.5.7 Important Limitation: When Gradual Scaling Does Not Apply

It is important to emphasize that the gradual scaling mechanisms discussed throughout this section only apply while the Horizontal Pod Autoscaler is actively responsible for managing replica adjustments. More specifically, scaling policies, stabilization windows and all native HPA behaviors are effective only when the number of running replicas is greater than zero and the HPA is controlling the scaling process. There are, however, two important scenarios in which KEDA bypasses the Horizontal Pod Autoscaler entirely:

- Scale from **0** to **1** replica
- Scale from **N** to **0** replicas

In these situations, the HPA cannot perform the scaling operation because it is intrinsically unable to manage workloads with zero running replicas. Consequently, KEDA directly updates the target Deployment by modifying the `spec.replicas` field through the Kubernetes API Server. The resulting behavior can be summarized as follows:

- The HPA is not involved in the scaling operation
- Scaling policies are not applied
- Stabilization windows are not evaluated
- The replica count is updated directly by KEDA

As a result, transitions from zero to one replica and from one (or more) replicas to zero occur independently of the gradual scaling mechanisms implemented by the Horizontal Pod Autoscaler.

Default Behavior
If no advanced configuration is explicitly provided inside the `ScaledObject`, KEDA automatically generates a Horizontal Pod Autoscaler using the default Kubernetes behavior. Consequently, all default HPA mechanisms previously discussed—including the default scaling policies and stabilization windows—remain fully applicable whenever the number of running replicas is greater than zero and the HPA is responsible for the scaling decision.

8.5.8 Difference Between Stabilization Window and Cooldown Period

Although both the stabilization window and the cooldown period contribute to improving autoscaling stability, they operate at different levels of the scaling process and serve distinct purposes. The stabilization window is a native Kubernetes mechanism implemented by the Horizontal Pod Autoscaler. It affects how scaling recommendations are filtered and applied over time, both for scale-up and scale-down operations. Rather than immediately applying every newly computed recommendation, the HPA analyzes the recommendations generated during the stabilization window and selects a stable value according to the scaling direction, as discussed in the previous sections. Conversely, the cooldown period is a KEDA-specific parameter that determines how long

the system should wait before attempting a zero replicas scale-down operation after the last scaling activity. Unlike the stabilization window, the cooldown period does not modify the scaling decision itself. Instead, it delays the execution of the zero replicas scale-down operation. This mechanism is especially relevant in event-driven scenarios, where periods of inactivity may only be temporary. By postponing scale-down operations, the cooldown period prevents unnecessary transitions to zero replicas when new events arrive shortly after the workload becomes idle.

Aspect	**Stabilization Window**	**Cooldown Period**
Implemented by	Horizontal Pod Autoscaler	KEDA
Purpose	Filters scaling recommendations before they are applied	Delays scale-down operations after the last scaling activity
Acts on	Scale-up and scale-down recommendations	Scale-down actions, especially towards zero replicas
Main objective	Improve decision stability	Prevent premature scale-to-zero operations

Table 8.7: Comparison between the HPA stabilization window and the KEDA cooldown period.

In summary, the stabilization window controls decision stability within the Horizontal Pod Autoscaler, whereas the cooldown period controls when KEDA allows scale-down operations to occur. Although both mechanisms improve autoscaling stability, they address different aspects of the scaling process and therefore complement each other rather than overlapping.

8.5.9 Cooldown Period in Event-Driven Scaling: Queue-Based Example

The concept of the cooldown period becomes particularly relevant in event-driven scaling scenarios, such as those in which KEDA scales applications according to the length of a message queue (e.g., RabbitMQ, Kafka, or cloud-based queue services). Consider an application that processes messages from a queue, where each Pod is capable of processing a fixed number of messages.

Suppose that, at a given time, the queue contains 100 messages and that each Pod is configured to process up to 10 messages. Based on these values, KEDA determines that ten replicas are required and scales the Deployment accordingly.

$$\frac{100 \text{ messages}}{10 \text{ messages/Pod}} = 10 \text{ Pods}$$

As the application progressively processes the queued messages, the queue length gradually decreases until it eventually becomes empty. At this point, no further events are available to drive the autoscaling process. Without any additional stabilization mechanism, KEDA could immediately trigger a scale-down operation, eventually reducing the Deployment to zero replicas as soon as the queue becomes empty.

Why Immediate Scale-to-Zero May Be Undesirable
Although immediately scaling down to zero replicas maximizes resource savings, this behavior may not always be desirable in real-world applications. Message arrivals are often bursty or intermittent. Consequently, a new batch of messages may arrive shortly after the queue becomes empty. If the application had already been scaled down to zero replicas, KEDA would first need to recreate one or more Pods before message processing could resume. This introduces additional latency due to Pod scheduling, image startup, and application initialization. Therefore, immediately scaling down to zero replicas may unnecessarily increase response times for workloads characterized by short inactivity periods.

Role of the Cooldown Period
The cooldown period addresses this problem by delaying scale-down operations after the workload becomes inactive. When a cooldown period is configured (300 seconds by default), KEDA does not immediately scale the Deployment down after the queue becomes empty. Instead, it waits for the configured interval while continuously monitoring the event source. If new messages arrive before the cooldown period expires, the Deployment remains active and continues processing the incoming workload without ever reaching zero replicas.

Example:
Assume that a cooldown period of 300 seconds is configured and consider the following sequence of events:

- At time t_0, the queue length becomes zero

- Between t_0 and $t_0 + 300$ seconds, a new batch of messages arrives

Since the cooldown period has not yet elapsed, KEDA does not scale the Deployment down to zero replicas. Instead, the application immediately resumes processing the new workload using the already-running Pods, while the Horizontal Pod Autoscaler continues to manage replica adjustments based on the observed workload. Conversely, suppose that no new messages arrive during the entire cooldown interval. Once the configured cooldown period has expired, KEDA determines that the workload has remained inactive long enough and proceeds with the scale-down operation. Consequently, the Deployment is eventually reduced to zero replicas.

Important Observation
The cooldown period does not influence how scaling decisions are computed. Instead, it determines *when* KEDA is allowed to perform scale-down operations after the workload becomes inactive. Consequently, the cooldown period is particularly valuable for event-driven applications characterized by intermittent workloads, where temporary inactivity should not immediately trigger scale-to-zero operations. Unlike the stabilization window implemented by the Horizontal Pod Autoscaler, which filters scaling recommendations before they are applied, the cooldown period introduces a form of temporal hysteresis within the event-driven autoscaling process. By delaying scale-down operations after the last observed activity, KEDA prevents premature transitions to zero replicas and significantly reduces unnecessary scale-in/scale-out oscillations caused by short-lived inactivity periods.

8.5.10 Cooldown Period in Time-Based (Cron) Scaling

The cooldown period can also be applied in time-based autoscaling scenarios, such as those implemented through the KEDA Cron scaler. Unlike queue-based autoscaling, where scaling decisions are driven by external events, the Cron scaler bases its decisions on predefined time windows. Consider a configuration in which a Deployment is scheduled to scale up to five replicas during working hours, for example from 9:00 to 18:00, and to scale down to zero replicas outside this interval. When the system exits the active time window at 18:00, the Cron scaler updates the desired state, indicating that the Deployment should eventually be reduced to zero replicas.

Without a Cooldown Period

If no cooldown period is configured, KEDA immediately starts the scale-down procedure as soon as the active time window expires. Consequently, the Deployment may reach zero replicas shortly after 18:00, depending on the interaction with the Horizontal Pod Autoscaler and the configured stabilization mechanisms.

Why a Cooldown Period May Be Beneficial
Although the active time window has ended, it may still be desirable to delay the transition to zero replicas. For example, some client requests or background tasks may still be in progress shortly after the scheduled end of the working period. Moreover, as will be illustrated in the experimental scenario presented in one of the next chapters, a properly configured stabilization window may temporarily preserve a running replica before KEDA ultimately performs the final scale-to-zero operation. Delaying the final scale-down therefore enables a smoother and more controlled transition between the active and inactive states of the application.

Configured Cooldown Period
Suppose that a cooldown period of 300 seconds is configured. Rather than immediately scaling the Deployment down after 18:00, KEDA postpones the scale-down operation while continuously evaluating the scaling conditions. If the system re-enters an active time window or if the scaling conditions change before the cooldown period expires, unnecessary scaling operations can be avoided. Only after the configured cooldown interval has elapsed does KEDA proceed with the final scale-to-zero operation.

Final Observation
Although the cooldown period is available for both event-driven and time-based autoscaling, its practical role differs depending on the nature of the workload. In event-driven scenarios, such as queue-based autoscaling, the cooldown period primarily prevents oscillations caused by intermittent workloads and bursty event arrivals. Conversely, in Cron-based autoscaling the workload schedule is deterministic. Consequently, the cooldown period mainly serves to smooth transitions between predefined operational states rather than reacting to unpredictable workload variations. Therefore, the effectiveness of the cooldown period is generally more evident in event-driven scenarios, whereas in time-based autoscaling it primarily contributes to achieving smoother transitions between scheduled scaling phases.

8.6 Chapter Summary

This chapter has presented the mechanisms that regulate gradual autoscaling in both native Kubernetes and KEDA-based environments. Starting from the native Horizontal Pod Autoscaler, the chapter introduced scaling policies and stabilization windows, illustrating how these mechanisms control the execution of scaling decisions in order to avoid abrupt replica variations and improve application stability. The discussion then showed how KEDA integrates with the native Kubernetes autoscaling framework. Rather than replacing the Horizontal Pod Autoscaler, KEDA automatically generates and manages an HPA from the corresponding `ScaledObject`, allowing the same scaling policies and stabilization windows to be configured through the `advanced.horizontalPodAutoscalerConfig` section. Consequently, whenever the number of running replicas is greater than zero, the gradual scaling behavior remains entirely governed by the native HPA mechanisms. The chapter also highlighted the intrinsic limitation of the Horizontal Pod Autoscaler in handling transitions involving zero replicas. In these situations, KEDA bypasses the HPA and directly updates the number of replicas of the target Deployment, enabling both scale-from-zero and scale-to-zero operations. Finally, the cooldown period was introduced as an additional KEDA mechanism that complements the HPA stabilization window. While the stabilization window filters scaling recommendations before they are applied, the cooldown period delays scale-down operations towards zero replicas after the workload becomes inactive. The queue-based and Cron-based examples demonstrated how these two mechanisms contribute to improving system stability under different workload characteristics, preventing unnecessary oscillations while maintaining application responsiveness. Overall, native Kubernetes and KEDA provide complementary capabilities for autoscaling. Kubernetes determines *how* scaling operations are performed through scaling policies and stabilization windows, whereas KEDA extends these capabilities by enabling autoscaling scenarios involving zero replicas and external event sources. The concepts presented in this chapter provide the theoretical foundation for the experimental demonstrations discussed in the following chapters, beginning with the implementation of the Cron scaler.

Chapter 9

Experimental Evaluation of Time-Based Autoscaling with KEDA Cron Scaler

9.1 Experimental Objective

The objective of this experiment is to evaluate the behavior of KEDA when using the Cron scaler, highlighting its ability to perform time-based autoscaling while seamlessly integrating with the native Kubernetes autoscaling framework. Unlike the previous CPU-based experiments, where scaling decisions were driven by resource utilization metrics, the Cron scaler performs autoscaling according to predefined time windows. This enables the implementation of scheduled scaling strategies, including scale-to-zero capabilities, which are not supported by the native Kubernetes Horizontal Pod Autoscaler. A simple NGINX-based Deployment is adopted as the experimental workload. As in the previous experiments, the application itself is intentionally kept simple because its internal logic does not influence the autoscaling process. Instead, scaling decisions are entirely determined by the time-based trigger configured in the corresponding ScaledObject.

9.2 Experimental Setup

The experiment is performed on a Kubernetes cluster configured with the components required to support time-based autoscaling through KEDA. Table 9.1 summarizes the main resources employed during the experiment:

Resource	**Purpose**
KEDA	Provides event-driven autoscaling through the Cron scaler and manages the generated Horizontal Pod Autoscaler.
Cron Scaler	Generates a synthetic external metric according to predefined time windows.
NGINX Deployment	Stateless workload used to observe the autoscaling behavior.
Horizontal Pod Autoscaler	Automatically generated by KEDA to manage scaling operations while at least one replica is running.
Metrics Server	Not required, since no resource-based metrics are used during the experiment.

Table 9.1: Resources used in the Cron scaler experiment.

The experimental environment is characterized by the following configuration:

- KEDA is installed as an add-on to the Kubernetes cluster
- The Metrics Server is not required because scaling decisions are entirely time-driven rather than resource-based
- A simple stateless NGINX Deployment is used as the target workload

9.3 Deployment Configuration

The first step consists of deploying the application that will be managed by KEDA throughout the experiment.

File: deployment.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: cron-demo
spec:
  replicas: 1
  selector:
    matchLabels:
      app: cron-demo
  template:
    metadata:
      labels:
        app: cron-demo
    spec:
      containers:
      - name: nginx
        image: nginx
```

Listing 9.1: Deployment manifest used in the Cron scaler experiment.

The manifest shown above defines the Kubernetes Deployment used throughout the experiment. The Deployment initially starts with a single replica, which represents the initial state before KEDA takes control of the autoscaling process. As the experiment progresses, the number of replicas is dynamically adjusted according to the Cron scaler configuration rather than workload metrics.

Most Important Fields Explanation:

- **replicas** Defines the initial number of Pod replicas created when the Deployment is first applied. In this experiment, the Deployment starts with a single replica, representing the initial state before KEDA takes control of the autoscaling process. Once the corresponding ScaledObject becomes active, this value is dynamically overridden according to the scaling decisions performed by KEDA and the generated Horizontal Pod Autoscaler.

- **selector/labels** The `selector` identifies the Pods managed by the Deployment by matching the labels defined in the Pod template. This association allows Kubernetes to correctly maintain the desired state of

the Deployment. The same labels are also indirectly used by the Horizontal Pod Autoscaler and KEDA to identify the target Pods associated with the managed Deployment.

- **container** Specifies the application executed inside each Pod. In this experiment, a standard NGINX container is adopted as a lightweight, stateless workload whose only purpose is to observe the autoscaling behavior. Since scaling decisions are entirely driven by the Cron trigger rather than resource utilization, no CPU or memory requests and limits are required.

Key Observation
As anticipated in the previous theoretical chapter, the application logic is intentionally irrelevant in this experiment. Unlike resource-based autoscaling, where CPU or memory utilization determines scaling decisions, the Cron scaler relies exclusively on predefined time conditions. Consequently, the NGINX container simply provides a lightweight and stateless workload that allows the autoscaling behavior to be observed without introducing workload-dependent effects.

9.4 ScaledObject Configuration

The second step consists of configuring the `ScaledObject`, which represents the core resource of the experiment. As discussed in the previous chapter, the ScaledObject defines both the trigger conditions that activate autoscaling and the configuration of the Horizontal Pod Autoscaler automatically generated by KEDA. In this experiment, the ScaledObject combines a Cron trigger with a customized HPA behavior. The Cron trigger determines when the workload should be active, while the generated Horizontal Pod Autoscaler manages scaling operations whenever at least one replica is running.

File: scaledobject.yaml

```
1 apiVersion: keda.sh/v1alpha1
2 kind: ScaledObject
3 metadata:
4   name: cron-demo-scaledobject
5 spec:
```

```
6   scaleTargetRef:
7     name: cron-demo
8   minReplicaCount: 0
9   maxReplicaCount: 10
10  cooldownPeriod: 60
11  pollingInterval: 15
12  advanced:
13    horizontalPodAutoscalerConfig:
14      behavior:
15        scaleUp:
16          stabilizationWindowSeconds: 0
17        scaleDown:
18          stabilizationWindowSeconds: 30
19  triggers:
20  - type: cron
21    metadata:
22      timezone: Europe/Rome
23      start: "*/2 * * * *"
24      end: "*/3 * * * *"
25      desiredReplicas: "10"
```

Listing 9.2: ScaledObject manifest used in the KEDA Cron scaler experiment.

The manifest shown above defines the KEDA ScaledObject responsible for managing the Deployment introduced in the previous section. Besides specifying the Cron trigger, the configuration customizes several autoscaling parameters, including the minimum and maximum number of replicas, the polling interval, the cooldown period, and the behavior of the Horizontal Pod Autoscaler automatically generated by KEDA. Unlike the previous CPU-based experiments, no resource metrics are monitored. Instead, scaling decisions are entirely determined by predefined time conditions expressed through Cron expressions.

Most Important Fields Explanation:

- **scaleTargetRef** Specifies the Kubernetes Deployment managed by the ScaledObject. In this experiment, KEDA continuously monitors the Cron trigger and applies all scaling operations to the `cron-demo` Deployment.

- **minReplicaCount** Defines the minimum number of replicas maintained by KEDA. Setting this value to zero enables the scale-to-zero capability outside the configured active time window, a feature that is not supported by the native Horizontal Pod Autoscaler.

- **maxReplicaCount** Defines the maximum number of replicas that may be created during scale-up operations. This value is automatically propagated to the Horizontal Pod Autoscaler generated by KEDA and therefore represents the upper bound for all scaling decisions.

- **cooldownPeriod** Defines the waiting time, expressed in seconds, before KEDA performs a scale-down operation after the last active trigger condition. In this experiment:

  $$\texttt{cooldownPeriod} = 60 \text{ s}$$

  Consequently, KEDA waits 60 seconds before performing the final transition towards zero replicas. This parameter serves two important purposes:

  - It prevents immediate scale-to-zero operations
  - It reduces unnecessary oscillations caused by temporary inactivity periods

- **pollingInterval** Specifies how frequently the KEDA controller evaluates the trigger condition. In this experiment:

  $$\texttt{pollingInterval} = 15 \text{ s}$$

  Every polling cycle:

  - The Cron trigger is evaluated
  - The corresponding external metric is recomputed

- **advanced.horizontalPodAutoscalerConfig** Allows customization of the Horizontal Pod Autoscaler automatically generated by KEDA. More specifically, this section configures the stabilization windows used by the HPA during scale-up and scale-down operations.

- **scaleUp.stabilizationWindowSeconds** The stabilization window for scale-up operations is explicitly set to zero seconds. Consequently, the Horizontal Pod Autoscaler immediately reacts to increasing workload conditions without delaying scale-up decisions.
- **scaleDown.stabilizationWindowSeconds** A stabilization window of 30 seconds is configured for scale-down operations. During this interval, the HPA stores recent scaling recommendations and selects the highest recommendation before reducing the number of replicas. Compared with the default Kubernetes value of 300 seconds, the stabilization window has intentionally been reduced to 30 seconds in order to:
  - Simplify the observation of the scaling process
  - Allow the HPA to progressively reduce the replica count toward one replica
  - Ensure that the stabilization phase completes before the 60-second KEDA cooldown period performs the final scale-to-zero operation

9.4.1 Default HPA Scaling Policies

No custom scaling policies are explicitly specified within the horizontalPodAutoscalerConfig section of the ScaledObject. Consequently, the Horizontal Pod Autoscaler automatically generated by KEDA adopts the default Kubernetes scaling policies discussed in the previous chapter. More specifically, the default scale-up policy allows the maximum between:

- A 100% increase of the current number of replicas
- An increase of four Pods every 15 seconds

Conversely, the default scale-down policy allows reducing up to 100% of the current replicas every 15 seconds. Although the default scale-down policy itself is relatively aggressive, the configured stabilization window of 30 seconds prevents immediate replica removal by filtering scale-down recommendations before they are applied. Overall, these default policies provide a suitable configuration for this experiment, enabling rapid scale-up reactions while still allowing controlled and progressive scale-down behavior through the stabilization window.

9.4.2 Cron Trigger Configuration

The `triggers` section defines the time-based condition that activates autoscaling. In this experiment, a Cron trigger is adopted to periodically generate an external metric according to predefined scheduling rules.

Most Important Fields

- **timezone** Specifies the time zone used to evaluate the Cron expressions. In this experiment, the value

  `Europe/Rome`

  ensures that all scheduling decisions are evaluated according to the local time zone.

- **start** Defines the Cron expression corresponding to the beginning of the active time window.

- **end** Defines the Cron expression corresponding to the end of the active time window.

- **desiredReplicas** Specifies the desired number of replicas maintained while the Cron trigger remains active. In this experiment, the Deployment is scaled to ten replicas during the active time window.

Important Clarification
Unlike resource-based scalers, the Cron scaler does not measure CPU utilization, memory consumption or any other workload-related metric. Instead, it periodically evaluates the configured Cron expressions and generates a synthetic external metric representing whether the current time falls inside or outside the configured active window. This synthetic metric is then consumed by the Horizontal Pod Autoscaler, which performs the actual scaling decisions according to the configured replica target.

9.4.3 Interpretation of the Time Window in the Cron Scaler Configuration

The time window defined by the Cron scaler is specified through standard Cron expressions that determine when autoscaling should become active. In the presented configuration, the following expressions are used:

```
start: "*/2 * * * *"
end:   "*/3 * * * *"
```

The first expression generates a start event every even minute, whereas the second generates an end event every minute divisible by three. Consequently, the active time window is not defined as a fixed interval but rather as the period between the most recent start event and the following end event. This produces a sequence of short recurring activation windows.

Example

Case 1

- Start event at minute 2
- End event at minute 3

The resulting active window is

$$[2, 3)$$

Case 2

- Start event at minute 4
- End event at minute 6

The resulting active window is

$$[4, 6)$$

These examples show that the duration of each activation window is not constant but depends on the relative occurrence of the independent start and end events.

Important Observation
Unlike traditional scheduling systems, where an active interval is explicitly defined, the KEDA Cron scaler relies on two independent recurring Cron schedules. The application becomes active whenever a start event occurs and remains active until the subsequent end event is triggered. Consequently, the active time window is implicitly determined by the interval between these two events rather than being explicitly specified as a continuous period.

Testing Configuration
The Cron expressions adopted in this experiment have intentionally been chosen to generate short recurring activation windows that simplify the observation of the autoscaling behavior. Such a configuration is particularly suitable for demonstration purposes, allowing multiple scale-up and scale-down cycles to be observed within a relatively short period of time. In production environments, however, more intuitive Cron expressions are generally preferred. For example, a typical business-hours configuration would be:

```
start: "0 9 * * *"
end:   "0 18 * * *"
```

This configuration defines a continuous daily activation window from 09:00 to 18:00.

Practical Example
A typical application of Cron-based autoscaling can be found in enterprise environments where workload demand follows predictable daily patterns. Consider, for instance, a company providing a web-based customer support platform that is primarily accessed during standard business hours. In this scenario, user traffic is significantly higher between 09:00 and 18:00, while it decreases to nearly zero during the night. By leveraging the Cron scaler, the Deployment can automatically scale up at the beginning of the working day to proactively allocate sufficient computing resources before user requests increase. Conversely, outside business hours the application can progressively scale down, potentially reaching zero replicas when no workload is expected. This strategy provides several important advantages:

- Significant resource optimization by avoiding unnecessary resource allocation during idle periods
- Reduced infrastructure costs, particularly in cloud environments where resources are billed according to actual usage
- Improved responsiveness during predictable peak hours by allocating replicas before demand increases
- Simplified autoscaling logic in deterministic scenarios, eliminating the need for more complex metric-based or event-driven triggers

9.5 Resource Deployment

Once both manifests have been defined, the corresponding Kubernetes resources can be deployed to the cluster. The Deployment is created first, followed by the ScaledObject, which activates the KEDA autoscaling logic and automatically manages the associated Horizontal Pod Autoscaler.

Apply Commands

```
kubectl apply -f deployment.yaml
kubectl apply -f scaledobject.yaml
```

Command Description

- **kubectl apply -f deployment.yaml** Creates the NGINX Deployment in the default namespace. At this stage, the application is deployed with the initial replica count specified in the Deployment manifest.
- **kubectl apply -f scaledobject.yaml** Creates the ScaledObject resource and activates the KEDA autoscaling logic. From this point onward, KEDA continuously monitors the configured Cron trigger and automatically manages the scaling behavior of the target Deployment.

Internal Workflow
After the ScaledObject has been applied, several internal operations are automatically performed before autoscaling becomes active.

1. The submitted ScaledObject is validated by the KEDA Admission Webhook, which verifies the correctness of its configuration.
2. The KEDA Controller detects the new ScaledObject, registers the Cron scaler, and automatically creates the corresponding Horizontal Pod Autoscaler associated with the target Deployment.
3. The KEDA Metrics Adapter exposes the synthetic external metric generated by the Cron scaler through the Kubernetes External Metrics API, allowing the generated Horizontal Pod Autoscaler to retrieve the metric and perform the corresponding scaling decisions.

The interaction between the KEDA Controller, the Metrics Adapter, and the automatically generated Horizontal Pod Autoscaler follows the architecture described in Chapter 3, where the role of each component has been discussed in detail. At this point, the autoscaling system is fully operational and continuously evaluates the configured Cron expressions according to the specified polling interval.

9.6 Monitoring the Experiment

Once the Deployment and the corresponding ScaledObject have been successfully created, the experiment can be monitored in real time. Several Kubernetes commands are used to observe the behavior of the generated Horizontal Pod Autoscaler, verify the evolution of the Deployment replica count, and correlate scaling decisions with the actual Pod lifecycle.

Monitoring Commands

```
kubectl get hpa -w
kubectl get deployment cron-demo -w
kubectl get pods -w
```

Command Description

- **kubectl get hpa -w** The `-w` option enables watch mode, allowing the command output to be continuously updated as the resource changes. This command represents the primary monitoring tool used throughout the experiment because it:
  - Displays the external metric exposed by KEDA
  - Reports the current number of replicas managed by the generated Horizontal Pod Autoscaler
  - Allows scale-up and scale-down decisions to be observed in real time
- **kubectl get deployment cron-demo -w** Also executed in watch mode, this command continuously reports changes to the Deployment. It is particularly useful for verifying when the scaling decisions computed by the Horizontal Pod Autoscaler are actually applied to the target Deployment.

- **kubectl get pods -w** This command continuously displays Pod creation and termination events. Consequently, it allows the Pod lifecycle to be directly correlated with the scaling decisions previously observed through the Horizontal Pod Autoscaler.

Monitoring Strategy
Throughout the experiments, the command

```
kubectl get hpa -w
```

was adopted as the primary monitoring tool because it provides direct visibility into both the external metric generated by the Cron scaler and the corresponding scaling decisions performed by the Horizontal Pod Autoscaler. The Deployment and Pod monitoring commands were instead used to verify when those scaling decisions were effectively propagated to the Kubernetes resources, allowing the entire autoscaling process to be observed from the generated metric down to the actual creation and termination of Pods.

9.7 Experimental Results

9.7.1 Observed Behavior

To simplify the analysis, only the `TARGETS` and `REPLICAS` fields obtained through the following monitoring command are reported:

```
kubectl get hpa -w
```

For readability, only the `TARGETS` and `REPLICAS` columns of the command output are shown below. The remaining fields remain constant throughout the experiment and therefore do not contribute to the interpretation of the autoscaling behavior. More specifically:

- **REFERENCE** = `Deployment/cron-demo`
- **NAME** = `keda-hpa-cron-demo-scaledobject`
- **MINPODS** = 1
- **MAXPODS** = 10

Similarly, the `AGE` column is omitted because its value continuously increases during the experiment and does not provide useful information for understanding the observed autoscaling behavior.

TARGETS	REPLICAS
<unknown>/1 (avg)	0
10/1 (avg)	1
2/1 (avg)	5
1/1 (avg)	10
0/1 (avg)	10
0/1 (avg)	10
0/1 (avg)	1
<unknown>/1 (avg)	0

Table 9.2: Simplified output of `kubectl get hpa -w` observed during the Cron scaler experiment.

9.7.2 Metric Interpretation

Before analysing the individual scaling phases, it is important to clarify how the metric values displayed in the Horizontal Pod Autoscaler output should be interpreted. As discussed in the previous chapter, KEDA configures the generated Horizontal Pod Autoscaler using the following metric specification:

```
type: AverageValue
target: 1
```

Consequently, the metric displayed by the command

```
kubectl get hpa -w
```

is normalized by the current number of running replicas. More specifically, the displayed metric corresponds to

$$\text{displayed metric} = \frac{\text{actual metric}}{\text{current replicas}}$$

where:

- **actual metric** represents the raw metric generated by the KEDA Cron scaler, corresponding to the desired number of replicas
- **current replicas** denotes the number of Pods currently running in the Deployment

Therefore, to reconstruct the original metric exposed by KEDA, the previous relationship can be inverted:

$$\text{actual metric} = \text{displayed metric} \times \text{current replicas}$$

The reconstructed value corresponds to the desired replica count generated by the Cron scaler during the active time window. For this reason, throughout the following analysis the raw metric produced by KEDA will be explicitly reconstructed by multiplying the observed `TARGETS` value by the corresponding `REPLICAS` value. This allows the scaling decisions performed by the generated Horizontal Pod Autoscaler to be interpreted directly in terms of the original metric generated by the Cron trigger.

9.7.3 Scale from Zero

Initially, the Deployment is scaled down to zero replicas, producing the following Horizontal Pod Autoscaler output:

TARGETS	REPLICAS
`<unknown>/1 (avg)`	0

Table 9.3: Initial state before the Cron trigger becomes active.

At this stage no Pods are running and, consequently, the Horizontal Pod Autoscaler cannot perform any scaling decisions. When the configured Cron window becomes active, KEDA detects the trigger condition and immediately activates the workload:

TARGETS	REPLICAS
`10/1 (avg)`	1

Table 9.4: First sample after the Cron trigger becomes active.

The metric generated by the Cron scaler is equal to the configured value

```
desiredReplicas: 10
```

Consequently, KEDA performs the initial transition

$$0 \rightarrow 1$$

and subsequently transfers the control of the autoscaling process to the generated Horizontal Pod Autoscaler.

9.7.4 Scale-Up Performed by the Horizontal Pod Autoscaler

Once the first replica has been created, the Horizontal Pod Autoscaler becomes responsible for all subsequent scaling operations. As discussed in the previous chapter, the generated HPA uses:

$$\text{desired replicas} = \left\lceil \frac{\text{current replicas} \times \text{current metric}}{\text{target metric}} \right\rceil$$

As discussed in the previous theoretical chapter, the Horizontal Pod Autoscaler always rounds the computed replica count up to the nearest integer. Consequently, even when the formula returns a non-integer value, the desired number of replicas is obtained by applying the ceiling function.

First HPA Evaluation
The first sample observed after the Cron trigger becomes active is

TARGETS	REPLICAS
`10/1 (avg)`	1

Table 9.5: First HPA evaluation after the initial scale-from-zero operation.

The Horizontal Pod Autoscaler therefore computes

$$1 \times \frac{10}{1} = 10$$

indicating that the desired Deployment size is equal to ten replicas. However, the Horizontal Pod Autoscaler cannot immediately perform this scaling operation because the configured scale-up policies limit the maximum number

of replicas that may be added during a single evaluation cycle. Since no custom scaling policies were specified in the ScaledObject, the default Kubernetes policies generated by KEDA are applied:

- Up to four additional Pods every 15 seconds
- Up to a 100% increase of the current replica count every 15 seconds

Starting from one running replica, the two policies produce:

$$1 + 4 = 5$$

$$1 + 100\% = 2$$

The Horizontal Pod Autoscaler selects the most permissive policy:

$$\max(5, 2) = 5$$

which produces the next observed sample:

TARGETS	REPLICAS
`2/1 (avg)`	5

Table 9.6: Second sample after applying the default scale-up policies.

As explained in the previous subsection, the metric displayed by the Horizontal Pod Autoscaler is normalized by the current number of replicas. Consequently,

$$\frac{10}{5} = 2$$

which explains the value reported in the `TARGETS` column.

Second HPA Evaluation
During the following evaluation cycle, the Horizontal Pod Autoscaler computes

$$5 \times \frac{2}{1} = 10$$

Once again, the desired Deployment size is equal to ten replicas. Applying the default scale-up policies yields

$$5 + 4 = 9$$

and

$$5 + 100\% = 10$$

The most permissive policy is therefore selected:

$$\max(9, 10) = 10$$

producing the following sample:

TARGETS	REPLICAS
`1/1 (avg)`	10

Table 9.7: Final scale-up phase reaching the desired number of replicas.

At this point, the Deployment has reached the desired size of ten replicas and remains stable until the Cron trigger becomes inactive.

9.7.5 Exit from the Active Time Window

When the configured Cron window expires, KEDA updates the synthetic metric generated by the Cron scaler, producing the following Horizontal Pod Autoscaler output:

TARGETS	REPLICAS
`0/1 (avg)`	10

Table 9.8: First sample after the Cron trigger becomes inactive.

Although the Deployment still consists of ten running replicas, the metric exposed by the Cron scaler becomes zero because the active time window has expired. Consequently, the desired replica count computed by the Horizontal Pod Autoscaler becomes

$$\left\lceil 10 \times \frac{0}{1} \right\rceil = 0.$$

Since the Horizontal Pod Autoscaler cannot scale below its minimum replica count, the effective desired value becomes

$$1.$$

However, as discussed in the previous chapter, this recommendation is not immediately applied because the configured stabilization window filters recent scale-down recommendations before allowing replica removal.

9.7.6 Stabilization Window Analysis

The ScaledObject configuration specifies the following scale-down stabilization window:

```
scaleDown:
  stabilizationWindowSeconds: 30
```

Since the default Horizontal Pod Autoscaler evaluation period is 15 seconds, the stabilization window evaluates all recommendations generated during the previous 30 seconds, including the current recommendation. The stabilization process begins when the first scale-down recommendation is generated, corresponding to the first occurrence of:

TARGETS	REPLICAS
0/1 (avg)	10

Table 9.9: First HPA sample triggering the activation of the stabilization window.

At this point, the metric generated by the Cron scaler drops to zero, causing the Horizontal Pod Autoscaler to compute its first scale-down recommendation. Consequently, the stabilization window becomes active and starts evaluating all recommendations generated within the configured 30-second observation interval.

First Scale-Down Sample
The stabilization window currently contains the following samples:

TARGETS	REPLICAS
2/1 (avg)	5
1/1 (avg)	10
0/1 (avg)	10

Table 9.10: Samples considered during the first scale-down evaluation.

The corresponding replica recommendations are reported in Table 9.11:

Evaluation Instant	Recommended Replicas
Previous recommendation	10
Previous recommendation	10
Current recommendation	1

Table 9.11: Replica recommendations considered during the first scale-down evaluation.

The current recommendation corresponds to one replica because

$$\left\lceil 10 \times \frac{0}{1} \right\rceil = 0,$$

and the Horizontal Pod Autoscaler cannot scale below its minimum replica count. Therefore, the stabilization window selects:

$$\max(10, 10, 1) = 10.$$

Consequently, no scale-down operation is performed.

Observed Result

TARGETS	REPLICAS
`0/1 (avg)`	10

Table 9.12: Observed result after the first scale-down evaluation.

Second Scale-Down Sample

During the following HPA evaluation cycle, the stabilization window contains the following samples:

TARGETS	REPLICAS
`1/1 (avg)`	10
`0/1 (avg)`	10
`0/1 (avg)`	10

Table 9.13: Samples considered during the second scale-down evaluation.

The corresponding recommendations become

Evaluation Instant	Recommended Replicas
Oldest recommendation inside the window	10
Previous recommendation	1
Current recommendation	1

Table 9.14: Replica recommendations considered during the second scale-down evaluation.

$$\max(10, 1, 1) = 10.$$

Therefore, the replica count remains unchanged.

Observed Result

TARGETS	REPLICAS
`0/1 (avg)`	10

Table 9.15: Observed result after the second scale-down evaluation.

Third Scale-Down Sample

During the following evaluation cycle, the oldest recommendation equal to ten replicas leaves the stabilization window. The remaining samples are

TARGETS	REPLICAS
`0/1 (avg)`	10
`0/1 (avg)`	10
`0/1 (avg)`	1

Table 9.16: Samples considered during the third scale-down evaluation.

The corresponding recommendations are

Evaluation Instant	Recommended Replicas
Oldest remaining recommendation	1
Previous recommendation	1
Current recommendation	1

Table 9.17: Replica recommendations considered during the third scale-down evaluation.

The stabilization window now returns

$$\max(1, 1, 1) = 1,$$

allowing the Horizontal Pod Autoscaler to finally perform the scale-down operation

$$10 \rightarrow 1.$$

Observed Result

TARGETS	REPLICAS
0/1 (avg)	1

Table 9.18: Observed result after the third scale-down evaluation.

9.7.7 Scale-to-Zero Performed by KEDA

After the workload has been reduced to a single replica, the Horizontal Pod Autoscaler cannot further decrease the replica count because its minimum number of replicas is equal to one. At this stage, KEDA waits for the configured cooldown period to expire:

```
1 cooldownPeriod: 60
```

Once the cooldown period has elapsed and the Cron trigger remains inactive, KEDA directly updates the Deployment, performing the final transition

$$1 \rightarrow 0.$$

This produces the final Horizontal Pod Autoscaler sample:

TARGETS	REPLICAS
<unknown>/1 (avg)	0

Table 9.19: Final HPA sample after KEDA performs the scale-to-zero operation.

The experiment clearly highlights the division of responsibilities between KEDA and the Horizontal Pod Autoscaler:

- KEDA manages the transitions from zero to one replica and from one replica to zero
- The Horizontal Pod Autoscaler manages all scaling operations while at least one replica is running
- The stabilization window governs the transition from ten replicas to one replica
- The cooldown period delays the final scale-to-zero operation

9.8 Chapter Summary

This chapter experimentally evaluated the behavior of the KEDA Cron scaler, demonstrating how time-based autoscaling can be achieved through predefined scheduling rules while maintaining seamless integration with the native Kubernetes autoscaling framework. The experimental results showed that, unlike native Kubernetes autoscaling, KEDA extends the Horizontal Pod Autoscaler by enabling both scale-from-zero and scale-to-zero operations, capabilities that are not natively supported by Kubernetes. The experiment also highlighted the complementary responsibilities of KEDA and the Horizontal Pod Autoscaler. Whenever at least one replica is running, scaling decisions are performed by the Horizontal Pod Autoscaler according to the synthetic metric generated by the Cron scaler and the configured scaling policies. Conversely, when the workload reaches zero replicas, KEDA becomes responsible for reactivating the application and restoring the control of the Horizontal Pod Autoscaler. The experimental observations further demonstrated the importance of both the stabilization window and the cooldown period. The stabilization window allows the Horizontal Pod Autoscaler to evaluate multiple past recommendations before applying scale-down operations, thereby preventing premature replica reductions caused by transient metric fluctuations. Once the workload has been reduced to the minimum replica count, the cooldown period determines how long KEDA waits before performing the final transition from one replica to zero. Overall, the experiment confirms that KEDA effectively extends the native Kubernetes autoscaling model by combining event-driven activation with the decision-making capabilities of the Horizontal Pod Autoscaler. Although the triggering condition considered in this chapter is purely time-based, the same architectural principles apply to more advanced event-driven scenarios.

These scenarios, based on external event sources rather than scheduled time windows, will be investigated in the following chapters.

Chapter 10

Application-Level Metrics and Monitoring: Prometheus and Grafana

10.1 Introduction

In the previous chapters, different autoscaling approaches in Kubernetes have been analyzed, starting from native CPU-based autoscaling and progressively introducing KEDA to enable event-driven scaling mechanisms. Although CPU and memory metrics are effective in many scenarios, they are not always sufficient to accurately represent the actual workload experienced by an application. In particular, several classes of cloud-native applications, including web servers, message consumers, and streaming services, may experience significant workload variations that are not directly reflected by resource utilization metrics. As a consequence, relying exclusively on CPU or memory consumption may lead to delayed or inaccurate scaling decisions. To overcome these limitations, modern cloud-native architectures increasingly rely on *application-level metrics*, which provide a more accurate representation of the application's runtime behavior. Within this context, Prometheus has become one of the most widely adopted monitoring and observability platforms for Kubernetes-based environments.

10.2 Prometheus Overview

Prometheus is an open-source monitoring system designed to collect, store, and query time-series data. Unlike traditional monitoring solutions, it adopts a pull-based model, periodically scraping metrics from configured targets through HTTP endpoints, typically exposed using the standard `/metrics` format. Each collected metric consists of a numerical value associated with a timestamp and a set of labels, allowing flexible filtering, aggregation, and querying operations. This time-series data model makes Prometheus particularly suitable for monitoring highly dynamic and distributed cloud-native applications. Unlike native Kubernetes resource metrics, such as CPU and memory utilization, which are automatically collected by Kubernetes components, application-level metrics must be explicitly exposed either by the application itself or through dedicated software components known as exporters.

10.3 Prometheus Exporters

A fundamental concept in the Prometheus ecosystem is the use of *exporters*, which act as intermediaries between applications and the monitoring system. Their primary responsibility is to retrieve application-specific statistics and transform them into a format compatible with Prometheus. For example, in the case of a web server such as NGINX, internal runtime statistics, including the number of active connections and processed requests, are not natively exposed using the Prometheus metric format. To overcome this limitation, a dedicated exporter, such as the NGINX Prometheus Exporter, can be deployed. A common deployment strategy consists of running the exporter as a sidecar container within the same Pod as the monitored application. In this configuration, the exporter retrieves the internal NGINX statistics, typically exposed through the `/stub_status` endpoint, converts them into Prometheus-compatible metrics, and exposes them through a standard `/metrics` endpoint that can subsequently be scraped by the Prometheus server. Once collected, these application-level metrics can be exploited for two main purposes. First, they provide advanced monitoring and visualization capabilities, often through tools such as Grafana, which allows the creation of customizable dashboards for real-time observation of the system behavior. Although Grafana does not directly participate in the autoscaling process, it represents a fundamental observability tool for validating and analysing the collected metrics. Second, and

most importantly within the context of this thesis, application-level metrics can be employed to implement autoscaling strategies based on the actual behavior of the application rather than on infrastructure-level resource consumption.

10.4 Prometheus Integration with Kubernetes Autoscaling

From an architectural perspective, Prometheus can be deployed either inside the Kubernetes cluster, typically as a set of Pods managed through a Deployment or installed using a Helm chart, or externally as a standalone monitoring service. In both cases, Prometheus is capable of automatically discovering and scraping metrics from multiple Pod replicas, making it particularly suitable for highly dynamic cloud-native environments. Within Kubernetes, Prometheus metrics can be integrated into the autoscaling process through two different approaches. Without KEDA, application-level metrics cannot be directly consumed by the Horizontal Pod Autoscaler. Instead, an additional component, known as the *Prometheus Adapter*, must be deployed. The adapter retrieves metrics from Prometheus and exposes them to the Kubernetes API Server through the API Aggregation Layer, allowing the Horizontal Pod Autoscaler to consume them as custom or external metrics. When KEDA is adopted, this architecture becomes considerably simpler. Rather than relying on an external Prometheus Adapter, KEDA can directly query Prometheus and expose the resulting metrics to the Horizontal Pod Autoscaler through its internal Metrics Adapter. Consequently, the number of required components is reduced and the overall autoscaling configuration becomes significantly simpler. The following chapter analyzes in detail the complete autoscaling workflow based on Prometheus metrics without KEDA. Subsequently, the same scenario will be revisited by introducing KEDA, highlighting the architectural differences as well as the advantages in terms of flexibility and configuration complexity.

10.5 Chapter Summary

This chapter introduced the role of application-level metrics in modern cloud-native environments, highlighting the limitations of relying exclusively on infrastructure-level resource metrics such as CPU and memory utilization.

In this context, Prometheus was presented as a monitoring and observability platform capable of collecting, storing, and querying application-specific time-series metrics. The chapter also described the role of exporters, which bridge the gap between applications and Prometheus by exposing application-specific statistics in a Prometheus-compatible format. Besides enabling advanced monitoring and visualization through Grafana, these metrics provide the foundation for implementing autoscaling strategies driven by the actual behavior of an application. Finally, the integration of Prometheus with Kubernetes autoscaling was introduced by comparing the native approach, which requires a dedicated Prometheus Adapter, with the KEDA-based solution, where metrics are directly retrieved from Prometheus through KEDA's internal Metrics Adapter. Building upon these concepts, the next chapter will examine the complete autoscaling workflow based on Prometheus metrics without KEDA before extending the same scenario using KEDA.

Chapter 11

Application-Level Metrics Autoscaling Based on Prometheus without KEDA

11.1 Introduction

In this chapter, we describe the complete theoretical workflow required to implement autoscaling in Kubernetes using application-level metrics collected by Prometheus, without relying on KEDA. Unlike native CPU- or memory-based autoscaling, this approach enables scaling decisions based on custom metrics that more accurately reflect the actual workload of an application. To make these metrics consumable by the Kubernetes control plane, an additional component known as the *Prometheus Adapter* must be deployed. This component plays a role analogous to that of the Metrics Server used in native CPU- and memory-based autoscaling. However, instead of exposing resource utilization metrics, the Prometheus Adapter bridges Prometheus and the Horizontal Pod Autoscaler (HPA) by exposing custom and external metrics through the Kubernetes API.

11.2 Architecture Overview

Figure 11.1 illustrates the high-level architecture enabling autoscaling in Kubernetes based on application-level metrics collected through Prometheus. The

diagram highlights the main components involved in the monitoring and autoscaling pipeline.

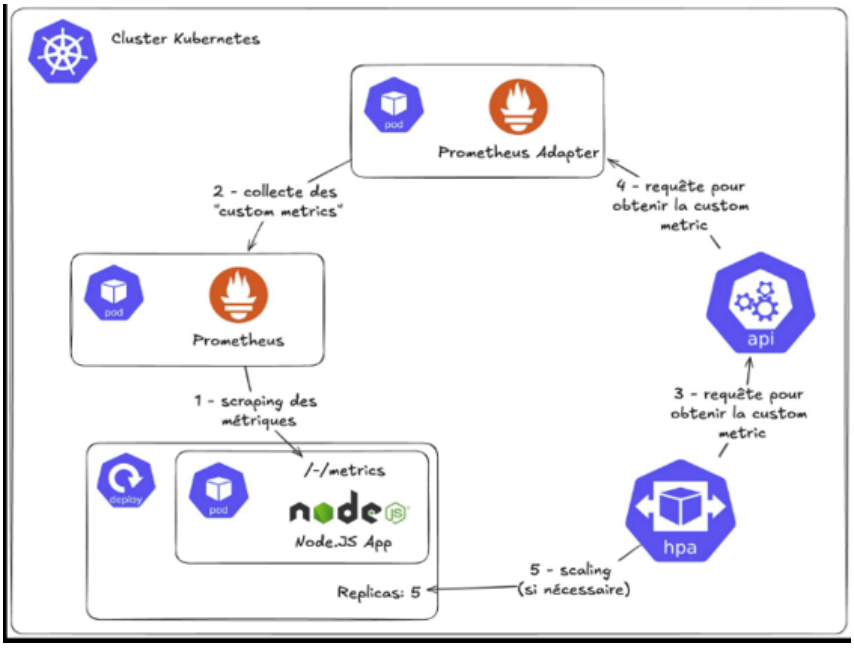


Figure 11.1: High-Level Architecture of Prometheus-Based Autoscaling in Kubernetes (without KEDA) [8].

At the application level, each Pod runs the target workload together with a component responsible for exposing metrics in a format compatible with Prometheus. These metrics are periodically collected by the Prometheus server and subsequently made available to Kubernetes through the Prometheus Adapter, which acts as a bridge between the monitoring infrastructure and the autoscaling mechanism. The Horizontal Pod Autoscaler then consumes these metrics to dynamically adjust the number of Pod replicas associated with a Deployment. It is important to note that the architecture shown in Figure 11.1 refers to a generic application (e.g., a Node.js service). Within the context of this thesis, however, this generic application corresponds to the NGINX-based workload adopted throughout the experimental evaluation. The architectural workflow remains unchanged regardless of the application type, provided that the application exposes metrics that can be collected and processed through the described monitoring pipeline. The architecture involves the following key

components:

- **Application Pods** (e.g., NGINX)
- **Exporter** (e.g., NGINX Prometheus Exporter)
- **Prometheus Server**
- **Prometheus Adapter**
- **Kubernetes API Server** (with the API Aggregation Layer)
- **Horizontal Pod Autoscaler (HPA)**
- **Deployment** resource

11.3 Application Metrics Exposure

The first step of the Prometheus-based autoscaling workflow consists of exposing application-level metrics that can later be collected by Prometheus and ultimately consumed by the Horizontal Pod Autoscaler. Each application Pod runs the main application container (e.g., NGINX) together with an exporter container following the sidecar deployment pattern. The exporter retrieves internal application statistics and exposes them in the Prometheus format through an HTTP endpoint, typically `/metrics`. For example, in the case of NGINX:

- NGINX exposes raw runtime statistics through the `/stub_status` endpoint
- The exporter converts these statistics into Prometheus metrics, such as:
  - `nginx_connections_active`;
  - `nginx_connections_waiting`.

Once converted, these metrics become accessible to the Prometheus server, which periodically collects them during its scraping process.

11.3.1 NGINX Metrics Exposure: The `/stub_status` Endpoint

Before describing how application-level metrics are collected and subsequently used for autoscaling, it is important to understand how these metrics are generated and exposed, particularly in the case of a web server such as NGINX. Unlike CPU and memory metrics, which are natively provided by Kubernetes components, application-specific metrics are not automatically available and must therefore be explicitly exposed. In the case of NGINX, this is achieved through an intermediate component known as the *NGINX Prometheus Exporter*. NGINX does not natively expose metrics in the Prometheus format. Instead, it provides a built-in module called `stub_status`, which exposes a minimal set of runtime statistics through an HTTP endpoint. To enable this functionality, the following configuration must be added to the NGINX configuration file (`nginx.conf`):

```
location /stub_status {
    stub_status;
    allow 127.0.0.1;
    deny all;
}
```

Listing 11.1: NGINX `stub_status` configuration.

Configuration Description

- `stub_status;` Enables the built-in NGINX status module
- `allow 127.0.0.1;` Restricts access to local requests only, following a common security best practice.
- `deny all;` Prevents external clients from accessing the endpoint

Once enabled, the endpoint is typically exposed internally through

```
http://localhost/stub\_status
```

11.3.2 Raw Output of the `/stub_status` Endpoint

When queried, the endpoint returns a plain-text response similar to the following:

```
Active connections: 10
server accepts handled requests
 100 100 200
Reading: 1 Writing: 2 Waiting: 7
```

Listing 11.2: Example output generated by the NGINX `/stub_status` endpoint.

11.3.3 Interpretation of the Exposed Fields

The information reported by the `/stub_status` endpoint can be interpreted as follows:

- **Active connections**: Total number of currently open client connections
- **accepts**: Total number of accepted client connections
- **handled**: Total number of successfully handled client connections
- **requests**: Total number of processed HTTP requests
- **Reading**: Connections for which NGINX is currently reading the request headers
- **Writing**: Connections for which NGINX is currently sending responses to clients.
- **Waiting**: Idle keep-alive connections waiting for additional client requests. This metric is particularly relevant for autoscaling scenarios because it provides an indication of the workload currently managed by the web server.

11.3.4 Role of the NGINX Prometheus Exporter

The NGINX Prometheus Exporter acts as a translation layer between NGINX and Prometheus, converting the raw statistics exposed by the web server into metrics compatible with the Prometheus data model.

Core Responsibilities
The exporter performs the following operations:

1. Queries the `/stub_status` endpoint exposed by NGINX
2. Parses the raw runtime statistics returned by the endpoint
3. Converts these statistics into the Prometheus metric format
4. Exposes the resulting metrics through an HTTP `/metrics` endpoint

11.3.5 Prometheus Metrics Exposed by the Exporter

Once the raw NGINX statistics have been processed, the exporter exposes several Prometheus-compatible metrics. Some representative examples are reported in Table 11.1.

Metric	Description
`nginx_connections_active`	Total active connections.
`nginx_connections_reading`	Connections currently reading request headers.
`nginx_connections_writing`	Connections currently sending responses.
`nginx_connections_waiting`	Idle keep-alive connections.
`nginx_http_requests_total`	Total number of processed HTTP requests.

Table 11.1: Representative Prometheus metrics exposed by the NGINX Prometheus Exporter.

These metrics exhibit the following characteristics:

- They are represented as time-series values
- They are labeled according to the originating Pod or instance
- They are directly consumable by the Prometheus server

11.3.6 Exporter Deployment Models

The NGINX Prometheus Exporter can be deployed according to different architectural models.

1. **Sidecar Container (Most Common)**
   - Runs within the same Pod as the NGINX container
   - Shares the same network namespace, allowing communication through `localhost`
   - Simplifies the communication between the application and the exporter
2. **Separate Deployment**
   - Runs as an independent Pod
   - Requires network connectivity to the NGINX Service
3. **Embedded in the Application**
   - Metrics are directly exposed by the application itself
   - This solution is not applicable to NGINX, which requires a dedicated exporter

For Kubernetes-based architectures, the sidecar deployment model is generally preferred because it simplifies communication while maintaining a clear separation of responsibilities between the application and the monitoring component.

11.3.7 Exporter Configuration in Kubernetes Deployment

The exporter is typically declared within the `PodSpec` section of a Kubernetes Deployment, as illustrated in Listing 11.3.

```
spec:
  containers:
  - name: nginx
    image: nginx:latest
    ports:
      - containerPort: 80

  - name: nginx-exporter
    image: nginx/nginx-prometheus-exporter:latest
    args:
      - -nginx.scrape-uri=http://127.0.0.1/stub_status
    ports:
      - containerPort: 9113
```

Listing 11.3: Example Kubernetes Deployment including the NGINX Prometheus Exporter.

Configuration Description

Container Definition

- `name:nginx-exporter` identifies the exporter container.
- `image:nginx/nginx-prometheus-exporter` specifies the official NGINX Prometheus Exporter image.

Arguments

- `-nginx.scrape-uri=http://127.0.0.1/stub_status` specifies the endpoint from which the exporter retrieves the NGINX statistics. Since no port number is explicitly specified, the default HTTP port (80) is assumed.

Key Observation

- The exporter communicates through `localhost` because both containers execute within the same Pod and therefore share the same network namespace.

Ports

- `containerPort:9113` specifies the default port used by the exporter to expose Prometheus metrics.

Consequently, the Prometheus server periodically scrapes metrics from:

`http://<pod-ip>:9113/metrics`

11.3.8 Exporter Output (`/metrics` Endpoint)

When the Prometheus server queries the exporter through the `/metrics` endpoint, it receives a response similar to the following:

```
# HELP nginx_connections_active Active connections
# TYPE nginx_connections_active gauge
nginx_connections_active 10

# HELP nginx_connections_waiting Idle connections
# TYPE nginx_connections_waiting gauge
nginx_connections_waiting 7

...
```

Listing 11.4: Example output generated by the NGINX Prometheus Exporter.

Important Conceptual Clarification
NGINX does not expose Prometheus metrics directly. Instead, the NGINX Prometheus Exporter acts as a mandatory intermediary component responsible for translating the native NGINX statistics into the Prometheus exposition format. This architecture reflects a general design pattern adopted by many monitoring systems:

- Applications expose internal runtime statistics using their native format
- Exporters translate these statistics into Prometheus-compatible metrics
- Prometheus periodically collects, stores, and standardizes the resulting time-series data

11.4 Prometheus-Based Autoscaling Workflow

Once application-level metrics have been exposed by the exporter, the autoscaling process proceeds through a sequence of interactions involving Prometheus, the Prometheus Adapter, the Kubernetes API Server, and the Horizontal Pod Autoscaler. The following subsections describe each step of this workflow in detail.

11.4.1 Step 1 – Metrics Collection by Prometheus

The Prometheus server periodically scrapes all configured targets (i.e., exporters) by issuing HTTP requests to their `/metrics` endpoints.

Key Characteristics

- Pull-based monitoring model
- Configurable scrape interval (e.g., every 15 seconds)
- Automatic discovery of Pod replicas through Kubernetes service discovery

After each scraping cycle, Prometheus stores:

- Time-series data
- Metrics labeled according to the originating Pod, namespace, and other metadata

11.4.2 Step 2 – Metric Aggregation and Querying

Prometheus provides its own query language, called *PromQL* (Prometheus Query Language), which enables users to retrieve, filter, and aggregate metrics collected from multiple Pod replicas. For example, the following query computes the average number of waiting connections across all running NGINX Pods:

```
avg(nginx_connections_waiting)
```

In this example, the average value is computed by considering the `nginx_connections_waiting` metric collected from all Pod replicas currently running within the cluster.

Important Clarification
The aggregation operation (e.g., average, sum, maximum, or minimum) is performed entirely by the Prometheus server rather than by the Prometheus Adapter. This design follows the separation of responsibilities adopted by the monitoring architecture:

- Prometheus is responsible for processing and aggregating time-series data
- The Prometheus Adapter simply retrieves values that have already been processed by Prometheus and exposes them to Kubernetes

11.4.3 Step 3 – Prometheus Adapter Integration

The Prometheus Adapter periodically queries the Prometheus server according to a set of predefined mapping rules.

Core Responsibilities

1. Execute PromQL queries
2. Map the query results to the Kubernetes Custom Metrics API or External Metrics API
3. Expose the resulting metrics through the Kubernetes API Server

A conceptual example of such a mapping is shown below:

```
1 metricsQuery: avg(nginx_connections_waiting)
2 name: nginx_connections_waiting_avg
```

In this example, the Prometheus Adapter executes the specified PromQL query and exposes the resulting aggregated value under the logical metric name `nginx_connections_waiting_avg`, which can subsequently be consumed by the Horizontal Pod Autoscaler.

11.4.4 Step 4 – API Aggregation Layer

After retrieving the processed metrics from Prometheus, the Prometheus Adapter exposes them through the Kubernetes API Server by leveraging the API Aggregation Layer.

API Endpoints

- `custom.metrics.k8s.io`
- `external.metrics.k8s.io`

This mechanism is analogous to the one employed by the Metrics Server for native CPU- and memory-based autoscaling.

Metric Retrieval Flow

1. The Horizontal Pod Autoscaler sends a metric request to the Kubernetes API Server.
2. The API Server forwards the request to the Prometheus Adapter through the API Aggregation Layer.
3. The Prometheus Adapter returns the requested metric value to the API Server, making it available to the Horizontal Pod Autoscaler.

11.4.5 Step 5 – HPA Metric Retrieval

The Horizontal Pod Autoscaler periodically queries the Kubernetes API Server to retrieve the application-level metrics exposed by the Prometheus Adapter.

Default Synchronization Interval

- Every 30 seconds (configured through the –horizontal-pod-autoscaler-sync-period parameter).

During each reconciliation cycle, the Horizontal Pod Autoscaler retrieves only the metrics associated with the Pods belonging to the target Deployment, ensuring that scaling decisions are based exclusively on the monitored workload.

11.4.6 Step 6 – Scaling Decision

Unlike native CPU utilization scaling, this approach relies on custom or external metrics collected by Prometheus and exposed through the Prometheus Adapter. A typical Horizontal Pod Autoscaler configuration is shown below.

```
type: External
external:
  metric:
    name: nginx_connections_waiting_avg
  target:
    type: Value
    value: 20
```

Important Remark
The metric name referenced by the Horizontal Pod Autoscaler must exactly match the logical metric name exposed by the Prometheus Adapter rather than the original Prometheus metric name. In this example, the Prometheus Adapter exposes the aggregated metric under the logical name.

```
nginx_connections_waiting_avg
```

which is therefore consistently referenced within the Horizontal Pod Autoscaler specification.

11.4.7 Step 7 – Scaling Formula

Once the application-level metric has been retrieved, the Horizontal Pod Autoscaler computes the desired number of replicas using the same scaling equation introduced for native autoscaling.

$$\text{desired replicas} = \left\lceil \frac{\text{current replicas} \times \text{current metric value}}{\text{target metric value}} \right\rceil$$

where

- **current metric value** is the value returned by the Prometheus Adapter (e.g., the average number of waiting connections)
- **target metric value** is the desired threshold configured in the Horizontal Pod Autoscaler (20 in the previous example)

As discussed in the previous theoretical chapters, the Horizontal Pod Autoscaler always applies the ceiling function to the computed replica count, ensuring that the desired number of replicas is rounded up to the nearest integer whenever a fractional value is obtained. Consequently, the overall scaling logic remains identical to native Kubernetes autoscaling; only the source of the metric changes, moving from infrastructure-level resource utilization to application-level metrics processed by Prometheus.

11.4.8 Step 8 – Deployment Update

After computing the desired number of replicas, the Horizontal Pod Autoscaler updates the

```
spec.replicas
```

field of the target Deployment through the Kubernetes API Server. Once the Deployment specification has been updated, Kubernetes automatically reconciles the cluster state by performing the following operations:

- The Kubernetes Scheduler allocates new Pods when additional replicas are required
- Pods are created or terminated according to the desired replica count
- The cluster progressively converges toward the desired state

11.5 Key Observations

Why CPU-Based Autoscaling is Not Always Sufficient
For applications such as NGINX, CPU utilization is not always an accurate indicator of the actual workload. Even under heavy traffic conditions, CPU usage may remain relatively low because request processing is predominantly I/O-bound and therefore limited by network operations rather than computational resources. Consequently, autoscaling strategies based exclusively on CPU utilization may fail to detect application saturation, resulting in delayed or inadequate scaling decisions. Application-level metrics collected through Prometheus overcome this limitation by providing direct visibility into the internal behavior of the application. As a result, scaling decisions can be based

on metrics that more accurately reflect the real workload experienced by the service.

Role Separation
The Prometheus-based autoscaling architecture follows a clear separation of responsibilities among its components, as summarized in Table 11.2.

Component	Responsibility
Exporter	Converts application-specific statistics into Prometheus-compatible metrics.
Prometheus	Collects, stores, and aggregates time-series metrics.
Prometheus Adapter	Exposes Prometheus metrics through the Kubernetes Metrics APIs.
Horizontal Pod Autoscaler (HPA)	Computes the desired number of replicas and performs scaling decisions.

Table 11.2: Role separation among the components involved in Prometheus-based autoscaling.

The Prometheus Adapter is Mandatory Without KEDA
When Prometheus-based autoscaling is implemented without KEDA, the Horizontal Pod Autoscaler cannot directly query the Prometheus server. An additional component, namely the Prometheus Adapter, is therefore required to bridge the gap between Prometheus and the Kubernetes autoscaling infrastructure by exposing application-level metrics through the Kubernetes Metrics APIs.

Internal and External Prometheus Deployments
The Prometheus server can be deployed according to different architectural solutions.

- Inside the Kubernetes cluster, which represents the most common deployment model
- Outside the Kubernetes cluster as an external monitoring service, provided that it is reachable through the network

Regardless of the deployment model, the Prometheus Adapter can retrieve metrics from both internal and external Prometheus servers, as long as the configured endpoint is accessible.

11.6 Chapter Summary

This chapter presented the complete theoretical workflow required to implement Kubernetes autoscaling based on application-level metrics collected by Prometheus, without relying on KEDA. Starting from the generation of application-specific metrics, the entire monitoring pipeline was analysed, including the role of the NGINX Prometheus Exporter, the Prometheus server, the Prometheus Adapter, the Kubernetes API Server, and the Horizontal Pod Autoscaler. Unlike native resource-based autoscaling, this approach enables scaling decisions based on metrics that more accurately represent the actual behavior of an application, thereby overcoming the limitations of CPU- and memory-based metrics in scenarios where resource utilization is not directly correlated with the workload. It is worth emphasizing that the Horizontal Pod Autoscaler scaling algorithm remains unchanged throughout this workflow. The only difference with respect to native autoscaling lies in the origin of the metrics, which are now derived from application-level behavior rather than infrastructure-level resource utilization. However, this solution also introduces additional architectural complexity, as the Prometheus Adapter must be deployed and configured to expose Prometheus metrics through the Kubernetes Metrics APIs before they can be consumed by the Horizontal Pod Autoscaler. In the next chapter, the same autoscaling scenario will be revisited by introducing KEDA. As will be shown, KEDA considerably simplifies the overall architecture by eliminating the need for the Prometheus Adapter while preserving the same Horizontal Pod Autoscaler decision-making process based on application-level metrics.

Chapter 12

Experimental Evaluation: Prometheus-Based Autoscaling without KEDA

12.1 Experimental Objective

In this chapter, we implement a complete autoscaling workflow in Kubernetes based on application-level metrics collected through Prometheus, without relying on KEDA. The objective is to demonstrate how scaling decisions can be driven by the actual behavior of an application through a custom metric exposed by an NGINX instance. To evaluate the application workload, the autoscaling mechanism relies on the following PromQL query:

```
sum(rate(nginx_http_requests_total[30s]))
```

The metric `nginx_http_requests_total` is a cumulative counter continuously maintained by NGINX and exposed through the NGINX Prometheus Exporter. Rather than representing the instantaneous number of active HTTP requests, this counter monotonically increases over time by incrementing each time an HTTP request is processed by a Pod. To derive a meaningful workload indicator suitable for autoscaling purposes, the Prometheus `rate()` function is applied over a 30-second observation window. For each active Pod replica, Prometheus evaluates the variation of the cumulative counter between the

beginning and the end of the observation interval according to

$$\text{rate}(x[30s]) \approx \frac{x(t_0 + 30s) - x(t_0)}{30}$$

where

- $x(t_0)$ represents the counter value at the beginning of the observation interval;
- $x(t_0 + 30s)$ represents the counter value at the end of the observation interval.

The resulting value corresponds to the average number of HTTP requests processed per second by a single Pod during the considered time window. Since Prometheus evaluates the metric independently for each replica, the `rate()` function generates one output time series for every active Pod in the cluster. Subsequently, the `sum()` aggregation operator combines all per-Pod request rates into a single scalar value:

```
sum(rate(nginx_http_requests_total[30s]))
```

This aggregation represents the total HTTP request rate collectively handled by the entire NGINX Deployment. Unlike the intermediate results produced by the `rate()` function, which generate one value for each Pod replica, the final `sum()` operation returns a single real-valued metric representing the overall workload of the application at cluster level. This aggregated metric is collected through the NGINX Prometheus Exporter and made available to the Horizontal Pod Autoscaler (HPA) via the Prometheus Adapter.

12.2 Prerequisites

Before starting the experimental setup, the following requirements must be satisfied:

- A running Kubernetes cluster
- `kubectl` correctly configured to access the cluster
- Helm installed (recommended for deploying the monitoring components)

KEDA Isolation Requirement
For the correct execution of this experiment, KEDA must **not** be installed in the Kubernetes cluster. The reason for this requirement is that KEDA introduces its own autoscaling control loop, automatically managing `ScaledObject` resources and generating dedicated Horizontal Pod Autoscaler (HPA) resources. If KEDA is already installed, it intercepts the autoscaling logic and may override or interfere with manually configured autoscaling mechanisms based on the Prometheus Adapter. More specifically, if KEDA remains installed during the Prometheus Adapter-based experiment, the following issues may arise:

- Automatic creation and reconciliation of KEDA-managed HPA resources
- Conflicts between metric sources used by KEDA scalers and the Prometheus Adapter
- Unexpected scaling behavior caused by two independent autoscaling control loops acting on the same workload
- Inability to correctly evaluate the native

  $$\text{HPA} \rightarrow \text{Prometheus Adapter} \rightarrow \text{Prometheus}$$

  workflow, since KEDA replaces this on-demand mechanism with its polling-based architecture.

For these reasons, KEDA must be completely removed from the cluster before deploying the Prometheus Adapter-based solution, ensuring that autoscaling decisions are performed exclusively by the native Kubernetes Horizontal Pod Autoscaler in combination with the Prometheus Adapter.

12.3 Prometheus Installation

The easiest and most robust way to install Prometheus in Kubernetes is through Helm using the `kube-prometheus-stack` chart.

```
helm repo add prometheus-community https://prometheus-community.
github.io/helm-charts
helm repo update
```

```
helm install monitoring prometheus-community/kube-prometheus-stack
\
  --namespace monitoring --create-namespace
```

Installed Components
The Helm chart automatically deploys the following monitoring components:

- Prometheus Server (metrics collection)
- Alertmanager
- Grafana
- Node Exporter
- Kubernetes monitoring components.

All these components are deployed as Pods within the `monitoring` namespace.

Installation Verification
The successful deployment of the monitoring stack can be verified through the following command:

```
kubectl get pods -n monitoring
```

A successful installation should display Pods similar to:

- `Prometheus-...`
- `Grafana-...`
- `Alertmanager-...`

12.4 Prometheus Stack Components

When the monitoring stack is installed through the `kube-prometheus-stack` Helm chart, several components are automatically deployed as independent Pods inside the Kubernetes cluster (typically within the `monitoring` namespace). Each component plays a specific role in the collection, processing, storage, and visualization of monitoring data.

1. Prometheus Server
The Prometheus Server represents the core component of the monitoring infrastructure. Its primary responsibilities include:

- Scraping metrics from configured targets (e.g., exporters, Pods, and Services)
- Storing metrics as time-series data
- Executing PromQL queries
- Providing metrics to external components such as the Prometheus Adapter and Grafana

How It Works

- Periodically sends HTTP requests to monitored targets (pull model).
- Collects metrics exposed through the `/metrics` endpoint.
- Stores the collected information inside its local time-series database.

Role in This Experiment

- Scrapes metrics exposed by the NGINX Prometheus Exporter
- Stores the application-level metrics (e.g., `nginx_connections_waiting`)
- Serves these metrics to the Prometheus Adapter

2. Alertmanager
Alertmanager is responsible for handling alerting logic based on the alerting rules defined within Prometheus.

Main Responsibilities

- Receives alerts generated by Prometheus
- Deduplicates and groups alerts
- Sends notifications through external channels (e.g., email or Slack)

Role in This Experiment

- It is not directly involved in the autoscaling workflow

- Nevertheless, it represents an essential component in production environments, enabling reactive monitoring (e.g., alerts generated when the number of active connections exceeds a predefined threshold)

3. Grafana
Grafana provides a graphical interface for visualizing metrics collected by Prometheus.

Main Responsibilities

- Queries the Prometheus server
- Displays customizable dashboards
- Enables real-time monitoring of infrastructure and application metrics

Role in This Experiment

- Optional, but useful for visualizing application-level metrics such as `nginx_connections_waiting`.
- Allows the observation of workload trends during the autoscaling experiments.

4. Node Exporter
The Node Exporter collects infrastructure-level metrics from Kubernetes worker nodes.

Collected Metrics

- CPU utilization
- Memory utilization
- Disk I/O
- Network statistics

Deployment Model

- Typically deployed as a DaemonSet
- One Pod runs on each Kubernetes node

Role in This Experiment

- Not directly involved in the autoscaling process
- Provides infrastructure-level observability, complementing the application-level metrics collected from NGINX

5. kube-state-metrics
The `kube-state-metrics` component exposes metrics describing the state of Kubernetes objects.

Observed Resources

- Deployments.
- Pods.
- ReplicaSets.
- Horizontal Pod Autoscalers.

Example Metrics

- Current number of replicas
- Pod lifecycle state (Running, Pending, etc.)

Role in This Experiment

- Supports the observation of scaling behavior.
- Allows tracking replica count variations during the experiments.

6. Prometheus Operator
The Prometheus Operator acts as a Kubernetes controller responsible for managing Prometheus instances.

Main Responsibilities

- Deploys and configures Prometheus
- Continuously watches Kubernetes custom resources such as:
  - `ServiceMonitor`
  - `PodMonitor`
- Dynamically updates the Prometheus scraping configuration

Instead of manually modifying Prometheus configuration files, users simply define Kubernetes custom resources, while the Prometheus Operator automatically updates the Prometheus configuration.

Role in This Experiment

- Enables automatic discovery of the NGINX Prometheus Exporter through a `ServiceMonitor` resource.

7. ServiceMonitor (Custom Resource Definition)
The `ServiceMonitor` custom resource specifies which Kubernetes Services should be scraped by the Prometheus server.

Configuration Parameters

- Target Services (selected through Kubernetes labels)
- Scraping interval
- Target port and endpoint

Role in This Experiment
The following selector is used:

```
selector:
  matchLabels:
    app: nginx
```

This configuration instructs Prometheus to automatically discover and periodically scrape the NGINX Prometheus Exporter. Although all these components are installed simultaneously through the Helm chart, only a subset of them actively participates in the autoscaling workflow developed in this chapter. In particular, the Prometheus Server, the Prometheus Operator, and the ServiceMonitor are directly involved in metrics collection, whereas the Prometheus Adapter, introduced later, exposes these metrics to the Kubernetes autoscaling infrastructure.

12.5 NGINX and Exporter Deployment

We now deploy an NGINX application instrumented with application-level metrics. The deployment consists of an NGINX web server configured to expose internal runtime statistics through the `/stub_status` endpoint, together with an NGINX Prometheus Exporter responsible for translating these statistics into Prometheus-compatible metrics.

12.5.1 NGINX Configuration with `/stub_status`

The NGINX configuration is provided through a Kubernetes `ConfigMap`, allowing the application configuration to be decoupled from the container image and enabling dynamic reconfiguration without rebuilding the application. The ConfigMap resource is shown below.

configmap.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: nginx-config
data:
  nginx.conf: |
    events {}
    http {
      server {
        listen 80;

        location / {
          return 200 "Hello from NGINX\n";
```

```
14        }
15
16        location /stub_status {
17          stub_status;
18          allow 127.0.0.1;
19          deny all;
20        }
21      }
22    }
```

Listing 12.1: ConfigMap manifest containing the customized NGINX configuration.

The configuration file (`nginx.conf`) is mounted inside the container at:

```
1 /etc/nginx/nginx.conf
```

replacing the default NGINX configuration.

Configuration Description
The configuration defines the following directives:

- `listen 80` configures the web server to accept incoming HTTP connections on port 80, which represents the standard port for HTTP traffic inside the container
- The `location /` block defines a default endpoint returning an HTTP 200 response containing a simple static message. This endpoint is mainly used for testing purposes and to generate traffic during the autoscaling experiments
- The `location /stub_status` block enables the built-in NGINX `stub_status` module, exposing internal runtime statistics such as active connections and request-handling counters. Although these statistics are not directly compatible with Prometheus, they provide the raw information required by the NGINX Prometheus Exporter

Access Restrictions
Access to the `/stub_status` endpoint is restricted through the following directives:

- `allow 127.0.0.1` permits only local requests

- `deny all` blocks every external request

This configuration guarantees that only processes running inside the same Pod can access the metrics endpoint. Since the NGINX Prometheus Exporter is deployed as a sidecar container within the same Pod, both containers share the same network namespace and the exporter can therefore access the endpoint through `localhost`. This design follows a common Kubernetes pattern in which application metrics are exposed only locally and subsequently collected by a dedicated exporter, improving both security and modularity.

12.5.2 Exporter Access to the NGINX Metrics Endpoint

The NGINX Prometheus Exporter retrieves application statistics through the following URI:

```
http://127.0.0.1/stub\_status
```

Although the port number is not explicitly specified, it is implicitly assumed to be port 80, which represents the default port for HTTP traffic. This allows the exporter configuration to remain concise while preserving the expected communication behavior between the two containers. This behavior follows the standard URL convention whereby the absence of an explicit port number causes the protocol default port to be used automatically. Since the NGINX server is configured with the directive

```
1 listen 80
```

the exporter successfully reaches the metrics endpoint without requiring any explicit port specification. If a non-standard port were used instead, it would have to be explicitly included in the URI.

12.5.3 NGINX Location Matching Behavior

The NGINX configuration defines two location blocks:

- `location /stub_status`, which exposes internal monitoring statistics
- `location /`, which provides the default HTTP response

NGINX evaluates location blocks according to prefix matching, always selecting the most specific matching rule. Consequently:

- requests directed to `/stub_status` are handled by the dedicated monitoring endpoint
- all other requests (e.g., `/`, `/test`, or `/api`) are served by the default `location /` block

This behavior ensures that application traffic and monitoring traffic remain logically separated while sharing the same NGINX server instance.

12.5.4 Deployment with NGINX and Exporter Sidecar

The application is deployed as a Kubernetes `Deployment` consisting of multiple replicas of a Pod containing two containers:

- The main NGINX container, responsible for serving HTTP traffic
- A sidecar container running the NGINX Prometheus Exporter, responsible for exposing application metrics in Prometheus format

The corresponding Deployment manifest is reported below:

deployment.yaml

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: nginx-deployment
5  spec:
6    replicas: 3
7    selector:
8      matchLabels:
9        app: nginx
10   template:
11     metadata:
12       labels:
13         app: nginx
14     spec:
15       containers:
```

```
16
17      - name: nginx
18        image: nginx:latest
19        volumeMounts:
20          - name: config
21            mountPath: /etc/nginx/nginx.conf
22            subPath: nginx.conf
23        ports:
24          - containerPort: 80
25
26      - name: nginx-exporter
27        image: nginx/nginx-prometheus-exporter:latest
28        args:
29          - -nginx.scrape-uri=http://127.0.0.1/stub_status
30        ports:
31          - containerPort: 9113
32
33      volumes:
34        - name: config
35          configMap:
36            name: nginx-config
```

Listing 12.2: Deployment manifest containing the NGINX application and the Prometheus Exporter sidecar.

Deployment Description

The NGINX container is configured through the ConfigMap mounted at

```
1 /etc/nginx/nginx.conf
```

allowing the server configuration to be modified dynamically while enabling the `/stub_status` endpoint for internal metrics exposure. The exporter container is configured to scrape metrics from the local endpoint

```
http://127.0.0.1/stub_status
```

Since both containers execute within the same Pod, they share the same network namespace, enabling communication through `localhost` without exposing the metrics endpoint externally. After collecting the internal NGINX statistics, the exporter exposes the corresponding Prometheus metrics through port

9113, making them available for scraping by the Prometheus server. This architecture follows the *sidecar pattern*, in which auxiliary functionality (metric collection) is decoupled from the primary application logic, improving modularity, security, and maintainability.

12.5.5 Service for Prometheus Scraping and Load Generation

A Kubernetes Service is created to expose both the NGINX application and the NGINX Prometheus Exporter within the cluster. The Service manifest is shown below:

service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
  labels:
    app: nginx
spec:
  selector:
    app: nginx
  ports:
    - name: http
      port: 80
      targetPort: 80
    - name: metrics
      port: 9113
      targetPort: 9113
```

Listing 12.3: ClusterIP Service manifest exposing the NGINX application and the Prometheus Exporter.

Since the type field is omitted, Kubernetes creates the Service as a ClusterIP resource by default. The Service exposes the NGINX Prometheus Exporter to the other components within the cluster, enabling metric collection by the Prometheus server. The Service selects the target Pods through the selector field, matching all Pods labeled with

```
app: nginx
```

thereby establishing the logical association between the Service and the Pods running both the NGINX application and the exporter sidecar.

Important Distinction
It is important to distinguish between the following fields:

- `metadata.labels` identifies the Service itself and is typically used by monitoring components such as the `ServiceMonitor`
- `spec.selector` identifies the target Pods associated with the Service

Port Mapping
The Service defines two independent port mappings:

- **Exporter**
  - `targetPort:9113`: port exposed by the exporter container inside the Pod
  - `port: 9113`: corresponding port exposed by the Service inside the cluster
- **NGINX Application**
  - `targetPort:80`: port exposed by the NGINX container inside the Pod
  - `port:80`: corresponding port exposed by the Service

This configuration allows internal components, such as the Prometheus server, to access the exporter through the stable endpoint

```
1 nginx-service:9113
```

independently of Pod IP changes. Furthermore, the NGINX application is also exposed through

```
1 nginx-service:80
```

allowing it to be reached by the internal BusyBox Pod used as a load generator during the autoscaling experiments (this aspect will be discussed later in the chapter). For simplicity, the same port number is adopted both for the Service and the target container. Nevertheless, Kubernetes allows these values to differ, providing flexible port remapping whenever required. Within this architecture, only the exporter endpoint is scraped by Prometheus, ensuring that monitoring traffic remains separated from the normal application traffic.

12.5.6 Service Networking and Pod Discovery

Kubernetes abstracts network communication through Services, which provide stable virtual IP addresses (ClusterIPs) and DNS names for accessing dynamic sets of Pods. When a Service is associated with multiple Pod replicas, it performs load balancing by forwarding each incoming request to a single Pod instance. Consequently, a standard Kubernetes Service does not broadcast requests to every replica simultaneously. However, monitoring follows a different mechanism. Through the integration between the Prometheus Operator and the `ServiceMonitor` resource, Prometheus performs service discovery and retrieves the list of all Pod endpoints associated with the Service. As a consequence, Prometheus directly scrapes every Pod individually using its corresponding Pod IP address, ensuring that metrics are collected from every replica rather than from a single instance selected by the Service. This distinction is particularly important. While the Service provides a stable abstraction layer for client communication, the actual metric collection is performed at the Pod level, enabling complete and accurate monitoring of distributed applications.

Key Observation
Although client requests are distributed by the Kubernetes Service to a single Pod at a time, Prometheus bypasses this load-balancing mechanism during metrics collection by discovering and scraping every Pod endpoint individually. This ensures that autoscaling decisions are based on metrics collected from the entire Deployment rather than from a single replica.

12.6 Prometheus Scraping Configuration

Once the NGINX application and the Prometheus Exporter have been deployed, Prometheus must be instructed to periodically scrape the exporter metrics. This is achieved by creating a `ServiceMonitor`, a Kubernetes Custom Resource Definition (CRD) automatically installed by the `kube-prometheus-stack` Helm chart. The corresponding manifest is reported below:

servicemonitor.yaml

```
1 apiVersion: monitoring.coreos.com/v1
2 kind: ServiceMonitor
3 metadata:
4   name: nginx-monitor
```

```
5   labels:
6     release: monitoring
7 spec:
8   selector:
9     matchLabels:
10      app: nginx
11  endpoints:
12    - port: metrics
13      path: /metrics
14      interval: 15s
```

Listing 12.4: ServiceMonitor manifest used to configure Prometheus metric scraping.

12.6.1 ServiceMonitor Configuration

The `ServiceMonitor` instructs the Prometheus Operator to automatically discover the Kubernetes Service associated with the NGINX exporter and periodically scrape the metrics exposed through its `/metrics` endpoint.

Configuration Description

- `selector.matchLabels` identifies the Kubernetes Service exposing the exporter by matching the label `app:nginx`
- `port:metrics` specifies the Service port that Prometheus must use for metric collection
- `path:/metrics` identifies the HTTP endpoint exposing Prometheus metrics
- `interval:15s` configures the scraping interval, causing Prometheus to collect metrics every 15 seconds

12.6.2 Target Discovery Issue and Resolution

During the integration phase, an additional issue emerged concerning Prometheus target discovery. Although the `ServiceMonitor` resource was correctly deployed, the NGINX exporter did not appear among the active scraping targets configured in Prometheus. The issue was diagnosed by inspecting the Prometheus custom resource through the following command:

```
kubectl get prometheus -n monitoring -o yaml
```

The retrieved configuration contained the following selector:

```
serviceMonitorSelector:
  matchLabels:
    release: monitoring
```

This configuration indicates that Prometheus discovers only those `ServiceMonitor` resources containing the label

```
release: monitoring
```

Consequently, the ServiceMonitor manifest had to be updated by explicitly adding the required label within the metadata section.

```
metadata:
  name: nginx-monitor
  namespace: monitoring
  labels:
    release: monitoring
```

Without this label, the `ServiceMonitor` resource was ignored by Prometheus and no scraping target associated with the NGINX exporter was registered. After applying the modification, the exporter correctly appeared among the active Prometheus targets and metric collection started successfully.

Practical Observation
This issue highlights the importance of inspecting the Prometheus custom resource whenever newly created ServiceMonitor resources are not discovered. In practice, the label selector configured by the Prometheus Operator determines which ServiceMonitor resources are considered during the discovery process.

12.6.3 Port Name Resolution

Another important aspect concerns the `port` field defined within the `endpoints` section of the `ServiceMonitor` resource. Contrary to what might be expected, this field does not represent a numeric TCP port. Instead, it refers to the logical name of the port defined in the corresponding Kubernetes Service. This abstraction enables Prometheus to unambiguously select the appropriate endpoint even when a Service exposes multiple ports. By matching the port

name rather than the numeric port value, the `ServiceMonitor` guarantees that Prometheus scrapes the correct endpoint, namely the exporter metrics endpoint, ensuring proper metric collection from all monitored Pods.

12.6.4 Deployment of Monitoring Resources

After defining the Kubernetes manifests for the NGINX application, the exporter, and the monitoring resources, all components can be deployed using the standard `kubectl apply` command.

```
kubectl apply -f configmap.yaml
kubectl apply -f deployment.yaml
kubectl apply -f service.yaml
kubectl apply -f servicemonitor.yaml
```

Alternatively, all manifests contained in the current directory can be applied simultaneously through

```
kubectl apply -f .
```

12.6.5 Namespace Consistency Considerations

An important aspect identified during the experimental validation concerns namespace consistency among the deployed resources. In particular, the following resources should reside within the same Kubernetes namespace:

- NGINX Deployment
- NGINX Service
- NGINX Prometheus Exporter
- ServiceMonitor
- Horizontal Pod Autoscaler

The selected namespace may either be the default namespace or the `monitoring` namespace hosting the Prometheus stack components. The important requirement is not the specific namespace itself, but rather the consistency of namespace placement among all monitored resources. This behavior is related to the Prometheus discovery mechanism implemented through

`ServiceMonitor` resources. The ServiceMonitor selects Kubernetes Services by means of label selectors within its visibility scope. If the monitored Service resides in a different namespace that is not included in the Prometheus discovery configuration, Prometheus may fail to detect the exporter endpoint, resulting in missing targets and null metric values. Placing all related resources within the same namespace greatly simplifies service discovery and avoids cross-namespace selection issues during the scraping phase. In the implemented configuration, all resources were eventually deployed within the default namespace, ensuring successful target discovery and correct metric collection by Prometheus.

12.7 Verification of Metrics Collection in Prometheus

After deploying the monitoring infrastructure, it is important to verify that metrics are correctly exposed by the exporter, successfully collected by Prometheus, and that all scraping targets are properly registered before integrating the monitoring pipeline with the autoscaling mechanism.

12.7.1 Verification of the Exporter Endpoint

As a first debugging step, the exporter endpoint can be directly inspected by forwarding the exporter port from one of the running Pods to the local machine.

```
kubectl port-forward pod/<nginx-pod> 9113:9113
```

This command forwards port `9113` of the exporter container running inside the selected Pod to the local machine, allowing direct access to the exporter endpoint through

```
http://localhost:9113/metrics
```

which exposes all metrics in Prometheus format.

Inspecting Exporter Metrics
The exported metrics can then be inspected through:

```
curl http://localhost:9113/metrics
```

Accessing this endpoint displays all metrics generated by the exporter. This verification step confirms that the exporter is correctly scraping the NGINX `/stub_status` endpoint and successfully transforming the collected statistics into Prometheus-compatible metrics.

12.7.2 Verification of Registered Prometheus Targets

After deploying the monitoring infrastructure, it is equally important to verify that Prometheus successfully discovers and scrapes the configured exporter targets. This verification can be performed by exposing the Prometheus Server locally through Kubernetes port forwarding.

Inspecting the Prometheus Service
The services deployed inside the monitoring namespace can first be listed through:

```
kubectl get svc -n monitoring
```

Among the returned services, the Prometheus Server must be identified. In the adopted experimental configuration, the corresponding service name is:

```
1 monitoring-kube-prometheus-prometheus
```

This Service provides the internal Kubernetes DNS entry through which cluster components communicate with the Prometheus Server.

Port Forwarding the Prometheus Server
The Prometheus Web UI and HTTP API can then be exposed locally through:

```
kubectl port-forward \
svc/monitoring-kube-prometheus-prometheus \
9090:9090 -n monitoring
```

This command forwards the Prometheus Service running inside the Kubernetes cluster to the local machine, allowing the Prometheus Web UI to be accessed through

```
http://localhost:9090
```

even though the service is internally exposed as a ClusterIP and is therefore not directly reachable from outside the cluster.

Role of the ClusterIP Service

In Kubernetes, the Prometheus Server is typically exposed through a Service of type `ClusterIP`, automatically created during the installation of the monitoring stack. The Service provides a stable virtual IP address and an associated DNS name within the cluster. This design allows internal components, such as the Prometheus Adapter, to communicate reliably with the Prometheus Server without depending on the dynamic IP addresses assigned to individual Pods. In particular, the Prometheus Adapter, running as a Pod inside the cluster, performs HTTP requests against the Prometheus API using the Service DNS name (e.g., `monitoring-kube-prometheus-prometheus`). Kubernetes resolves this DNS name to the corresponding ClusterIP, which transparently forwards the requests to the underlying Prometheus Pod. This approach guarantees stable communication, decouples service discovery from the Pod lifecycle, and fully complies with Kubernetes networking principles, where Services provide stable abstractions for dynamically managed workloads.

12.7.3 Verification Through the Prometheus Targets API

Once port forwarding has been established, the registered scraping targets can be inspected through the Prometheus Targets API.

```
curl -s http://localhost:9090/api/v1/targets | \
jq '.data.activeTargets[] |
{job: .labels.job,
 health: .health,
 url: .scrapeUrl}'
```

The command returns a simplified view of all active targets currently scraped by Prometheus, including:

- The logical job name
- The health status of the target
- The scraping endpoint URL

A correctly configured NGINX exporter should appear with

```
1 health:"up"
```

indicating that Prometheus successfully reaches the exporter endpoint and collects metrics correctly.

12.7.4 Accessing the Prometheus Web UI

Once the Prometheus Server has been exposed locally through port forwarding, its graphical interface becomes available at:

```
http://localhost:9090
```

The interface can be opened using any web browser available on the host system or virtual machine, for example:

```
firefox http://localhost:9090
```

The Prometheus Web UI provides several useful functionalities, including:

- Interactive execution of PromQL queries
- Visualization of collected metrics
- Inspection of active scraping targets
- Verification of exporter status
- Real-time debugging of autoscaling metrics

In particular, the **Status** → **Targets** section provides a graphical overview of all registered scraping endpoints together with their current health status.

12.7.5 Verification of Active Targets

The same verification can also be performed graphically through the Prometheus Web UI by opening the

Status → **Targets**

page. Each Pod replica associated with the monitored Service should appear as an individual target with status `UP`, confirming that Prometheus has successfully discovered the Service through the ServiceMonitor resource and is actively collecting metrics from every Pod replica. Verifying that every Pod replica appears as an independent target is particularly important because Prometheus collects metrics at the Pod level rather than through the load-balanced Service endpoint. This confirms that the monitoring pipeline is correctly configured and that the collected metrics accurately represent the behavior of the entire Deployment.

12.7.6 Querying and Verifying Autoscaling Metrics

After exposing the Prometheus Server locally through port forwarding, the collected metrics can be verified either from the terminal using the Prometheus HTTP API or interactively through the Prometheus Web UI.

Querying Raw Metrics

The first verification consists of querying the HTTP request rate independently for each active NGINX Pod through the following PromQL expression:

```
rate(nginx_http_requests_total[30s])
```

The metric `nginx_http_requests_total` is a cumulative counter continuously incremented whenever an HTTP request is processed by an NGINX Pod. Since cumulative counters monotonically increase over time, the `rate()` function is used to derive a meaningful workload indicator suitable for autoscaling purposes. For each active Pod replica, Prometheus evaluates the variation of the cumulative counter over the specified 30-second observation window according to

$$\text{rate}(x[30s]) \approx \frac{x(t_0 + 30s) - x(t_0)}{30}$$

where:

- $x(t_0)$ represents the counter value at the beginning of the observation window
- $x(t_0 + 30s)$ represents the counter value at the end of the observation window

The resulting value represents the average number of HTTP requests processed per second by each individual Pod during the considered time interval. Since the query is evaluated independently for every replica, the output consists of one time series for each active Pod, confirming that metrics are collected separately from every exporter endpoint.

Query Execution

The query can be executed directly through the Prometheus HTTP API using:

```
curl -s -G http://localhost:9090/api/v1/query \
--data-urlencode \
'query=rate(nginx_http_requests_total[30s])'
```

For improved readability, the returned JSON response can be formatted using `jq`:

```
curl -s -G http://localhost:9090/api/v1/query \
--data-urlencode \
'query=rate(nginx_http_requests_total[30s])' | jq
```

The returned response contains:

- The timestamp corresponding to the evaluation instant of the PromQL query
- The associated metric value, expressed as the average number of HTTP requests processed per second

A non-empty result confirms that:

- The exporter is correctly exposing metrics
- Prometheus is successfully scraping the exporter endpoint
- The autoscaling metric is available for consumption by the autoscaler

Querying the Aggregated Metric

While the previous query returns one value for each active Pod, autoscaling typically relies on a single metric representing the overall workload handled by the entire Deployment. This aggregated metric can be obtained through the following PromQL expression:

```
sum(rate(nginx_http_requests_total[30s]))
```

The `sum()` aggregation operator combines all the per-Pod request rates generated by the `rate()` function into a single scalar value representing the overall HTTP request rate handled collectively by the entire NGINX Deployment. Unlike the previous query, which returns one time series for each replica, this aggregation produces a single real-valued metric representing the total average number of HTTP requests processed per second across all active Pods. This aggregated metric is particularly relevant for autoscaling because it reproduces the workload indicator that will later be consumed by the autoscaler to evaluate the current load of the application.

Query Execution

The aggregated metric can be queried through:

```
curl -s -G http://localhost:9090/api/v1/query \
--data-urlencode \
'query=sum(rate(nginx_http_requests_total[30s]))'
```

The returned JSON output can also be formatted through:

```
curl -s -G http://localhost:9090/api/v1/query \
--data-urlencode \
'query=sum(rate(nginx_http_requests_total[30s]))' | jq
```

Key Observation
The transition from the per-Pod query to the aggregated query mirrors the evolution from monitoring individual application instances to evaluating the workload of the Deployment as a whole, which is the information ultimately required by the autoscaler.

Instant Query Semantics

When executing aggregation queries such as

```
sum(rate(nginx_http_requests_total[30s]))
```

Prometheus computes the result directly through its native PromQL query engine. Within the context of an *instant query*, corresponding to the default behavior adopted by Kubernetes autoscaling mechanisms, the returned value represents the most recent evaluation of the metric at the current point in time. Although the result is associated with the current evaluation timestamp, the metric itself is computed using samples collected during the preceding 30-second observation window. Consequently, the returned scalar represents the average HTTP request rate observed during the interval

$$[T - 30s,\ T]$$

where T denotes the evaluation instant associated with the query result.

Verification Through the Prometheus Web UI

The same verification process can also be performed graphically through the Prometheus Web UI. Once port forwarding is active, the interface becomes available at

```
http://localhost:9090
```

using any web browser available on the host system or virtual machine. Inside the Prometheus Web UI, the previously described PromQL expressions can be entered into the query editor and executed through the **Execute** button. The following expressions can be tested interactively:

- `nginx_http_requests_total`
- `rate(nginx_http_requests_total[30s])`
- `sum(rate(nginx_http_requests_total[30s]))`

12.8 Prometheus Adapter Installation and Configuration

Once Prometheus has been configured and verified, the next step consists of installing the Prometheus Adapter, which acts as the bridge between the Prometheus monitoring system and the Kubernetes Metrics API. Its role is to execute PromQL queries against the Prometheus Server and expose the resulting values as Kubernetes external metrics consumable by the Horizontal Pod Autoscaler.

12.8.1 Adding the Helm Repository

The Prometheus Adapter can be installed through the official Helm repository.

```
helm repo add prometheus-community https://prometheus-community.
github.io/helm-charts
helm repo update
```

These commands add the official Helm repository containing the Prometheus Adapter chart and update the local Helm repository index, ensuring that the latest version of the adapter package is available for installation.

12.8.2 Custom Adapter Configuration

Before installing the adapter, a custom `values.yaml` configuration file is created to define how Prometheus metrics are mapped into Kubernetes external metrics.

values.yaml

```
rules:
  default: false
  external:
    - seriesQuery: 'nginx_http_requests_total'
      resources:
        namespaced: false
      name:
        as: "nginx_http_requests_rate_total"
      metricsQuery: 'sum(rate(nginx_http_requests_total[30s]))'
```

Listing 12.5: Custom Prometheus Adapter configuration.

This configuration specifies how the Prometheus Adapter exposes Prometheus metrics through the Kubernetes External Metrics API.

Configuration Description

- `rules.default:false` disables the default metric mappings provided by the Helm chart, allowing the adapter configuration to be fully customized
- `external.seriesQuery` specifies how the adapter discovers the desired metric inside Prometheus. In this case, it searches for the metric `ngin x\_http\_requests\_total`
- `resources.namespaced:false` specifies that the exposed metric is cluster-wide and is therefore not associated with a specific Kubernetes namespace or Pod resource. This configuration is required because the PromQL expression

  ```
  sum(rate(nginx_http_requests_total[30s]))
  ```

  returns a single globally aggregated scalar value without namespace or Pod labels. Consequently, the metric is exposed through the `external .metrics.k8s.io` API as a non-namespaced external metric consumable by the Horizontal Pod Autoscaler.

- `name.as` defines the logical metric name exposed to Kubernetes
- `metricsQuery` is the most important configuration field. It specifies the PromQL query executed by the adapter. In this implementation, it computes the sum of the HTTP request rate across all active Pods, producing a single aggregated metric suitable for autoscaling

Key Observation
Unlike Prometheus, which stores all collected metrics, the Prometheus Adapter does not maintain its own metric database. Instead, it simply executes the configured PromQL query on demand whenever a request is received through the Kubernetes External Metrics API, returning the resulting value to the requesting component.

12.8.3 Installing the Prometheus Adapter

The Prometheus Adapter can then be installed using Helm together with the custom configuration file.

```
helm install prometheus-adapter \
prometheus-community/prometheus-adapter \
-f values.yaml
```

This command installs the Prometheus Adapter inside the Kubernetes cluster using the Helm chart together with the custom configuration defined in the `values.yaml` file. During the installation:

- The adapter is deployed as a Pod in the default namespace
- The custom configuration contained in `values.yaml` is automatically applied
- The adapter registers itself with the Kubernetes API Server through the API Aggregation Layer, exposing metrics under the `external.metrics .k8s.io` API group

12.8.4 Verification of the Adapter Deployment

The successful deployment of the Prometheus Adapter can be verified through:

```
kubectl get pods
```

A successful installation is indicated by the Prometheus Adapter Pod being in the `Running` state.

12.8.5 Verification of the Prometheus Server Service

Before validating the adapter configuration, it is necessary to identify the Kubernetes Service exposing the Prometheus Server.

```
kubectl get svc -n monitoring
```

The output should contain the Service exposing the Prometheus Server, for example:

```
1 monitoring-kube-prometheus-prometheus
```

This Service is particularly important because the Prometheus Adapter connects to the Prometheus Server through the `--prometheus-url` startup argument defined inside its Deployment manifest.

12.8.6 Verification of the Adapter Deployment Configuration

The adapter Deployment can be inspected through:

```
kubectl get deployment prometheus-adapter \
-n default -o yaml
```

Inside the container startup arguments, the following parameter should be present: `--prometheus-url=http://monitoring-kube-prometheus-prometheus.monitoring:9090`. This argument specifies the internal DNS endpoint used by the Prometheus Adapter to communicate with the Prometheus Server.

12.8.7 Correcting an Incorrect Prometheus URL

During the experimental implementation, it was observed that the default value automatically configured by Helm did not match the actual Service name exposing the Prometheus Server. Consequently, the Prometheus Adapter was

unable to retrieve metrics correctly. To solve this issue, the Deployment was manually patched through:

```
kubectl patch deployment prometheus-adapter -n default \
  --type='json' \
  -p='[
    {
      "op": "replace",
      "path": "/spec/template/spec/containers/0/args/3",
      "value": "--prometheus-url=http://monitoring-kube-prometheus-
prometheus.monitoring:9090"
    }
  ]'
```

Listing 12.6: Patching the Prometheus Adapter Deployment.

The path:

```
/spec/template/spec/containers/0/args/3
```

identifies the fourth startup argument of the container, considering that argument indexing starts from zero.

Key Observation
Unlike Prometheus, which stores all collected metrics, the Prometheus Adapter does not maintain its own metric database. Instead, it simply executes the configured PromQL query on demand whenever a request is received through the Kubernetes External Metrics API, returning the resulting value to the requesting component.

12.9 Verification of the External Metrics API

12.9.1 Verification of the API Registration

Once the Prometheus Adapter has been deployed, the first verification consists of checking that it has been correctly registered within the Kubernetes API Server through the API Aggregation Layer.

```
kubectl get apiservice | grep metrics
```

This command verifies whether the `external.metrics.k8s.io` API group has been successfully registered by the Prometheus Adapter. A successful configuration should return an entry similar to:

```
v1beta1.external.metrics.k8s.io
```

confirming that external metrics are available and can be consumed by Kubernetes components such as the Horizontal Pod Autoscaler.

12.9.2 Verification of Available External Metrics

The external metrics currently exposed by the Prometheus Adapter can be inspected through:

```
kubectl get --raw "/apis/external.metrics.k8s.io/v1beta1" | jq
```

This command queries the Kubernetes API Server to retrieve all the external metrics currently exposed by the Prometheus Adapter. In particular, it allows verifying that:

- The Prometheus Adapter has been correctly registered within the Kubernetes API Server
- The metric generated by the PromQL query

  ```
  sum(rate(nginx_http_requests_total[30s]))
  ```

  is available for consumption by the Horizontal Pod Autoscaler under the logical name:

  ```
  nginx_http_requests_rate_total
  ```

- The JSON output is formatted through the `jq` utility for improved readability.

12.9.3 Validation of the Exposed External Metric

The metric can then be queried directly through the Kubernetes Metrics API by executing:

```
kubectl get --raw \
"/apis/external.metrics.k8s.io/v1beta1/namespaces/default/
nginx_http_requests_rate_total" \
| jq
```

This request reproduces the same workflow that will later be executed automatically by the Horizontal Pod Autoscaler once autoscaling is enabled. For this reason, it represents an important debugging step, allowing verification that:

- The Prometheus Adapter is correctly registered within the Kubernetes API Server
- The Adapter successfully communicates with the Prometheus Server

12.9.4 Pull-Based Behaviour of the Prometheus Adapter

Unlike the Prometheus Server, which continuously scrapes exporters at fixed time intervals, the Prometheus Adapter does not retrieve metrics periodically. Instead, it follows a pull-based model and executes the configured PromQL query only when a request is received through the Kubernetes External Metrics API. Within the complete autoscaling workflow, the sequence of operations is therefore the following:

1. The Horizontal Pod Autoscaler periodically queries the Kubernetes External Metrics API (every 15 seconds by default)
2. The Kubernetes API Server forwards the request to the Prometheus Adapter
3. The Adapter dynamically executes the configured PromQL query against the Prometheus Server
4. The resulting metric value is returned to the HPA, which uses it to evaluate whether a scaling action is required

Consequently, manually invoking the external metrics endpoint through the previous `kubectl get --raw` command reproduces exactly the same workflow that would normally be triggered automatically by the Horizontal Pod Autoscaler.

12.9.5 Interpretation of the Observed Metric Value

During the experimental validation, an interesting behaviour was observed. Even in the absence of externally generated traffic, the metric returned by the Kubernetes Metrics API was approximately

$$0.2 = 200m.$$

This behaviour is caused by the Prometheus scraping mechanism itself. The Prometheus Server was configured with a scraping interval of 15 seconds, while the metric exposed through the Adapter is computed using the following PromQL query:

```
sum(rate(nginx_http_requests_total[30s]))
```

Every scraping operation performed by Prometheus targets the `NGINXPrometheusExporter`. Upon receiving each scrape request, the exporter internally contacts the NGINX container through the `/stub_status` endpoint in order to retrieve the latest runtime statistics. These internal HTTP requests generated by the exporter are themselves counted by the native NGINX counter `nginx_http_requests_total`. Consequently, each Pod receives approximately one internal request every 15 seconds generated indirectly by the Prometheus scraping process itself. Since the `rate()` function computes the average number of HTTP requests per second over a 30-second observation window, the resulting value for each Pod becomes approximately

$$\frac{2}{30} \approx 0.067 \text{ requests/sec.}$$

Considering that three Pod replicas are active, the global aggregation performed by the `sum()` function produces

$$0.067 \times 3 \approx 0.2 \text{ requests/sec.}$$

This value corresponds to the approximately **200m** observed through the Kubernetes External Metrics API, even though no external user-generated traffic is present.

Experimental Observation
This result highlights an important practical aspect of Prometheus-based autoscaling. Even when the application is idle, monitoring activities themselves

generate a small amount of traffic that contributes to the observed metric. Consequently, when defining autoscaling thresholds, it is important to account for this baseline value in order to avoid undesired scaling actions caused exclusively by the monitoring infrastructure.

12.10 Horizontal Pod Autoscaler Configuration

Once the Prometheus Adapter has been successfully configured and the external metric has been validated, the Horizontal Pod Autoscaler can be deployed to automatically adjust the number of application replicas according to the HTTP request rate exposed through the Kubernetes External Metrics API.

12.10.1 Horizontal Pod Autoscaler Manifest

hpa.yaml

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: nginx-hpa
  namespace: default
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: nginx-deployment
  minReplicas: 3
  maxReplicas: 5
  behavior:
    scaleDown:
      stabilizationWindowSeconds: 60
      policies:
        - type: Percent
          value: 20
          periodSeconds: 15
    scaleUp:
      stabilizationWindowSeconds: 0
```

```
22      policies:
23        - type: Percent
24          value: 100
25          periodSeconds: 15
26  metrics:
27    - type: External
28      external:
29        metric:
30          name: nginx_http_requests_rate_total
31        target:
32          type: Value
33          value: 20
```

Listing 12.7: Horizontal Pod Autoscaler manifest based on an external Prometheus metric.

12.10.2 Description of the Configuration Fields

API Version
The field

```
1 apiVersion: autoscaling/v2
```

specifies the API version used for the Horizontal Pod Autoscaler resource. Version `autoscaling/v2` enables advanced autoscaling features, including support for custom and external metrics.

Resource Type
The field

```
1 kind: HorizontalPodAutoscaler
```

defines the Kubernetes resource responsible for automatically adjusting the number of Pod replicas according to the observed metric values.

Metadata
The metadata section

```
1 name: nginx-hpa
2 namespace: default
```

identifies the Horizontal Pod Autoscaler and specifies the namespace in which it operates. This namespace must coincide with the namespace containing the target Deployment.

Scale Target
The `scaleTargetRef` section identifies the Kubernetes resource managed by the autoscaler.

```
1 apiVersion: apps/v1
2 kind: Deployment
3 name: nginx-deployment
```

The HPA modifies the `spec.replicas` field of this Deployment according to the value of the monitored metric.

Minimum and Maximum Replicas
The fields

```
1 minReplicas: 3
2 maxReplicas: 5
```

define the lower and upper bounds of the autoscaling process. The number of running Pods will never fall below three replicas nor exceed five replicas.

Scaling Behaviour
The `behavior` section customizes how aggressively the Horizontal Pod Autoscaler reacts during both scale-up and scale-down operations.

Scale-Down Behaviour
The configuration

```
1 stabilizationWindowSeconds: 60
```

defines a stabilization period of 60 seconds before applying any scale-down operation. During this interval, the HPA continuously evaluates incoming metric values while delaying replica removal in order to avoid oscillations caused by temporary workload fluctuations. By default, the stabilization window is equal to 300 seconds (5 minutes). In this implementation, it was intentionally reduced to 60 seconds in order to accelerate the experimental evaluation and make the scale-down behaviour observable within a shorter time interval while preserving the correctness of the autoscaling mechanism. The scale-down policy

```
type: Percent
value: 20
periodSeconds: 15
```

specifies that replica removal is expressed as a percentage of the currently running Pods. More precisely, the HPA is allowed to remove at most 20% of the currently running replicas every 15 seconds.

Scale-Up Behaviour
The scale-up configuration specifies

```
stabilizationWindowSeconds:0
```

which disables any stabilization delay before increasing the number of replicas. Consequently, the HPA reacts immediately whenever the observed metric exceeds the configured threshold. The corresponding scale-up policy

```
value: 100
periodSeconds: 15
```

allows the HPA to increase the number of running replicas by up to 100% during each evaluation interval, enabling aggressive reactions under sudden workload spikes. The evaluation interval is equal to 15 seconds, matching the default polling interval adopted by the Horizontal Pod Autoscaler.

Metrics Section
The autoscaling metric is defined within the `metrics` section. The field

```
type: External
```

indicates that scaling is driven by an external metric rather than by Pod resource utilization. Unlike CPU or memory metrics, the monitored value is globally aggregated and exposed through the Prometheus Adapter. The metric

```
name:nginx_http_requests_rate_total
```

corresponds to the logical metric exposed by the Prometheus Adapter through the Kubernetes API Server. It is important to emphasize that this metric name must exactly match the logical name specified through the `as:` field inside the Prometheus Adapter configuration.

Target Configuration
The field

```
type: Value
```

specifies that the current value of the external metric is directly compared against a fixed threshold. The configured threshold is

```
value: 20
```

Therefore,

- If the observed metric exceeds 20, the HPA performs a scale-out operation
- If the observed metric falls below 20, the HPA performs a scale-in operation

12.10.3 Deploying the Horizontal Pod Autoscaler

Once the manifest has been created, the HPA can be deployed through:

```
kubectl apply -f hpa.yaml
```

This command creates the Horizontal Pod Autoscaler resource inside the cluster. After deployment, Kubernetes continuously evaluates the external metric and automatically adjusts the number of replicas according to the configured scaling policies.

Key Observation
Unlike the Prometheus Adapter validation performed in the previous section, from this point onward the metric retrieval process becomes fully automatic. The HPA periodically queries the Kubernetes External Metrics API, which in turn triggers the execution of the configured PromQL query through the Prometheus Adapter.

12.10.4 Monitoring the Horizontal Pod Autoscaler

The HPA behaviour can be monitored in real time through:

```
kubectl get hpa -w
```

This command continuously displays the current metric value, the configured target threshold and the number of running replicas, making it possible to observe scaling decisions as they occur.

Typical Output

A typical output may appear as follows:

NAME	TARGETS	MINPODS	MAXPODS	REPLICAS
nginx-hpa	25/20	3	5	4

Table 12.1: Representative output of the `kubectl get hpa -w` command during the experiment.

where

- 25 represents the current external metric value
- 20 represents the configured target threshold
- The HPA consequently performs a scale-out operation

Expected Behaviour Under Idle Conditions

In the absence of externally generated traffic, the `TARGETS` column is expected to display approximately

$$200m/20$$

which corresponds to

$$0.2/20.$$

Here,

- 0.2 represents the current value of the aggregated external metric `nginx_http_requests_rate_total`;
- 20 is the target threshold configured inside the Horizontal Pod Autoscaler.

Even when the application is idle, the observed metric does not reach zero because the Prometheus scraping mechanism periodically generates internal HTTP requests through the NGINX Prometheus Exporter. These requests are included in the PromQL evaluation window and therefore contribute to the metric value used by the Horizontal Pod Autoscaler.

12.11 Experimental Load Generation and Autoscaling Validation

12.11.1 Generating the Workload

Once the Horizontal Pod Autoscaler has been deployed, an artificial workload can be generated in order to validate the autoscaling mechanism.

Creating the Load Generator

A temporary BusyBox Pod can be created through:

```
kubectl run -it load-generator --image=busybox -- /bin/sh
```

This command creates a temporary Pod inside the Kubernetes cluster using the lightweight BusyBox image and immediately opens an interactive shell inside it through the `-it` option. Unlike the application Pods, this Pod is created directly through the `kubectl run` command and is therefore not managed by a Deployment resource. Its sole purpose is to generate traffic during the experimental evaluation of the autoscaling mechanism. Since the Pod executes within the cluster network, it can directly access Kubernetes Services using their DNS names.

Generating Continuous Traffic

Inside the BusyBox container, the following command is executed:

```
while true; do
    wget -q -O- http://nginx-service
done
```

This infinite loop continuously sends HTTP requests to the NGINX Service. Since no port number is explicitly specified, the request is automatically directed to the default HTTP port (port 80). The `wget` utility is executed in quiet mode through the `-q` option so that no output is printed while continuously generating traffic. This artificial workload increases the number of HTTP requests processed by the application, thereby increasing the value of the monitored metric used by the Horizontal Pod Autoscaler.

Reconnecting to the BusyBox Pod

If the interactive shell has already been closed, it can be reopened through:

```
kubectl exec -it load-generator -- /bin/sh
```

This command opens an interactive shell inside the running BusyBox Pod, allowing additional commands to be executed without recreating the container.

12.11.2 Observing the Scale-Out Process

The evolution of the Deployment can be monitored in real time through:

```
kubectl get pods -w
```

This command continuously displays the number of running Pods, allowing observation of the scaling actions performed by the Horizontal Pod Autoscaler.

Expected behaviour

- Increasing HTTP traffic raises the external metric.
- The HPA reacts by increasing the number of running replicas.
- The expected scale-out transition is $3 \rightarrow 5$ Pods.

The observed transition from three to five replicas reflects the configuration specified in the `scaleUp` section of the Horizontal Pod Autoscaler. More specifically, the absence of a stabilization window

```
1 stabilizationWindowSeconds: 0
```

allows the autoscaler to react immediately whenever the observed metric exceeds the configured threshold. Furthermore, the percentage-based policy

```
1 value: 100
```

permits the HPA to increase the number of replicas by up to 100% during each evaluation interval. Consequently, once the workload exceeds the configured threshold, the HPA can rapidly double the number of running replicas within a single evaluation cycle (15 seconds), allowing the Deployment to converge quickly toward the desired state.

12.11.3 Terminating the Workload

Interrupting the Traffic Generator

Inside the BusyBox container, the workload generation can be stopped by pressing:

CTRL + C

This keyboard interrupt terminates the infinite loop generating HTTP requests, causing the workload applied to the application to decrease.

Leaving the BusyBox Container

The interactive shell can then be terminated through

```
exit
```

This command closes the shell session and returns control to the local terminal.

Deleting the Temporary Pod

Finally, the temporary Pod can be removed through

```
kubectl delete pod load-generator
```

This command removes the temporary load generator from the cluster, ensuring that no additional traffic is generated after the experiment and cleaning up the test environment.

Alternative Ephemeral Load Generator

As an alternative, the complete traffic generation process can be executed through a single command:

```
kubectl run -i --tty load-generator \
  --rm \
  --image=busybox \
  --restart=Never \
  -- /bin/sh -c \
"while true; do wget -q -O- http://nginx-service; done"
```

This command dynamically creates an ephemeral BusyBox Pod incorporating all the features previously discussed. The selected options provide several useful behaviours:

- `-i --tty` enables interactive terminal attachment
- `--restart=Never` prevents Kubernetes from automatically recreating the Pod after termination
- `--rm` automatically removes the Pod once execution finishes

Consequently, pressing

`CTRL + C`

immediately terminates the traffic generation process and automatically removes the temporary Pod from the cluster, leaving no residual testing resources.

12.11.4 Observing the Scale-In Process

After stopping the workload generation, the scale-in behaviour can be monitored through

```
kubectl get pods -w
```

As the HTTP request rate falls below the configured threshold, the Horizontal Pod Autoscaler gradually reduces the number of running replicas.

Expected behaviour

$$5 \rightarrow 4 \rightarrow 3$$

The observed scale-in sequence is governed by the configuration specified in the `scaleDown` section of the Horizontal Pod Autoscaler. Unlike scale-up operations, scale-down is constrained by two complementary mechanisms:

- A stabilization window of 60 seconds;
- A percentage-based scaling policy allowing a maximum reduction of 20% every evaluation interval.

The stabilization window delays replica removal, ensuring that temporary traffic fluctuations do not trigger premature scale-in operations. Once the stabilization period expires, the percentage-based policy limits the maximum reduction applied during each evaluation cycle, producing a gradual transition from five replicas to four and finally to the minimum configured value of three replicas. This behaviour guarantees a stable and controlled scale-in process, avoiding oscillations under low-load conditions while progressively converging toward the desired number of replicas.

12.12 Key Observations and Conclusions

Configuration Complexity Compared with native CPU-based autoscaling, implementing autoscaling through Prometheus without KEDA introduces additional configuration complexity. Besides deploying the monitoring infrastructure, this approach requires the installation and configuration of the Prometheus Adapter together with the definition of explicit metric mapping rules that expose Prometheus metrics through the Kubernetes External Metrics API. Consequently, the overall autoscaling pipeline involves a higher configuration overhead than native resource-based scaling.

Lack of Scale-to-Zero Support As in standard Kubernetes Horizontal Pod Autoscaler implementations, this architecture does not support scale-to-zero operations. Since the Horizontal Pod Autoscaler requires

$$\texttt{minReplicas} \geq 1$$

at least one Pod replica must always remain active, even when the monitored workload becomes negligible.

12.13 Chapter Summary

The experimental validation presented in this chapter demonstrates the complete integration between Prometheus, the Prometheus Adapter and the Horizontal Pod Autoscaler. By generating synthetic HTTP traffic through a temporary BusyBox Pod, it was possible to increase the monitored application-level metric and observe the corresponding autoscaling decisions in real time.

Throughout the experiment, the Horizontal Pod Autoscaler continuously compared the current value of the aggregated external metric against the configured target threshold, automatically adjusting the number of running replicas according to the defined scaling policies. This confirms that Kubernetes is capable of performing autoscaling based on application-level metrics rather than relying exclusively on infrastructure-level resource utilization. The experimental results also highlight both the strengths and the limitations of this architecture. On one hand, Prometheus and the Prometheus Adapter enable highly flexible autoscaling strategies driven by application-specific metrics. On the other hand, achieving this functionality requires the deployment and configuration of additional components, increasing the overall complexity of the monitoring and autoscaling infrastructure. Furthermore, similarly to native Horizontal Pod Autoscaler deployments, this solution cannot scale workloads down to zero replicas, since the HPA always requires at least one running Pod. Overall, this chapter demonstrates that Kubernetes can successfully implement autoscaling based on application-level metrics through the integration of Prometheus and the Prometheus Adapter. In the next chapter, the same experimental scenario will be revisited using KEDA, showing how the architecture can be significantly simplified while preserving the same scaling capabilities and additionally enabling scale-to-zero and scale-from-zero operations.

Chapter 13

NGINX Autoscaling with KEDA and Prometheus Metrics

13.1 Introduction

In this scenario, the same NGINX-based application introduced in the previous chapter is deployed within a Kubernetes cluster together with a Prometheus-based monitoring stack. As discussed previously, the application exposes runtime metrics through an exporter, which are periodically scraped and stored by Prometheus. From the perspective of metric collection, no architectural changes are introduced compared with the workflow presented in the previous chapter. The NGINX application continues to expose runtime statistics through the `/stub_status` endpoint, the exporter translates these statistics into Prometheus-compatible metrics, and Prometheus periodically scrapes and stores the resulting time-series data. Consequently, the monitoring infrastructure remains unchanged, allowing the focus to shift exclusively to the autoscaling architecture introduced by KEDA. The following sections describe how KEDA integrates with the existing monitoring pipeline, how external metrics are retrieved from Prometheus, and how the interaction with the Horizontal Pod Autoscaler is simplified through the KEDA Controller and the KEDA Metrics Adapter.

13.2 Common Monitoring Components

The following components remain identical to those described in the Prometheus Adapter-based workflow:

- NGINX Deployment and Pods
- Exporter sidecar exposing application-level metrics
- Prometheus scraping mechanism
- PromQL query definition (e.g., `sum(rate(nginx_http_requests_total[30s]))`)
- Horizontal Pod Autoscaler scaling formula and Deployment replica update logic

Maintaining the same monitoring infrastructure ensures continuity with the previously described application-level autoscaling workflow and allows the architectural differences introduced by KEDA to be isolated and analysed independently from the metric collection process.

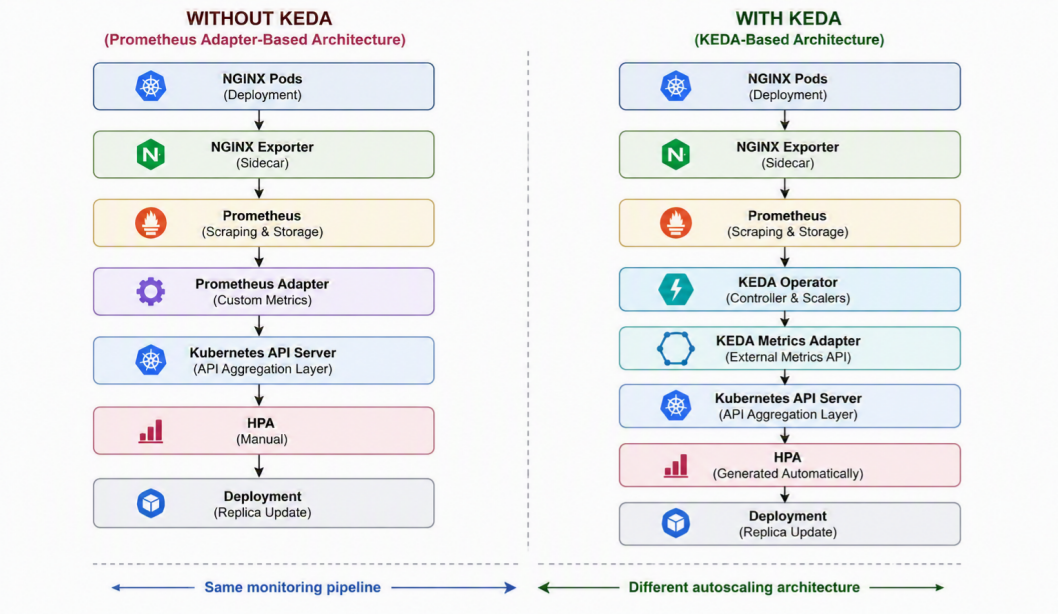


Figure 13.1: Comparison between Prometheus Adapter and KEDA-based autoscaling architectures.

As shown in Figure 13.1, the monitoring pipeline remains unchanged in both architectures. Application metrics are still collected by the NGINX Prometheus Exporter, scraped by Prometheus, and stored in the Prometheus time-series database. The main architectural difference lies in the autoscaling component. In the Prometheus Adapter-based approach, the Prometheus Adapter is responsible for translating PromQL query results into Kubernetes external or custom metrics that can be consumed by the Horizontal Pod Autoscaler. In contrast, KEDA removes the need for the Prometheus Adapter by directly querying Prometheus through its built-in Prometheus scaler. Based on the retrieved metric values, KEDA automatically manages the creation and configuration of the HPA, simplifying the overall autoscaling architecture while reducing the number of required components and the associated configuration effort.

13.3 KEDA Integration and Architectural Changes

The main architectural difference introduced in this workflow is the adoption of KEDA, which fundamentally changes how autoscaling is configured and orchestrated. Instead of manually configuring both a Horizontal Pod Autoscaler (HPA) resource and a Prometheus Adapter, the user now defines a single higher-level Kubernetes resource `ScaledObject`. This resource abstracts the autoscaling configuration by encapsulating both:

- The target workload to be scaled (the NGINX Deployment)
- The scaling logic, defined through a Prometheus-based trigger

Consequently, the autoscaling configuration becomes more declarative, as the user only specifies the desired scaling behaviour while KEDA automatically manages the underlying autoscaling infrastructure.

13.3.1 ScaledObject Creation and Validation

The workflow starts with the creation of the `ScaledObject` resource through the Kubernetes API Server. The manifest includes, among others, the following key fields:

- `scaleTargetRef`, specifying the target NGINX Deployment

- `triggers`, defining the Prometheus query together with the scaling threshold
- Optional parameters such as `pollingInterval`, controlling how frequently KEDA evaluates the external metric.

Once submitted, the ScaledObject follows the standard Kubernetes resource lifecycle:

- It is validated by the Kubernetes API Server
- It is stored in `etcd`
- It becomes observable by the KEDA Controller, which starts managing its lifecycle

13.3.2 KEDA Controller and Metric Polling

After detecting the newly created ScaledObject, the KEDA Controller starts its reconciliation loop. At this stage:

- The Prometheus scaler is initialized
- KEDA establishes communication with the Prometheus Server
- Metric retrieval begins according to the configured polling interval

The polling frequency is controlled through the parameter

```
1 pollingInterval
```

whose default value is approximately 30 seconds. This represents one of the first major architectural differences with respect to the previous Prometheus Adapter-based solution. Previously:

- The Prometheus Adapter queried Prometheus only when requested by the Horizontal Pod Autoscaler.

With KEDA:

- The KEDA Controller directly and periodically queries Prometheus independently of the HPA reconciliation cycle.

As a consequence, metric collection becomes proactive rather than request-driven, anticipating one of the key architectural characteristics that distinguish KEDA from the traditional Prometheus Adapter approach.

13.3.3 Metric Evaluation and Transformation

At each polling cycle, the KEDA Controller instructs the Prometheus scaler to execute the configured PromQL query:

```
sum(rate(nginx_http_requests_total[30s]))
```

The query returns a single aggregated value representing the overall HTTP request rate handled collectively by all replicas of the NGINX application. Unlike per-Pod metrics, this aggregated value provides a global view of the workload across the entire Deployment. After retrieving the query result, KEDA:

- Converts the aggregated value into an external metric
- Stores the computed metric internally
- Makes the metric available to the Kubernetes autoscaling pipeline

13.3.4 Metric Exposure Through KEDA Metrics Adapter

Unlike the Prometheus Adapter-based architecture discussed in the previous chapter, no standalone Prometheus Adapter is required. Instead, KEDA relies on its own internal component, the **KEDA Metrics Adapter**, which is responsible for exposing the computed metrics to Kubernetes. The KEDA Metrics Adapter:

- Integrates with the Kubernetes API Aggregation Layer
- Exposes external metrics through the endpoint:

  ```
  /apis/external.metrics.k8s.io
  ```

- Provides the latest metric value whenever requested by the Horizontal Pod Autoscaler

The interaction between Kubernetes components follows the workflow below:

1. The Horizontal Pod Autoscaler sends a metric request to the Kubernetes API Server

2. The Kubernetes API Server forwards the request to the KEDA Metrics Adapter

3. The KEDA Metrics Adapter immediately returns the most recently computed metric value

A fundamental architectural difference emerges when comparing this workflow with the Prometheus Adapter-based solution presented in the previous chapter. In the traditional architecture, metric evaluation is performed *on demand*. Every time the Horizontal Pod Autoscaler executes its reconciliation loop, it requests the current metric value through the Kubernetes API Server. The Prometheus Adapter then executes the corresponding PromQL query against the Prometheus Server and returns the result to the HPA. KEDA adopts a completely different approach. Rather than evaluating metrics only when requested by the HPA, the KEDA Controller continuously polls the external data source according to the configured `pollingInterval`. The computed metric values are therefore pre-fetched, maintained internally by KEDA, and made immediately available through the KEDA Metrics Adapter whenever the HPA requests them. Consequently, metric collection and metric consumption become fully decoupled. While the Horizontal Pod Autoscaler still periodically requests metrics through the Kubernetes Metrics API, the requested values have already been computed and cached by KEDA, eliminating the need to execute a PromQL query during each HPA reconciliation cycle. This polling-based architecture represents one of the fundamental design principles of KEDA. Although Prometheus is used as the external data source in this experiment, the same workflow applies to all supported KEDA scalers, including Kafka, RabbitMQ, Redis, Azure Service Bus, AWS SQS, and many others. Independently of the underlying event source, KEDA continuously retrieves and maintains the metric values before exposing them through its internal Metrics Adapter to the Kubernetes autoscaling pipeline.

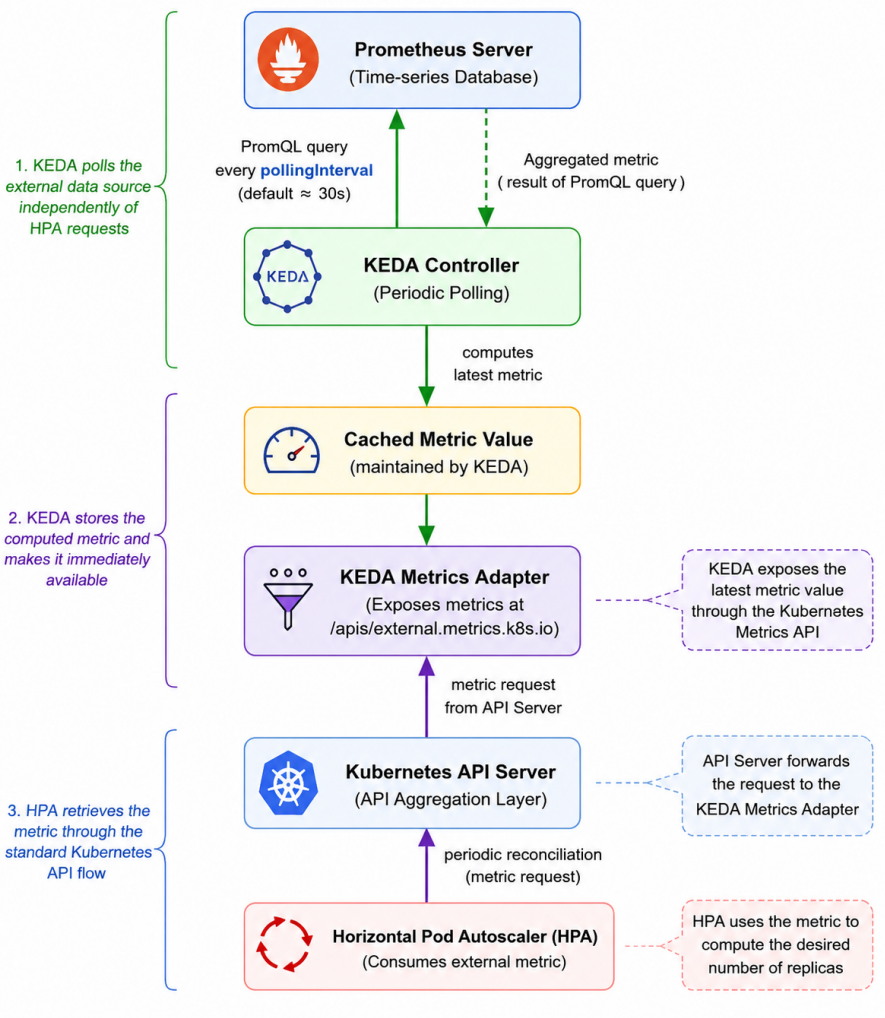


Figure 13.2: Internal metric polling workflow in KEDA.

As shown in Figure 13.2, KEDA adopts a proactive metric polling approach. Instead of querying Prometheus whenever the Horizontal Pod Autoscaler requests a metric, the KEDA Operator periodically retrieves the metric value from the external data source and stores it internally. When the HPA requests the corresponding external metric through the External Metrics API, the KEDA Metrics Adapter immediately returns the cached value.

Aspect	**Prometheus Adapter**	**KEDA**
Metric evaluation timing	On-demand, triggered during each HPA reconciliation cycle.	Periodic polling performed independently of HPA activity.
Control flow	HPA → API Server → Prometheus Adapter → Prometheus	KEDA Controller → Prometheus → Cached metric → HPA (through the KEDA Metrics Adapter).
Coupling between components	Tightly coupled to HPA requests (request-driven model).	Metric collection and scaling decisions are fully decoupled.
Latency characteristics	Latency depends on PromQL execution performed at request time.	Near-zero latency during HPA reconciliation since the metric has already been computed and cached.
Scalability model	Reactive approach with frequent PromQL execution under intensive HPA activity.	Proactive polling model that reduces the number of Prometheus queries by periodically pre-computing metrics.
Extensibility	Limited to metrics exposed through Prometheus and mapped to the Kubernetes Metrics API.	Supports numerous event sources (Prometheus, Kafka, RabbitMQ, Redis, Azure Service Bus, AWS SQS, etc.) through a unified scaler abstraction.
Configuration effort	Requires manual deployment and configuration of the Prometheus Adapter, explicit metric mappings, and a manually defined Horizontal Pod Autoscaler.	Only a ScaledObject needs to be defined. KEDA automatically creates and manages the Horizontal Pod Autoscaler together with the metric exposure pipeline.

Table 13.1: Summarizes the main architectural differences between the traditional Prometheus Adapter-based approach and the KEDA-based autoscaling workflow discussed in this chapter.

The comparison highlights that the fundamental scaling algorithm implemented by the Horizontal Pod Autoscaler remains unchanged in both architectures. The primary differences concern the way external metrics are collected, processed, and exposed to Kubernetes. By continuously polling external systems, caching the resulting metric values, and automatically managing the Horizontal Pod Autoscaler, KEDA significantly simplifies the autoscaling architecture while reducing configuration complexity and improving extensibility.

13.4 Automatic HPA Generation

Another important architectural difference introduced by KEDA concerns the creation of the Horizontal Pod Autoscaler. In the Prometheus Adapter-based workflow discussed in the previous chapter, the HPA had to be explicitly defined and deployed by the user through a dedicated Kubernetes manifest. In contrast, KEDA automatically generates and manages the corresponding HPA based on the configuration specified in the `ScaledObject` resource. Once the ScaledObject is reconciled, KEDA creates an HPA configured according to the trigger definition and continuously keeps it synchronized with the desired autoscaling configuration. In the considered scenario, the generated HPA uses:

- An external metric

```
type: External
```

and compares it against a fixed target value defined as:

```
target:
type: Value
```

The resulting Horizontal Pod Autoscaler is therefore functionally equivalent to the manually created HPA presented in the previous chapter, while requiring no direct user intervention.

13.5 Scaling Decision

Despite the architectural changes introduced by KEDA, the core scaling algorithm remains exactly the same. After receiving the external metric through the Kubernetes Metrics API, the Horizontal Pod Autoscaler:

- Retrieves the latest metric value
- Applies the standard Horizontal Pod Autoscaler scaling formula
- Updates the `spec.replicas` field of the target Deployment

Consequently:

- Scale-out occurs whenever the observed metric exceeds the configured threshold
- Scale-in occurs whenever the observed metric falls below the configured threshold

Therefore, although KEDA significantly simplifies the autoscaling architecture by managing metric retrieval and HPA generation, the final scaling decision continues to be performed by the native Kubernetes Horizontal Pod Autoscaler according to its standard reconciliation algorithm. This observation confirms that KEDA does not replace the Horizontal Pod Autoscaler. Instead, it extends the native Kubernetes autoscaling framework by providing a simplified mechanism for retrieving external metrics and automatically configuring the HPA, while delegating the final scaling decisions to the standard Kubernetes autoscaling algorithm.

13.6 Chapter Summary

The previous sections have demonstrated that the introduction of KEDA does not modify the fundamental autoscaling algorithm implemented by Kubernetes. Instead, it changes the way external metrics are collected, processed, and delivered to the Horizontal Pod Autoscaler. Without KEDA, implementing application-level autoscaling requires several manually configured components, including the Prometheus Adapter, the Horizontal Pod Autoscaler, and the corresponding metric mappings. The complete control flow follows the sequence:

Prometheus → Prometheus Adapter → HPA → Deployment

With KEDA, the architecture becomes considerably simpler. The Prometheus Adapter is no longer required, the Horizontal Pod Autoscaler is automatically

generated from the `ScaledObject` definition, and metric retrieval is entirely managed by the KEDA Controller and its internal Metrics Adapter. The simplified workflow is illustrated in Figure 13.3:

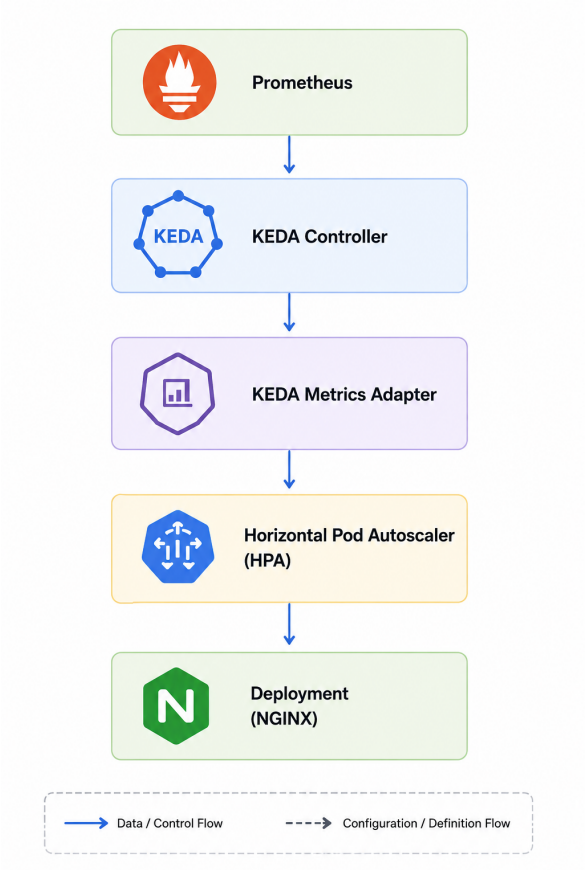


Figure 13.3: Simplified KEDA autoscaling workflow.

As discussed throughout this chapter, the Horizontal Pod Autoscaler continues to execute the native Kubernetes scaling algorithm, while KEDA assumes responsibility for metric retrieval, transformation, and orchestration. This separation of responsibilities greatly reduces configuration complexity without altering the underlying autoscaling logic. Overall, KEDA provides a more modular, extensible, and declarative autoscaling architecture. By abstracting the integration with external event sources and automatically managing the interaction with the Horizontal Pod Autoscaler, it significantly simplifies the

implementation of event-driven autoscaling while preserving the robustness of the native Kubernetes scaling mechanism. These characteristics reduce the operational effort required to configure application-level autoscaling and improve the maintainability of the overall solution. These architectural advantages become particularly evident in the experimental implementation presented in the following chapter.

Chapter 14

Experimental Evaluation of KEDA-Based Prometheus Autoscaling (NGINX Use Case)

14.1 Experimental Objective

In this experiment, the same NGINX-based application introduced in the previous chapter is deployed within a Kubernetes cluster and monitored through a Prometheus-based monitoring stack. The objective is to experimentally validate the KEDA-based autoscaling architecture presented in Chapter 13 by enabling autoscaling based on the application-level metric `nginx_http_requests_total`. In particular, the experiment demonstrates how KEDA replaces the Prometheus Adapter while preserving the existing monitoring pipeline and the underlying Horizontal Pod Autoscaler scaling logic. Before introducing KEDA, it is important to emphasize that the entire metric collection pipeline remains unchanged with respect to the previous implementation based on the Prometheus Adapter. Therefore, all Kubernetes resources responsible for metric generation, exposure, and collection are reused without modification. Since these resources have already been discussed in detail in Chapter 12, they are not reported again in full in this chapter. Instead, only the main architectural considerations are summarized below.

14.2 Reusing the Existing Monitoring Infrastructure

NGINX Configuration
The ConfigMap defining the NGINX configuration is identical to the one presented in Chapter 12 and is therefore not repeated here. The configuration enables the `/stub_status` endpoint, which exposes the raw application statistics generated by NGINX. Access to this endpoint is restricted to the loopback interface (`127.0.0.1`), ensuring that only containers running within the same Pod can access it. This design is fundamental because the NGINX Prometheus Exporter, deployed as a sidecar container, retrieves application metrics locally through the shared network namespace.

Deployment with Exporter Sidecar
The Deployment manifest also remains unchanged with respect to the previous experiment (see Chapter 12) and continues to adopt the sidecar pattern. The application Pod still contains two containers:

- The NGINX container, responsible for serving HTTP traffic on port 80
- The NGINX Prometheus Exporter, which retrieves metrics locally from the `/stub_status` endpoint and exposes them in Prometheus format through port 9113

No modification is required at this layer because KEDA operates exclusively on the autoscaling mechanism, while metric generation and exposure remain completely unchanged.

Service Definition
The Kubernetes Service is also reused without any modification. As in the previous implementation, it exposes two ports:

- port 80, used for application traffic and for workload generation during the experimental validation
- port 9113, used exclusively by the Prometheus Server to scrape metrics exposed by the exporter

The dual-port configuration preserves a clear separation between application traffic and monitoring traffic while providing stable endpoints for both communication flows.

ServiceMonitor
The ServiceMonitor resource is identical to the one previously introduced and therefore is not repeated. Its purpose remains to instruct the Prometheus Server to periodically scrape the metrics endpoint exposed by the NGINX Prometheus Exporter. As discussed in the previous chapter, the ServiceMonitor selects the Kubernetes Service through label matching and leverages Kubernetes service discovery to retrieve the corresponding Pod endpoints. Consequently, Prometheus does not scrape metrics through the Service ClusterIP, but instead directly contacts every exporter instance using its Pod IP address and the exposed metrics port. The resulting monitoring workflow therefore remains unchanged:

$$\begin{aligned} \text{Prometheus} &\rightarrow \text{PodIP}_1{:}9113 \\ \text{Prometheus} &\rightarrow \text{PodIP}_2{:}9113 \\ \text{Prometheus} &\rightarrow \text{PodIP}_3{:}9113 \end{aligned}$$

At this stage, the entire monitoring pipeline is identical to the architecture previously validated without KEDA.

Deployment of the Monitoring Resources
Since the monitoring infrastructure is unchanged, the same Kubernetes manifests discussed in Chapter 12 are deployed using:

```
kubectl apply -f configmap.yaml
kubectl apply -f deployment.yaml
kubectl apply -f service.yaml
kubectl apply -f servicemonitor.yaml
```

These commands deploy the NGINX configuration, the application Pods with the exporter sidecar, the Kubernetes Service, and the ServiceMonitor resource, preserving exactly the same metric generation and collection workflow adopted in the previous experiment.

Metric Verification
Before enabling autoscaling, it is necessary to verify that application metrics are correctly exposed and collected throughout the monitoring pipeline. Since the monitoring architecture remains unchanged, the validation procedure is conceptually identical to that presented in Chapter 12. Therefore, the detailed verification steps are not repeated in this section. Briefly, the Prometheus Server can be temporarily exposed through a `kubectl port-forward` operation, allowing access to the Prometheus Web UI for interactive execution of

PromQL queries and real-time inspection of collected metrics. Through this interface, both raw and aggregated PromQL queries can be executed to validate the correct functioning of the monitoring pipeline and the metric computation logic used by the autoscaling mechanism. Consequently, the same debugging methodology described in the previous experiment can be applied, including:

- Verification of metric collection
- Validation of PromQL query results
- Inspection of aggregated autoscaling metrics
- Runtime observation of scaling behaviou

Nevertheless, one important architectural difference should already be highlighted. In the Prometheus Adapter-based implementation, PromQL queries are executed on demand whenever the Horizontal Pod Autoscaler requests a metric through the Kubernetes Metrics API. In contrast, KEDA adopts a decoupled polling-based architecture in which external scalers periodically retrieve metrics independently of HPA reconciliation cycles. Consequently, when the Horizontal Pod Autoscaler requests the current metric value, the latest value has already been computed and made available through the KEDA Metrics Adapter.

14.3 ScaledObject Configuration

As discussed in the previous chapter, the introduction of KEDA modifies exclusively the autoscaling layer, while leaving the monitoring infrastructure unchanged. Consequently, the Prometheus Adapter and the manually defined Horizontal Pod Autoscaler are no longer required. Instead, the complete autoscaling configuration is encapsulated within a single Kubernetes custom resource, namely the `ScaledObject`. This resource defines both the workload to be scaled and the event-driven scaling logic based on Prometheus metrics. The ScaledObject adopted in this experiment is reported below:

scaledobject.yaml

```
1 apiVersion: keda.sh/v1alpha1
2 kind: ScaledObject
3 metadata:
```

```
4   name: nginx-scaledobject
5 spec:
6   scaleTargetRef:
7     name: nginx-deployment
8   minReplicaCount: 3
9   maxReplicaCount: 5
10  pollingInterval: 30
11
12  advanced:
13    horizontalPodAutoscalerConfig:
14      behavior:
15        scaleDown:
16          stabilizationWindowSeconds: 60
17          policies:
18            - type: Percent
19              value: 100
20              periodSeconds: 15
21        scaleUp:
22          stabilizationWindowSeconds: 0
23          policies:
24            - type: Percent
25              value: 100
26              periodSeconds: 15
27
28  triggers:
29    - type: prometheus
30      metricType: Value
31      metadata:
32        serverAddress: http://monitoring-kube-prometheus-prometheus.
    monitoring:9090
33        metricName: nginx_rps
34        query: sum(rate(nginx_http_requests_total[30s]))
35        threshold: "20"
36        ignoreNullValues: "false"
```

Listing 14.1: ScaledObject manifest used for Prometheus-based autoscaling with KEDA.

Compared with the previous implementation based on the Prometheus Adapter, the configuration becomes significantly more compact. Instead of defining

separate resources for the Prometheus Adapter and the Horizontal Pod Autoscaler, KEDA centralizes the entire autoscaling configuration within a single declarative resource. The `ScaledObject` references the target Deployment, specifies the desired scaling policies, defines the polling interval adopted by the KEDA Controller, and configures the Prometheus trigger used to retrieve the application-level metric that drives autoscaling. The following sections analyze each part of the manifest in detail, illustrating how KEDA automatically orchestrates the complete autoscaling workflow starting from this single resource definition.

14.3.1 Scaling Parameters

The first group of parameters contained in the `ScaledObject` specifies the workload to be managed together with the basic scaling boundaries adopted by KEDA.

scaleTargetRef
The `scaleTargetRef` field specifies the Kubernetes workload managed by the autoscaling mechanism. In this experiment, it references the NGINX Deployment previously introduced, whose replica count will be dynamically adjusted according to the Prometheus metric evaluated by KEDA.

minReplicaCount
The `minReplicaCount` parameter defines the minimum number of Pod replicas that must always remain active.

In the adopted configuration:

- The Deployment starts with three running replicas
- The autoscaler is not allowed to reduce the replica count below this value

Unlike the previous Cron-based experiment, where KEDA was configured to perform scale-to-zero operations, the objective of this experiment is to evaluate autoscaling exclusively within the interval between three and five replicas.

maxReplicaCount
The `maxReplicaCount` field specifies the maximum number of replicas that KEDA is allowed to create. In this experiment, the upper bound is set to five replicas, preventing uncontrolled scale-out while providing a sufficiently wide range to observe the autoscaling behaviour under increasing workload.

pollingInterval
The `pollingInterval` parameter defines how frequently the KEDA Controller polls the external metric source. In this configuration:

```
1 pollingInterval = 30s
```

Consequently, every 30 seconds the Prometheus scaler:

- Executes the configured PromQL query
- Retrieves the most recent application-level metric
- Updates the metric value exposed through the KEDA Metrics Adapter

This parameter represents one of the main architectural differences between KEDA and the Prometheus Adapter-based approach discussed in the previous chapter. Rather than executing PromQL queries only when requested by the Horizontal Pod Autoscaler, KEDA continuously polls Prometheus according to the configured polling interval, ensuring that an up-to-date metric value is already available whenever the HPA performs a scaling evaluation.

14.3.2 Advanced HPA Behaviour Configuration

The `advanced` section allows the customization of the Horizontal Pod Autoscaler automatically generated by KEDA. More specifically, the horizontalPodAutoscalerConfig field provides direct access to the HPA configuration, enabling the definition of custom scaling behaviours while leaving the HPA generation process entirely managed by KEDA. The `behavior` field specifies the scaling policies adopted during both scale-up and scale-down operations.

Scale-Down Configuration
The `scaleDown` section defines the behaviour adopted when the application workload decreases and the number of running Pod replicas must be reduced. The configuration specifies:

- `stabilizationWindowSeconds:60`
- a percentage-based scaling policy (`type:Percent`)
- `value:100`
- `periodSeconds:15`.

As discussed in Chapter 12, the stabilization window delays scale-down operations by continuously evaluating the scaling recommendations generated during the configured time interval. This mechanism prevents unnecessary oscillations caused by temporary workload fluctuations. Although Kubernetes adopts a default stabilization window of 300 seconds, the value has been intentionally reduced to 60 seconds in this experiment in order to accelerate the validation process and make scale-in events observable within a shorter time interval. Unlike the previous experiment, where a conservative policy (`value: 20`) produced the gradual transition:

$$5 \rightarrow 4 \rightarrow 3$$

the current configuration adopts a much more aggressive policy by setting:

```
value: 100
```

Consequently, once the stabilization window expires and the observed metric falls below the configured threshold, the autoscaler is allowed to immediately converge to the desired replica count, producing the direct transition:

$$5 \rightarrow 3$$

The parameter `periodSeconds:15` specifies that this policy may be evaluated every 15 seconds, consistently with the Horizontal Pod Autoscaler reconciliation interval.

Scale-Up Configuration
The `scaleUp` section defines the behaviour adopted when the workload increases and additional replicas are required. The configuration specifies:

- `stabilizationWindowSeconds:0`
- a percentage-based scaling policy (`value:100`)
- `periodSeconds:15`.

Setting the stabilization window to zero disables any delay before scaling out, allowing the autoscaler to react immediately whenever the observed metric exceeds the configured threshold. Moreover, the policy value equal to 100 permits the replica count to increase by up to 100% during a single evaluation

cycle. As a result, under high incoming traffic conditions, the Deployment can rapidly transition from:

$$3 \rightarrow 5$$

replicas, providing a highly responsive scale-out behaviour. Overall, these parameters demonstrate that, although the Horizontal Pod Autoscaler is automatically generated by KEDA, its behaviour remains fully configurable through the `ScaledObject` resource, allowing the same level of control available when manually defining an HPA.

14.3.3 Prometheus Trigger Configuration

The `triggers` section defines the event source driving the autoscaling process. In this experiment, a Prometheus-based trigger is configured, allowing KEDA to periodically execute a PromQL query and use the resulting metric to drive scaling decisions.

Trigger Type
The field:

```
type: prometheus
```

specifies that the external metric source is a Prometheus Server. Consequently, during each polling cycle, the KEDA Prometheus scaler executes the configured PromQL query and retrieves the corresponding application-level metric.

Metric Interpretation
One of the most important parameters of the trigger configuration is:

```
metricType: Value
```

This parameter explicitly specifies how the retrieved metric must be interpreted by the Horizontal Pod Autoscaler automatically generated by KEDA. When the metric type is set to `Value`, the metric is interpreted as a single global value associated with the entire workload rather than as an average value distributed across the running replicas. Consequently, KEDA generates an HPA configured as:

```
target:
  type: Value
```

instead of the default configuration:

```
target:
  type: AverageValue
```

This distinction is particularly important when using aggregated Prometheus queries such as:

```
sum(rate(nginx_http_requests_total[30s]))
```

since the query already returns a single metric representing the overall HTTP request rate handled collectively by the entire NGINX Deployment. If the metric were interpreted as an `AverageValue`, the Horizontal Pod Autoscaler would automatically normalize the aggregated value by dividing it by the current number of running replicas before evaluating the scaling algorithm. During the experimental validation, this behaviour was directly observed. In the absence of external traffic, the aggregated Prometheus query returned the expected value:

$$0.2 = 200\text{m}$$

which corresponds to the total request rate generated by the periodic `Prometheusscraping` operations. However, when the default `AverageValue` interpretation was adopted, the HPA internally divided this aggregated value by the initial three running replicas, obtaining:

$$\frac{0.2}{3} \approx 0.067 = 67\text{m}$$

instead of the expected:

$$200\text{m}$$

By explicitly configuring:

```
metricType: Value
```

the aggregated metric is consumed directly without any normalization across replicas, preserving the original semantics of the PromQL query and ensuring consistency between metric computation and autoscaling behaviour. Finally, it is worth emphasizing that this capability is not specific to the Prometheus scaler. According to the official KEDA documentation, all scalers, except the CPU and Memory scalers, support both `AverageValue` and `Value` metric types, whereas CPU and Memory scalers support the `AverageValue` and

`Utilization` modes. Therefore, the possibility of explicitly selecting the metric interpretation represents a general feature of KEDA and can be exploited across most supported event sources.

Trigger Metadata
The remaining fields contained in the trigger `metadata` section define how the KEDA Prometheus scaler communicates with the Prometheus Server and interprets the metric returned by the configured PromQL query.

serverAddress
The `serverAddress` parameter specifies the endpoint used by KEDA to communicate with the Prometheus Server running inside the Kubernetes cluster. In this implementation, the Prometheus instance is reached through its internal Kubernetes Service DNS:

```
monitoring-kube-prometheus-prometheus.monitoring:9090
```

This endpoint allows the KEDA Controller to periodically execute PromQL queries without exposing the Prometheus Server outside the cluster.

metricName
The `metricName` field defines the logical name assigned to the metric inside KEDA. Although independent of the original Prometheus metric name, this identifier is subsequently exposed through the KEDA Metrics Adapter and consumed by the automatically generated Horizontal Pod Autoscaler when requesting external metrics from the Kubernetes Metrics API.

Query
The field:

```
1 query: sum(rate(nginx_http_requests_total[30s]))
```

defines the PromQL expression periodically executed by the KEDA Prometheus scaler. As already discussed in the previous chapter, the `rate()` function computes the average HTTP request rate over a 30-second observation window, while the `sum()` aggregation combines the contribution of all running replicas into a single scalar value representing the overall workload of the NGINX Deployment. Consequently, KEDA expects this query to return exactly one aggregated value, which is then exposed to the Horizontal Pod Autoscaler through the KEDA Metrics Adapter.

threshold
The parameter:

```
threshold: "20"
```

defines the target value used by the autoscaler during scaling decisions. More specifically:

- If the observed metric exceeds the configured threshold, the autoscaler performs a scale-out operation
- If the metric falls below the threshold, the autoscaler initiates a scale-in operation according to the configured scaling policies

ignoreNullValues
One of the most useful parameters during the experimental activity is:

```
ignoreNullValues: "false"
```

This field controls how the KEDA Prometheus scaler behaves whenever the configured PromQL query returns an empty result. By default, this parameter is implicitly configured as:

```
ignoreNullValues: "true"
```

meaning that empty Prometheus query results are silently interpreted as valid zero-valued metrics. During the experimental validation, this behaviour complicated the debugging process because an empty Prometheus response could not be distinguished from a legitimate metric value equal to zero. For this reason, the parameter was explicitly configured as:

```
ignoreNullValues: "false"
```

Under this configuration, the scaler explicitly reports an invalid metric whenever:

- The Prometheus Server cannot be reached
- The Prometheus query returns an empty vector because metrics are not correctly collected from the exporters.

In both situations, the generated Horizontal Pod Autoscaler reports:

```
<unknown>
```

inside the `TARGETS` column of:

```
kubectl get hpa -w
```

clearly indicating that the monitoring pipeline is malfunctioning rather than simply observing a zero-valued metric. Conversely, when the default configuration

```
ignoreNullValues: "true"
```

is adopted, empty Prometheus responses are interpreted as zero-valued metrics. As a consequence, the Horizontal Pod Autoscaler may display:

```
0 / threshold
```

even though the PromQL query was not successfully evaluated. Therefore, a displayed metric value equal to zero does not necessarily imply that the actual application workload is zero, but may instead indicate that the monitoring pipeline failed to return valid metric data. This behaviour proved particularly useful during the debugging phase of the experimental implementation, as it allowed monitoring failures to be immediately distinguished from legitimate low-load conditions.

14.4 ScaledObject Deployment and Verification

The configured `ScaledObject` is deployed through the standard Kubernetes command:

```
kubectl apply -f scaledobject.yaml
```

Once the resource has been created, the KEDA Controller automatically detects the new `ScaledObject`, initializes the corresponding Prometheus scaler, and generates the associated Horizontal Pod Autoscaler according to the specified configuration. The correct creation of the automatically generated HPA, together with the propagation of the external metric through the Kubernetes autoscaling pipeline, can be verified using:

```
kubectl get hpa -w
```

This command continuously monitors the generated Horizontal Pod Autoscaler, displaying the current metric value together with the configured scaling threshold and the current number of running replicas.

Verification of the Idle-State Metric
In the absence of externally generated traffic, the `TARGETS` column is expected to display approximately:

$$200\text{m}/20$$

corresponding to:

$$0.2/20$$

As discussed in the previous chapter, this baseline value is produced by the monitoring pipeline itself. The periodic scraping operations performed by the Prometheus Server generate internal requests towards the NGINX exporter, which are ultimately reflected by the `nginx_http_requests_total` counter. When the configured PromQL query

```
sum(rate(nginx_http_requests_total[30s]))
```

is evaluated, the aggregated request rate corresponds to approximately:

$$0.2 = 200\text{m}$$

which therefore represents the expected idle-state metric. Unlike the initial behaviour observed when using the default `AverageValue` metric interpretation, the Horizontal Pod Autoscaler now correctly reports the aggregated value without any normalization across replicas. This confirms that the explicit configuration:

```
1 metricType: Value
```

is correctly propagated from the `ScaledObject` to the automatically generated HPA.

Verification of the Generated HPA
The configuration automatically generated by KEDA can be further inspected in two complementary ways. First, the absence of the `(avg)` notation in the `TARGETS` column confirms that the metric is interpreted as a global value rather than as an average value across replicas. A complete inspection of the generated resource can then be performed through:

```
kubectl get hpa keda-hpa-nginx-scaledobject -o yaml
```

This command displays the complete Horizontal Pod Autoscaler manifest automatically generated by KEDA, allowing verification of both the metric configuration and the scaling policies derived from the corresponding `ScaledObject`. Overall, these verification steps confirm that KEDA has successfully generated the Horizontal Pod Autoscaler and correctly propagated the Prometheus trigger configuration through the Kubernetes autoscaling pipeline.

14.5 Load Generation and Autoscaling Validation

To experimentally validate the KEDA-based autoscaling mechanism, the same workload generation procedure adopted in the previous chapter is used. A temporary BusyBox Pod is created inside the Kubernetes cluster to continuously generate HTTP requests towards the NGINX Service. The load generator can be started through:

```
kubectl run -i --tty load-generator \
  --rm \
  --image=busybox \
  --restart=Never \
  -- /bin/sh
```

The `--rm` option automatically removes the temporary Pod once the interactive session terminates, simplifying the cleanup phase after the experiment. Inside the BusyBox container, continuous traffic is generated by executing:

```
while true; do
    wget -q -O- http://nginx-service
done
```

This loop continuously sends HTTP requests to the NGINX Service, progressively increasing the value of the aggregated metric `nginx_rps` computed by the Prometheus query configured inside the `ScaledObject`. As soon as the observed metric exceeds the configured threshold, KEDA detects the increased workload during the next polling cycle and updates the external metric exposed through the KEDA Metrics Adapter. Unlike the previous implementation based on the Prometheus Adapter, the metric used by the HPA has already been computed by the KEDA Controller during its periodic polling

cycle. Consequently, the HPA simply retrieves the latest available metric through the KEDA Metrics Adapter without triggering a new Prometheus query. The automatically generated Horizontal Pod Autoscaler subsequently evaluates the new metric and performs a scale-out operation. According to the configuration presented in the previous sections, the Deployment is expected to transition from:

$$3 \rightarrow 5$$

replicas, consistently with the aggressive scale-up policy configured through the `horizontalPodAutoscalerConfig` section of the `ScaledObject`. Once the load generation process is interrupted by pressing:

`CTRL+C`

the HTTP traffic immediately stops and, thanks to the `--rm` option, the temporary BusyBox Pod is automatically removed from the cluster. Consequently, the observed request rate rapidly decreases below the configured threshold. After the configured stabilization window expires, the Horizontal Pod Autoscaler performs the corresponding scale-in operation, reducing the number of replicas from:

$$5 \rightarrow 3.$$

This behaviour is fully consistent with the scale-down policy configured in the `horizontalPodAutoscalerConfig` section of the `ScaledObject`, confirming that KEDA correctly orchestrates the complete autoscaling workflow while delegating the final scaling decision to the automatically generated Horizontal Pod Autoscaler.

14.6 Chapter Summary

This chapter presented the complete implementation and experimental validation of a KEDA-based autoscaling workflow driven by application-level metrics collected through Prometheus. The results demonstrate that the adoption of KEDA does not modify the underlying monitoring pipeline. The collection of application-level metrics remains entirely based on the Prometheus ecosystem, including the NGINX exporter, the Prometheus Server, and the corresponding PromQL queries. Consequently, the generation and aggregation of monitoring data are identical to those described in the previous implementation based

on the Prometheus Adapter. The main architectural differences are instead concentrated in the autoscaling layer. By introducing the `ScaledObject` abstraction, KEDA removes the need for manually deploying and configuring both a dedicated Prometheus Adapter and a Horizontal Pod Autoscaler for each autoscaling scenario. The entire scaling configuration is instead encapsulated within a single declarative Kubernetes resource, which is automatically translated by KEDA into the corresponding autoscaling infrastructure. Despite this simplification, the underlying scaling logic remains unchanged. Autoscaling decisions continue to be driven by Prometheus metrics computed through PromQL expressions, while the Horizontal Pod Autoscaler still applies the standard Kubernetes scaling algorithm to determine the desired number of replicas. The principal architectural difference lies in the metric retrieval model. In the Prometheus Adapter-based approach, metric evaluation is performed on demand whenever the Horizontal Pod Autoscaler requests an updated metric through the Kubernetes Metrics API. Conversely, KEDA adopts a decoupled polling-based architecture in which external scalers periodically query the monitoring system, pre-compute the required metrics, and make the latest values immediately available through the KEDA Metrics Adapter. Overall, the experimental results confirm that KEDA preserves the robustness of Kubernetes native autoscaling while significantly reducing configuration complexity and improving the modularity, maintainability, and extensibility of the autoscaling architecture. These characteristics make KEDA particularly well suited for cloud-native applications requiring event-driven scaling based on heterogeneous external data sources.

Chapter 15

Monitoring KEDA Autoscaling Metrics through Prometheus and Grafana

15.1 Introduction

In the previous chapters, the application-level metrics exposed by the NGINX workload and collected through Prometheus were exploited by KEDA to drive autoscaling decisions. In this chapter, the opposite perspective is considered. Rather than focusing on the application metrics consumed by KEDA, the attention shifts to the operational metrics generated by KEDA itself. These metrics can be collected by Prometheus and visualized through Grafana, providing valuable insight into the internal behavior of the KEDA control plane and enabling advanced monitoring and observability of event-driven autoscaling systems. The same NGINX-based autoscaling scenario presented in the previous chapter is reused throughout this experiment. Consequently, all Kubernetes manifests related to the application deployment, the NGINX Prometheus Exporter, the ServiceMonitor, and the ScaledObject remain unchanged and are therefore not reported again in this section. The discussion focuses exclusively on the additional configuration required to monitor the KEDA infrastructure itself. When KEDA is installed in a Kubernetes cluster, several distributed components are automatically deployed within the `keda` namespace. The main components include:

- The KEDA Operator
- The KEDA Metrics API Server
- The KEDA Admission Webhooks

Together with these Pods, Kubernetes Service resources are automatically created to expose the internal communication endpoints required by the KEDA control plane. Each component exposes Prometheus-compatible metrics through the standard `/metrics` HTTP endpoint, typically available on port `8080`. These metrics provide operational information regarding several aspects of the autoscaling process, including:

- Scaler activity
- Trigger execution
- Autoscaling state
- Metric retrieval
- Scaler errors
- Interactions with the Kubernetes Horizontal Pod Autoscaler

Unlike the application metrics discussed in the previous chapters, these metrics describe the internal state and operation of the KEDA infrastructure itself, making them particularly valuable for debugging, monitoring, and analyzing the behavior of event-driven autoscaling within Kubernetes environments.

15.2 Enabling Prometheus Monitoring for KEDA Components

Although the KEDA components automatically expose Prometheus-compatible metrics through their `/metrics` endpoints, these metrics are not scraped by the Prometheus Server by default. To enable metric collection, dedicated `ServiceMonitor` resources must be created so that the Prometheus Operator can discover the KEDA Services and periodically scrape their metrics. The automatic generation of the required `ServiceMonitor` resources can be enabled

by installing or updating KEDA through a dedicated Helm configuration file. In the adopted setup, the following configuration was used:

values.yaml

```
prometheus:
  operator:
    enabled: true
    serviceMonitor:
      enabled: true

  metricServer:
    enabled: true
    serviceMonitor:
      enabled: true

  webhooks:
    enabled: true
    serviceMonitor:
      enabled: true
```

The updated configuration can then be applied through:

```
helm upgrade keda kedacore/keda \
  -n keda \
  -f values.yaml
```

After the installation (or upgrade) process, the automatically generated `ServiceMonitor` resources can be verified by executing:

```
kubectl get servicemonitor -n keda
```

In the adopted monitoring infrastructure, an additional configuration step was required. As already discussed in the previous chapters, the Prometheus Operator was configured with a restrictive `serviceMonitorSelector`. Consequently, only `ServiceMonitor` resources containing the appropriate label were discovered by Prometheus. The configured selector can be inspected through:

```
kubectl get prometheus -n monitoring -o yaml | grep -A20
serviceMonitorSelector
```

which reveals that the generated `ServiceMonitor` resources must contain the label

```
release: monitoring
```

to be selected by the Prometheus Operator. Since this label was not automatically added during the KEDA installation, it was manually assigned to each generated `ServiceMonitor`:

```
kubectl label servicemonitor -n keda \
    keda-operator release=monitoring

kubectl label servicemonitor -n keda \
    keda-admission-webhooks release=monitoring

kubectl label servicemonitor -n keda \
    keda-operator-metrics-apiserver release=monitoring
```

Once the generated `ServiceMonitor` resources satisfy the selector configured in the Prometheus Operator, the Prometheus Server automatically discovers the KEDA Services and periodically scrapes the corresponding `/metrics` endpoints. From this point onward, all operational metrics exposed by the KEDA components become available for querying through Prometheus and, consequently, for visualization within Grafana. Figure 15.1 summarizes the complete workflow enabling the collection and visualization of the operational metrics exposed by the KEDA control plane.

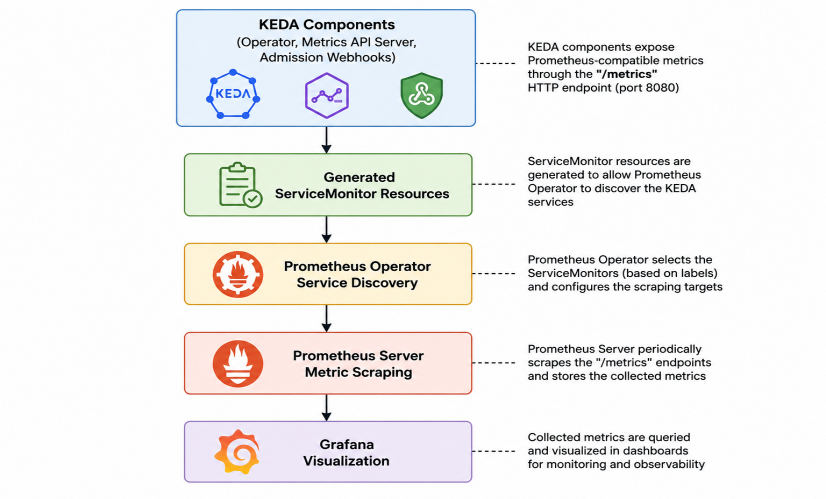


Figure 15.1: Monitoring workflow for KEDA components.

Figure 15.1 illustrates the workflow used to monitor the KEDA components. Prometheus periodically scrapes the metrics exposed by the KEDA Operator, Metrics Adapter, and Admission Webhooks after discovering the corresponding `ServiceMonitor` resources. The collected metrics are then stored in the Prometheus time-series database and visualized through Grafana dashboards, providing real-time insights into the behavior of the KEDA control plane and the autoscaling process.

Scraping Interval
Another important aspect concerns the scraping periodicity adopted by `Prometheus` for collecting the metrics exposed by the KEDA components. The automatically generated `ServiceMonitor` resources do not explicitly define the `interval` field within the `endpoints` section. Consequently, the scraping frequency defaults to the global `scrapeInterval` configured in the Prometheus instance managed by the Prometheus Operator. The configured global scraping interval can be verified by executing:

```
kubectl get prometheus -n monitoring -o yaml | grep scrapeInterval
```

In the adopted setup, the returned value was:

```
1 scrapeInterval: 30s
```

This configuration indicates that the Prometheus Server scrapes the `/metrics` endpoints exposed by the KEDA components every 30 seconds. Therefore, unless an explicit `interval` is specified within the corresponding `ServiceMonitor` definition, all operational metrics exposed by the KEDA components are collected with the same 30-second scraping frequency defined globally by the Prometheus monitoring stack. As a result, these metrics become available in both Prometheus and Grafana with the same refresh granularity.

15.3 Verification of KEDA Metrics Collection

Verification of Active Targets
Once the `ServiceMonitor` resources have been correctly configured, the discovery of the KEDA components by the Prometheus Server can be verified through the Prometheus web interface. The Prometheus Service can first be exposed locally by executing:

```
kubectl port-forward svc/monitoring-kube-prometheus-prometheus \
    -n monitoring 9090:9090
```

The Prometheus Targets page then becomes accessible at:

```
http://localhost:9090/targets
```

At this point, the targets associated with the following KEDA components should appear with status `UP`:

- `keda-operator`
- `keda-admission-webhooks`
- `keda-operator-metrics-apiserver`

As an alternative to the Prometheus web interface, the same verification can be performed directly from the command line through the Prometheus HTTP API. After exposing the Prometheus Service locally via port forwarding, the list of active scraping targets can be retrieved with:

```
curl -s http://localhost:9090/api/v1/targets | \
jq '.data.activeTargets[] | {job: .labels.job, health: .health, url
: .scrapeUrl}'
```

This command queries the Prometheus REST API endpoint responsible for exposing all currently discovered scraping targets together with their health status and scraping endpoint. In the returned output, the targets associated with the KEDA components should appear with the state `"up"`, confirming that the Prometheus Server is successfully discovering and scraping the metrics exposed by the KEDA infrastructure.

Representative KEDA Metrics

Before moving to Grafana-based visualization, it is useful to briefly analyze some of the most representative metrics exposed by the KEDA components and collected by Prometheus. These metrics provide valuable insight into the internal runtime behavior of the KEDA control plane and can be exploited for debugging, monitoring, and operational analysis of the autoscaling process. Among the most relevant metrics are those summarized in Table 15.1:

Metric	Purpose
keda_scaler_active	Indicates whether the scaler associated with a ScaledObject is currently active (1) or inactive (0).
keda_scaler_metrics_value	Reports the latest metric value retrieved by the scaler from the configured external metric source.
keda_scaled_object_errors_total	Counts the errors generated for each ScaledObject.
rest_client_requests_total	Monitors the HTTP REST requests executed by the Metrics API Server toward the Kubernetes API Server.
controller_runtime_webhook_requests_total	Counts the validation requests processed by the Admission Webhooks component.

Table 15.1: Representative metrics exposed by the KEDA components.

The first three metrics are exposed by the KEDA Operator, whereas `rest_client_requests_total` and `controller_runtime_webhook_requests_total` are exposed by the KEDA Metrics API Server and the Admission Webhooks, respectively. Among these metrics, `keda_scaler_active` and `keda_scaler_metrics_value` are particularly useful during experimental validation, as they directly reflect the operational state of the scaler and the metric values used by KEDA to drive autoscaling decisions.

15.3.1 Verification via the Prometheus HTTP API

Before accessing Grafana, the correct scraping behavior can also be verified directly from the terminal through the Prometheus HTTP API. After exposing the Prometheus Service locally via port forwarding, representative metrics exposed by the various KEDA components can be queried using the following commands.

KEDA Operator

```
curl -G http://localhost:9090/api/v1/query \
  --data-urlencode 'query=keda_scaler_active' | jq
```

KEDA Metrics API Server

```
curl -G http://localhost:9090/api/v1/query \
  --data-urlencode 'query=rest_client_requests_total' | jq
```

KEDA Admission Webhooks

```
curl -G http://localhost:9090/api/v1/query \
  --data-urlencode 'query=controller_runtime_webhook_requests_total
' | jq
```

The presence of valid metric vectors in the returned output confirms that the Prometheus Server is correctly scraping the metrics exposed by the main KEDA components and that the monitoring pipeline is operating as expected.

15.4 Visualization through Grafana

Once the metrics exposed by the KEDA components have been successfully collected by Prometheus, they can be visualized through Grafana. The Grafana Service can be exposed locally by executing:

```
kubectl port-forward svc/monitoring-grafana \
    -n monitoring 3000:80
```

The Grafana web interface then becomes accessible at:

```
http://localhost:3000
```

The default username is:

```
admin
```

whereas the initial password can be retrieved through:

```
kubectl get secret monitoring-grafana -n monitoring \
    -o jsonpath="{.data.admin-password}" | base64 --decode
```

Once authenticated, the *Explore* section can be used together with the `Prometheus` data source to execute PromQL queries and visualize the metrics exposed by the KEDA components in real time.

15.5 Chapter Summary

This chapter demonstrated how the operational metrics exposed by the KEDA control plane can be collected by Prometheus and visualized through Grafana, extending the monitoring infrastructure previously adopted for application-level metrics. By enabling the automatic generation of `ServiceMonitor` resources and configuring their discovery by the Prometheus Operator, the metrics exposed by the KEDA Operator, the Metrics API Server, and the Admission Webhooks become available for both programmatic querying and graphical visualization. The resulting monitoring infrastructure enables continuous observation of KEDA's internal behavior and verification that the autoscaling process operates as expected at runtime. In particular, the collected metrics provide valuable information about scaler activity, metric retrieval, and interactions with the Horizontal Pod Autoscaler, enabling the correct operation of the autoscaling pipeline to be validated. Overall, the integration of KEDA, Prometheus, and Grafana provides a complete observability layer for event-driven autoscaling systems, facilitating debugging, operational monitoring, and performance analysis of the autoscaling infrastructure within Kubernetes environments.

Chapter 16

RabbitMQ-Based Event-Driven Autoscaling with KEDA

16.1 Introduction

This chapter presents a second application-level autoscaling scenario aimed at illustrating the event-driven scaling capabilities provided by KEDA through its native RabbitMQ scaler. Similarly to the Prometheus-based NGINX use case discussed in the previous chapter, autoscaling decisions are driven by an application-level workload metric rather than by traditional infrastructure-level resource indicators such as CPU or memory utilization. Unlike the previous scenario, however, the workload is no longer characterized by the HTTP request rate observed by the application and collected through Prometheus queries. Instead, the proposed architecture adopts an asynchronous communication model in which workload requests are buffered inside a distributed message queue managed by RabbitMQ. Autoscaling decisions are therefore based on the number of pending messages waiting to be processed rather than on the request rate itself. More specifically, RabbitMQ is used to decouple workload generation from workload processing by temporarily storing incoming tasks inside a queue. A pool of worker Pods consumes these tasks asynchronously, while KEDA continuously monitors the queue length and dynamically adjusts the number of worker replicas according to the amount of pending work. This architecture is particularly well suited to cloud-native applications implementing asynchronous communication patterns, where workload requests may arrive in bursts and individual processing operations may require significant

execution time. In such scenarios, CPU utilization often represents a delayed and indirect indication of the actual workload pressure, whereas the queue backlog provides a direct measure of the amount of work still awaiting processing. Consequently, queue length represents a more effective scaling metric for event-driven applications than traditional resource utilization metrics.

16.2 RabbitMQ Architecture and Message Flow

RabbitMQ Architecture and AMQP Protocol
RabbitMQ is an open-source distributed message broker implementing the AMQP (Advanced Message Queuing Protocol) standard. AMQP is specifically designed to support reliable asynchronous communication between distributed applications through queues, exchanges, routing mechanisms, acknowledgments, and persistent message delivery semantics. One of the most important characteristics introduced by AMQP is the possibility of multiplexing multiple logical communication channels over a single persistent TCP connection. Instead of establishing one dedicated TCP connection for every producer or consumer interaction, AMQP allows multiple independent logical channels to coexist within the same connection. Each channel behaves as an isolated bidirectional communication stream associated with specific publish, consume, acknowledgment, or queue-management operations. This mechanism significantly reduces networking overhead because establishing and closing TCP connections is considerably more expensive than creating lightweight logical channels. Consequently, producers and consumers can execute multiple independent messaging operations concurrently while sharing the same underlying TCP session with the broker. Conceptually, the TCP connection can be interpreted as a physical communication pipe between the application and RabbitMQ, whereas AMQP channels represent lightweight logical communication streams multiplexed over the same transport connection. Within a Kubernetes cluster, RabbitMQ is deployed as a standard containerized application exposing multiple logical software components operating within the same runtime environment. The most important component is the AMQP server itself, exposed on port 5672, which is responsible for handling producer and consumer connections, queue management, message routing, acknowledgments, and task delivery. A second important component is the RabbitMQ Management Plugin, exposed on port 15672. This plugin provides both a graphical

web interface and a REST-based HTTP API used for broker administration and monitoring activities. Through this interface, users can inspect queues, exchanges, consumers, message rates, backlog statistics, acknowledgment rates, and the overall broker activity in real time. RabbitMQ can also expose a Prometheus-compatible metrics exporter, typically available on port 15692 when the Prometheus plugin is enabled. This component allows a Prometheus server to periodically scrape broker-related metrics through a dedicated ServiceMonitor resource deployed within the Kubernetes cluster. Consequently, RabbitMQ can simultaneously support both Prometheus-based observability workflows and direct KEDA-based autoscaling strategies. In the considered use case, however, the KEDA RabbitMQ scaler does not rely on Prometheus queries or on the HTTP Management API exposed by the Management Plugin. Instead, it directly communicates with the AMQP server through the AMQP protocol on port 5672 to retrieve queue-related information from the broker itself.

Producer, Broker and Consumer Interaction Model

The implemented architecture follows the classical producer–broker–consumer communication paradigm. Producer applications asynchronously publish workload requests into a RabbitMQ queue, whereas consumer applications independently retrieve and process these tasks. More specifically, producers publish messages toward a queue identified by a logical queue name. In the implemented use case, all messages are sent to the queue named `task-queue`. The queue name acts as a logical identifier associated with a specific workload stream managed internally by the broker. RabbitMQ stores all messages belonging to the same queue until they are successfully processed and acknowledged by a consumer. The introduction of a message queue enables temporary workload bursts to be absorbed by the broker, even when the instantaneous processing throughput of the consumers is temporarily insufficient. Consequently, producer applications remain completely decoupled from consumer execution speed, while the autoscaling mechanism progressively adapts the number of worker replicas in order to drain the accumulated backlog. On the consumer side, RabbitMQ adopts a push-based delivery model. Rather than continuously polling the queue searching for new tasks, consumers register a callback function through an AMQP channel and remain passively waiting for incoming message deliveries. Whenever new messages become available and a consumer satisfies the configured Quality of Service (QoS) policy, the broker actively dispatches the corresponding tasks. RabbitMQ also supports a pull-

based consumption model, in which consumers explicitly request new messages through synchronous polling operations. However, the push-based approach was adopted in this work because it provides a more efficient event-driven communication model, reducing unnecessary polling overhead while minimizing message delivery latency. In the implemented configuration, the QoS parameter `prefetch_count=1` ensures that each consumer processes at most one unacknowledged message at a time. Consequently, every consumer replica behaves as a sequential worker capable of processing only one task concurrently. This aspect is particularly relevant for the autoscaling strategy because the overall processing throughput directly depends on the number of active consumer replicas. The lifecycle of each message is managed directly by RabbitMQ through multiple internal states. Newly published messages initially enter the *Ready* state, meaning that they are waiting inside the queue and have not yet been assigned to any consumer. Once RabbitMQ dispatches a message to a consumer, the message transitions to the *Unacked* state, indicating that processing has started but has not yet been acknowledged. After completing the assigned task, the consumer explicitly sends an acknowledgment (ACK) to the broker through the AMQP channel. Only after receiving this acknowledgment does RabbitMQ permanently remove the message from the queue. Conversely, if the consumer crashes, disconnects, or terminates before sending the ACK, RabbitMQ automatically restores the corresponding message to the *Ready* state, allowing it to be reassigned to another available consumer replica. This mechanism guarantees reliable at-least-once delivery semantics while preventing message loss during failures. The message lifecycle described above is illustrated in Figure 16.1:

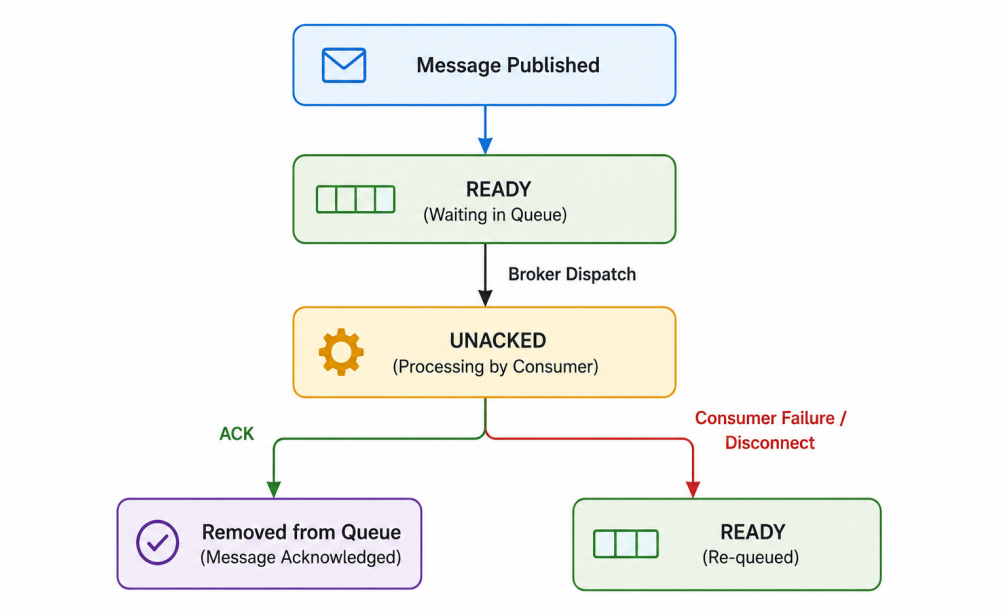


Figure 16.1: RabbitMQ message lifecycle.

For example, assume that the queue initially contains 10 messages in the *Ready* state and that two consumer replicas are currently active. Since the consumers are configured with `prefetch_count=1`, RabbitMQ can assign at most one unacknowledged message to each consumer simultaneously. Consequently, the broker dispatches two messages, resulting in the following queue state:

- 8 messages in the *Ready* state
- 2 messages in the *Unacked* state

If both consumers successfully complete their processing tasks and transmit the corresponding ACK messages, RabbitMQ permanently removes the processed messages from the queue, leaving:

- 8 messages in the *Ready* state
- 0 messages in the *Unacked* state

Conversely, assume that one consumer crashes before sending its ACK message. RabbitMQ automatically restores the corresponding task from the *Unacked* state to the *Ready* state, obtaining:

- 9 messages in the *Ready* state
- 1 message in the *Unacked* state

The restored message can then be reassigned to another available consumer replica, ensuring reliable task processing while preventing message loss.

16.3 RabbitMQ Scaler and Autoscaling Workflow

The autoscaling workflow implemented through the KEDA RabbitMQ scaler is based on the number of queue messages currently in the *Ready* state, namely the tasks still waiting to be assigned to consumers. This quantity directly represents the current workload backlog accumulated inside the broker. As already discussed for the previous KEDA examples, the RabbitMQ scaler periodically connects to the RabbitMQ AMQP server through the AMQP protocol on port 5672 in order to retrieve the current number of messages in the *Ready* state. Once collected, this metric is exposed to the Kubernetes Horizontal Pod Autoscaler through the KEDA Metrics Adapter. More specifically, the KEDA Metrics Adapter integrates with the Kubernetes API Server through the Aggregation API Layer, allowing external metrics generated by KEDA scalers to become accessible through the standard Kubernetes External Metrics API. Consequently, the HPA periodically queries the Kubernetes API Server, which transparently forwards the request to the KEDA Metrics Adapter. The HPA can then retrieve the queue-related workload metric and apply its standard scaling algorithm to compute the desired number of consumer replicas. A particularly important aspect of the RabbitMQ scaler is the way in which it adapts the HPA scaling semantics. KEDA automatically configures the generated HPA metric type as `AverageValue` and sets the HPA target equal to the `value` parameter specified inside the `ScaledObject` manifest. In this use case, such parameter represents the desired average number of waiting queue messages tolerated by each consumer replica. Consequently, the HPA first normalizes the total number of *Ready* messages by the current number of active replicas before applying its scaling formula:

$$desiredReplicas = \left\lceil currentReplicas \times \frac{currentMetricValue}{targetMetricValue} \right\rceil$$

Since the metric type is configured as `AverageValue`, the metric exposed to the HPA becomes:

$$currentMetricValue = \frac{readyMessages}{currentReplicas}$$

while the target value corresponds to:

$$targetMetricValue = value$$

By substituting the normalized metric into the HPA scaling formula, the resulting desired number of replicas becomes:

$$desiredReplicas = \left\lceil \frac{readyMessages}{value} \right\rceil$$

Therefore, the final scaling behavior is coherent with the queue-processing semantics of the application. The `value` parameter does not represent the maximum number of messages that a single consumer replica can process in parallel. Instead, it defines the acceptable backlog threshold tolerated for each worker Pod before additional replicas should be created. In the implemented scenario, each consumer processes tasks sequentially because `prefetch_count=1` limits concurrent processing to a single message at a time. Nevertheless, a target value greater than one can still be configured in order to tolerate a limited queue backlog per replica before triggering additional scaling operations. Consequently, the autoscaling process becomes proportional to the amount of pending workload accumulated inside the queue rather than to the instantaneous parallelism capability of individual consumers.

Overall Autoscaling Workflow

Producer applications continuously publish workload messages into the RabbitMQ queue, causing the number of *Ready* messages to increase whenever the incoming production rate exceeds the aggregate processing throughput of the active consumers. The RabbitMQ scaler periodically retrieves the queue backlog metric from the broker through AMQP and exposes it to the HPA as an external metric. The HPA then computes the desired number of replicas according to its standard scaling formula, using the `AverageValue` configuration together with the configured stabilization windows and scaling policies. As additional consumer replicas become active, the aggregate processing throughput increases, progressively reducing the queue backlog and eventually triggering

gradual scale-down operations once the workload decreases again. The overall interaction between RabbitMQ, KEDA and the HPA is summarized in Figure 16.2.

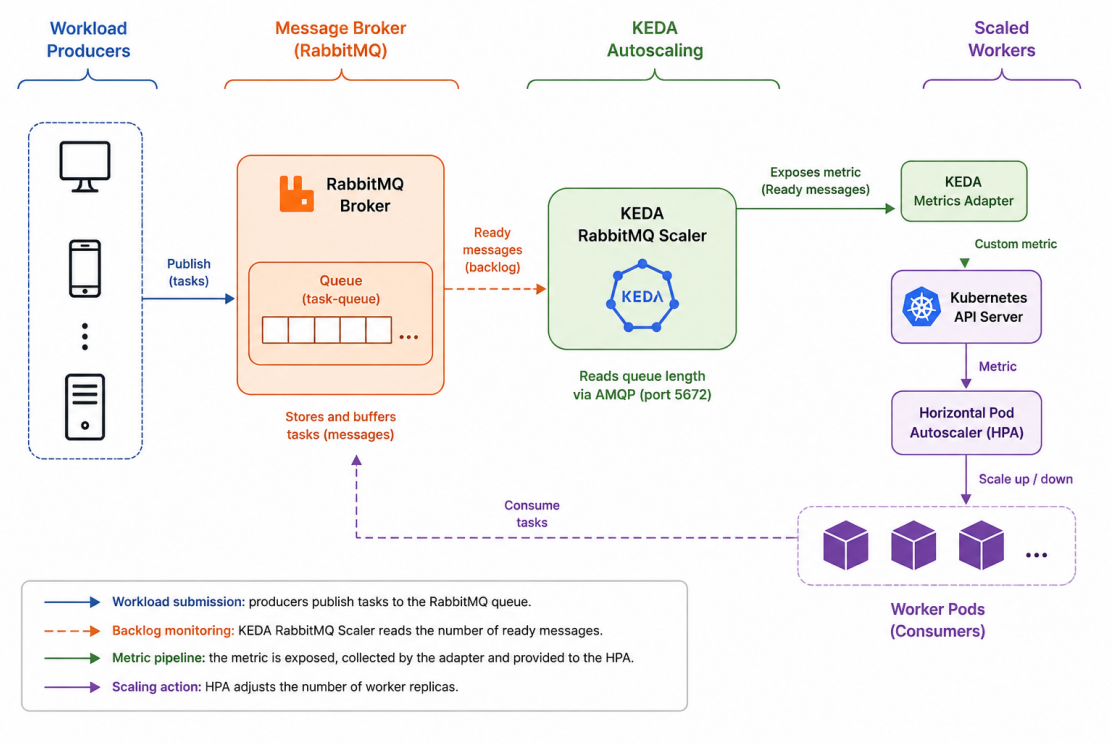


Figure 16.2: RabbitMQ-based autoscaling workflow with KEDA.

Scale-to-Zero and Cooldown Period

Similarly to the previous KEDA scenarios, the RabbitMQ scaler also supports scale-to-zero capabilities. When no consumer replicas are active, the standard HPA can no longer operate because no Pods exist from which metrics can be normalized. In this condition, KEDA temporarily takes direct control of the scaling process and autonomously reactivates the Deployment whenever new messages appear inside the queue. Once at least one consumer replica becomes active again, control is delegated back to the HPA, which resumes the standard autoscaling workflow. Another particularly important aspect in this scenario is represented by the `cooldownPeriod` parameter configured inside the KEDA `ScaledObject`. This parameter defines the amount of time KEDA

waits after the workload metric becomes inactive, namely equal to zero, before directly scaling the Deployment down to zero replicas. This behavior is especially relevant in queue-based workloads because temporary fluctuations in the number of pending messages may occur frequently during normal operation. Without a proper cooldown interval, the system could continuously oscillate between zero and one replica, causing unnecessary Pod recreations and increasing message-processing latency. More specifically, once the queue becomes empty and the number of *Ready* messages remains equal to zero for a sufficiently long period, the HPA first performs gradual scale-down operations according to its stabilization windows and scale-down policies until the Deployment reaches a single active replica. Only after the configured KEDA `cooldownPeriod` expires does KEDA bypass the HPA and directly scale the Deployment from one replica to zero replicas. Conversely, when new messages appear while no consumers are active, KEDA immediately bypasses the HPA again and directly scales the Deployment from zero to one replica, thus reactivating the standard autoscaling workflow. All these aspects concerning KEDA direct intervention during scale-to-zero transitions, HPA interaction, stabilization windows, scaling policies and cooldown period behavior will be analyzed experimentally and discussed in detail in the next chapter dedicated to the practical implementation and evaluation of the RabbitMQ autoscaling demo.

16.4 Chapter Summary

This chapter introduced the RabbitMQ-based event-driven autoscaling model supported by KEDA, highlighting the differences with the previously discussed Prometheus-based autoscaling approach. Unlike request-rate driven scenarios, autoscaling decisions are based on the queue backlog, which directly represents the amount of pending work awaiting processing and therefore provides a more accurate workload indicator for asynchronous applications. The chapter first described the RabbitMQ architecture and the AMQP communication model, illustrating how producers, the broker, and consumers interact through reliable message queues and acknowledgment mechanisms. Particular attention was devoted to the lifecycle of queue messages and to the distinction between the *Ready* and *Unacked* states, which guarantee reliable at-least-once message delivery. The integration between RabbitMQ and KEDA was then analyzed by describing the operation of the native RabbitMQ scaler and its interaction

with the Kubernetes Horizontal Pod Autoscaler. In particular, the chapter discussed how KEDA periodically retrieves the queue backlog, exposes it through the Kubernetes External Metrics API, and enables the HPA to dynamically adapt the number of consumer replicas according to the desired backlog threshold. Finally, the mechanisms governing scale-to-zero transitions and the role of the `cooldownPeriod` parameter were presented, providing the theoretical foundations required for the experimental implementation discussed in the next chapter.

Chapter 17

Experimental Evaluation of RabbitMQ-Based Autoscaling with KEDA

17.1 Experimental Objective

This section presents the practical implementation of the RabbitMQ-based event-driven autoscaling scenario introduced in the previous chapter. The objective of the experiment is to validate the complete integration between RabbitMQ, KEDA and the Kubernetes Horizontal Pod Autoscaler by dynamically scaling a pool of consumer Pods according to the workload accumulated inside a RabbitMQ queue. The experimental setup consists of the following main components:

- A RabbitMQ broker deployed inside the Kubernetes cluster
- A producer application continuously generating workload messages
- A scalable consumer deployment processing queued tasks
- A KEDA `ScaledObject` dynamically scaling the consumer replicas according to the queue backlog.

The following sections describe the deployment procedure, application implementation, containerization process, autoscaling configuration and experimental evaluation adopted throughout this work.

RabbitMQ Deployment

The first step consists of creating a dedicated namespace used to isolate all RabbitMQ-related resources from the remaining cluster components:

```
kubectl create namespace rabbitmq
```

RabbitMQ is then deployed through the manifest file:

rabbitmq.yaml

```
apiVersion: apps/v1
kind: Deployment

metadata:
  name: rabbitmq
  namespace: rabbitmq

spec:
  replicas: 1

  selector:
    matchLabels:
      app: rabbitmq

  template:
    metadata:
      labels:
        app: rabbitmq

    spec:
      containers:
      - name: rabbitmq

        image: rabbitmq:3-management

        ports:
        - containerPort: 5672
        - containerPort: 15672

        env:
        - name: RABBITMQ_DEFAULT_USER
```

```
          value: user

        - name: RABBITMQ_DEFAULT_PASS
          value: password

---
apiVersion: v1
kind: Service

metadata:
  name: rabbitmq
  namespace: rabbitmq

spec:
  selector:
    app: rabbitmq

  ports:
  - name: amqp
    port: 5672
    targetPort: 5672

  - name: management
    port: 15672
    targetPort: 15672
```

The manifest defines both the RabbitMQ Deployment and the corresponding Kubernetes Service required to expose the broker inside the cluster. The Deployment creates a single RabbitMQ broker replica within the dedicated namespace:

```
replicas: 1
```

Since the purpose of the experiment is to evaluate the autoscaling behavior of the consumer applications rather than broker fault tolerance, a single RabbitMQ instance is sufficient for the considered use case. The Deployment uses the official RabbitMQ image including the Management Plugin:

```
image: rabbitmq:3-management
```

This image automatically enables both the AMQP server responsible for message handling and the RabbitMQ Management Plugin used for administration and monitoring activities. Two container ports are explicitly exposed:

```
ports:
- containerPort: 5672
- containerPort: 15672
```

Port **5672** exposes the AMQP server, which manages producer and consumer connections, AMQP channels, queue operations, message routing, acknowledgments and task delivery. Both the producer/consumer applications and the KEDA RabbitMQ scaler communicate directly with this endpoint through the AMQP protocol. Port **15672** exposes the RabbitMQ Management Plugin, providing both a graphical web interface and a REST API for administration and debugging. Through this interface it is possible to inspect queue states, connected consumers, message statistics, acknowledgment rates, the number of *Ready* and *Unacked* messages, and the overall broker activity in real time. The default authentication credentials are configured through the following environment variables:

```
env:
- name: RABBITMQ_DEFAULT_USER
  value: user

- name: RABBITMQ_DEFAULT_PASS
  value: password
```

These credentials are subsequently used by the producer application, the consumer applications, the KEDA RabbitMQ scaler and the RabbitMQ Management UI. The second part of the manifest defines the Kubernetes Service associated with the Deployment. The Service exposes the broker through stable virtual networking endpoints that can be reached by any Pod inside the cluster using the internal DNS name:

```
rabbitmq.rabbitmq.svc.cluster.local
```

The Service exposes two ports:

- Port **5672**, corresponding to the AMQP server
- Port **15672**, corresponding to the RabbitMQ Management Plugin

In both cases, the Service forwards incoming traffic received on the Service port to the corresponding container port exposed by the RabbitMQ Pod. The complete manifest can then be deployed through:

```
kubectl apply -f rabbitmq.yaml
```

The status of the deployed resources can be verified with:

```
kubectl get pods -n rabbitmq
kubectl get svc -n rabbitmq
```

Finally, the RabbitMQ Management UI can be temporarily exposed outside the cluster for monitoring and debugging purposes using Kubernetes port forwarding:

```
kubectl port-forward svc/rabbitmq 15672:15672 -n rabbitmq
```

The graphical interface then becomes accessible through:

```
http://localhost:15672
```

17.2 Consumer and Producer Implementation

Consumer Application Implementation
The consumer application is responsible for asynchronously processing workload messages delivered by the RabbitMQ broker. Unlike the producer component, which simply generates queue tasks, the consumer implements the actual task-processing workflow and directly influences the autoscaling dynamics of the entire system, since the aggregate processing throughput depends on the number of active consumer replicas. The adopted Python implementation is reported below:

consumer.py

```
1 import pika # Python AMQP client library used to interact
      with the RabbitMQ broker.
2 import time # Standard Python module used to introduce
     delays between message publications.
3
4 # Create a blocking TCP connection toward the RabbitMQ
     broker
```

```
# exposed through the Kubernetes Service DNS name.
connection = pika.BlockingConnection(
    pika.ConnectionParameters(
        host='rabbitmq.rabbitmq.svc.cluster.local',
        port=5672, # Default port number if not specified: 5672
        credentials=pika.PlainCredentials('user', 'password')
    )
)

# Create an AMQP logical channel over the TCP connection.
channel = connection.channel()

# Declare the queue if it does not already exist.
# Producer and consumers must reference the same queue name.
channel.queue_declare(queue='task-queue')

# Callback function automatically executed whenever
# a new message is delivered to the consumer.
def callback(ch, method, properties, body):
    # Print the received message payload.
    print(f"Received: {body.decode()}")

    # Simulate message processing workload.
    # Each consumer processes one message in about 5 seconds.
    time.sleep(5)

    # Send an explicit ACK to RabbitMQ after 5 seconds processing completion.
    # Achieved by retrieving the delivery tag from method (metadata),
    # which is a unique id of that message in the queue.
    # The Rabbitmq broker can now safely remove the message from the queue.
    ch.basic_ack(delivery_tag=method.delivery_tag)

```

```
# Configure RabbitMQ QoS policy.
#
# prefetch_count=1 ensures that each consumer receives
# at most one unacknowledged message at a time.
#
# This prevents a single consumer from receiving multiple
# concurrent tasks from the Rabbitmq broker before
    completing the current one.
channel.basic_qos(prefetch_count=1)

# Register the consumer and the callback function
    associated with the queue.
#
# Whenever a new message becomes available in the queue,
    the RabbitMQ broker
# delivers it to one registered consumer. Once the
    message reaches the consumer
# side, Pika automatically invokes the callback function
    by populating its
# parameters with the corresponding AMQP metadata and
    message payload.
channel.basic_consume(
    queue='task-queue',
    on_message_callback=callback
)

# Print startup status information.
print("Waiting for messages...")

# Start the infinite consumer loop.
#
# The consumer now continuously waits for incoming
    messages in listening mode
# and processes them asynchronously through the callback
    function.
channel.start_consuming()
```

Listing 17.1: RabbitMQ consumer implementation.

The application initially establishes a persistent TCP connection toward the

RabbitMQ broker through the Kubernetes internal DNS name:

```
connection = pika.BlockingConnection(...)
```

More specifically, the connection targets the AMQP server exposed on port 5672, which is responsible for queue management, message routing, acknowledgments and producer–consumer communication. Once the TCP connection has been established, the application creates an AMQP logical channel:

```
channel = connection.channel()
```

As discussed in the previous chapter, AMQP channels represent lightweight bidirectional communication streams multiplexed over a single TCP connection. This mechanism avoids the overhead associated with creating multiple independent TCP sessions while still allowing logically isolated messaging operations. The consumer then declares the queue:

```
channel.queue_declare(queue='task-queue')
```

If the queue does not already exist, RabbitMQ automatically creates it; otherwise, the existing queue is reused. Consequently, the queue becomes the shared asynchronous communication buffer between producers and consumers. The actual processing logic is implemented through the callback function:

```
def callback(ch, method, properties, body):
```

This function is automatically invoked by the Pika client library whenever RabbitMQ delivers a new message to the consumer. The callback parameters are populated with the AMQP metadata associated with the delivered message, including: `ch`, representing the AMQP communication channel; `method`, containing the delivery metadata generated by RabbitMQ; `properties`, storing optional message-level metadata; `body`, containing the actual workload payload produced by the producer application. After receiving the message, the payload is decoded and printed:

```
print(f"Received: {body.decode()}")
```

The consumer then simulates workload execution by introducing an artificial processing delay:

```
time.sleep(5)
```

Each consumer replica therefore requires approximately five seconds to process a single task. This artificial delay intentionally limits the processing throughput of every worker, allowing the RabbitMQ queue backlog to grow during the experiments and making the autoscaling behavior observable. Once processing has been completed, the consumer explicitly sends an acknowledgment (ACK) to RabbitMQ:

```
ch.basic_ack(delivery_tag=method.delivery_tag)
```

The `delivery_tag` uniquely identifies the delivered message within the AMQP channel and enables RabbitMQ to permanently remove the corresponding task from the queue. If the consumer terminates before sending the ACK, RabbitMQ automatically restores the message to the *Ready* state, ensuring reliable at-least-once delivery semantics. The following Quality of Service (QoS) policy is then configured:

```
channel.basic_qos(prefetch_count=1)
```

The parameter `prefetch_count=1` ensures that RabbitMQ assigns at most one unacknowledged message at a time to each consumer replica. Consequently, the broker waits for the ACK associated with the current task before dispatching a new one to the same consumer. As a result, every consumer Pod behaves as a sequential worker capable of processing only one task concurrently. This configuration plays a fundamental role in the autoscaling strategy because the overall processing capacity becomes directly proportional to the number of active consumer replicas. For example:

- 1 consumer replica processes approximately one message every 5 seconds
- 5 consumer replicas process approximately five messages every 5 seconds
- 10 consumer replicas process approximately ten messages every 5 seconds

Consequently, whenever the producer generates workload messages faster than the aggregate consumer throughput, the number of messages in the *Ready* state progressively increases, eventually triggering scale-out operations through the KEDA RabbitMQ scaler. The consumer is then registered to the queue through:

```
channel.basic_consume(
    queue='task-queue',
```

```
    on_message_callback=callback
)
```

At this stage, the consumer subscribes to the queue and enters a passive listening state. RabbitMQ therefore adopts the push-based delivery model previously discussed, actively dispatching new messages whenever the consumer becomes eligible to receive additional tasks according to the configured QoS policy. Finally, the following instruction starts the infinite event-processing loop:

```
channel.start_consuming()
```

The consumer continuously waits for incoming messages and asynchronously processes them through the callback mechanism for the entire lifetime of the Pod replica.

Consumer Containerization

The consumer application is containerized through the Dockerfile reported in Listing 17.2:

Dockerfile

```
FROM python:3.11-slim

RUN pip install pika

COPY consumer.py /consumer.py

CMD ["python", "/consumer.py"]
```

Listing 17.2: Dockerfile used to build the RabbitMQ consumer image.

The Dockerfile is intentionally minimal. It uses the lightweight `python:3.11-slim` image as the execution environment, installs the Pika library required for AMQP communication with RabbitMQ, copies the consumer application into the container filesystem and finally executes the Python script when the container starts. Once the Dockerfile has been created, the container image can be locally built through:

```
docker build -t matteo4400/rabbitmq-consumer:1.0 .
```

In the adopted experimental setup, the generated image was uploaded to the public Docker Hub container registry. This approach allows Kubernetes worker

nodes to automatically retrieve the container image from an external centralized repository during Pod creation. Before uploading an image to Docker Hub, it is necessary to:

- Create a Docker account
- Create or own a Docker Hub repository
- Authenticate the local Docker client using the account credentials

Authentication is performed through:

```
docker login
```

After successful authentication, the local Docker client becomes authorized to push container images to the repositories associated with the authenticated account. The generated image can then be uploaded through:

```
docker push matteo4400/rabbitmq-consumer:1.0
```

where:

- `matteo4400` identifies the Docker Hub username
- `rabbitmq-consumer` represents the repository name
- `1.0` denotes the image version tag

Once the upload procedure completes successfully, the image becomes publicly available through Docker Hub and can be verified from the *Repositories* section associated with the authenticated account. An alternative solution would consist of using a private container registry hosted directly within the Kubernetes infrastructure. However, for the considered experimental scenario, Docker Hub was preferred because of its simplicity and ease of integration. Once uploaded, the image can be automatically retrieved by Kubernetes through the Deployment manifest reported in Listing 17.3:

consumer-deployment.yaml

```
1 apiVersion: apps/v1
2 kind: Deployment
3
4 metadata:
```

```
  name: rabbitmq-consumer

spec:
  replicas: 1

  selector:
    matchLabels:
      app: rabbitmq-consumer

  template:
    metadata:
      labels:
        app: rabbitmq-consumer

    spec:
      containers:
      - name: consumer
        image: matteo4400/rabbitmq-consumer:1.0
        imagePullPolicy: Always
```

Listing 17.3: Deployment manifest for the RabbitMQ consumer application.

The Deployment initially creates a single consumer replica within the default namespace:

```
replicas: 1
```

This initial replica count is subsequently managed dynamically by the KEDA-generated Horizontal Pod Autoscaler during the experimental evaluation. The container image used by each Pod is specified through:

```
image: matteo4400/rabbitmq-consumer:1.0
```

The image reference must exactly match the Docker Hub username, repository name and image tag associated with the uploaded container image. During Pod creation, Kubernetes instructs the underlying container runtime to retrieve the specified image directly from Docker Hub before starting the container instance. The following parameter controls the image retrieval policy:

```
imagePullPolicy: Always
```

This configuration forces Kubernetes to always attempt a new image pull whenever a Pod replica is created, even if the same image version is already cached

locally on the worker node. Such behavior is particularly useful during experimental activities and iterative software development because it guarantees that newly created Pods always execute the latest version associated with the selected image tag. The Deployment can then be created through:

```
kubectl apply -f consumer-deployment.yaml
```

Finally, the status of the deployed resources can be verified through:

```
kubectl get deployment
kubectl get pods
```

Producer Application Implementation

The producer application is responsible for continuously generating workload messages and publishing them to the RabbitMQ queue. Unlike the consumer component, which processes tasks already stored inside the broker, the producer acts as the workload generator of the entire experimental scenario. The adopted implementation is reported in Listing 17.4:

producer.py

```
import pika # Python AMQP client library used to interact with the RabbitMQ broker.
import time # Standard Python module used to introduce delays between message publications.

# Create a blocking TCP connection toward the RabbitMQ broker
# exposed through the Kubernetes Service DNS name.
connection = pika.BlockingConnection(
    pika.ConnectionParameters(
        host='rabbitmq.rabbitmq.svc.cluster.local',
        port=5672, # Default port number if not specified: 5672
        credentials=pika.PlainCredentials('user', 'password')
    )
)

# Create an AMQP logical channel over the TCP connection.
channel = connection.channel()
```

```
16
17 # Declare the queue if it does not already exist.
18 # Producer and consumers must reference the same queue
       name.
19 channel.queue_declare(queue='task-queue')
20
21 # Local counter used to generate progressive message
       identifiers.
22 i = 0
23
24 # Infinite loop continuously generating workload messages
       .
25 while True:
26
27     # Create the payload associated with the current
           message.
28     msg = f"message-{i}"
29
30     # Publish the message through the default RabbitMQ
           exchange.
31     # The routing_key matches the destination queue name.
32     channel.basic_publish(
33         exchange='',
34         routing_key='task-queue',
35         body=msg
36     )
37
38     # Print the produced message for debugging purposes.
39     print(f"Sent {msg}")
40
41     # Increment message identifier.
42     i += 1
43
44     # Wait 0.2 seconds before sending the next message
45     # (~5 messages per second generation rate).
46     time.sleep(0.2)
```

Listing 17.4: RabbitMQ producer implementation.

Similarly to the consumer application, the producer initially establishes a per-

sistent TCP connection toward the RabbitMQ broker through the Kubernetes internal DNS service. More specifically, the connection targets the AMQP server exposed on port 5672, which is responsible for message publishing and queue management operations. Once the connection has been established, the application creates an AMQP logical channel through which all subsequent queue operations are performed. As previously discussed, AMQP channels represent lightweight bidirectional communication streams multiplexed over a single TCP connection, allowing multiple messaging operations without requiring additional TCP sessions. The producer then declares the destination queue. If the queue does not already exist, RabbitMQ automatically creates it; otherwise, the existing queue is reused. Consequently, both producer and consumer applications reference the same logical queue, enabling asynchronous communication through the broker. A local counter is initialized to generate progressive message identifiers. The workload generation process is then implemented through an infinite loop that continuously creates and publishes new messages. During each iteration, the producer:

- Generates a new payload in the form `message-`i
- Publishes the message to RabbitMQ using `channel.basic_publish()`
- Prints the generated message for debugging purposes
- Increments the local message identifier
- Waits 200 ms before generating the next workload message

Message publication is performed through the default RabbitMQ exchange, while the destination queue is identified by the routing key:

- `exchange=''` selects the default RabbitMQ exchange
- `routing_key='task-queue'` identifies the destination queue
- `body=msg` contains the message payload

Once published, each message becomes managed by the RabbitMQ broker and immediately enters the queue lifecycle discussed in the previous chapter. Depending on the current system state:

- If no consumer replicas are available, the message remains in the *Ready* state

- If an eligible consumer exists, RabbitMQ dispatches the message according to the configured QoS policy
- Once assigned to a consumer but not yet acknowledged, the message transitions to the *Unacked* state

Finally, the producer waits 200 ms before generating the next workload message. This configuration produces approximately five messages per second, intentionally exceeding the processing throughput of a single consumer replica configured with a five-second processing delay. Consequently, the queue backlog progressively increases, triggering the KEDA-based autoscaling mechanism analyzed throughout the experimental evaluation.

Producer Containerization

The producer application is containerized through the Dockerfile reported in Listing 17.5:

Dockerfile

```
FROM python:3.11-slim

RUN pip install pika

COPY producer.py /producer.py

CMD ["python", "/producer.py"]
```

Listing 17.5: Dockerfile used to build the RabbitMQ producer image.

Since the containerization workflow is identical to that previously described for the consumer application, only the main commands are reported below. The Docker image can be built through:

```
docker build -t matteo4400/rabbitmq-producer:1.0 .
```

After authenticating with Docker Hub:

```
docker login
```

the image can be uploaded to the remote repository through:

```
docker push matteo4400/rabbitmq-producer:1.0
```

Unlike the consumer application, no Kubernetes `Deployment` resource was created for the producer component. This design choice was intentional because the producer is not subject to autoscaling operations and is only required during workload generation phases. Consequently, instead of deploying the producer as a persistent Kubernetes Deployment, the producer container is dynamically instantiated as a temporary Pod during the experimental evaluation by means of the `kubectl run` command discussed later in the testing section. Conversely, the consumer application necessarily requires a Kubernetes Deployment because the Horizontal Pod Autoscaler and KEDA can only manage scalable workload controllers exposing a mutable `replicas` field. Running the producer as an ephemeral Pod also simplifies the experimental workflow by avoiding unnecessary cluster resources when workload generation is not required. The different deployment strategies adopted for the producer and consumer components are summarized in Table 17.1:

Aspect	**Producer**	**Consumer**
Kubernetes resource	Ephemeral Pod	Deployment
Managed by KEDA/HPA	No	Yes
Purpose	Workload generation	Task processing

Table 17.1: Comparison of the deployment strategies adopted for the producer and consumer components.

17.3 KEDA Configuration

The autoscaling behavior of the consumer deployment is implemented through the KEDA `ScaledObject` resource reported in Listing 17.6:

scaledobject.yaml

```
1 apiVersion: keda.sh/v1alpha1
2 kind: ScaledObject
3
4 metadata:
5   name: rabbitmq-scaledobject
6
7 spec:
8
```

```
  scaleTargetRef:
    name: rabbitmq-consumer

  minReplicaCount: 0
  maxReplicaCount: 10

  pollingInterval: 15

  cooldownPeriod: 120

  advanced:
    horizontalPodAutoscalerConfig:
      behavior:

        scaleUp:
          stabilizationWindowSeconds: 0

        scaleDown:
          stabilizationWindowSeconds: 60

  triggers:
  - type: rabbitmq

    metadata:
      protocol: amqp
      queueName: task-queue
      mode: QueueLength
      value: "5"
      activationValue: "0"
      host: amqp://user:password@rabbitmq.rabbitmq.svc.cluster.local
    :5672/
```

Listing 17.6: KEDA ScaledObject used for RabbitMQ-based autoscaling.

The `ScaledObject` associates the RabbitMQ scaler with the consumer Deployment and defines all the parameters governing the autoscaling process, including the target workload, the minimum and maximum number of replicas, polling frequency, scale-to-zero behavior, HPA policies and the trigger configuration used to retrieve queue-related metrics from the RabbitMQ broker. Unlike the previous Prometheus-based scenario, no PromQL query is required

in this case. Instead, the RabbitMQ scaler directly connects to the broker through the AMQP protocol and periodically retrieves the number of pending messages contained in the configured queue. The collected metric is subsequently exposed to the Kubernetes Horizontal Pod Autoscaler through the KEDA Metrics Adapter, allowing the HPA to compute the desired number of consumer replicas according to the standard autoscaling algorithm. The role of each configuration parameter is discussed in detail in the following paragraphs. The configuration is analyzed by first considering the parameters defining the target workload and the general autoscaling behavior, followed by the RabbitMQ-specific trigger configuration. The following field identifies the Kubernetes workload dynamically managed by KEDA and the generated Horizontal Pod Autoscaler:

```
scaleTargetRef:
  name: rabbitmq-consumer
```

In this configuration, the scaling target corresponds to the consumer Deployment introduced in the previous section. Consequently, KEDA dynamically updates the `replicas` field of this Deployment according to the workload accumulated in the RabbitMQ queue. The autoscaling boundaries are defined through:

```
minReplicaCount: 0
maxReplicaCount: 10
```

The lower bound was intentionally configured to zero in order to enable scale-to-zero capabilities. Consequently, when the queue remains empty for a sufficiently long period, all consumer replicas can be removed from the cluster, eliminating unnecessary resource consumption. Conversely, the upper bound limits the deployment to at most ten consumer replicas, preventing uncontrolled resource allocation during workload bursts. The polling frequency adopted by the RabbitMQ scaler is configured through:

```
pollingInterval: 15
```

Every 15 seconds, the RabbitMQ scaler connects to the RabbitMQ AMQP server and retrieves the current queue metric used for autoscaling decisions. This value was intentionally aligned with the default HPA reconciliation period of 15 seconds in order to simplify the interpretation of the autoscaling behavior during the experimental evaluation. The following parameter defines the KEDA cooldown interval:

```
cooldownPeriod: 120
```

This parameter becomes particularly relevant when scale-to-zero is enabled. After the queue becomes empty, KEDA does not immediately scale the Deployment from one replica to zero. Instead, it waits for the configured cooldown interval before directly performing the final scale-to-zero transition. In the adopted experimental setup, the cooldown period was intentionally configured to 120 seconds in order to:

- Avoid aggressive oscillations between zero and one replica
- Allow the HPA stabilization window to complete its gradual scale-down process before KEDA intervenes
- Clearly observe the interaction between HPA-controlled scaling and KEDA-controlled scale-to-zero behavior during the experimental evaluation

The behavior of the automatically generated Horizontal Pod Autoscaler is customized through the following configuration:

```
advanced:
  horizontalPodAutoscalerConfig:
```

The scale-up stabilization window is explicitly configured as:

```
scaleUp:
  stabilizationWindowSeconds: 0
```

This configuration allows scale-up operations to be applied immediately without waiting for additional metric samples, ensuring a rapid reaction to workload increases. No explicit scale-up policies are defined. Consequently, Kubernetes automatically applies the default HPA scale-up policies, allowing the number of replicas to increase every 15 seconds by the maximum between:

- 100% of the current replica count
- 4 additional replicas

Conceptually, the default configuration corresponds to:

```
policies:
- type: Percent
  value: 100
  periodSeconds: 15

- type: Pods
  value: 4
  periodSeconds: 15
```

where:

- `type:Percent` enables proportional scaling
- `type:Pods` allows adding a fixed number of replicas independently of the current deployment size.

The HPA automatically selects the most aggressive policy. For example:

- Starting from one replica, the `+4 Pods` policy dominates, resulting in five replicas
- Starting from five replicas, the `+100%` policy dominates, resulting in ten replicas

Conversely, the scale-down stabilization window is configured as:

```
scaleDown:
  stabilizationWindowSeconds: 60
```

This parameter is fundamental for preventing unstable oscillatory behavior during workload decreases. During scale-down operations, the HPA continuously stores the replica recommendations generated during the previous 60 seconds and temporarily applies the maximum recommendation instead of immediately adopting the newest lower value. Since the default HPA reconciliation period is 15 seconds, the stabilization window effectively considers approximately the last five recommendation samples:

- Current recommendation
- Recommendation generated 15 seconds earlier
- Recommendation generated 30 seconds earlier

- Recommendation generated 45 seconds earlier
- Recommendation generated 60 seconds earlier

This mechanism intentionally delays aggressive scale-down transitions, resulting in smoother workload stabilization. Similarly to the scale-up case, no explicit scale-down policies are defined. Therefore, Kubernetes automatically adopts the default HPA scale-down policy:

```
policies:
- type: Percent
  value: 100
  periodSeconds: 15
```

Although this policy theoretically allows reducing up to 100% of the current replicas every 15 seconds, in practice the stabilization window dominates the scale-down behavior by delaying aggressive reductions until the configured stabilization interval has expired. The main configuration parameters governing the autoscaling behavior adopted in this experimental setup are summarized in Table 17.2:

Parameter	**Value**	**Purpose**
`minReplicaCount`	0	Enable scale-to-zero
`maxReplicaCount`	10	Limit the maximum number of consumer replicas
`pollingInterval`	15 s	Queue metric polling frequency
`cooldownPeriod`	120 s	Delay before the final scale-to-zero operation
`scaleUp stabilization`	0 s	Immediate scale-out response
`scaleDown stabilization`	60 s	Prevent oscillatory scale-down behavior

Table 17.2: Main KEDA autoscaling parameters adopted in the RabbitMQ experimental setup.

The second part of the `ScaledObject` configuration defines the RabbitMQ trigger used to drive the autoscaling process:

```
triggers:
- type: rabbitmq
```

The communication protocol adopted by the scaler is specified through:

```
protocol: amqp
```

In this configuration, the scaler directly communicates with the RabbitMQ AMQP server exposed on port 5672. Alternatively, an HTTP-based configuration could also be adopted. In that case, the scaler would interact with the RabbitMQ Management Plugin exposed on port 15672 instead of the AMQP server. The monitored queue is identified by:

```
queueName: task-queue
```

The RabbitMQ scaler continuously monitors the number of messages associated with this queue and periodically exposes the corresponding metric to the Kubernetes Horizontal Pod Autoscaler. The operating mode of the scaler is configured through:

```
mode: QueueLength
```

When operating in `QueueLength` mode, the scaler uses the number of queue messages currently in the *Ready* state as the autoscaling metric. Consequently:

- Messages already assigned to consumers (*Unacked*) are excluded from the metric
- Only tasks still waiting to be processed contribute to autoscaling decisions

One of the most important parameters of the entire configuration is:

```
value: "5"
```

This parameter defines the target average number of queue messages tolerated by each consumer replica. KEDA automatically propagates this value to the generated HPA by configuring the metric type as `AverageValue`. Consequently, the autoscaling process becomes proportional to the queue backlog normalized over the number of running consumer replicas. As discussed in the previous chapter, each consumer processes tasks sequentially because:

- `prefetch_count=1` prevents concurrent task assignment
- Each task requires approximately five seconds to complete.

Therefore, the configured value does not represent the true parallel processing capacity of a consumer replica. Instead, it defines the acceptable queue backlog tolerated per replica before additional scaling operations become necessary. The activation threshold is configured through:

```
activationValue: "0"
```

This parameter specifies the minimum metric value required before KEDA reactivates the deployment from the scale-to-zero state. Since the threshold is configured to zero (which is also the default value), the appearance of a single *Ready* message is sufficient to trigger the transition from zero to one consumer replica, after which control is delegated back to the Horizontal Pod Autoscaler. Finally, the RabbitMQ connection string used by the scaler is specified through:

```
host: amqp://user:password@rabbitmq.rabbitmq.svc.cluster.local:5672/
```

Using this connection string, the scaler authenticates directly against the RabbitMQ AMQP server through:

- The Kubernetes internal DNS service name
- The configured authentication credentials
- The AMQP communication endpoint exposed on port 5672

After retrieving the queue metric, KEDA exposes it to the Kubernetes Horizontal Pod Autoscaler through the KEDA Metrics Adapter and the Kubernetes API Aggregation Layer discussed in the previous chapters. The HPA can then execute its standard autoscaling algorithm to dynamically compute the desired number of consumer replicas according to the configured stabilization windows and scaling policies. Before starting the experimental evaluation, the autoscaling configuration is applied through:

```
kubectl apply -f scaledobject.yaml
```

The successful creation of the `ScaledObject` can be verified through:

```
kubectl get scaledobject
```

while the automatically generated Horizontal Pod Autoscaler can be inspected through:

```
kubectl get hpa
```

17.4 Experimental Evaluation

Once all infrastructure components have been deployed, workload generation can be initiated by dynamically creating the producer Pod directly from the previously uploaded Docker Hub image:

```
kubectl run rabbitmq-producer \
  --image=matteo4400/rabbitmq-producer:1.0 \
  --image-pull-policy=Always
```

Unlike the consumer application, the producer is instantiated as a standalone Pod rather than through a Kubernetes Deployment. This design choice is intentional because the producer is only required to artificially generate workload during the experimental evaluation and therefore does not need to be managed by either the Horizontal Pod Autoscaler or KEDA. The parameter:

```
1 --image=matteo4400/rabbitmq-producer:1.0
```

specifies the Docker Hub image used to instantiate the producer container, while

```
1 --image-pull-policy=Always
```

forces Kubernetes to always retrieve the latest image version from Docker Hub before creating the Pod, even if the image is already cached locally on the worker node. Once started, the producer immediately begins publishing workload messages at approximately five messages per second, rapidly increasing the number of *Ready* messages inside the RabbitMQ queue. Owing to:

- The intentionally high producer generation rate
- The sequential consumer processing model
- The five-second processing delay configured within each consumer replica

the queue backlog grows rapidly. Consequently, only a short producer execution time is sufficient to generate enough workload to experimentally observe the entire autoscaling process. For this reason, the producer Pod can be manually removed shortly after startup through:

```
kubectl delete pod rabbitmq-producer
```

Once the producer has been terminated:

- No new workload messages are generated
- The consumer replicas continue processing the remaining queue backlog
- The Horizontal Pod Autoscaler progressively performs scale-down operations
- KEDA eventually executes the final scale-to-zero transition after the configured `cooldownPeriod` expires.

The autoscaling process can be continuously monitored through:

```
kubectl get hpa -w
```

This command continuously displays:

- The current external metric value
- The configured target value
- The current number of replicas
- The autoscaling decisions computed by the Horizontal Pod Autoscaler

Similarly, Pod creation and termination events can be monitored through:

```
kubectl get pods -w
```

Finally, queue statistics together with the *Ready* and *Unacked* message states can be inspected directly from the RabbitMQ Management UI throughout the workload generation and processing phases. The autoscaling behavior of the consumer deployment was experimentally analyzed by continuously monitoring the Horizontal Pod Autoscaler through:

```
kubectl get hpa -w
```

Throughout the experiment, the generated HPA retained the name keda-hpa-rabbitmq-scaledobject, while the minimum and maximum replica limits remained fixed at one and ten replicas, respectively. Since these values remained constant during the entire experiment, they have been omitted from the reported outputs for the sake of brevity. The representative output collected during the experimental evaluation is reported in Listing 17.7:

```
TARGETS                 REPLICAS
<unknown>/5 (avg)       0
52/5 (avg)              1
22400m/5 (avg)          5
15900m/5 (avg)          10
16300m/5 (avg)          10
13300m/5 (avg)          10
10300m/5 (avg)          10
7300m/5 (avg)           10
4300m/5 (avg)           10
1300m/5 (avg)           10
0/5 (avg)               10
0/5 (avg)               10
0/5 (avg)               9
0/5 (avg)               3
0/5 (avg)               1
<unknown>/5 (avg)       0
```

Listing 17.7: Representative HPA output collected during the RabbitMQ autoscaling experiment.

Initially, the consumer Deployment was completely scaled down to zero replicas because no messages were present inside the RabbitMQ queue:

```
<unknown>/5 (avg)     ...     REPLICAS 0
```

Under these conditions, no consumer Pods were running inside the cluster. Consequently:

- No HPA metric normalization could be performed
- The metric appeared as `<unknown>` from the HPA perspective
- The standard HPA control loop was temporarily unable to operate.

At this stage, KEDA directly controls the scaling process. As soon as the producer starts publishing workload messages, the RabbitMQ scaler detects the presence of *Ready* messages inside the queue. More specifically, KEDA continuously compares the retrieved queue metric with the configured `activationValue` defined in the `ScaledObject`:

```
activationValue: "0"
```

Since the activation threshold is equal to zero, the presence of a single *Ready* message is sufficient to reactivate the Deployment. In this phase, KEDA temporarily bypasses the Horizontal Pod Autoscaler and directly updates the Deployment replica count:

$$0 \rightarrow 1$$

Once one consumer replica becomes available, metric normalization is again possible and control of the autoscaling process is delegated back to the HPA, which resumes the standard scale-up and scale-down workflow according to the configured stabilization windows and scaling policies. This transition becomes visible in the following output:

```
52/5 (avg)      ...      REPLICAS 1
```

At this stage:

- One consumer replica is active
- The HPA has acquired the external metric exposed by KEDA
- The reported metric corresponds to 52 *Ready* messages currently waiting inside the queue

Since the metric type is configured as `AverageValue`, the displayed value is already normalized with respect to the current number of replicas. Therefore:

$$\text{Current replicas} = 1$$

$$\text{Displayed metric} = 52$$

$$\text{Actual queue backlog} = 52 \text{ messages}$$

The HPA immediately applies its standard scaling formula:

$$desiredReplicas = \left\lceil \frac{1 \times 52}{5} \right\rceil = 11$$

The computed value is then limited by:

- The configured `maxReplicaCount=10`
- The default HPA scale-up policies

Since the Deployment currently contains only one replica, the default *+4 Pods* policy is more aggressive than the proportional *+100%* policy. Consequently, the HPA immediately updates the Deployment from:

$$1 \rightarrow 5$$

Although the scaling decision has already been computed, the current HPA output still reports the previous Deployment state because metric acquisition and scaling computation occur during the same reconciliation cycle. The updated replica count therefore becomes visible only during the subsequent HPA refresh interval. This behavior is confirmed by the next output:

```
22400m/5 (avg)      ...      REPLICAS 5
```

At this stage:

- Five consumer replicas are active
- The normalized metric reported by the HPA equals 22.4

Since the metric is normalized with respect to the number of replicas:

$$22.4 \times 5 = 112$$

meaning that approximately 112 *Ready* messages are currently stored inside the RabbitMQ queue. Applying the HPA scaling formula again yields:

$$desiredReplicas = \left\lceil \frac{5 \times 22.4}{5} \right\rceil = 23$$

which is again limited by the configured upper bound of ten replicas. Unlike the previous case, the Deployment now already contains five replicas. Consequently:

- The proportional *+100%* policy allows scaling from 5 to 10 replicas
- The *+4 Pods* policy would only allow scaling from 5 to 9 replicas

Therefore, the proportional policy becomes dominant and the Deployment immediately scales to the configured upper limit:

$$5 \rightarrow 10$$

This transition becomes visible during the next reconciliation cycle:

```
15900m/5 (avg)    ...    REPLICAS 10
```

From this point onward, the Deployment remains saturated at the configured upper bound:

```
1 maxReplicaCount: 10
```

The reported metric values are expressed in milli-units because the normalized queue backlog may assume fractional values after being divided by the current number of running replicas. For example, the metric reported in the previous reconciliation cycle:

```
15900m/5 (avg)
```

corresponds to a normalized metric value of 15.9. Since ten consumer replicas are currently active, the actual queue backlog can be estimated as:

$$15.9 \times 10 = 159$$

indicating that approximately 159 *Ready* messages are still waiting inside the RabbitMQ queue. During this phase:

- The producer continuously publishes new workload messages
- Consumer replicas process tasks sequentially
- Each task requires approximately five seconds to complete
- The queue backlog grows faster than it is depleted

Consequently, the Deployment remains saturated at ten replicas while the queue backlog progressively increases. After the producer Pod is terminated, no additional workload messages are generated. The consumer replicas therefore continue processing the remaining queue backlog until the normalized metric reported by the HPA gradually decreases:

```
16300m/5 (avg)    ...    REPLICAS 10
13300m/5 (avg)    ...    REPLICAS 10
10300m/5 (avg)    ...    REPLICAS 10
7300m/5 (avg)     ...    REPLICAS 10
4300m/5 (avg)     ...    REPLICAS 10
```

The first reconciliation cycle effectively triggering the scale-down stabilization mechanism is:

4300m/5 (avg) ... REPLICAS 10

At this stage:

- Current replicas = 10
- Normalized metric = 4300m = 4.3
- Target value = 5

The Horizontal Pod Autoscaler therefore computes:

$$desiredReplicas = \left\lceil \frac{10 \times 4.3}{5} \right\rceil = \lceil 8.6 \rceil = 9$$

This is the first reconciliation cycle producing a recommendation lower than the currently active replica count. However, the `ScaledObject` was configured with the following scale-down stabilization window:

```
scaleDown:
  stabilizationWindowSeconds: 60
```

Consequently, the newly computed recommendation is not applied immediately. Instead, the HPA activates the stabilization mechanism and temporarily stores the recommendation. Since:

- The HPA reconciliation period is 15 seconds
- The stabilization window lasts 60 seconds

approximately five consecutive recommendation samples are simultaneously considered during each reconciliation cycle. More specifically, at every scale-down evaluation:

- The current recommendation is added to the stabilization history
- Recommendations generated during the previous 60 seconds are retained
- The maximum recommendation among them is selected
- The selected maximum immediately becomes the effective Deployment replica count

During the first stabilization cycle, the recommendation history is summarized in Table 17.3:

Observed Metric	Replica Recommendation
16300m/5 (avg)	10
13300m/5 (avg)	10
10300m/5 (avg)	10
7300m/5 (avg)	10
4300m/5 (avg)	9

Table 17.3: Replica recommendations retained during the first stabilization window.

Consequently, the stabilization algorithm applies:

$$\max(10, 10, 10, 10, 9) = 10$$

Therefore, despite the newly computed recommendation equal to nine replicas, the Deployment temporarily remains stabilized at ten replicas. Therefore, the Deployment temporarily remains stabilized at:

REPLICAS = 10

During the following reconciliation cycle, the queue is not yet completely empty:

1300m/5 (avg) ... REPLICAS 10

The newest HPA recommendation now becomes:

$$desiredReplicas = \left\lceil \frac{10 \times 1.3}{5} \right\rceil = 3$$

The stabilization window slides forward by one recommendation while preserving the recommendations generated during the previous 60 seconds. The retained recommendation history is reported in Table 17.4:

Observed Metric	Replica Recommendation
13300m/5 (avg)	10
10300m/5 (avg)	10
7300m/5 (avg)	10
4300m/5 (avg)	9
1300m/5 (avg)	3

Table 17.4: Second scale-down stabilization window.

Consequently,

$$\max(10, 10, 10, 9, 3) = 10$$

and the Deployment remains temporarily stabilized at ten replicas. During the next reconciliation cycle, the stabilization window slides forward once again:

```
0/5 (avg)    ...    REPLICAS 10
```

The retained recommendations now become those reported in Table 17.5:

Observed Metric	Replica Recommendation
10300m/5 (avg)	10
7300m/5 (avg)	10
4300m/5 (avg)	9
1300m/5 (avg)	3
0/5 (avg)	1

Table 17.5: Third scale-down stabilization window.

Therefore,

$$\max(10, 10, 9, 3, 1) = 10$$

and the Deployment remains stabilized at ten replicas for one additional reconciliation cycle. After two further reconciliation intervals (30 seconds), the oldest recommendation equal to ten replicas expires from the stabilization history. The next reconciliation cycle therefore reports:

```
0/5 (avg)    ...    REPLICAS 9
```

The retained recommendations are now summarized in Table 17.6.

Observed Metric	Replica Recommendation
4300m/5 (avg)	9
1300m/5 (avg)	3
0/5 (avg)	1
0/5 (avg)	1
0/5 (avg)	1

Table 17.6: Fourth scale-down stabilization window.

Consequently,

$$\max(9, 3, 1, 1, 1) = 9$$

and the Deployment is immediately scaled down to nine replicas. Unlike the scale-up phase, during stabilization-window-controlled scale-down operations the selected maximum recommendation is directly applied within the current reconciliation cycle. Consequently, the updated replica count immediately appears in the `REPLICAS` column of the HPA output. The following reconciliation cycle reports:

```
0/5 (avg)     ...     REPLICAS 3
```

The stabilization history now becomes that reported in Table 17.7:

Timestamp	Recommendation
1300m/5 (avg)	3
0/5 (avg)	1
0/5 (avg)	1
0/5 (avg)	1
0/5 (avg)	1

Table 17.7: Replica recommendations retained during the final stabilization window.

Therefore,

$$\max(3, 1, 1, 1, 1) = 3$$

and the Deployment is immediately scaled down to three replicas. After one additional reconciliation cycle, the last recommendation greater than one also expires from the stabilization history:

0/5 (avg) ... REPLICAS 1

The retained recommendations are now:

Observed Metric	Replica Recommendation
0/5 (avg)	1
0/5 (avg)	1
0/5 (avg)	1
0/5 (avg)	1
0/5 (avg)	1

Table 17.8: Final scale-down stabilization window.

Consequently,

$$\max(1, 1, 1, 1, 1) = 1$$

At this stage:

- The HPA-controlled scale-down process has completed;
- One idle consumer replica is still active inside the cluster.

The Deployment is not immediately scaled to zero because the configured `cooldownPeriod = 120` seconds is still active. More specifically, the cooldown timer started when the RabbitMQ scaler first detected a queue backlog equal to zero, corresponding to the first reconciliation cycle reporting:

0/5 (avg)

Since both the KEDA polling interval and the HPA reconciliation interval were intentionally configured to the same value of 15 seconds, the scaler and the HPA operate approximately synchronously throughout the experiment. Consequently, the cooldown timer effectively starts from the first HPA reconciliation cycle reporting an empty queue. During the cooldown interval:

- KEDA postpones the final scale-to-zero transition
- Transient workload spikes may still be processed without requiring a complete cold start
- Unnecessary oscillations between zero and one replica are avoided

Once the configured cooldown period expires, KEDA directly takes control of the Deployment and performs the final scale-to-zero transition:

<unknown>/5 (avg) ... REPLICAS 0

At this point:

- All consumer replicas have been removed
- The HPA can no longer normalize the external metric
- The metric returns to `<unknown>`
- KEDA passively continues monitoring the RabbitMQ queue waiting for future workload activations

17.4.1 Additional Remarks on Cooldown Behavior and Transient Workload Spikes

The configured `cooldownPeriod` not only delays the final scale-to-zero transition, but also improves the system's ability to react to transient workload spikes occurring shortly after the queue becomes empty. Avoiding an immediate transition to zero replicas prevents the repeated execution of complete cold-start procedures. Indeed, when scaling from zero, the arrival of new workload messages would require:

- Deployment reactivation by KEDA
- Kubernetes Pod scheduling
- Eventual container image retrieval
- Container startup
- Re-establishment of the AMQP connection with RabbitMQ
- Application initialization and readiness completion

Keeping one consumer replica temporarily active during the cooldown interval therefore significantly reduces workload reactivation latency. Two representative scenarios may occur during the cooldown interval.

Workload Spike After Stabilization Window Completion Assume that the HPA has already completed the gradual scale-down process imposed by the stabilization window, for example:

$$10 \rightarrow 4 \rightarrow 1$$

while the configured KEDA cooldown timer has not yet expired. In this condition, one consumer replica is still active inside the cluster. If new workload messages suddenly appear before the expiration of the `cooldownPeriod`, the RabbitMQ scaler immediately detects the increased queue backlog and the HPA resumes its normal autoscaling workflow without requiring a scale-from-zero reactivation. For example, assuming:

$$\text{Current replicas} = 1$$

$$\text{Queue backlog} = 40$$

$$\text{Target value} = 5$$

the HPA computes

$$desiredReplicas = \left\lceil \frac{1 \times 40}{5} \right\rceil = 8$$

The Deployment can therefore immediately scale up again according to the configured scale-up policies.

Workload Spike During Stabilization Window Execution A second scenario occurs when new workload messages appear while:

- The cooldown period is still active
- The HPA scale-down stabilization window has not yet completed
- The Deployment replica count is still progressively decreasing.

For example, the Deployment may currently be transitioning through

$$10 \rightarrow 4$$

while previous replica recommendations are still retained inside the stabilization window. If a new workload spike suddenly increases the queue backlog,

the HPA immediately computes a larger replica recommendation. Since the configured stabilization window only affects scale-down operations, scale-up decisions are not delayed. Consequently, the Deployment can immediately reverse its scaling direction and rapidly increase again, for example:

$$4 \rightarrow 8$$

or

$$4 \rightarrow 10$$

depending on the computed recommendation and the active scale-up policies. These scenarios highlight how the combined effects of the following mechanisms:

- HPA stabilization windows
- Default HPA scaling policies
- KEDA cooldown periods
- Scale-to-zero capabilities

allows the autoscaling system to simultaneously achieve stability during workload decreases while maintaining rapid responsiveness to sudden workload increases in highly dynamic queue-based applications.

17.5 Chapter Summary

This chapter presented the complete experimental implementation and evaluation of a RabbitMQ-based event-driven autoscaling scenario using KEDA. Starting from the deployment of the RabbitMQ broker and the implementation of producer and consumer applications, the chapter described the complete containerization workflow, the Kubernetes deployment configuration, and the definition of the corresponding KEDA `ScaledObject`. The experimental analysis demonstrated the interaction between the RabbitMQ scaler, the Kubernetes Horizontal Pod Autoscaler, and KEDA during both scale-up and scale-down operations. In particular, the conducted experiments validated the effects of the default HPA scaling policies, scale-down stabilization windows, KEDA cooldown mechanisms, and scale-to-zero capabilities under realistic queue-based workloads. The experimental results confirmed that the

RabbitMQ scaler accurately reacts to variations in queue length while preserving stable autoscaling behaviour through the mechanisms provided by Kubernetes and KEDA. Overall, the proposed architecture proved capable of rapidly adapting the number of consumer replicas to workload fluctuations while simultaneously maintaining efficiency, responsiveness, and stability. These results further demonstrate the suitability of KEDA for implementing event-driven autoscaling strategies in asynchronous cloud-native applications based on message queues.

Chapter 18

RabbitMQ Native Prometheus Exporter and KEDA Autoscaling

18.1 Experimental Objective

As discussed in the previous chapter, RabbitMQ workloads can be autoscaled through the native KEDA RabbitMQ scaler, which directly retrieves queue-related metrics from the broker using the AMQP protocol. RabbitMQ, however, also provides native integration with the Prometheus monitoring ecosystem through its built-in Prometheus exporter. This enables the same queue metrics to be periodically collected by a Prometheus Server and subsequently consumed by the KEDA Prometheus scaler for autoscaling purposes. From a conceptual perspective, the workload considered in this experiment remains identical to that presented in the previous RabbitMQ autoscaling scenario. The same producer–consumer architecture is adopted, and the autoscaling process is driven by the same application-level metric, namely the number of messages currently stored in the queue in the *Ready* state. Consequently, the only significant difference introduced in this experiment concerns the metric acquisition path rather than the autoscaling logic itself.

18.2 RabbitMQ Prometheus Integration

The interaction among the KEDA components remains unchanged with respect to the previous experiments. As discussed throughout the previous chapters, KEDA periodically retrieves the workload metric through the configured scaler, exposes it to the Kubernetes Horizontal Pod Autoscaler (HPA) via the KEDA Metrics Adapter and the Kubernetes API Aggregation Layer, and finally delegates the replica computation process to the HPA. The main architectural difference introduced in this experiment concerns the source from which the autoscaling metric is obtained. Instead of directly querying the RabbitMQ broker through the AMQP protocol, the metric is now retrieved from a Prometheus Server, which periodically scrapes RabbitMQ metrics through the native Prometheus exporter integrated into the broker itself. Figure 18.1 summarizes the overall metric acquisition workflow adopted in this experiment:

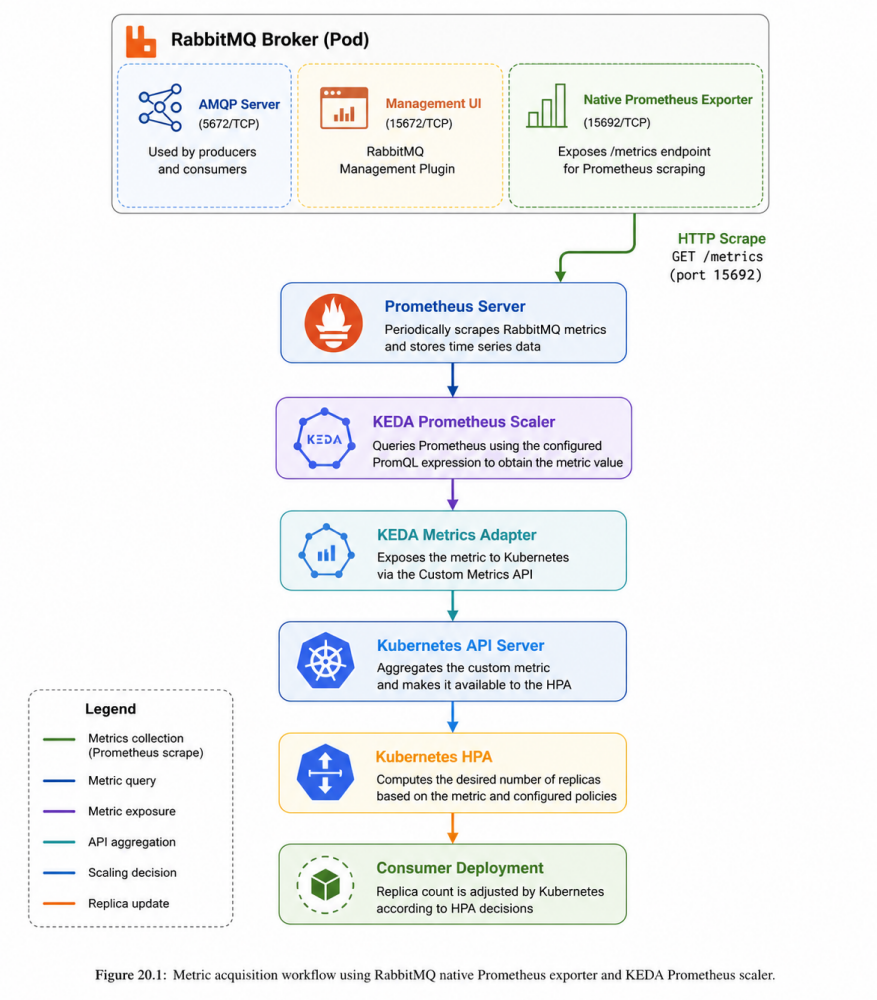


Figure 20.1: Metric acquisition workflow using RabbitMQ native Prometheus exporter and KEDA Prometheus scaler.

Figure 18.1: Metric acquisition workflow using the RabbitMQ native Prometheus exporter and the KEDA Prometheus scaler.

The RabbitMQ Prometheus exporter is exposed on port `15692` and provides the standard Prometheus-compatible HTTP endpoint:

```
http://<rabbitmq-host>:15692/metrics
```

Unlike the NGINX monitoring scenario discussed previously, RabbitMQ does not require an external exporter container. The Prometheus exporter is directly integrated into the RabbitMQ runtime and therefore has direct access

to the broker internal state. Consequently, when the Prometheus Server performs a scraping operation through the `/metrics` endpoint, the exporter does not interact with either the RabbitMQ Management Plugin exposed on port `15672` or the AMQP server component exposed on port `5672`. Instead, it dynamically retrieves the latest metric values directly from the broker internal runtime structures and generates the corresponding Prometheus response on demand. This behavior follows the standard Prometheus pull-based monitoring model. Metrics are neither periodically pushed to Prometheus nor proactively collected by the exporter. Instead, metric acquisition only occurs when the Prometheus Server issues an HTTP scraping request. The exporter then generates a fresh snapshot of the current RabbitMQ runtime state and returns it as a Prometheus-compatible response.

18.3 Experimental Implementation

Before deploying the resources discussed in this section, a Prometheus monitoring infrastructure must already be available within the Kubernetes cluster. More specifically, the Prometheus Server, the Prometheus Operator and the additional monitoring components provided by the monitoring stack must already be correctly installed and operational. This prerequisite was already satisfied through the monitoring environment deployed during the previous Prometheus-based autoscaling experiments. Since the producer application, consumer application, Docker containerization workflow, Docker Hub image publication procedure and consumer Deployment configuration remain unchanged with respect to the previous RabbitMQ experiment, they are not reported again in this chapter. Likewise, both the workload generation process and the consumer processing logic remain identical. Consequently, only the Kubernetes resources requiring modifications to enable Prometheus integration are discussed below.

18.3.1 RabbitMQ Deployment and Service Modifications

To expose Prometheus-compatible metrics, the RabbitMQ Deployment was extended by adding the native exporter endpoint on port `15692`. More specifically, the following container port was added to the `rabbitmq.yaml` manifest:

```
1 ports:
```

```
2 - containerPort: 5672
3 - containerPort: 15672
4 - containerPort: 15692
```

Listing 18.1: Additional container port enabling the RabbitMQ native Prometheus exporter.

where:

- **5672** exposes the AMQP server component used by producers, consumers and the native RabbitMQ scaler
- **15672** exposes the RabbitMQ Management Plugin
- **15692** exposes the native Prometheus exporter endpoint

The Kubernetes Service associated with the RabbitMQ Deployment was extended accordingly by exposing the same endpoint:

```
1 - name: prometheus
2   port: 15692
3   targetPort: 15692
```

Listing 18.2: Additional Service port exposing the RabbitMQ Prometheus exporter.

This additional Service port allows the Prometheus Server to periodically scrape the exporter endpoint from within the Kubernetes cluster. The RabbitMQ Deployment was further extended by introducing the following environment variable:

```
1 - name: RABBITMQ_SERVER_ADDITIONAL_ERL_ARGS
2   value: "-rabbitmq_prometheus return_per_object_metrics true"
```

Listing 18.3: Environment variable enabling the RabbitMQ native Prometheus exporter.

This configuration enables the integrated Prometheus exporter and instructs RabbitMQ to expose metrics at queue-level granularity, allowing each queue to be monitored independently.

18.3.2 ServiceMonitor Configuration

To enable automatic target discovery by the Prometheus Operator, a dedicated `ServiceMonitor` resource was created:

servicemonitor.yaml

```
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor

metadata:
  name: rabbitmq-servicemonitor
  namespace: rabbitmq

  labels:
    release: monitoring

spec:
  selector:
    matchLabels:
      app: rabbitmq

  namespaceSelector:
    matchNames:
    - rabbitmq

  endpoints:
  - port: prometheus
    path: /metrics
    interval: 5s
```

Listing 18.4: ServiceMonitor used to scrape the RabbitMQ native Prometheus exporter.

The `release:monitoring` label ensures that the `ServiceMonitor` is discovered by the Prometheus Operator according to the `serviceMonitorSelector` configured in the Prometheus Custom Resource. Before applying the ServiceMonitor, it is important to observe that Prometheus target discovery relies on Kubernetes label matching. More specifically, the selector defined inside the `ServiceMonitor`

```
selector:
```

```
2   matchLabels:
3     app: rabbitmq
```

must match the labels associated with the RabbitMQ Service deployed in the cluster. In the implemented setup, the RabbitMQ Deployment, RabbitMQ Service and `ServiceMonitor` were all deployed inside the same `rabbitmq` namespace. Although this is not a strict requirement, this configuration simplifies resource organization and target discovery. The `namespaceSelector` field explicitly instructs the Prometheus Operator to search for matching Services within the `rabbitmq` namespace. Once the configuration has been completed, the `ServiceMonitor` can be applied through:

```
kubectl apply -f servicemonitor.yaml
```

After the resource has been applied, the Prometheus Operator automatically registers the new monitoring target and configures the Prometheus Server to periodically scrape the RabbitMQ exporter endpoint every five seconds, according to the configured scraping interval.

18.3.3 Validation

Before performing the validation steps, the Prometheus Service is exposed locally through:

```
kubectl port-forward svc/monitoring-kube-prometheus-prometheus \
9090:9090 -n monitoring
```

The first validation consists of verifying that the RabbitMQ exporter has been successfully discovered by Prometheus:

```
curl -s http://localhost:9090/api/v1/targets | \
jq '.data.activeTargets[] |
{job: .labels.job, health: .health, url: .scrapeUrl}'
```

The RabbitMQ target should appear with status `UP`, confirming that the `ServiceMonitor`, the label matching process and the exporter endpoint have all been configured correctly. A second validation consists of directly querying the metric that will later be consumed by the KEDA Prometheus scaler:

```
curl -s -G http://localhost:9090/api/v1/query \
--data-urlencode \
'query=rabbitmq_queue_messages_ready{queue="task-queue"}' | jq
```

This command effectively simulates the behavior of the KEDA Prometheus scaler by directly executing the corresponding PromQL query against the Prometheus Server. When no messages are present inside the queue, the returned value is expected to be zero. During workload generation, instead, the query returns the number of messages currently stored in the queue in the *Ready* state. This is exactly the metric later consumed by the KEDA Prometheus scaler to compute the desired number of consumer replicas.

18.3.4 Prometheus-based ScaledObject Configuration

To retrieve the RabbitMQ queue metric through Prometheus instead of directly communicating with the broker via AMQP, the `ScaledObject` configuration was modified by replacing the native RabbitMQ scaler with the KEDA Prometheus scaler. The complete trigger configuration adopted in this experiment is reported below:

```
triggers:
- type: prometheus

  metadata:
    serverAddress: http://monitoring-kube-prometheus-prometheus.
    monitoring:9090
    ignoreNullValues: "false"
    metricName: rabbitmq-ready-messages
    query: rabbitmq_queue_messages_ready{queue="task-queue"}
    threshold: "5"
    activationThreshold: "0"
```

Listing 18.5: Prometheus trigger configuration adopted by the KEDA ScaledObject.

The role of each configuration parameter is discussed below: The serverAddress parameter specifies the endpoint of the Prometheus Server queried by the KEDA Prometheus scaler to retrieve the autoscaling metric. The metricName parameter defines the name under which the metric is exposed by KEDA to the Kubernetes Horizontal Pod Autoscaler (HPA) through the KEDA Metrics Adapter. The query parameter specifies the PromQL query used to retrieve the number of messages currently stored in the *Ready* state inside the `task-queue` queue. The threshold parameter defines the desired average num-

ber of *Ready* messages tolerated by each consumer replica. Unlike the native RabbitMQ scaler, where the equivalent target was specified through the value parameter of the QueueLength mode and automatically propagated by KEDA to the generated HPA configuration, the Prometheus scaler requires the target value to be explicitly configured through the dedicated threshold parameter. Although the parameter names differ, both configurations conceptually represent the same autoscaling target and therefore produce the same HPA target value. The activationThreshold parameter specifies the minimum metric value required to reactivate the Deployment from zero replicas. In the native RabbitMQ scaler, the same functionality was provided through the activationValue parameter. Despite the different naming adopted by the two scalers, both parameters represent the same activation threshold used by KEDA to determine when the Deployment should transition from zero to one replica. Finally, the ignoreNullValues parameter controls the behavior adopted when the Prometheus query does not return any value. By setting it to `"false"`, the scaler treats missing metric values as an error condition rather than silently ignoring them. This configuration is particularly useful during testing and debugging because it immediately highlights potential issues affecting the Prometheus query, target discovery process or scraping configuration. In the implemented experiment, the parameter was explicitly configured to `"false"` to ensure that any metric acquisition problem could be promptly detected. The remaining `ScaledObject` parameters, including the polling interval, cooldown period, stabilization windows, minimum replica count and maximum replica count, remain identical to those adopted in the previous RabbitMQ autoscaling experiment. From a structural perspective, the overall `ScaledObject` remains almost identical to the one adopted in the previous experiment. The only substantial difference lies in the trigger configuration, which now retrieves the autoscaling metric from the Prometheus Server instead of directly querying the RabbitMQ broker through the AMQP protocol.

18.4 Experimental Evaluation

After deploying the modified resources, workload generation was initiated using the same procedure adopted in the previous RabbitMQ autoscaling experiment. More specifically, the producer pod was dynamically instantiated

through:

```
kubectl run rabbitmq-producer \
  --image=matteo4400/rabbitmq-producer:1.0 \
  --image-pull-policy=Always
```

After a sufficiently large queue backlog had been generated, the producer pod was manually removed through:

```
kubectl delete pod rabbitmq-producer
```

For completeness, the output of the generated HPA collected through continuous monitoring with:

```
kubectl get hpa -w
```

is reported below. Although some numerical values differ slightly from those observed in the previous RabbitMQ scaler experiment, the overall autoscaling workflow and the sequence of scaling decisions remain unchanged because the same workload metric, target threshold, polling interval, stabilization windows and cooldown period were preserved. Similarly to the previous experiment, the generated HPA remained named `keda-hpa-rabbitmq-scaledobject`, while the `MINPODS = 1` and `MAXPODS = 10` parameters remained constant throughout the experiment and have therefore been omitted from the reported output for brevity:

TARGETS	REPLICAS
<unknown>/5 (avg)	0
75/5 (avg)	1
28200m/5 (avg)	5
16300m/5 (avg)	10
13300m/5 (avg)	10
10300m/5 (avg)	10
7300m/5 (avg)	10
4300m/5 (avg)	10
1300m/5 (avg)	10
0/5 (avg)	10
0/5 (avg)	10
0/5 (avg)	9
0/5 (avg)	3
0/5 (avg)	1

```
<unknown>/5 (avg)      0
```

Listing 18.6: HPA output collected during the RabbitMQ autoscaling experiment using the KEDA Prometheus scaler.

As can be observed from the reported HPA output, the overall autoscaling behavior remained equivalent to that discussed in the previous RabbitMQ experiment. Scale-up operations, scale-down stabilization, HPA recommendation computations, cooldown period handling, scale-to-zero transitions and reactivation from zero replicas all followed the same sequence of events and produced the expected scaling decisions. This result is consistent with the experimental configuration adopted throughout the two implementations. The workload generation process, queue-processing logic, autoscaling thresholds, HPA policies, stabilization windows, polling interval and cooldown period remained unchanged. Consequently, both implementations expose the HPA to the same application-level workload metric and therefore naturally produce equivalent autoscaling decisions. The only architectural difference concerns the metric acquisition path. In the previous experiment, the metric was directly retrieved by the native RabbitMQ scaler through the AMQP protocol. Conversely, in the present implementation, the metric is first collected by the Prometheus Server through the RabbitMQ native Prometheus exporter and is subsequently retrieved by the KEDA Prometheus scaler through a PromQL query. Although this additional monitoring layer introduces an extra component into the metric acquisition pipeline, it does not alter the autoscaling logic itself. The choice between the two approaches therefore mainly depends on the characteristics of the target environment. The native RabbitMQ scaler provides a simpler and more direct solution when autoscaling is the only objective, whereas the Prometheus-based approach becomes particularly attractive in Kubernetes environments where a Prometheus monitoring infrastructure is already available. In such scenarios, the same monitoring platform can simultaneously support observability activities and provide the workload metrics required for event-driven autoscaling, avoiding direct communication between KEDA and the RabbitMQ broker while centralizing metric collection within the existing monitoring ecosystem. For clarity, Table 18.1 summarizes the main differences between the native RabbitMQ scaler and the KEDA Prometheus scaler adopted throughout the two experimental implementations:

Feature	**RabbitMQ Native Scaler**	**Prometheus Scaler**
Metric source	RabbitMQ broker	Prometheus Server
Communication protocol	AMQP	HTTP (PromQL query)
Additional monitoring infrastructure	Not required	Prometheus monitoring stack required
Metric acquisition	Directly from RabbitMQ	Retrieved from Prometheus after scraping
Autoscaling logic	KEDA + Kubernetes HPA	KEDA + Kubernetes HPA
Scale-to-zero support	Supported	Supported
Main advantage	Simple and direct integration	Leverages an existing monitoring infrastructure
Typical use case	RabbitMQ-centric deployments	Clusters already using Prometheus for monitoring

Table 18.1: Comparison between the native RabbitMQ scaler and the KEDA Prometheus scaler.

18.5 Chapter Summary

This chapter presented the implementation and experimental evaluation of a RabbitMQ-based event-driven autoscaling solution using the KEDA Prometheus scaler. Unlike the previous implementation based on the native RabbitMQ scaler, the autoscaling metric was no longer retrieved directly from the broker through the AMQP protocol, but instead collected by the Prometheus Server through the native RabbitMQ Prometheus exporter and subsequently queried by the Prometheus scaler using PromQL. The experimental results demonstrated that replacing the metric acquisition path does not affect the overall autoscaling behavior. Scale-up operations, HPA recommendation computations, scale-down stabilization, cooldown handling and scale-to-zero transitions remained fully consistent with those observed in the previous RabbitMQ ex-

periment. Consequently, while the native RabbitMQ scaler provides a simple and direct solution for broker-specific deployments, the Prometheus-based approach offers a more flexible alternative for Kubernetes environments where a Prometheus monitoring infrastructure is already available, enabling a unified platform for both observability and event-driven autoscaling.

Chapter 19

Comparative Evaluation of Static and Event-Driven Scaling Strategies in a RabbitMQ Scenario

19.1 Experimental Objective

The previous chapters presented the complete implementation of the event-driven application and its supporting infrastructure, including RabbitMQ, Prometheus, Grafana, and the KEDA-based autoscaling mechanism. Building upon this environment, the objective of this chapter is to experimentally evaluate the effectiveness of different resource allocation strategies for event-driven applications deployed on Kubernetes. To provide a comprehensive comparison, four deployment scenarios are analysed under identical workload conditions. Two scenarios rely on static resource allocation, representing underprovisioned and overprovisioned configurations, while the remaining two leverage KEDA to dynamically adjust the number of consumer replicas according to the workload. In particular, the autoscaling scenarios investigate the impact of two different minimum replica configurations, namely scale-to-zero and warm-start deployments. The evaluation is performed by submitting the same workload to each configuration and monitoring the system through Prometheus and Grafana. The collected measurements are then used to analyse queue dynamics, consumer utilization, resource consumption, workload completion time,

and overall scaling behaviour. Particular attention is devoted to the estimation of the computational overhead introduced by idle consumer replicas, allowing the comparison to consider not only application performance but also resource efficiency. By combining theoretical considerations with experimental measurements, this chapter aims to highlight the trade-offs between static provisioning and event-driven autoscaling, identifying the advantages and limitations of each deployment strategy in terms of responsiveness, scalability, and infrastructure utilization.

19.2 Experimental Architecture

The experimental platform adopted for the comparative evaluation is based on the same infrastructure presented in the previous chapter. A RabbitMQ broker acts as the message queue, while a set of consumer pods retrieves and processes the incoming messages. RabbitMQ metrics are continuously collected by Prometheus and visualized through Grafana dashboards, which provide both time-series plots and summary metrics describing the behaviour of the system. In the autoscaling scenarios, KEDA is deployed within the Kubernetes cluster to dynamically adjust the number of consumer replicas according to an external metric collected from Prometheus. As in the previous experimental setup, the selected scaling metric is the number of RabbitMQ messages in the *ready* state, corresponding to the messages waiting in the queue that have not yet been delivered to any consumer. The Kubernetes Horizontal Pod Autoscaler (HPA) automatically generated by KEDA performs the actual scaling decisions between the configured minimum and maximum number of replicas, whereas KEDA is responsible for managing the activation and deactivation of the Deployment whenever scale-to-zero is enabled. To maintain consistency between monitoring and autoscaling, the Prometheus scaler was selected for all KEDA-based experiments. Both Grafana and KEDA retrieve the required RabbitMQ metrics directly from the Prometheus server, ensuring that the metric used to visualize the queue state is exactly the same metric used to trigger autoscaling decisions. Since the selected scaling signal corresponds to the number of RabbitMQ messages in the *ready* state, any increase in this value immediately indicates that the available consumer capacity is insufficient to process the incoming workload as soon as it arrives, making it particularly suitable for event-driven autoscaling. The consumer implementa-

tion and the interaction logic with RabbitMQ remain unchanged with respect to the previous experimental setup. Each consumer establishes a connection with the RabbitMQ broker, subscribes to the target queue, and processes the received messages sequentially. This behaviour is enforced by configuring the `prefetch_count` parameter to one, ensuring that RabbitMQ delivers at most one unacknowledged message to each consumer at any given time. Consequently, a consumer receives a new message only after the previous one has been completely processed and the corresponding acknowledgement has been returned to the broker. This configuration establishes a direct relationship between the number of unacknowledged messages and the number of busy consumers. Since each consumer can process only one message at a time, every unacknowledged message corresponds to one actively processing consumer. As a result, the number of unacknowledged messages can be directly interpreted as the number of consumers currently executing a task.

19.3 Producer Redesign

The main modification introduced in this chapter concerns the producer component. During the initial validation experiments, the producer acted as a temporary traffic generator: it published a predefined workload to RabbitMQ and terminated immediately afterwards. Although this behaviour was sufficient to validate the interaction among RabbitMQ, the consumer application, Prometheus, Grafana, and KEDA, it proved unsuitable for conducting repeated comparative experiments under controlled and reproducible conditions. To overcome this limitation, the producer was redesigned as a persistent Flask-based HTTP service. Instead of generating messages automatically when the container starts, the new implementation exposes an HTTP endpoint that can be invoked on demand to trigger workload generation. The request allows both the total number of messages and the desired production rate to be specified, making the workload generation process fully parameterizable. This approach enables exactly the same traffic pattern to be reproduced across all experimental scenarios, ensuring fair and repeatable comparisons. The updated implementation of the workload generator is reported in Listing 19.1:

producer.py

```
1 from flask import Flask, request, jsonify
```

```
import pika  # Python AMQP client library used to interact with the RabbitMQ broker.
import time  # Standard Python module used to control the message generation rate.

# Create the Flask application instance.
app = Flask(__name__)


@app.route('/health', methods=['GET'])
def health():
    """
    Simple endpoint used to verify that the producer service
    is running and reachable.
    """
    return jsonify({"status": "ok"}), 200


@app.route('/generate', methods=['POST'])
def generate_messages():
    """
    Generate a finite number of messages toward RabbitMQ.

    Expected JSON payload:

    {
        "messages": 300,
        "rate": 5
    }

    where:
    - messages = total number of messages to generate
    - rate = message generation rate (messages per second)

    If one or both parameters are omitted,
    default values are used.
    """
```

```
37
38      # Create a blocking TCP connection toward the
            RabbitMQ broker
39      # exposed through the Kubernetes Service DNS name.
40      connection = pika.BlockingConnection(
41          pika.ConnectionParameters(
42              host='rabbitmq.rabbitmq.svc.cluster.local',
43              port=5672,  # Default RabbitMQ AMQP port.
44              credentials=pika.PlainCredentials('user', '
                    password')
45          )
46      )
47
48      # Create an AMQP logical channel over the TCP
            connection.
49      channel = connection.channel()
50
51      # Declare the queue if it does not already exist.
52      # Producer and consumers must reference the same
            queue name.
53      channel.queue_declare(queue='task-queue')
54
55      # Retrieve JSON payload from the incoming HTTP
            request.
56      payload = request.get_json(silent=True) or {}
57
58      # Total number of messages to generate.
59      # Default value: 300 messages.
60      total_messages = int(payload.get("messages", 300))
61
62      # Message generation rate expressed in messages per
            second.
63      # Default value: 5 messages/second.
64      rate = float(payload.get("rate", 5))
65
66      # Prevent invalid configurations.
67      if total_messages <= 0:
68          return jsonify({
69              "error": "messages must be greater than zero"
```

```
70          }), 400
71
72      if rate <= 0:
73          return jsonify({
74              "error": "rate must be greater than zero"
75          }), 400
76
77      # Compute the delay between two consecutive messages.
78      sleep_time = 1 / rate
79
80      print(
81          f"Starting workload generation: "
82          f"{total_messages} messages at {rate} msg/s"
83      )
84
85      # Generate the configured workload.
86      try:
87
88          for i in range(total_messages):
89
90              # Create the payload associated with the
                    current message.
91              msg = f"message-{i}"
92
93              # Publish the message through the default
                    RabbitMQ exchange.
94              # The routing_key matches the destination
                    queue name.
95              channel.basic_publish(
96                  exchange='',
97                  routing_key='task-queue',
98                  body=msg
99              )
100
101             # Print the produced message for debugging
                    purposes.
102             print(f"Sent {msg}")
103
104             # Wait according to the configured generation
```

```
                rate.
105             time.sleep(sleep_time)
106
107         print("Workload generation completed")
108
109         # Return a summary of the executed workload.
110         return jsonify({
111             "status": "completed",
112             "messages_sent": total_messages,
113             "rate": rate
114         }), 200
115
116     finally:
117
118         # Always close the AMQP connection when the
                request terminates.
119         # This guarantees proper resource cleanup both in
                normal execution
120         # and in case of exceptions during message
                generation or publication.
121         connection.close()
122
123         print("RabbitMQ connection closed")
124
125 if __name__ == '__main__':
126     # Start the Flask development server.
127     # The service is exposed on all network interfaces at
            port 5000
128     # to allow access from outside the container.
129     app.run(
130         host='0.0.0.0',
131         port=5000
132     )
```

Listing 19.1: Flask-based producer implementation used for controlled workload generation.

The Flask-based workload generator exposes two HTTP endpoints. The first one, `/health`, is intended exclusively for debugging and service verification. It accepts `GET` requests and returns a simple JSON response confirming that

the producer service is running and reachable. This endpoint is particularly useful for verifying that the producer pod has started correctly and that the corresponding Kubernetes Service is accessible before initiating workload generation. The second endpoint, `/generate`, represents the main enhancement introduced for the comparative evaluation. It accepts `POST` requests carrying a JSON payload that specifies the workload parameters. In particular, the `messages` field defines the total number of messages to be generated, whereas the `rate` field specifies the production rate expressed in messages per second. When one or both parameters are omitted, predefined default values are automatically adopted. This design enables different traffic patterns to be generated without modifying the application code, rebuilding the container image, or restarting the producer pod. When the `/generate` endpoint is invoked, the producer establishes a connection with the RabbitMQ broker using the same communication logic adopted in the previous experimental setup. Likewise, the queue declaration and message publication mechanisms remain unchanged: messages are published to the `task-queue` queue through the default RabbitMQ exchange. The main difference with respect to the previous implementation is that the workload generation process is now initiated through an HTTP request and entirely driven by the parameters contained in the received JSON payload. The desired production rate is enforced by computing the delay between two consecutive messages as the inverse of the configured rate. For example, when the generation rate is set to 2 messages per second, the producer waits 0.5 seconds before publishing the next message. This simple mechanism provides a sufficiently deterministic traffic generation process, ensuring that identical workloads can be reproduced across all four experimental scenarios analysed in this chapter. Once the workload has been completely generated, the producer returns a JSON response containing the execution status, the total number of published messages, and the configured generation rate. Finally, the connection to RabbitMQ is closed within a `finally` block, guaranteeing proper resource cleanup regardless of whether the execution completes successfully or an exception occurs during message publication. The Flask server listens on port `5000` and is bound to all network interfaces, allowing it to be reached through the corresponding Kubernetes Service.

19.4 Producer Deployment and Service Exposure

Compared with the previous experimental setup, the producer is no longer executed as a temporary pod whose sole purpose is to generate a predefined workload and terminate immediately afterwards. Instead, it is deployed as a persistent application component running inside the Kubernetes cluster and exposing an HTTP service. This architectural modification enables workload generation to be triggered on demand in a controlled, repeatable, and fully parameterizable manner, without recreating the pod or modifying the application code before each experiment. The containerization process follows the same methodology adopted for the other application components presented in the previous chapters. The Docker image is built from the Python source code together with its execution environment and subsequently pushed to Docker Hub, allowing it to be directly referenced by the Kubernetes Deployment manifest. Since the image build and publication procedure has already been described in detail, it is not repeated here. For completeness, Listing 19.2 reports the Dockerfile used to build the producer image. The Dockerfile is intentionally lightweight. It starts from the official Python 3.11 slim image, installs the two required Python libraries, namely `pika` for the interaction with RabbitMQ and `Flask` for exposing the HTTP API, copies the `producer.py` script into the container filesystem, exposes port `5000`, and finally starts the workload generator when the container is launched.

Dockerfile

```
# Base image containing Python 3.11 in a lightweight Debian-based
    environment.
FROM python:3.11-slim

# Install the Python libraries required by the workload generator:
# - pika: Communication with RabbitMQ
# - flask: REST API used to trigger workload generation
RUN pip install --no-cache-dir pika flask

# Copy the workload generator application script inside the container
    .
COPY producer.py /producer.py
```

```
# Expose the HTTP port used by the Flask server.
EXPOSE 5000

# Execute the workload generator service script when the container
    starts.
CMD ["python", "/producer.py"]
```

Listing 19.2: Dockerfile used to build the Flask-based workload generator.

As for the other containerized application components, once the image had been successfully built, it was pushed to Docker Hub and referenced within the producer Deployment manifest. Unlike the producer adopted during the initial validation experiments, the new implementation is deployed through a dedicated Kubernetes Deployment. Since the producer acts exclusively as a workload generator, a single replica is sufficient for all experiments presented in this chapter. The producer itself is not intended to be scaled; instead, it serves as a controlled traffic injection tool capable of generating workloads according to the parameters specified by the user. Maintaining a single replica also eliminates any ambiguity in the workload generation process, ensuring that the message production rate specified through the HTTP request exactly matches the overall traffic injected into RabbitMQ. The Kubernetes Deployment used for the producer is reported in Listing 19.3:

producer-deployment.yaml

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: producer
5  spec:
6    replicas: 1
7
8    selector:
9      matchLabels:
10       app: producer
11
12   template:
13     metadata:
14       labels:
15         app: producer
16
```

```
    spec:
      containers:
      - name: producer
        image: matteo4400/rabbitmq-producer:1.0
        imagePullPolicy: Always
        ports:
        - containerPort: 5000
```

Listing 19.3: Kubernetes Deployment manifest of the producer service.

The Deployment manifest defines a single producer replica. The pod template references the Docker image previously built and published to Docker Hub, namely `matteo4400/rabbitmq-producer:1.0`. The container exposes port `5000`, which corresponds to the port used by the Flask HTTP server. Furthermore, the field `imagePullPolicy: Always` instructs Kubernetes to always retrieve the image from the registry when the pod is created, a convenient choice during the experimental phase, where the image may be rebuilt and updated multiple times. Since the structure and semantics of Kubernetes Deployment manifests have already been discussed in the previous chapters, they are not analysed further here. The relevant aspect for this experimental setup is that the producer is now maintained as a persistent application component running with a single replica, rather than being executed as a temporary pod exclusively for workload generation. Since the producer exposes HTTP endpoints, a Kubernetes Service is also required to provide stable network access. A `ClusterIP` Service forwards incoming HTTP requests to the Flask application running inside the producer pod. Unlike the previous implementation, where the producer terminated immediately after generating the workload, the new architecture requires the producer to remain continuously reachable so that workload generation can be triggered on demand. The corresponding Service manifest is reported in Listing 19.4:

producer-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: producer-service
spec:
  selector:
    app: producer
```

```
8   ports:
9   - port: 5000
10    targetPort: 5000
```

Listing 19.4: ClusterIP Service exposing the producer HTTP API.

The Service selects the producer pod through the label `app: producer`, matching the label defined in the corresponding Deployment. Incoming traffic is accepted on port `5000` and forwarded to the same port inside the container, where the Flask application is listening. Consequently, the producer can always be reached through a stable Kubernetes Service name, independently of the pod IP dynamically assigned by Kubernetes. As for the Deployment, the Service manifest is intentionally simple. Its only purpose is to expose the HTTP endpoints implemented by the producer within the cluster and, whenever required, make them accessible from the host machine through Kubernetes port forwarding. This allows workload generation to be initiated externally by sending an HTTP request to the `/generate` endpoint. All the remaining Kubernetes resources employed throughout the experimental platform are unchanged with respect to those presented in the previous chapters. In particular, the RabbitMQ Deployment, the associated Services, and the integrated Prometheus exporter are identical to the previously adopted configuration. Likewise, the ServiceMonitor continues to target the RabbitMQ exporter Service, enabling Prometheus to periodically collect the broker metrics. The consumer application, together with its container image and Kubernetes Deployment, also remains unchanged. Each consumer continues to retrieve messages from RabbitMQ and process them sequentially according to the logic already described. Consequently, neither the consumer implementation nor its Deployment manifest are reported again in this chapter. Across all four experimental scenarios, the consumer application remains exactly the same; the only aspect that changes is the strategy used to control the number of active replicas. In the two static deployment strategies, the number of consumer replicas is fixed by configuring the `replicas` field of the Deployment manifest. Conversely, in the KEDA-based scenarios, the number of running consumers is dynamically managed through the `ScaledObject` resource, whose configuration will be presented in the following sections. For this reason, the RabbitMQ manifests, the ServiceMonitor, and the unchanged portions of the consumer Deployment are omitted from this chapter. The only new Kubernetes resources introduced for the comparative evaluation are the producer

Deployment and the corresponding Service. Together, they transform the producer from a simple one-shot traffic generator into a persistent HTTP-based workload generation service, which is employed consistently across all four experimental scenarios analysed in this chapter.

19.5 Controlled Workload Triggering

Once the producer Deployment and the corresponding Service have been deployed, workload generation can be initiated by sending an HTTP request to the `/generate` endpoint. Since all experiments were conducted on a local Kubernetes cluster, the producer `ClusterIP` Service was accessed through Kubernetes port forwarding. This mechanism maps a local port on the host machine to the HTTP port exposed by the Kubernetes Service, allowing the producer to be reached through standard HTTP requests issued from the local environment. Before triggering the workload, a port-forwarding session must be established and kept active throughout the experiment. The command used to expose the producer Service on the local machine is reported in Listing 19.5:

```
kubectl port-forward svc/producer-service 5000:5000
```

This command forwards port `5000` of the local machine to port `5000` of the Kubernetes Service named `producer-service`. Consequently, the Flask application running inside the producer pod becomes accessible through `localhost:5000`, allowing HTTP requests to be sent directly from the host machine. The port-forwarding session must remain active while workload generation is being performed. Once the forwarding session has been established, the workload can be triggered through the HTTP request reported in Listing 19.5:

```
curl -X POST http://localhost:5000/generate \
     -H "Content-Type: application/json" \
     -d '{"messages":100,"rate":2}'
```

The request is sent to the `/generate` endpoint exposed by the Flask-based producer, while the JSON payload specifies the workload parameters adopted throughout the experimental campaign. More specifically, the producer generates a total of **100 messages** at a constant rate of **2 messages per second**. Under these conditions, the theoretical duration of the traffic generation phase is approximately **50 seconds**. Using an identical workload across all four deployment strategies ensures that the comparison is performed under exactly

the same input traffic conditions. Since each consumer requires approximately **5 seconds** to process a single message, its processing capacity corresponds to

$$\frac{1}{5} = 0.2 \text{ messages/s}.$$

Consequently, sustaining an incoming traffic rate of **2 messages/s** without accumulating a queue backlog theoretically requires

$$\frac{2}{0.2} = 10$$

simultaneously active consumers. This value represents an important theoretical reference for interpreting the experimental results presented in the following sections. Configurations employing significantly fewer than ten consumers are expected to be underprovisioned and therefore unable to sustain the incoming workload without queue growth. Conversely, configurations with substantially more than ten active consumers provide processing capacity exceeding the workload requirements and are consequently expected to exhibit resource overprovisioning.

19.6 Metrics and Evaluation Methodology

To ensure a fair comparison among the four deployment strategies, all experiments are evaluated under the same workload conditions using a common set of metrics collected by Prometheus and visualized through Grafana. The objective of the analysis is twofold: to assess the performance of each deployment strategy and to evaluate its resource efficiency. Performance is primarily related to the ability of the system to process the incoming workload with limited backlog accumulation, whereas resource efficiency is assessed by analysing how many consumer replicas are allocated, how many are actively processing messages, and how many remain idle during the experiment. The Grafana dashboard developed for this analysis is composed of two complementary types of panels:

- **Time-series panels**, which illustrate the temporal evolution of the system during the workload generation, queue filling, and queue draining phases.

- **Bar gauge panels**, which provide compact summary metrics to facilitate the comparison among the four deployment strategies.

Overall, the dashboard comprises **17 metrics**, grouped into five panels: Three time-series panels and two bar gauge panels. Although the main text focuses on the interpretation and analysis of these metrics, the complete PromQL queries used to compute each metric and build the Grafana dashboard are reported in Appendix A. The queries are organized according to the corresponding dashboard panels in order to facilitate the reproducibility of the monitoring configuration and the experimental results.

19.6.1 Time-Series Panels

The time-series panels describe the dynamic evolution of the system throughout the experiment, allowing the interaction between workload generation, queue behavior, and consumer activity to be analyzed over time.

Messages Rates:

The *Messages Rates* panel reports the evolution of the message rates observed during the experiment and is primarily used to analyze the dynamic behavior of the system during both workload generation and message processing. The panel includes the following metrics:

Incoming Rate The *Incoming Rate* represents the rate at which the producer publishes messages to RabbitMQ. Throughout all experiments presented in this chapter, the producer generates messages at a constant rate of **2 messages/s**. Consequently, this metric is mainly used to verify that the workload is injected into the queue according to the expected traffic profile.

Delivery Rate The *Delivery Rate* measures the rate at which RabbitMQ delivers messages from the queue to the consumers. It indicates how quickly messages leave the *ready* state and are assigned to available consumers. Whenever sufficient processing capacity is available, the delivery rate is expected to closely follow the incoming rate. Conversely, in underprovisioned or cold-start scenarios, it may be limited by the number of available consumers.

Ack Rate The *Ack Rate* represents the rate at which consumers acknowledge messages after completing their processing. Since each consumer processes one message at a time and sends an acknowledgment only after processing has finished, this metric directly reflects the effective processing throughput of the consumer application.

Queue State and Backlog:

The *Queue State and Backlog* panel illustrates the evolution of the queue throughout the experiment. Its main purpose is to distinguish messages still waiting to be processed from those that have already been assigned to consumers. The panel includes the following metrics:

Ready Queue Messages The *Ready Queue Messages* metric represents the number of messages waiting in RabbitMQ that have not yet been delivered to any consumer. It is the primary indicator of backlog accumulation: a large number of ready messages indicates that the available processing capacity is insufficient to immediately sustain the incoming workload. This is also the metric employed by KEDA in the autoscaling scenarios. As described in the previous chapter, KEDA retrieves this value from Prometheus through the Prometheus scaler and uses it as the external metric to control the scaling of the consumer Deployment.

Unacked Queue Messages The *Unacked Queue Messages* metric represents the number of messages that have already been delivered to consumers but have not yet been acknowledged. Although these messages no longer contribute to the queue backlog, they are still being processed by the consumers. Since each consumer operates with `prefetch_count = 1`, RabbitMQ delivers at most one unacknowledged message to each consumer. Consequently, the number of unacknowledged messages directly corresponds to the number of consumers currently processing a task.

Total Queue Messages The *Total Queue Messages* metric represents the total workload still present within the system, including both messages waiting in the queue (*ready*) and those currently being processed (*unacknowledged*). This metric is particularly useful for evaluating the overall workload completion

time. Even after the backlog has been completely drained, messages may still remain in the unacknowledged state while consumers finish processing them.

Consumer State:

The *Consumer State* panel describes the evolution of the consumer Deployment during the experiment, allowing the number of active, busy, and idle consumers to be monitored over time. The panel includes the following metrics:

Active Consumers The *Active Consumers* metric represents the number of consumer replicas currently connected to RabbitMQ. In the static scenarios, this value remains constant because the number of replicas is fixed in the Deployment manifest. In contrast, during the KEDA-based experiments, it varies dynamically according to the scaling decisions performed by the Horizontal Pod Autoscaler generated by KEDA.

Busy Consumers The *Busy Consumers* metric represents the number of consumers actively processing messages. Since the consumers operate sequentially with `prefetch_count = 1`, each consumer can process only one message at any given time. Therefore, the number of busy consumers coincides with the number of unacknowledged messages.

Idle Consumers The *Idle Consumers* metric represents the number of active consumer replicas that are not processing any message. It is computed as the difference between the number of active consumers and the number of busy consumers. This metric plays an important role in evaluating resource efficiency, since it directly quantifies the amount of overprovisioned processing capacity allocated during the experiment.

19.6.2 Bar Gauge Panels

While the time-series panels provide a detailed view of the system evolution over time, the bar gauge panels summarize the most relevant indicators of each experiment through a set of aggregate metrics. These panels facilitate the comparison among the four deployment strategies by highlighting both resource utilization and workload processing performance.

Consumer Efficiency Summary:

The *Consumer Efficiency Summary* panel provides a compact overview of consumer allocation and utilization during the workload processing window. The reported metrics are primarily intended to evaluate the resource efficiency of the different deployment strategies. The panel includes the following metrics:

Peak Active Consumers The *Peak Active Consumers* metric represents the maximum number of consumer replicas observed during the experiment. It identifies the highest scale reached by the system. In the static scenarios, this value coincides with the fixed number of deployed replicas, whereas in the KEDA-based scenarios it corresponds to the maximum number of replicas reached during the autoscaling process.

Avg Active Consumers The *Avg Active Consumers* metric represents the average number of active consumer replicas during the workload processing window. It indicates the average amount of processing capacity allocated while messages are present in the system. This metric is particularly useful for comparing static provisioning with dynamic autoscaling, since it quantifies the average amount of resources allocated throughout the experiment.

Avg Busy Consumers The *Avg Busy Consumers* metric represents the average number of consumers actively processing messages during the workload processing window. It therefore quantifies the fraction of the allocated processing capacity that is effectively used to execute the workload.

Avg Idle Consumers The *Avg Idle Consumers* metric represents the average number of active consumers that are not processing any message during the workload processing window. It provides a direct measure of overprovisioned processing capacity: large values indicate that several consumers remained allocated without contributing to workload execution.

Queue and Processing Summary:

The *Queue and Processing Summary* panel provides a compact overview of the queue dynamics and the overall processing performance of the system. The panel includes the following metrics:

Load Duration The *Load Duration* metric represents the total time required to process the workload. More precisely, it measures the interval between the arrival of the first message in the queue system and the acknowledgement of the last processed message. Consequently, this metric includes both the queue filling phase and the complete queue draining phase. The workload is therefore considered active as long as at least one message is still present in the system, either waiting in the *ready* state or being processed in the *unacknowledged* state.

Peak Ready Messages The *Peak Ready Messages* metric represents the maximum number of messages simultaneously accumulated in the *ready* state during the experiment. It therefore quantifies the maximum backlog reached by the system. This metric is particularly relevant in underprovisioned and KEDA-based scenarios, where the available processing capacity may initially be insufficient to sustain the incoming workload.

Backlog Duration The *Backlog Duration* metric measures the time interval during which messages remain in the *ready* state. It starts when the first message enters the queue and ends when the last ready message is delivered to a consumer, thus leaving the queue and entering the *unacknowledged* state. Unlike the *Load Duration*, this metric excludes the final processing tail, during which messages may still be present in the *unacknowledged* state. Consequently, it quantifies only the duration of the actual backlog, namely the period during which at least one message is waiting to be assigned to a consumer.

Peak Busy Consumers The *Peak Busy Consumers* metric represents the maximum number of consumers simultaneously processing messages during the experiment. Since each consumer can process at most one message at a time, this value directly corresponds to the peak number of unacknowledged messages. This metric can be directly compared with the theoretical number of consumers required to sustain the incoming workload without backlog accumulation, providing a useful reference for evaluating the effectiveness of each deployment strategy.

19.6.3 Computation Methodology

Although both instantaneous and aggregate metrics are reported by the Grafana dashboard, particular attention must be paid to the methodology adopted for computing the average values. Defining a consistent observation window is essential to ensure a fair comparison among the different deployment strategies, especially when comparing static deployments with event-driven autoscaling.

Workload Processing Window

Average-based metrics are computed exclusively over the *workload processing window*, namely the time interval during which the queue contains messages either in the *ready* state or in the *unacknowledged* state. Restricting the computation to this interval avoids including periods that are not representative of the actual workload execution, such as:

- The interval preceding workload generation, during which consumer replicas may already be active although no messages have yet been produced
- The interval following the complete processing of the workload, during which some consumer replicas may still remain active even though no messages are left in the system

This distinction is particularly important for the KEDA-based scenarios. After the queue has been completely drained, KEDA and the Horizontal Pod Autoscaler (HPA) may intentionally keep consumer replicas active because of mechanisms such as the HPA scale-down stabilization window and the KEDA cooldown period. Including these post-processing intervals in the computation of average consumer metrics would bias the comparison between static and autoscaled deployments, since the resulting averages would be influenced by autoscaling control policies rather than by the actual workload processing activity. For this reason, metrics such as *Avg Active Consumers*, *Avg Busy Consumers*, and *Avg Idle Consumers* are evaluated exclusively during the queue filling and draining phases. The behaviour introduced by the cooldown period and the stabilization window is analysed separately in the sections dedicated to the KEDA-based deployment strategies.

Theoretical Reference

The theoretical reference adopted throughout the analysis is derived directly from the workload configuration. The producer generates a total of **100 messages** at a constant rate of **2 messages/s**. Consequently, the workload generation phase lasts approximately

$$\frac{100}{2} = 50 \text{ s.}$$

Since each consumer requires approximately **5 seconds** to process a single message, the processing capacity of one consumer is:

$$\frac{1}{5} = 0.2 \text{ messages/s.}$$

Therefore, sustaining an incoming workload of **2 messages/s** without accumulating backlog theoretically requires

$$\frac{2}{0.2} = 10$$

simultaneously busy consumers. This theoretical value serves as a reference throughout the experimental analysis. Configurations employing fewer than ten active consumers are expected to accumulate backlog, whereas configurations with significantly more than ten consumers provide excess processing capacity at the expense of resource efficiency.

Measurement Resolution

The reported durations and peak values should be interpreted considering the temporal resolution of the monitoring infrastructure. RabbitMQ metrics are collected by Prometheus with a scrape interval of approximately **5 seconds**, while Grafana evaluates the corresponding queries over the selected observation window using the same sampling interval. Consequently, metrics such as *Load Duration*, *Backlog Duration*, and the reported peak values may be affected by a small quantization error. Nevertheless, all deployment strategies are evaluated using the same monitoring configuration and identical dashboard queries. Although individual measurements may therefore exhibit a limited timing uncertainty, the comparison among the different scenarios remains consistent and statistically meaningful.

19.6.4 Summary of Dashboard Metrics

Table 19.1 summarizes all the metrics presented in the previous sections together with their corresponding meaning. This table provides a concise reference for the interpretation of the experimental results discussed throughout the remainder of the chapter.

Panel	**Metric**	**Meaning**
Consumer Efficiency Summary	Peak Active Consumers	Maximum number of active consumer replicas observed during the experiment.
Consumer Efficiency Summary	Avg Active Consumers	Average number of active consumers during the workload processing window.
Consumer Efficiency Summary	Avg Busy Consumers	Average number of consumers processing messages during the workload processing window.
Consumer Efficiency Summary	Avg Idle Consumers	Average number of active consumers not processing messages during the workload processing window.
Queue & Processing Summary	Load Duration	Time from the arrival of the first message until the last unacknowledged message is processed and acknowledged.
Queue & Processing Summary	Peak Ready Messages	Maximum number of messages accumulated in the *ready* state.
Queue & Processing Summary	Backlog Duration	Time during which messages remain in the *ready* state, excluding the final unacknowledged processing phase.
Queue & Processing Summary	Peak Busy Consumers	Maximum number of consumers simultaneously processing messages.

Continued on next page

Panel	Metric	Meaning
Messages Rates	Incoming Rate	Rate at which the producer publishes messages to RabbitMQ.
Messages Rates	Delivery Rate	Rate at which RabbitMQ delivers messages to consumers.
Messages Rates	Ack Rate	Rate at which consumers acknowledge processed messages.
Queue State and Backlog	Total Queue Messages	Total number of messages present in the system, including both *ready* and *unacknowledged* messages.
Queue State and Backlog	Unacked Queue Messages	Messages delivered to consumers but not yet acknowledged.
Queue State and Backlog	Ready Queue Messages	Messages waiting in the queue that have not yet been delivered to consumers.
Consumer State	Active Consumers	Consumer replicas currently connected to the RabbitMQ queue.
Consumer State	Busy Consumers	Consumers currently processing messages.
Consumer State	Idle Consumers	Active consumers not currently processing messages.

Table 19.1: Summary of the Grafana dashboard metrics used throughout the experimental evaluation.

19.7 Reference Estimation of Idle Consumer CPU Usage

Before analysing the four deployment strategies, an additional experiment was conducted to estimate the average CPU usage of a generic consumer while it

is active but not processing any messages. The resulting value constitutes the reference CPU consumption of a generic idle consumer and is subsequently used throughout the remainder of the chapter to estimate the processing resources associated with idle consumer replicas. This preliminary measurement is necessary because RabbitMQ metrics allow the number of idle consumers to be determined, but they do not identify which specific consumer pods are idle at any given time. During workload execution, messages may be assigned to different consumers as processing progresses. Consequently, although the dashboard reports the number of idle consumers, the identity of those consumers continuously changes throughout the experiment. For this reason, monitoring the CPU usage of a single consumer during workload execution would not provide a reliable estimate of the CPU consumption of a permanently idle consumer. To eliminate this source of variability, the measurement was performed under stable idle conditions. In this configuration, fifteen consumer pods were deployed, connected to the RabbitMQ broker, and kept active without generating any workload. The queue therefore remained empty, with no messages in either the *ready* or the *unacknowledged* state. As a consequence, every consumer remained idle throughout the entire observation period, and its CPU usage was exclusively associated with maintaining the running process and the connection to the RabbitMQ broker. Although all consumer replicas execute the same application code and are therefore functionally equivalent, their CPU usage is not perfectly constant over time. Small temporal fluctuations were observed for each consumer, mainly due to the normal runtime activity of the process and the lightweight operations required to maintain the connection with the message broker. Similar variations were also observed among different consumer pods. Nevertheless, all measured values remained extremely low and within the same order of magnitude. The base metric adopted for this analysis is `container_cpu_usage_seconds_total`, a cumulative Prometheus counter representing the total CPU time consumed by each container. Since cumulative counters cannot be directly interpreted as CPU utilization, the CPU usage was obtained by applying a rate function over an appropriate observation window. During the experiments, the container CPU metric was observed to be scraped by Prometheus with an effective interval of approximately **15 seconds**. After deploying the fifteen idle consumers, the system was left running for approximately **5–6 minutes** before collecting the final measurements. This stabilization period ensured that the five-minute rate window used to compute CPU utilization contained only samples corresponding to the

steady idle condition. Consequently, the measured values were not affected by the consumer startup phase or by any transient behaviour preceding the experiment.

19.7.1 Time-Series Analysis of Idle Consumer CPU Usage

The first Grafana panel was used to analyse the temporal evolution of the CPU usage of the idle consumer pods. The PromQL query employed to generate the corresponding time-series visualization is reported in Listing 19.5:

```
1  1000 *
2  sum(
3      max by (cluster, namespace, pod, container)(
4        node_namespace_pod_container:
       container_cpu_usage_seconds_total:sum_rate5m{
5          cluster="$cluster",
6          namespace="$namespace"
7        }
8      )
9    * on(cluster, namespace, pod)
10     group_left(workload, workload_type)
11     namespace_workload_pod:kube_pod_owner:relabel{
12       cluster="$cluster",
13       namespace="$namespace",
14       workload=~"$workload",
15       workload_type=~"$type"
16     }
17 ) by (pod)
```

Listing 19.5: PromQL query used to visualize the CPU usage of idle consumer pods over time.

The query is based on the Kubernetes recording rule `node_namespace_pod_container:container_cpu_usage_seconds_total:sum_rate5m`, which computes the five-minute rate of the cumulative CPU usage counter. The resulting values are grouped by pod and multiplied by **1000** to express CPU usage in millicores (mCPU). This visualization was used to verify that the CPU consumption of idle consumers remained consistently low throughout

the observation period and was affected only by small runtime fluctuations.

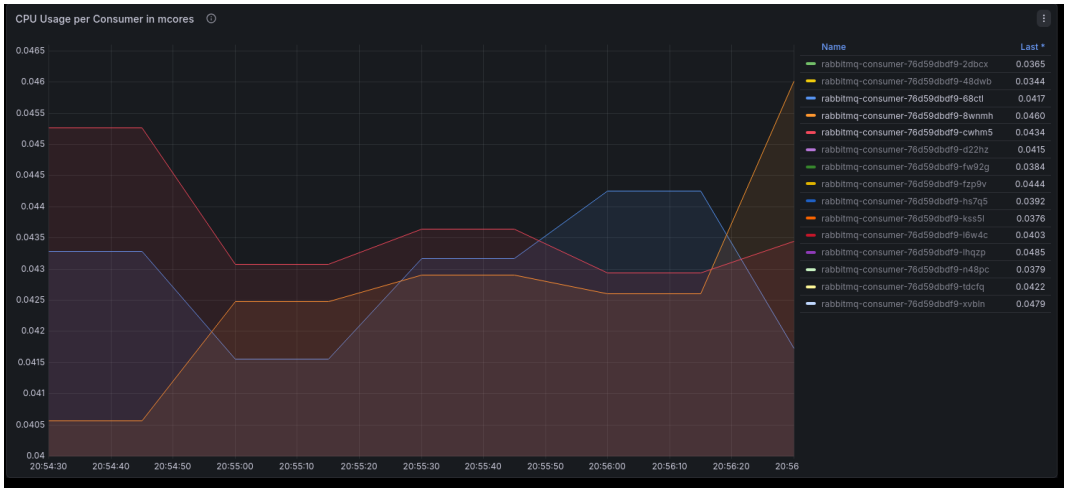


Figure 19.1: CPU usage time series of representative idle consumer pods.

Figure 19.1 illustrates the CPU usage of a representative subset of idle consumer pods during the selected two-minute observation window. Only a limited number of time series is displayed to improve readability, since plotting all fifteen consumers simultaneously would produce an excessively crowded graph. Although small temporal fluctuations can be observed, the CPU usage remains consistently low for all the reported consumer pods, confirming that the idle overhead is both limited and comparable across different replicas.

19.7.2 Average CPU Usage of Individual Idle Consumers

After verifying the stability of the idle CPU usage trend, the final observation window in Grafana was restricted to **2 minutes**. Since CPU usage samples were available approximately every **15 seconds**, this observation window contained about eight measurements for each consumer pod. The PromQL query used to compute the average CPU usage of each idle consumer over the selected observation window is reported in Listing 19.6:

```
1 1000 *
2 avg_over_time(
3   (
```

```
    sum by (pod) (
      rate(container_cpu_usage_seconds_total{
        namespace="default",
        pod=~"rabbitmq-consumer.*",
        container!="",
        container!="POD"
      }[5m])
    )
  )[2m:15s]
)
```

Listing 19.6: PromQL query used to compute the average CPU usage of each idle consumer pod.

The query first computes the CPU usage of each consumer pod by applying the `rate()` function over a five-minute window to the cumulative metric `container_cpu_usage_seconds_total`. Subsequently, the `avg_over_time()` function computes the average CPU usage over the selected two-minute observation window using a 15-second evaluation step. As a result, the query returns one average CPU usage value for each idle consumer pod.

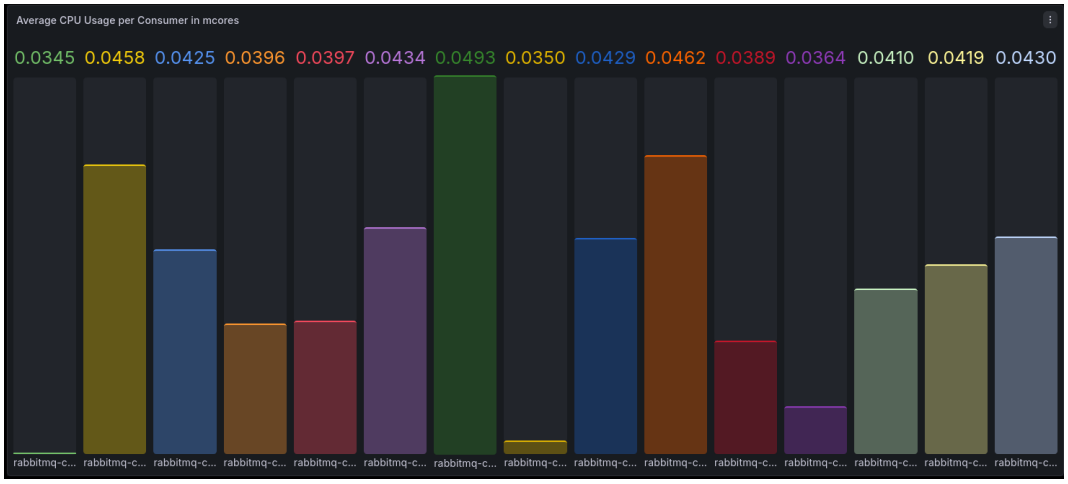


Figure 19.2: Average CPU usage of each idle consumer pod over the selected observation window.

Figure 19.2 reports the average CPU usage measured for each of the fifteen idle consumer pods during the selected observation window. Each bar corresponds

to one consumer replica and the reported values are expressed in millicores (mCPU). Although all consumers execute the same application and remain idle throughout the experiment, small differences can be observed among the measured values due to normal runtime fluctuations. Nevertheless, all measurements remain extremely low and belong to the same order of magnitude, confirming the consistency of the experimental setup. The average CPU usage values measured for the fifteen idle consumer pods are reported below:

$$0.0345,\ 0.0458,\ 0.0425,\ 0.0396,\ 0.0397,$$
$$0.0434,\ 0.0493,\ 0.0350,\ 0.0429,\ 0.0462,$$
$$0.0389,\ 0.0364,\ 0.0410,\ 0.0419,\ 0.0430 \text{ mCPU}.$$

19.7.3 Reference CPU Usage of a Generic Idle Consumer

Finally, the PromQL query reported in Listing 19.7 was used to compute the average of the per-pod average CPU usage values, thus obtaining a single reference value representative of a generic idle consumer.

```
1000 *
avg(
  avg_over_time(
    (
      sum by (pod) (
        rate(container_cpu_usage_seconds_total{
          namespace="default",
          pod=~"rabbitmq-consumer.*",
          container!="",
          container!="POD"
        }[5m])
      )
    )[2m:15s]
  )
)
```

Listing 19.7: PromQL query used to compute the reference CPU usage of a generic idle consumer.

The query first computes the average CPU usage of each consumer pod over the selected two-minute observation window and subsequently averages these values across the fifteen idle consumer replicas. The resulting value therefore represents the estimated average CPU usage of a generic consumer while it is active, connected to RabbitMQ, and not processing any messages.



Figure 19.3: Reference CPU usage of a generic idle consumer.

Figure 19.3 reports the final reference value obtained by averaging the per-pod CPU usage measurements collected from the fifteen idle consumer replicas. The resulting average CPU usage is approximately **0.0413 mCPU**, which is subsequently adopted throughout the remainder of the chapter to estimate the CPU consumption associated with idle consumer replicas in each deployment strategy. The final reference value obtained from the experimental measurement is therefore:

$$\overline{CPU}_{idle} = 0.0413 \text{ mCPU}$$

Expressed in CPU cores, this corresponds to:

$$0.0413 \text{ mCPU} = 0.0000413 \text{ cores.}$$

Since one CPU core corresponds to one CPU-second per second, the estimated CPU consumption of a generic idle consumer can also be expressed as:

$$0.0000413 \text{ CPU-seconds/s.}$$

Throughout the analysis of the four deployment strategies, the CPU usage associated with idle consumers is estimated according to:

$$CPU_{idle} = AvgIdleConsumers \times 0.0413 \text{ mCPU}$$

where `AvgIdleConsumers` denotes the average number of idle consumers measured during the workload processing window of the corresponding experiment.

Whenever an estimate of the cumulative CPU time is required, the average idle CPU usage can be converted from millicores to CPU cores and multiplied by the workload duration, yielding

$$CPUTime_{idle} = CPU_{idle}^{(cores)} \times LoadDuration$$

The measurements discussed throughout this section were collected using a dedicated Grafana dashboard specifically developed for the idle CPU characterization experiment. Although each panel has been analysed separately in order to explain the associated metrics and PromQL queries, the complete dashboard is shown in Figure 19.4. It combines the CPU time-series measurements, the per-pod average CPU usage, and the final reference CPU value used throughout the remainder of the experimental evaluation.

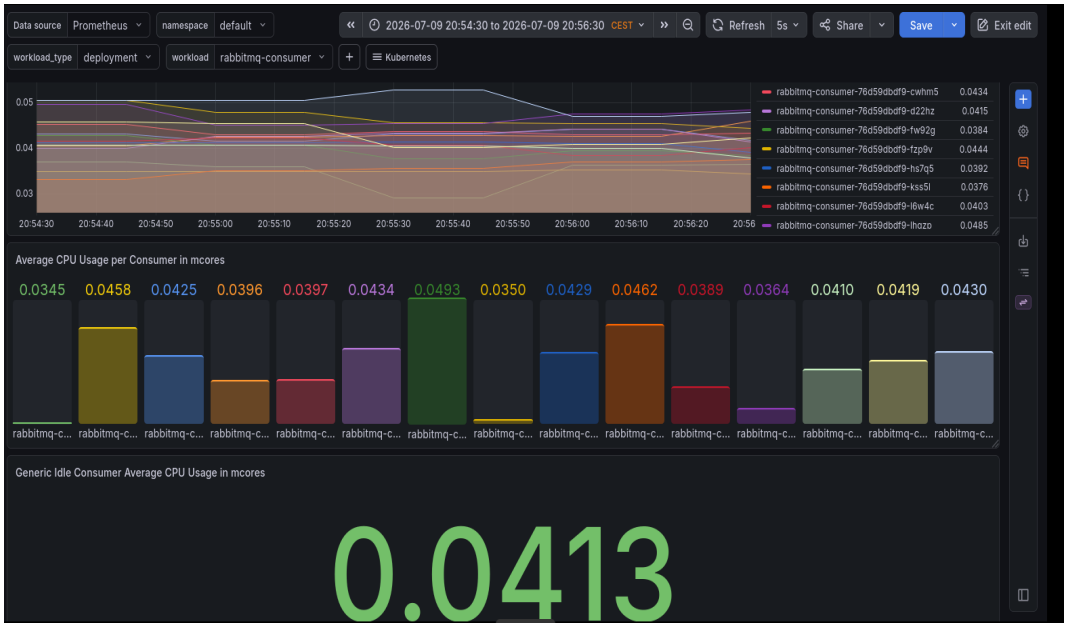


Figure 19.4: Complete Grafana dashboard used to estimate the reference CPU usage of a generic idle consumer. The dashboard combines the CPU usage time series of representative idle consumer pods, the average CPU usage measured for each idle consumer, and the final reference CPU value adopted throughout the comparative evaluation of the four deployment strategies.

This methodology makes it possible to distinguish between *logical overprovisioning*, quantified by the number of active but idle consumer replicas, and

the corresponding *actual computational cost*, quantified by the CPU resources effectively consumed by those idle processes.

19.8 Static Underprovisioned Scenario

19.8.1 Scenario Description

The first deployment strategy corresponds to a *static underprovisioned configuration*. In this scenario, the number of consumer replicas is manually fixed to **2** through the Kubernetes Deployment manifest. Since no autoscaling mechanism is enabled, the number of active consumers remains constant throughout the entire experiment. This scenario serves as a baseline for evaluating the behaviour of a system whose allocated processing capacity is intentionally lower than the incoming workload. The objective is to analyse how the RabbitMQ queue evolves when the available consumers are unable to sustain the message production rate. The workload configuration is identical to that adopted throughout the experimental campaign. The producer generates **100 messages** at a constant rate of **2 messages/s**. Since each consumer requires approximately **5 seconds** to process a message, the processing capacity of a single consumer is

$$1 \text{ consumer} = \frac{1 \text{ message}}{5 \text{ s}} = 0.2 \text{ messages/s}$$

With two active consumers, the overall processing capacity becomes

$$2 \text{ consumers} \times 0.2 \text{ messages/s} = 0.4 \text{ messages/s}$$

which is significantly lower than the incoming traffic rate

$$\text{Incoming rate} = 2 \text{ messages/s}$$
$$\text{Processing capacity} = 0.4 \text{ messages/s}$$

Consequently, during the workload generation phase, the queue is expected to accumulate backlog at an average rate of

$$2 \text{ messages/s} - 0.4 \text{ messages/s} = 1.6 \text{ messages/s}$$

Since the producer injects messages for approximately **50 seconds**, the expected backlog accumulated during the traffic generation phase is

$$1.6 \text{ messages/s} \times 50 \text{ s} = 80 \text{ messages}$$

This theoretical value provides an important reference for interpreting the experimental results presented in the following sections.

19.8.2 Summary Metrics

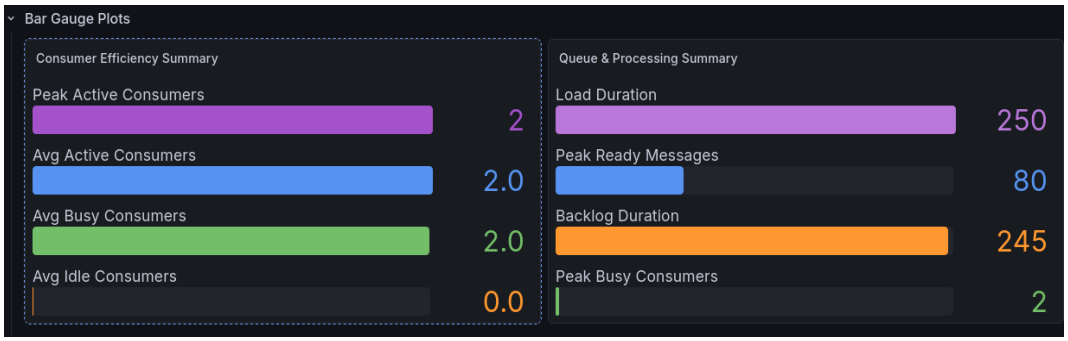


Figure 19.5: Summary metrics for the static underprovisioned scenario.

Figure 19.5 summarizes the most relevant consumer efficiency and queue processing metrics for the static deployment with two consumer replicas. As expected, both consumers remain continuously busy throughout the workload processing window, resulting in no idle consumers. However, the limited processing capacity leads to a substantial backlog, with a peak of approximately **80 ready messages** and an overall workload duration of about **250 seconds**. The reported metrics confirm the expected behaviour of an underprovisioned deployment. The *Peak Active Consumers* metric is equal to **2**, corresponding to the fixed number of replicas configured in the Deployment. Since no autoscaling mechanism is employed, the number of active consumers remains constant for the entire duration of the experiment. Likewise, both *Avg Active Consumers* and *Avg Busy Consumers* are equal to **2.0**, indicating that both consumer replicas remain continuously active and busy while messages are present in the system. Consequently, the *Avg Idle Consumers* metric is equal to **0.0**, confirming that no allocated consumer remains unused during workload execution. From the perspective of resource utilization, this configuration is therefore highly efficient, as every allocated consumer actively contributes to message processing. Nevertheless, maximum utilization should not be interpreted as optimal system performance. In this case, the available processing

capacity is simply insufficient to sustain the incoming workload. The queue-related metrics clearly highlight this limitation. The measured *Peak Ready Messages* value is equal to **80**, perfectly matching the theoretical backlog derived previously. Similarly, the measured *Load Duration* is approximately **250 seconds**, which is consistent with the theoretical processing time:

$$\frac{100}{0.4} = 250 \text{ s.}$$

The *Backlog Duration* is approximately **245 seconds**. This value is slightly shorter than the overall load duration because it only accounts for the period during which messages remain in the *ready* state. Once the final ready messages have been delivered to the consumers, they continue to be processed in the *unacknowledged* state before the corresponding acknowledgements are sent. Given the five-second processing time of each consumer, the observed difference between the two metrics is fully consistent with the expected system behaviour. Finally, the *Peak Busy Consumers* metric reaches the value of **2**, confirming that both available consumers are simultaneously processing messages during the busiest phase of the experiment. Since only two consumer replicas are deployed, this represents the maximum processing capacity achievable in this configuration.

19.8.3 Message Rate Analysis

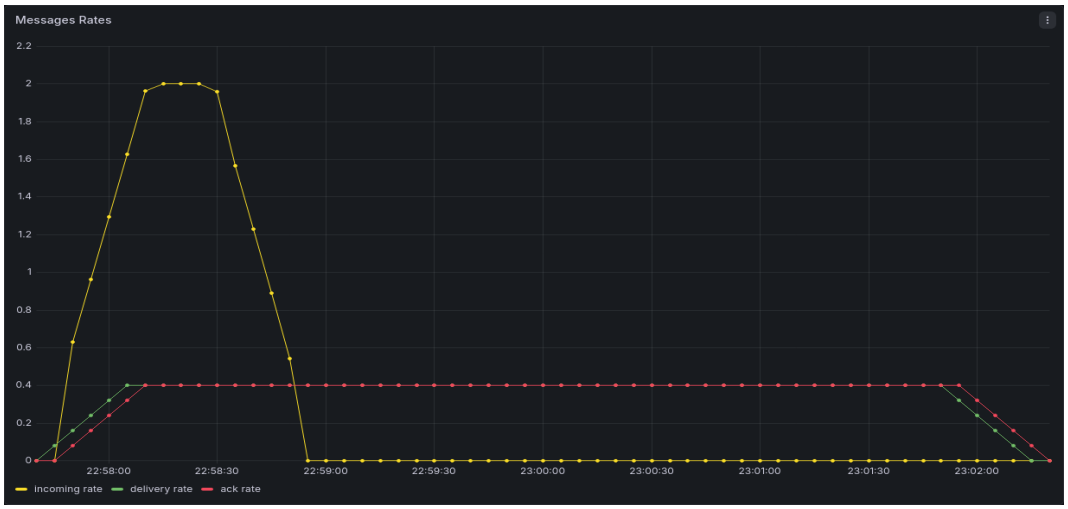


Figure 19.6: Message rates in the static underprovisioned scenario. The incoming rate reaches approximately 2 messages/s during the workload generation phase, whereas both the delivery and acknowledgement rates remain close to 0.4 messages/s. This confirms that the available consumer capacity is insufficient to sustain the incoming workload.

Figure 19.6 illustrates the temporal evolution of the message rates throughout the experiment, providing a direct comparison between the workload generated by the producer and the processing capacity of the consumers. The *Incoming Rate* reaches the configured value of approximately **2 messages/s** during the workload generation phase, confirming that the HTTP producer injects traffic into RabbitMQ according to the expected workload profile. By contrast, both the *Delivery Rate* and the *Ack Rate* remain close to **0.4 messages/s** for most of the experiment. This value coincides with the theoretical processing capacity of the two deployed consumers:

$$2 \text{ consumers} \times 0.2 \text{ messages/s} = 0.4 \text{ messages/s}.$$

The *Delivery Rate* represents the rate at which RabbitMQ assigns messages to the consumers, whereas the *Ack Rate* corresponds to the rate at which consumers complete message processing and send the corresponding acknowledgements to the broker. Since each consumer requires approximately five

seconds to process a message and only two consumer replicas are available, the acknowledgement rate cannot exceed approximately **0.4 messages/s**. The graph clearly highlights the main bottleneck of this deployment strategy. During the workload generation phase, messages enter the system at approximately **2 messages/s**, while they are processed at only about **0.4 messages/s**. The difference between these two rates continuously accumulates as *ready* messages within the RabbitMQ queue, leading to the backlog discussed in the previous section. Once the producer completes the workload generation, the *Incoming Rate* immediately drops to zero. However, both the *Delivery Rate* and the *Ack Rate* remain active for a considerably longer period, as the consumers continue processing the backlog accumulated during the injection phase. This behaviour is characteristic of an underprovisioned queue-based system: although the workload generation phase is relatively short, the overall execution time is dominated by the backlog draining phase, which results from the insufficient processing capacity of the consumer deployment.

19.8.4 Queue State and Backlog Analysis

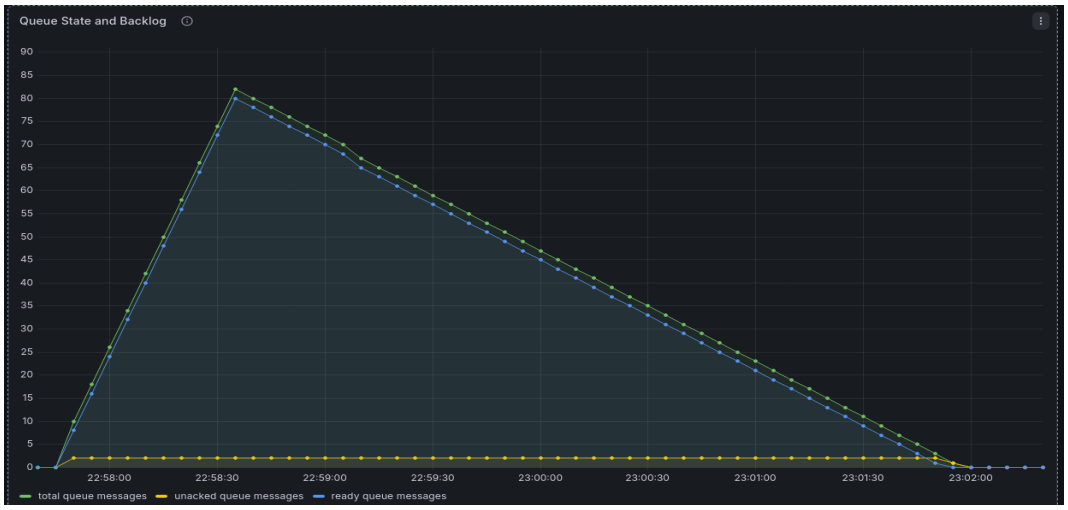


Figure 19.7: Queue state and backlog in the static underprovisioned scenario. The number of ready messages grows rapidly during the workload generation phase, reaching a peak of approximately 80 messages. After the producer stops generating traffic, the backlog is gradually drained by the two consumers, whose combined processing capacity is approximately 0.4 messages/s.

Figure 19.7 illustrates the evolution of the queue throughout the experiment and clearly highlights the effects of the processing capacity mismatch between the producer and the consumer deployment. During the workload generation phase, the number of *Ready Queue Messages* increases almost linearly. This behaviour is expected, since the producer injects messages at a higher rate than the consumers are able to process them. The peak number of ready messages reaches approximately **80**, perfectly matching the theoretical estimation. During the 50-second workload generation phase, the producer injects a total of 100 messages, whereas the two consumers can process only

$$0.4 \text{ messages/s} \times 50 \text{ s} = 20 \text{ messages}.$$

Consequently, the expected backlog at the end of the generation phase is

$$100 \text{ produced messages} - 20 \text{ processed messages} = 80 \text{ ready messages},$$

which is fully consistent with the peak value reported by the dashboard. The *Total Queue Messages* curve remains slightly above the *Ready Queue Messages* curve throughout the experiment. This behaviour is expected because the total number of queue messages includes both the messages waiting in the *ready* state and those already delivered to consumers but still present in the *unacknowledged* state. Since both consumers remain continuously busy, approximately two messages are almost always being processed while the backlog exists. Consequently, when the ready queue reaches about 80 messages, the total number of messages still present in the system is slightly higher because it also accounts for the messages currently under processing. After the producer completes the workload generation, no additional messages enter the system and the ready queue begins to decrease almost linearly. From this point onward, the only active process is the gradual draining of the accumulated backlog by the two consumers. Given the available processing capacity, draining approximately 80 ready messages theoretically requires

$$\frac{80 \text{ messages}}{0.4 \text{ messages/s}} = 200 \text{ s}.$$

Adding this draining time to the initial workload generation phase yields an expected total workload duration of

$$50 \text{ s} + 200 \text{ s} = 250 \text{ s},$$

which is fully consistent with the measured *Load Duration* reported in the summary metrics. The queue-state plot also explains the difference between the *Backlog Duration* and the *Load Duration*. The backlog duration ends when the last message leaves the *ready* state and is delivered to a consumer. At that moment, however, the final messages are still being processed and therefore remain in the *unacknowledged* state. Consequently, the workload is not considered completed until the last acknowledgement is sent. This explains why the measured *Backlog Duration* is approximately **245 seconds**, whereas the corresponding *Load Duration* is approximately **250 seconds**.

19.8.5 Consumer State Analysis



Figure 19.8: Consumer state in the static underprovisioned scenario. The number of active consumers remains fixed at two throughout the experiment. Both consumers remain busy for almost the entire workload processing window, while the number of idle consumers stays equal to zero, confirming that the deployment is fully utilized but underprovisioned.

Figure 19.8 illustrates the evolution of the consumer deployment during the experiment and confirms that both deployed consumer replicas remain continuously engaged throughout the workload processing window. The *Active Consumers* metric remains constant at **2**, as expected for a static deployment configured with a fixed number of replicas and no autoscaling mechanism.

Similarly, the *Busy Consumers* metric remains equal to **2** for almost the entire duration of the experiment, indicating that both consumer replicas are continuously processing messages. This behaviour is consistent with the presence of a persistent backlog: whenever a consumer completes the processing of a message and sends the corresponding acknowledgement, RabbitMQ immediately assigns a new ready message to that consumer. Conversely, the *Idle Consumers* metric remains equal to **0** throughout the workload processing window, indicating that no allocated processing capacity remains unused. From the perspective of resource utilization, this deployment therefore achieves maximum utilization, since every available consumer actively contributes to workload processing. However, such a high utilization should not be interpreted as an indicator of an optimal system configuration. In this scenario, the absence of idle consumers is simply a consequence of insufficient processing capacity. The consumer replicas remain fully occupied because the incoming workload continuously exceeds the service rate that the deployment can sustain, causing the system to operate under saturation conditions. This distinction is particularly important from an engineering perspective. A deployment may achieve nearly 100% resource utilization while still exhibiting poor overall performance if the available processing capacity is insufficient to satisfy the workload demand. Although all consumer replicas are efficiently utilized in this scenario, the resulting queueing delay and overall workload completion time remain significantly higher than desired due to the persistent backlog accumulated during message generation.

19.8.6 Idle CPU Usage Estimation

Using the reference idle CPU usage estimated in the previous section, the CPU consumption associated with idle consumers in this scenario can be computed as:

$$\text{Estimated idle CPU usage} = \text{Avg Idle Consumers} \times 0.0413 \text{ mCPU}.$$

Since the average number of idle consumers is equal to **0.0**, the estimated idle CPU usage is:

$$0.0 \times 0.0413 \text{ mCPU} = 0 \text{ mCPU}.$$

Consequently, the corresponding cumulative idle CPU time is also equal to zero:

$$0 \text{ cores} \times 250 \text{ s} = 0 \text{ core} \cdot \text{s}.$$

This result confirms that the static underprovisioned configuration does not waste CPU resources on idle consumer replicas during workload execution. Nevertheless, this apparent efficiency is achieved only because all available consumers remain continuously busy. As highlighted throughout the previous analysis, the absence of idle CPU usage is therefore a direct consequence of resource saturation rather than an indication of an optimal deployment, leading to significant backlog accumulation and a considerably longer workload completion time.

19.8.7 Engineering Discussion

The static underprovisioned scenario represents the lower bound in terms of resource allocation. With only two consumer replicas, it minimizes the number of running pods and, consequently, eliminates any CPU consumption associated with idle consumers. This apparent efficiency, however, comes at a significant performance cost. While the producer injects messages at a rate of **2 messages/s**, the available consumers can collectively process only **0.4 messages/s**. The resulting capacity mismatch generates a backlog that grows at approximately **1.6 messages/s** during the workload generation phase. Consequently, the queue reaches a peak of about **80 ready messages**, and the overall workload completion time increases to approximately **250 seconds**. This scenario clearly highlights the trade-off between resource utilization and system performance. On the one hand, all available consumers remain continuously busy, resulting in maximum utilization and no idle processing capacity. On the other hand, the limited number of replicas causes long queueing delays, as messages spend a considerable amount of time waiting before being processed. From an engineering perspective, this deployment represents a system operating under saturation conditions. Although the available resources are fully exploited, the processing capacity is insufficient to sustain the incoming workload, leading to poor responsiveness and excessive workload completion times. For this reason, the static underprovisioned configuration provides a useful baseline for the subsequent comparisons. It validates the theoretical model presented earlier and demonstrates that achieving high resource utilization alone is not sufficient to obtain good overall system performance. Instead, the allocated processing capacity must be properly dimensioned with respect to the workload characteristics, either through a higher static replica count or by adopting an event-driven autoscaling mechanism capable of increasing the

number of consumers as backlog starts to accumulate.

19.9 Static Overprovisioned Scenario

19.9.1 Scenario Description

The second deployment strategy corresponds to a *static overprovisioned configuration*. In this scenario, the number of consumer replicas is manually fixed to **15** through the Kubernetes Deployment manifest. As in the previous static scenario, no autoscaling mechanism is enabled and, consequently, the number of active consumers remains constant throughout the entire experiment. This configuration represents the opposite condition with respect to the underprovisioned scenario. Instead of allocating fewer consumers than required by the workload, the system is intentionally configured with more processing capacity than strictly necessary. The objective is to verify whether this deployment completely avoids backlog accumulation and to quantify the amount of unused consumer capacity. The workload configuration is identical to that adopted in the previous scenario. The producer generates **100 messages** at a constant rate of **2 messages/s**. Since each consumer requires approximately **5 seconds** to process one message, the processing capacity of a single consumer is:

$$1 \text{ consumer} = \frac{1 \text{ message}}{5 \text{ s}} = 0.2 \text{ messages/s}.$$

With fifteen active consumers, the total theoretical processing capacity becomes:

$$15 \text{ consumers} \times 0.2 \text{ messages/s} = 3 \text{ messages/s}.$$

This value is higher than the incoming workload rate:

$$\text{Incoming rate} = 2 \text{ messages/s},$$
$$\text{Processing capacity} = 3 \text{ messages/s}.$$

Therefore, the available processing capacity is expected to sustain the incoming workload without generating backlog. Nevertheless, the theoretical number of consumers required to process the incoming traffic is:

$$2 \text{ messages/s} \times 5 \text{ s/message} = 10 \text{ consumers}.$$

Consequently, although fifteen consumer replicas remain active throughout the experiment, only about ten are expected to process messages on average, while the remaining five consumers are expected to remain idle.

19.9.2 Summary Metrics

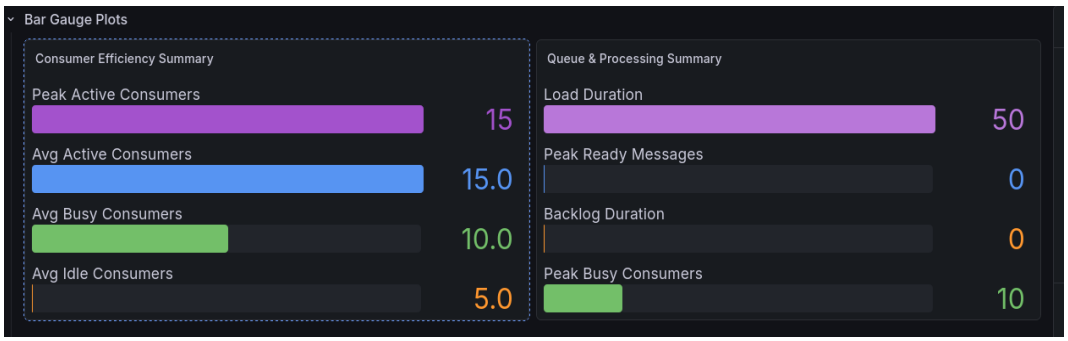


Figure 19.9: Summary metrics for the static overprovisioned scenario. The bar gauge panels report the main consumer efficiency and queue processing metrics for the static deployment with fifteen consumer replicas. The workload is completed in approximately 50 seconds without backlog accumulation. However, only about ten consumers are busy on average, while the remaining five stay idle throughout the workload processing window.

Figure 19.9 summarizes the most relevant consumer efficiency and queue processing metrics for the static overprovisioned deployment. As expected, the workload is completed in approximately **50 seconds** without any backlog accumulation. Nevertheless, only about ten consumers actively process messages on average, while the remaining five remain idle. The reported metrics confirm the expected behavior of an overprovisioned deployment. The *Peak Active Consumers* metric is equal to **15**, corresponding to the fixed number of replicas configured in the Deployment. Since no autoscaling mechanism is employed, the number of active consumers remains constant throughout the experiment. Likewise, the *Avg Active Consumers* metric is equal to **15.0**, confirming that all consumer replicas remain allocated during the workload processing window. In contrast, the *Avg Busy Consumers* metric is equal to **10.0**, perfectly matching the theoretical number of consumers required to sustain the incoming

workload:

$$2 \text{ messages/s} \times 5 \text{ s/message} = 10 \text{ busy consumers.}$$

The remaining replicas are therefore not required to process the incoming traffic. This is reflected by the *Avg Idle Consumers* metric, which is equal to **5.0**. Consequently, approximately one-third of the allocated consumer replicas remain idle during the experiment:

$$\frac{5 \text{ idle consumers}}{15 \text{ active consumers}} = 33.3\%.$$

From a performance perspective, the queue-related metrics exhibit the expected behaviour. The *Peak Ready Messages* metric is equal to **0**, indicating that no messages accumulate in the *ready* state. Similarly, the *Backlog Duration* is equal to **0 seconds**, confirming that the queue never experiences a waiting backlog. The *Load Duration* is approximately **50 seconds**, corresponding to the duration of the workload generation phase. Since the available processing capacity exceeds the incoming traffic rate, messages are processed immediately after being delivered to the consumers, allowing the workload to complete almost simultaneously with the end of message generation. Finally, the *Peak Busy Consumers* metric reaches the value of **10**. This result is particularly significant because it confirms the theoretical analysis: although fifteen consumer replicas are deployed, only about ten are actually required to sustain the configured workload. The remaining five replicas therefore represent overprovisioned processing capacity.

19.9.3 Message Rate Analysis



Figure 19.10: Message rates in the static overprovisioned scenario. The incoming, delivery, and acknowledgement rates closely follow one another throughout the workload generation phase, indicating that the available consumer capacity is sufficient to sustain the incoming workload without backlog accumulation.

Figure 19.10 illustrates the evolution of the message rates throughout the experiment, providing a direct comparison between the workload generated by the producer and the processing capacity of the consumer deployment. The *Incoming Rate* reaches the configured value of approximately **2 messages/s**, confirming that the HTTP producer generates the expected workload. Unlike the underprovisioned scenario, both the *Delivery Rate* and the *Ack Rate* closely follow the incoming traffic throughout the workload generation phase. This behaviour indicates that RabbitMQ is able to deliver messages to available consumers as soon as they arrive, while the consumers process them without generating a persistent backlog. The *Delivery Rate* remains aligned with the incoming traffic because sufficient idle consumers are always available to receive newly published messages. Similarly, the *Ack Rate* follows the expected processing throughput, with only a slight temporal shift caused by the five-second processing time required for each message. The small delay between the delivery and acknowledgement curves is expected. A message first enters the *unacknowledged* state when it is delivered to a consumer and is acknowl-

edged only after the processing phase has been completed. Consequently, the acknowledgement curve is slightly shifted with respect to the delivery curve, while the two rates remain closely aligned throughout the experiment thanks to the available parallel processing capacity. Overall, the message-rate panel confirms that the consumer deployment is not the bottleneck of the system. Since the available processing capacity exceeds the incoming workload, the producer traffic is sustained without queue buildup, allowing messages to be processed almost immediately after they are generated.

19.9.4 Queue State and Backlog Analysis



Figure 19.11: Queue state and backlog in the static overprovisioned scenario. The *Ready Queue Messages* remain equal to zero throughout the experiment, indicating that messages are immediately assigned to available consumers. The *Unacked Queue Messages* reach approximately ten, corresponding to the number of consumers simultaneously processing messages.

Figure 19.11 illustrates the evolution of the queue throughout the experiment and clearly demonstrates the absence of backlog. The *Ready Queue Messages* metric remains equal to **0** for the entire duration of the workload, indicating that messages are delivered to available consumers immediately after they are published, without spending time waiting in the queue. Conversely, the

Unacked Queue Messages metric reaches approximately **10** and remains almost constant during the central part of the experiment. This behaviour is fully consistent with the theoretical model. Since the producer generates messages at a rate of **2 messages/s** and each message requires approximately **5 seconds** to be processed, the expected number of messages being processed simultaneously is:

$$2 \text{ messages/s} \times 5 \text{ s} = 10 \text{ messages}.$$

Since each consumer processes only one message at a time, these ten unacknowledged messages directly correspond to ten busy consumers. This observation is also consistent with the *Peak Busy Consumers* metric reported in the summary panel. The *Total Queue Messages* curve almost completely overlaps the *Unacked Queue Messages* curve because no messages accumulate in the *ready* state. Consequently, the total number of messages present in the system is almost entirely represented by messages currently being processed by the consumer replicas. This result highlights the important distinction between a *backlog-free* system and an *empty* system. During workload execution, the queue does not contain waiting messages, yet the system is far from idle. Instead, the active workload is almost entirely represented by messages in the *unacknowledged* state, which are already assigned to consumers and are actively being processed.

19.9.5 Consumer State Analysis



Figure 19.12: Consumer state in the static overprovisioned scenario. The number of active consumers remains fixed at fifteen throughout the experiment. During the workload processing window, approximately ten consumers are busy while the remaining five stay idle, confirming that the deployment provides more processing capacity than required by the incoming workload.

Figure 19.12 illustrates the evolution of the consumer deployment during the experiment and clearly confirms the overprovisioned nature of this configuration. The *Active Consumers* metric remains constant at **15**, as expected for a static deployment configured with a fixed number of replicas and no autoscaling mechanism. During the workload processing window, the *Busy Consumers* metric rapidly reaches approximately **10** and remains close to this value for most of the experiment. This result is fully consistent with the theoretical analysis, which predicts that approximately ten consumers are sufficient to sustain the configured workload without backlog accumulation. Conversely, the *Idle Consumers* metric remains close to **5** throughout the same interval. This indicates that the deployment provides more processing capacity than actually required. Although these additional consumer replicas remain active and connected to RabbitMQ, they do not contribute to message processing during the workload execution. From an engineering perspective, this deployment achieves excellent performance at the expense of resource allocation

efficiency. The available processing capacity is sufficient to eliminate queueing delays and complete the workload almost simultaneously with the end of traffic generation. However, part of the allocated capacity remains continuously unused. Compared with the static underprovisioned scenario, this configuration represents the opposite side of the resource-performance trade-off. There, all available consumers were fully utilized but the system suffered from severe backlog accumulation and long completion times. Here, the workload is processed without backlog, but this improvement is achieved by allocating more consumer replicas than are actually required.

19.9.6 Idle CPU Usage Estimation

Using the reference idle CPU usage estimated in the previous section, the CPU consumption associated with idle consumers in this scenario can be computed as:

$$\text{Estimated idle CPU usage} = \text{Avg Idle Consumers} \times 0.0413 \text{ mCPU}.$$

Since the average number of idle consumers is equal to **5.0**, the estimated idle CPU usage is:

$$5.0 \times 0.0413 \text{ mCPU} = 0.2065 \text{ mCPU}.$$

Expressed in CPU cores, this corresponds to:

$$0.2065 \text{ mCPU} = 0.0002065 \text{ cores}.$$

The corresponding cumulative idle CPU time over the measured workload duration is:

$$0.0002065 \text{ cores} \times 50 \text{ s} = 0.010325 \text{ core} \cdot \text{s}.$$

Although this value is extremely small in absolute terms, it should not be interpreted as evidence that overprovisioning is cost-free. An idle consumer consumes very little CPU while simply maintaining its runtime state and the connection with the RabbitMQ broker. However, the main cost of this deployment does not necessarily arise from CPU consumption itself, but rather from keeping additional pods, processes, and RabbitMQ connections active despite not contributing to workload processing. Furthermore, if CPU requests are configured for the consumer pods, idle replicas also reserve schedulable CPU capacity, even though their actual CPU utilization remains negligible.

Consequently, the overall cost of overprovisioning depends not only on CPU consumption, but also on resource requests, memory occupancy, connection overhead, and the scheduling constraints imposed on the Kubernetes cluster.

19.9.7 Engineering Discussion

The static overprovisioned scenario represents the upper static baseline in terms of resource allocation. By deploying fifteen consumer replicas, the system provides a theoretical processing capacity of **3 messages/s**, exceeding the incoming workload of **2 messages/s**. As a result, the available capacity is sufficient to process messages as soon as they arrive, completely preventing backlog accumulation. From a performance perspective, this configuration achieves the best behaviour among the static deployments analysed in this chapter. The *Ready Queue Messages* remain equal to zero, the *Backlog Duration* is null, and the workload is completed in approximately **50 seconds**, corresponding almost entirely to the workload generation phase. Consequently, messages experience virtually no queueing delay. This level of performance, however, is achieved by allocating more consumer replicas than the workload actually requires. While the incoming traffic can be sustained by approximately ten busy consumers, all fifteen replicas remain active throughout the experiment. Consequently, about five consumers remain idle on average, corresponding to roughly one third of the allocated processing capacity. This scenario therefore highlights the opposite side of the trade-off observed in the static underprovisioned configuration. There, resource utilization was maximized at the expense of system performance; here, performance is maximized at the expense of resource allocation efficiency. Although the measured idle CPU usage is extremely small (approximately **0.2065 mCPU**), overprovisioning still incurs additional costs associated with maintaining unnecessary pods, RabbitMQ connections, memory occupancy, and schedulable cluster resources. From an engineering perspective, this deployment demonstrates that static provisioning can guarantee excellent performance only when the system is dimensioned for the worst-case workload, or even beyond it. While this approach is effective for constant and predictable traffic patterns, it becomes increasingly inefficient when the workload is variable or intermittent, as excess replicas remain allocated even when they are not required. These observations naturally motivate the adoption of event-driven autoscaling mechanisms such as KEDA. Rather than permanently allocating resources for peak demand,

KEDA dynamically adjusts the number of consumer replicas according to the actual queue state, aiming to preserve the performance benefits of the over-provisioned configuration while reducing the resource overhead associated with permanently idle consumers.

19.10 KEDA Autoscaling Configuration

Before analysing the two KEDA-based deployment strategies, it is useful to summarize the autoscaling configuration adopted throughout the experiments. In both scenarios, KEDA uses the *Prometheus scaler* to retrieve the number of RabbitMQ messages in the *ready* state from the Prometheus server. This metric represents the backlog currently waiting in the queue and is used as the external signal driving the scaling of the consumer Deployment. Most autoscaling parameters are identical in the two KEDA configurations. The polling interval is set to **5 seconds**, meaning that KEDA queries the Prometheus metric every five seconds. The maximum number of replicas is fixed at **20**, providing an upper bound for the autoscaling process. The Prometheus scaler is configured with a **threshold equal to 5**. This value represents the target used by the Horizontal Pod Autoscaler generated by KEDA to compute the desired number of consumer replicas. Since the monitored metric corresponds to the number of RabbitMQ messages in the *ready* state, the threshold directly affects how aggressively the system reacts to backlog accumulation. From a practical point of view, it can be interpreted as a virtual backlog target per consumer replica. For a given number of ready messages, a lower threshold leads to a higher desired replica count, making the autoscaler more aggressive. This can reduce queueing delay and accelerate backlog draining, but it may also allocate more consumer replicas than strictly necessary, increasing temporary resource usage and the number of idle consumers. Conversely, a higher threshold produces a lower desired replica count for the same backlog level. This can improve resource saving, but it may also slow down backlog absorption and increase the time messages spend waiting in the ready queue. Therefore, the threshold introduces an important trade-off between responsiveness and resource efficiency. If the threshold is too low, the system may overreact to short backlog spikes and create excessive replicas. If the threshold is too high, the system may underreact and keep too few consumers active, causing the queue to grow for a longer time. In these experiments, the value **5** was selected as

a balanced configuration for the considered workload and was intentionally kept unchanged in both KEDA scenarios. This ensures that the comparison between the scale-to-zero and warm-start configurations depends only on the different value of `minReplicaCount`, rather than on a different autoscaling sensitivity. The scaler also defines an **activationThreshold equal to 0**. This parameter mainly affects the activation behaviour when the Deployment has been scaled down to zero. With this configuration, the trigger becomes active as soon as the number of ready messages exceeds zero. Consequently, whenever at least one message appears in the queue, KEDA immediately detects the workload and the autoscaling process can restart. This parameter is particularly relevant for the configuration with `minReplicaCount = 0`, where the consumer Deployment can be completely scaled down after the queue has been emptied. The **cooldown period** is configured to **120 seconds**. This parameter determines how long KEDA waits before scaling the Deployment from one replica to zero after the trigger becomes inactive. Therefore, it only affects the final scale-to-zero transition of the first KEDA scenario. In the warm-start configuration with `minReplicaCount = 2`, the Deployment cannot be reduced below two replicas, and the cooldown period does not lead to a final transition to zero. The `ScaledObject` also specifies the behaviour of the Horizontal Pod Autoscaler generated by KEDA. The **scale-up stabilization window** is set to **0 seconds**, allowing the deployment to react immediately whenever additional replicas are required. Conversely, the **scale-down stabilization window** is configured to **60 seconds**, delaying scale-in decisions in order to smooth temporary metric fluctuations and avoid unnecessary replica oscillations. This behaviour is useful in bursty workloads, since it prevents the system from immediately removing replicas as soon as the queue becomes empty. If new traffic arrives shortly after the previous workload has been processed, some replicas may still be active and available to serve the new messages without requiring a complete scale-up cycle. The two KEDA configurations differ only in the value of `minReplicaCount`. In the first scenario, `minReplicaCount` is set to **0**, enabling the Deployment to scale down completely when no messages remain in the queue. This configuration provides maximum idle-time resource saving, but introduces a cold-start effect when new workload arrives. In the second scenario, `minReplicaCount` is set to **2**, ensuring that two consumer replicas always remain active. This second configuration provides a *warm start*, since consumer pods are already running and connected to RabbitMQ when new messages arrive, although complete scale-to-zero is no longer

possible. For the KEDA experiments, the Grafana observation window was intentionally extended beyond the workload processing phase. This makes it possible to observe not only the queue filling and draining phases, but also the autoscaling behaviour after the workload has been completed. In particular, the extended observation window allows the effects of the HPA scale-down stabilization window and, in the scale-to-zero configuration, the KEDA cooldown period to be analysed. Nevertheless, the average metrics reported in the summary bar gauges, namely *Avg Active Consumers*, *Avg Busy Consumers*, and *Avg Idle Consumers*, are still computed exclusively over the workload processing window. This ensures a fair comparison with the static scenarios, since these metrics are not influenced by post-processing autoscaling mechanisms such as stabilization windows or cooldown periods. The autoscaling tail observed after the queue becomes empty is therefore analysed separately through the time-series panels and the HPA timeline.

Parameter	**Value**	**Role**
Scaler	Prometheus	Retrieves RabbitMQ ready messages from the Prometheus server.
Scaling metric	Ready messages	Number of RabbitMQ messages waiting in the queue and not yet delivered to consumers.
Polling interval	5 s	Frequency at which KEDA queries the Prometheus metric.
Threshold	5 ready messages	Target value used by the HPA to compute the desired number of consumer replicas.
Activation threshold	0 ready messages	Activates the scaler as soon as the ready-message metric becomes greater than zero.
Maximum replicas	20	Upper bound for the consumer Deployment.
Cooldown period	120 s	Delay before scaling from one replica to zero after the trigger becomes inactive.
Scale-up stabilization window	0 s	Allows immediate scale-up reactions.
Scale-down stabilization window	60 s	Delays and smooths scale-in decisions.
Scenario 1: `minReplicaCount`	0	Enables scale-to-zero.
Scenario 2: `minReplicaCount`	2	Keeps two warm consumer replicas permanently active.

Table 19.2: Main KEDA autoscaling parameters adopted in the experimental evaluation.

19.11 KEDA Scale-to-Zero Scenario

19.11.1 Scenario Description

The first KEDA-based scenario enables the *scale-to-zero* feature by setting `minReplicaCount = 0` in the corresponding `ScaledObject`. In this configuration, the consumer Deployment can be completely scaled down when no workload is present in the RabbitMQ queue. As soon as new messages arrive, KEDA reactivates the Deployment, while the Horizontal Pod Autoscaler (HPA) generated by KEDA progressively increases the number of consumer replicas according to the external metric retrieved from Prometheus. As in the previous scenarios, the workload consists of **100 messages** generated at a constant rate of **2 messages/s**. Since each consumer requires **5 seconds** to process a single message, the processing capacity of one consumer is:

$$1 \text{ consumer} = \frac{1 \text{ message}}{5 \text{ s}} = 0.2 \text{ messages/s}.$$

Consequently, the theoretical number of busy consumers required to sustain the incoming workload without accumulating backlog is:

$$2 \text{ messages/s} \times 5 \text{ s/message} = 10 \text{ busy consumers}.$$

Unlike the static overprovisioned scenario, however, the system does not start with consumer replicas already running. Since `minReplicaCount = 0`, the Deployment initially contains no active consumer pods. Therefore, when the workload starts, the first messages inevitably accumulate in the queue before KEDA detects the active trigger and begins scaling the Deployment. As a result, an initial backlog is expected during the first phase of the experiment. The scaling signal used by KEDA is the number of RabbitMQ messages in the *ready* state, retrieved from Prometheus through the Prometheus scaler. As described in the previous section, the scaler threshold is configured to **5 ready messages**, while the activation threshold is set to **0**. Consequently, the trigger becomes active as soon as at least one message enters the ready state, after which the HPA computes the desired number of consumer replicas according to the configured target value.

19.11.2 HPA Timeline and Autoscaling Behaviour

Besides the Grafana dashboard, the autoscaling behaviour was also monitored directly from Kubernetes by observing the HPA generated by KEDA through

the `kubectl get hpa -w` command. While the Grafana dashboard illustrates the effects of autoscaling on the RabbitMQ queue and on the consumer Deployment, the HPA output provides the corresponding Kubernetes control-plane view, showing the replica decisions applied throughout the experiment. The HPA evolution observed during the `minReplicaCount = 0` scenario is reported below:

TARGETS	MINPODS	MAXPODS	REPLICAS
<unknown>/5 (avg)	1	20	0
<unknown>/5 (avg)	1	20	0
39/5 (avg)	1	20	1
12400m/5 (avg)	1	20	5
7300m/5 (avg)	1	20	10
2534m/5 (avg)	1	20	15
0/5 (avg)	1	20	15
0/5 (avg)	1	20	15
0/5 (avg)	1	20	8
0/5 (avg)	1	20	1
<unknown>/5 (avg)	1	20	0

Listing 19.8: HPA evolution observed during the KEDA scale-to-zero scenario.

The output illustrates the complete autoscaling lifecycle of the consumer Deployment, from zero replicas to the maximum number of replicas reached during backlog recovery, and finally back to zero after the workload has been completely processed. The observed replica sequence is:

$$0 \to 1 \to 5 \to 10 \to 15 \to 8 \to 1 \to 0.$$

At the beginning of the experiment, the Deployment is scaled down to zero replicas. Consequently, the HPA reports the external metric as `<unknown>`, since no consumer pod is running and the target Deployment is inactive. This behaviour is expected in a KEDA scale-to-zero configuration. Although the HPA object already exists, it cannot directly manage a Deployment with zero replicas. Therefore, the transition from zero to one replica is entirely performed by KEDA when the external metric, namely the number of RabbitMQ ready messages, becomes active. Once the first consumer replica has been created, the HPA takes control of the Deployment and manages scaling between one replica and the configured maximum number of replicas. During

the scale-out phase, the number of replicas increases from 1 to 5, then to 10, and finally to 15. This behaviour is fully consistent with the backlog growth observed in the queue-state plots. As the number of ready messages increases, the external metric rises accordingly, leading the HPA to progressively allocate additional consumer replicas. The observed scale-up sequence is also coherent with the default HPA scaling policies. The transition from 1 to 5 replicas is allowed by the default policy that permits adding up to four pods during a single scaling period. The subsequent increase from 5 to 10 replicas follows the default 100% scale-up policy, while the final transition to 15 replicas reflects the large backlog accumulated during the initial scale-from-zero phase. Since the scale-up stabilization window is configured to **0 seconds**, the Deployment is allowed to react immediately to the increasing external metric. After the queue has been completely drained, the external metric drops to zero. Nevertheless, the number of replicas does not decrease immediately. As shown by the HPA output, the Deployment remains at 15 replicas for a short period even after the metric reaches `0/5`. This behaviour is consistent with the configured **60-second scale-down stabilization window**, which prevents aggressive downscaling and reduces the likelihood of replica oscillations caused by temporary workload fluctuations. The subsequent scale-in process proceeds through intermediate replica counts, reducing the Deployment from 15 to 8 replicas and then from 8 to 1 replica. The same behaviour can also be observed qualitatively in the Consumer State panel, where the number of active consumers decreases progressively after the workload has been processed. Finally, the transition from one replica back to zero is performed directly by KEDA rather than by the HPA, since the HPA only manages Deployments with at least one active replica. This final transition occurs after the configured **120-second cooldown period**. Overall, the observed timeline clearly confirms the separation of responsibilities between KEDA and the HPA. KEDA manages the activation from zero replicas and the final transition back to zero, whereas the HPA controls the scaling process between one replica and the configured maximum number of replicas. This also explains the autoscaling tail visible in the extended Grafana observation window after the queue has become empty. As discussed in the methodology section, the average consumer metrics reported in the summary dashboard are computed exclusively over the workload processing window and are therefore unaffected by the HPA stabilization window and the KEDA cooldown period.

19.11.3 Summary Metrics

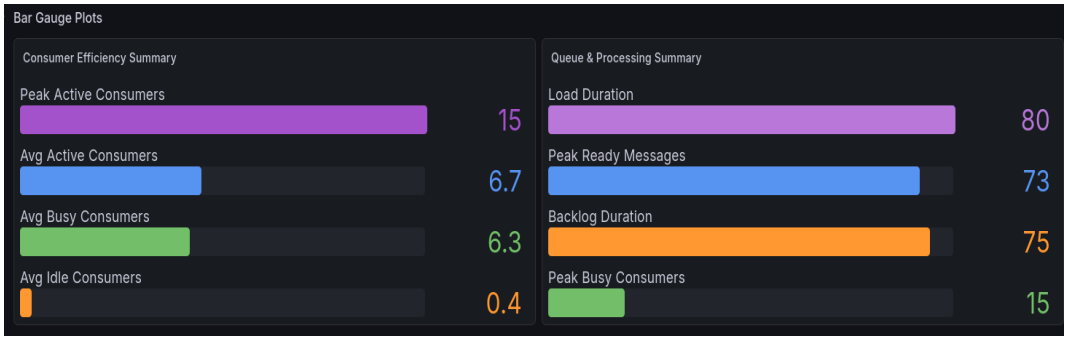


Figure 19.13: Summary metrics for the KEDA scale-to-zero scenario. The bar gauge panels report the main consumer efficiency and queue processing metrics for the KEDA configuration with `minReplicaCount = 0`. During the experiment, the Deployment scales from zero to fifteen consumer replicas, reaches an average of approximately 6.7 active consumers during the workload processing window, and accumulates a peak backlog of 73 ready messages before the queue is completely drained.

Figure 19.13 summarizes the most relevant consumer efficiency and queue processing metrics for the KEDA scale-to-zero scenario. The *Peak Active Consumers* metric reaches **15**, indicating that the consumer Deployment scales up to fifteen replicas during the experiment. This value is fully consistent with the HPA timeline discussed in the previous section, where the maximum replica count also reaches fifteen during the scale-out phase. Consequently, the maximum number of active consumers reported by Grafana matches the highest replica count managed by the HPA. This peak exceeds the theoretical steady-state requirement of ten busy consumers. The reason is that the autoscaler is not only reacting to the incoming traffic rate, but also to the backlog accumulated during the initial scale-from-zero phase. Once a significant number of messages has accumulated in the *ready* state, additional consumer replicas are temporarily required to recover the backlog and reduce the overall processing delay. The *Avg Active Consumers* metric is equal to **6.7**, while the *Avg Busy Consumers* metric is equal to **6.3**. As described in the methodology section, these averages are computed exclusively over the workload processing window. The small difference between the two values indicates that most

active consumer replicas effectively contribute to message processing during the experiment. The contrast between the peak and average number of active consumers also highlights the transient nature of the autoscaling process. The Deployment reaches fifteen replicas only during the most demanding phase of backlog recovery, whereas the average number of active consumers remains considerably lower throughout the workload processing window. This confirms that the highest replica count is required only temporarily and does not characterize the overall execution. The *Avg Idle Consumers* metric is equal to only **0.4**. This represents a substantial improvement over the static overprovisioned scenario, where the average number of idle consumers was **5.0**. From the perspective of resource allocation, the KEDA scale-to-zero configuration therefore achieves a much more efficient utilization of consumer replicas during workload execution. The queue-related metrics also reveal the cost associated with starting from zero replicas. The *Peak Ready Messages* metric reaches **73**, showing that a significant backlog accumulates before enough consumers become available to process the incoming workload. The *Backlog Duration* is approximately **75 seconds**, whereas the *Load Duration* is approximately **80 seconds**. As observed in the previous scenarios, the difference between these two values corresponds to the final processing tail, during which the last messages have already left the *ready* state but are still being processed in the *unacknowledged* state. Finally, the *Peak Busy Consumers* metric also reaches **15**. This indicates that, at the peak of the scale-out phase, all active consumer replicas are effectively processing messages. Consequently, the additional replicas created during autoscaling are fully exploited to accelerate backlog recovery, rather than remaining idle during the workload processing window.

19.11.4 Message Rate Analysis

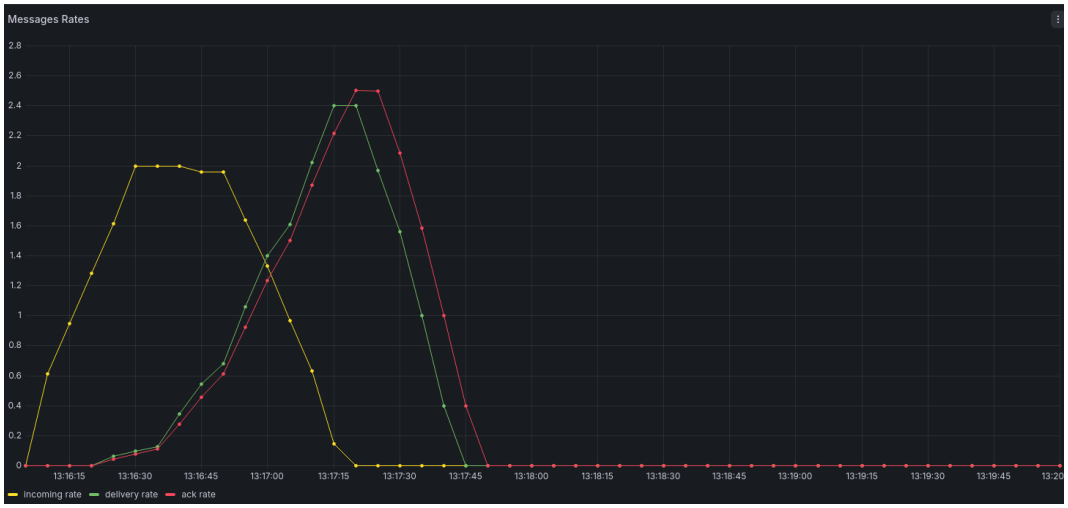


Figure 19.14: Message rates in the KEDA scale-to-zero scenario. The incoming rate reaches approximately 2 messages per second during the workload generation phase. The delivery and acknowledgement rates increase after the autoscaler creates consumer replicas, eventually exceeding the incoming rate while the accumulated backlog is being drained.

Figure 19.14 illustrates the evolution of the message rates throughout the experiment and highlights the dynamic behaviour introduced by event-driven autoscaling. During the initial phase, the *Incoming Rate* rapidly reaches the configured value of approximately **2 messages/s**. Since the Deployment initially contains no active consumer replicas, the consumer side is unable to process incoming messages immediately. Consequently, both the *Delivery Rate* and the *Ack Rate* remain close to zero during the first seconds of the experiment, allowing messages to accumulate in the RabbitMQ queue. As soon as KEDA activates the Deployment and the HPA progressively increases the number of consumer replicas, the *Delivery Rate* starts increasing as well. This indicates that RabbitMQ begins assigning the accumulated messages to the newly available consumers. The *Ack Rate* follows the same trend with a slight delay, reflecting the approximately five seconds required by each consumer to process a message before sending the corresponding acknowledgement. Once a sufficient number of consumer replicas has been deployed, both the delivery and

acknowledgement rates temporarily exceed the incoming workload rate, reaching approximately **2.4–2.5 messages/s** in the observed experiment. This behaviour is expected during the backlog recovery phase. While the producer continues generating messages at a constant rate of **2 messages/s**, the autoscaled consumer group is able to process messages at a higher rate, gradually reducing the backlog accumulated during the initial scale-from-zero phase. After the producer completes the workload generation, the *Incoming Rate* drops to zero. Nevertheless, the delivery and acknowledgement rates remain active for a short period while the remaining queued messages are delivered, processed, and acknowledged by the consumer replicas. Once the backlog has been completely drained and no messages remain in the system, all three rates converge to zero.

19.11.5 Queue State and Backlog Analysis



Figure 19.15: Queue state and backlog in the KEDA scale-to-zero scenario. The *Ready Queue Messages* increase during the initial phase because the Deployment starts with zero consumer replicas. After KEDA progressively scales out the consumer Deployment, the accumulated backlog is rapidly drained. The peak number of ready messages reaches approximately 73, while the maximum number of unacknowledged messages reaches 15.

Figure 19.15 illustrates the evolution of the queue throughout the experiment and clearly highlights the cold-start effect introduced by configuring minReplicaCount = 0. At the beginning of the workload, no consumer replicas are active. Consequently, incoming messages initially remain in the *ready* state, causing the ready queue to grow rapidly. This initial backlog accumulation is the direct consequence of starting from zero replicas and waiting for KEDA to activate the Deployment. This behaviour is fully consistent with the HPA evolution discussed in the previous section. The observed scale-out sequence,

$$0 \rightarrow 1 \rightarrow 5 \rightarrow 10 \rightarrow 15,$$

explains the change in the slope of the *Ready Queue Messages* curve. During the first phase of the experiment, only a limited number of consumer replicas is available, allowing the backlog to continue growing. As additional replicas are progressively created, the available processing capacity increases and the backlog begins to drain at an increasingly faster rate. The *Peak Ready Messages* metric reaches approximately **73**. This value is lower than the **80 ready messages** observed in the static underprovisioned scenario because, in the KEDA configuration, the Deployment starts scaling out while the producer is still generating traffic. Consequently, although a significant backlog still accumulates, the autoscaler begins reacting before the workload generation phase has completed. The relatively high peak backlog is therefore not caused by insufficient processing capacity, but rather by the latency introduced by the initial scale-from-zero activation and the subsequent scale-up process. At the same time, the measured backlog remains considerably higher than in the static overprovisioned scenario, where the number of ready messages remained essentially equal to zero throughout the experiment. This comparison clearly illustrates the fundamental trade-off of the scale-to-zero approach: resource allocation is minimized during idle periods, but this comes at the cost of an initial cold-start delay whenever a new workload arrives. As the autoscaler progressively increases the number of consumer replicas, the *Unacked Queue Messages* curve also rises. This indicates that an increasing number of messages is being delivered to consumers and actively processed. The peak value reaches approximately **15**, matching the maximum number of busy consumers observed during the experiment. In other words, at the peak of the scale-out phase, all consumer replicas created by the autoscaler are effectively engaged in processing the accumulated workload. After the scale-out phase, the ready queue decreases rapidly. At this point, the available processing capacity ex-

ceeds the incoming message rate, allowing the system not only to sustain the incoming traffic but also to progressively recover the backlog accumulated during the initial phase. Once the ready queue becomes empty, some messages still remain in the *unacknowledged* state for a short period while their processing is completed. This final processing tail explains why the measured *Backlog Duration* is slightly shorter than the overall *Load Duration*. Overall, the observed queue behaviour closely follows the expected dynamics of a KEDA scale-to-zero deployment. The backlog initially grows because the system starts with no active consumers, and is subsequently drained rapidly once the Deployment scales out. This panel therefore provides a clear visualization of both the main strength and the principal limitation of this approach: excellent resource efficiency during idle periods, balanced against the unavoidable cold-start penalty experienced when new traffic reaches the system.

19.11.6 Consumer State Analysis



Figure 19.16: Consumer state in the KEDA scale-to-zero scenario. The consumer Deployment starts from zero replicas and progressively scales out as backlog accumulates. The number of active consumers reaches fifteen before the accumulated workload is drained. After the workload has been processed, consumer replicas remain active for a short period due to the HPA stabilization window and the KEDA cooldown period before the Deployment eventually scales back to zero.

Figure 19.16 illustrates the complete lifecycle of the consumer Deployment from the application perspective, highlighting how the number of active, busy, and idle consumers evolves throughout the autoscaling process. At the beginning of the experiment, both the *Active Consumers* and *Busy Consumers* metrics are equal to zero because the Deployment has been completely scaled down. This corresponds to the idle state of the system, where no consumer pod is running and, consequently, no resources are consumed by the consumer application. When the first messages begin accumulating in the RabbitMQ queue, KEDA detects the external metric retrieved from Prometheus and activates the target Deployment. This corresponds to the first transition observed in the HPA timeline, where the Deployment moves from zero to one active replica. From that point onward, the HPA takes control of the scaling process and progressively adjusts the number of consumer replicas according to the value of the external metric. The *Active Consumers* metric progressively increases until reaching a peak value of **15**. This result is fully consistent with the HPA evolution discussed previously, where the Deployment also reaches fifteen replicas during the backlog recovery phase. Throughout the scale-out process, the *Busy Consumers* curve closely follows the number of active replicas. This indicates that the additional replicas created by the autoscaler are effectively used to process the accumulated workload rather than remaining idle. Consequently, the scale-out operation successfully increases the processing capacity required to drain the backlog. Once the queue has been completely drained, the number of busy consumers rapidly falls to zero. At this point, the workload has already been processed, yet the number of active consumers does not immediately decrease. As previously discussed, this behaviour is expected and results from the combined action of the HPA scale-down stabilization window and the KEDA cooldown period. The HPA first performs a gradual scale-in through intermediate replica counts, whereas the final transition from one replica back to zero is carried out directly by KEDA. During this post-processing interval, the *Idle Consumers* metric temporarily increases because consumer replicas remain active despite having no messages left to process. This behaviour is an intentional consequence of the stabilization and cooldown mechanisms, whose purpose is to prevent unnecessary replica oscillations if new workload arrives shortly after the queue has been emptied. As discussed in the methodology section, this post-processing interval is intentionally excluded from the computation of the average consumer metrics reported in the summary dashboard. The *Avg Active Consumers*, *Avg Busy Consumers*,

and *Avg Idle Consumers* metrics are computed exclusively over the workload processing window, namely while messages are still present either in the *ready* or in the *unacknowledged* state. This ensures a fair comparison with the static scenarios by evaluating consumer utilization only during the effective workload execution. Overall, the Consumer State panel and the HPA timeline provide two complementary views of the same autoscaling process. Grafana illustrates the effects of autoscaling on the RabbitMQ consumers, whereas the HPA output exposes the corresponding scaling decisions taken by Kubernetes. The strong agreement between these two perspectives confirms that the observed consumer behaviour is a direct consequence of the autoscaling policy defined in the `ScaledObject`.

19.11.7 Idle CPU Usage Estimation

Using the reference idle CPU usage estimated previously, the CPU consumption associated with idle consumers during the workload can be computed as:

$$\text{Estimated idle CPU usage} = \text{Avg Idle Consumers} \times 0.0413 \text{ mCPU}.$$

Since the average number of idle consumers during the workload processing window is equal to **0.4**, the estimated idle CPU usage is:

$$0.4 \times 0.0413 \text{ mCPU} = 0.01652 \text{ mCPU}.$$

Expressed in CPU cores, this corresponds to:

$$0.01652 \text{ mCPU} = 0.00001652 \text{ cores}.$$

The corresponding cumulative idle CPU time over the measured workload duration is:

$$0.00001652 \text{ cores} \times 80 \text{ s} = 0.0013216 \text{ core} \cdot \text{s}.$$

This value is extremely small and considerably lower than that measured in the static overprovisioned scenario. It confirms that, during the workload processing window, the KEDA scale-to-zero configuration keeps the number of unused consumer replicas to a minimum. Nevertheless, after the workload has been completed, the autoscaling tail introduces a temporary period during which consumer replicas remain active even though no messages are left to process. As discussed previously, this behaviour results from the HPA scale-down stabilization window and the KEDA cooldown period. Although this

interval is intentionally excluded from the computation of the average workload metrics, it remains an important aspect of the overall autoscaling lifecycle. From an engineering perspective, this behaviour represents a deliberate trade-off: stabilization and cooldown improve scaling stability while delaying the release of resources after the workload has finished.

19.11.8 Engineering Discussion

Among the evaluated deployment strategies, the KEDA scale-to-zero configuration provides the highest level of resource efficiency during inactive periods. When the RabbitMQ queue is empty, the consumer Deployment can be completely scaled down, eliminating idle consumer replicas and avoiding unnecessary resource allocation. This makes the configuration particularly suitable for intermittent or bursty workloads, where permanently running consumers would remain unused for long periods. During workload execution, the average number of active consumers is only **6.7**, substantially lower than the fifteen replicas maintained in the static overprovisioned configuration. At the same time, the average number of busy consumers reaches **6.3**, while the average number of idle consumers is only **0.4**. These results indicate that most of the allocated consumer replicas actively contribute to message processing, leading to a significantly more efficient utilization of resources. This behaviour is also consistent with the HPA timeline discussed previously. Although the Deployment temporarily reaches a peak of fifteen replicas during backlog recovery, this maximum value is maintained only for a limited portion of the experiment. The average number of active consumers remains considerably lower throughout the workload processing window, confirming that the additional replicas are allocated only when required to recover the accumulated backlog. The price paid for this resource efficiency is the cold-start effect. Since the Deployment initially contains no active consumer replicas, messages begin accumulating before sufficient consumers have been created, scheduled, and connected to RabbitMQ. Consequently, the experiment exhibits a peak backlog of approximately **73 ready messages** and a backlog duration of about **75 seconds**. Compared with the static overprovisioned scenario, the scale-to-zero configuration therefore sacrifices immediate responsiveness in exchange for eliminating idle replicas during inactive periods. The maximum number of active consumers reaches **15**, exceeding the theoretical steady-state requirement of ten busy consumers. This behaviour is expected because the autoscaler

reacts not only to the current incoming workload, but also to the backlog accumulated during the initial scale-from-zero phase. Once a significant queue has formed, additional consumer replicas are temporarily required to recover the accumulated delay. In other words, the autoscaler dynamically increases the processing capacity not only to sustain the incoming traffic, but also to eliminate the backlog generated during the startup phase. The consistency between the Grafana dashboard and the HPA timeline further validates the observed behaviour. Grafana illustrates the effects of autoscaling on the RabbitMQ queue and on the consumer Deployment, whereas the HPA output reports the corresponding scaling decisions taken by Kubernetes. Together, these complementary views clearly show the complete autoscaling lifecycle: KEDA activates the Deployment from zero replicas, the HPA manages scale-out according to the Prometheus external metric, the HPA subsequently performs scale-in after the stabilization window, and KEDA finally returns the Deployment to zero once the cooldown period has expired. Overall, this scenario clearly illustrates both the main strength and the principal limitation of event-driven scale-to-zero. On one hand, it almost eliminates idle resource allocation during inactive periods while maintaining a high utilization of the deployed consumer replicas during workload execution. On the other hand, the unavoidable cold-start delay introduces an initial backlog whenever new traffic arrives. Consequently, this configuration is particularly suitable for workloads where resource efficiency and zero-idle operation are prioritized over immediate response time, or where a limited backlog buildup at startup is considered acceptable.

19.12 KEDA Warm-Start Scenario

The fourth scenario evaluates the behaviour of KEDA when autoscaling is enabled while preventing the consumer Deployment from scaling down to zero replicas. In this configuration, the `ScaledObject` is configured with `minReplicaCount = 2`, meaning that two consumer replicas are always kept active, even when no messages are present in the RabbitMQ queue. This configuration represents a *warm-start* autoscaling strategy. Unlike the previous scale-to-zero scenario, the consumer Deployment does not need to be activated from zero when new workload arrives. Since two consumer replicas are already running and connected to RabbitMQ, the first messages can be delivered immediately. The objective is to evaluate whether maintaining a small baseline

processing capacity can reduce the cold-start effect and limit the initial backlog while still allowing KEDA and the HPA to dynamically increase the number of consumers when the workload exceeds the available processing capacity. The workload configuration is identical to that adopted in the previous scenarios. The producer generates 100 messages at a constant rate of 2 messages per second, while each consumer requires 5 seconds to process a single message. Consequently, the processing capacity of one consumer is:

$$1 \text{ consumer} = \frac{1 \text{ message}}{5 \text{ s}} = 0.2 \text{ messages/s}.$$

With two consumer replicas initially active, the available processing capacity becomes:

$$2 \text{ consumers} \times 0.2 \text{ messages/s} = 0.4 \text{ messages/s}.$$

Although this value remains lower than the incoming traffic rate of 2 messages per second, the system no longer starts from zero processing capacity. The two baseline consumers immediately begin processing incoming messages while KEDA monitors the increasing number of ready messages and the HPA progressively scales the Deployment out. The theoretical number of busy consumers required to sustain the incoming workload without accumulating backlog therefore remains:

$$2 \text{ messages/s} \times 5 \text{ s/message} = 10 \text{ busy consumers}.$$

Accordingly, the autoscaler is expected to increase the number of replicas toward approximately ten consumers, or slightly beyond this value if an initial backlog has already accumulated and must be recovered.

19.12.1 HPA Timeline and Autoscaling Behaviour

The evolution of the Horizontal Pod Autoscaler was monitored throughout the experiment using the `kubectl get hpa -w` command. The observed HPA output is reported below.

TARGETS	MINPODS	MAXPODS	REPLICAS
<unknown>/5 (avg)	2	20	0
4/5 (avg)	2	20	2
16/5 (avg)	2	20	2
8500m/5 (avg)	2	20	6

```
5091m/5 (avg)       2       20      11
2091m/5 (avg)       2       20      11
0/5 (avg)           2       20      11
0/5 (avg)           2       20      11
0/5 (avg)           2       20      5
0/5 (avg)           2       20      2
```

Listing 19.9: HPA evolution observed during the KEDA warm-start scenario with `minReplicaCount = 2`.

The HPA output illustrates the complete evolution of the consumer Deployment during the warm-start experiment. The Deployment begins with two active replicas, scales out to a maximum of eleven replicas while the backlog increases, and finally scales back down to the configured minimum once the workload has been processed. The HPA confirms the expected warm-start behaviour. Since the `ScaledObject` specifies `minReplicaCount = 2`, the generated HPA reports `MINPODS = 2` and `MAXPODS = 20`. Although the external metric is initially reported as `<unknown>`, the Deployment is immediately brought to its minimum size of two replicas. The resulting replica evolution can be summarized as:

$$0 \rightarrow 2 \rightarrow 6 \rightarrow 11 \rightarrow 5 \rightarrow 2.$$

The transition to two replicas represents the warm-start baseline. Unlike the previous scale-to-zero scenario, KEDA does not need to activate the Deployment from zero because the minimum number of consumer replicas is already maintained. During the scale-out phase, the number of replicas initially remains equal to two while the external metric increases from `4/5` to `16/5`. As the number of ready messages grows further, the HPA increases the Deployment first to six replicas and then to eleven. This behaviour is fully consistent with the default HPA scale-up policies. Starting from two replicas, the transition from two to six is permitted by the policy allowing an increase of up to four pods within a single policy period. The subsequent transition from six to eleven replicas is likewise coherent with the HPA scaling algorithm and with the external metric retrieved through the Prometheus scaler. The maximum replica count of eleven also matches the peak number of active consumers observed in the Grafana *Consumer State* panel, confirming the consistency between the Kubernetes control-plane view and the RabbitMQ monitoring metrics. After the queue has been completely drained, the external metric drops

to zero. Nevertheless, the Deployment does not immediately scale down. As shown in the HPA output, the number of replicas remains equal to eleven for a short interval before being progressively reduced to five and finally back to two replicas. This behaviour is consistent with the configured HPA scale-down stabilization window of 60 seconds, which delays aggressive scale-in operations and reduces the risk of oscillations. Unlike the scale-to-zero configuration analysed previously, the Deployment never returns to zero replicas. This behaviour is expected because the minimum replica count is fixed at two. Consequently, the KEDA cooldown period does not trigger a final scale-to-zero transition, since KEDA is not allowed to deactivate the Deployment completely. The HPA therefore manages scaling between the configured lower bound of two replicas and the number of replicas required by the external metric, while the two baseline consumers remain permanently active.

19.12.2 Summary Metrics

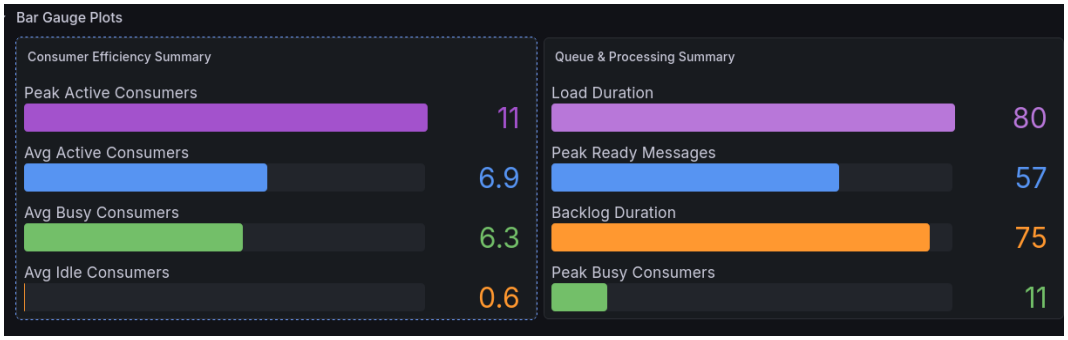


Figure 19.17: Summary metrics for the KEDA warm-start scenario. The bar gauge panels summarize the behaviour of the KEDA configuration with `minReplicaCount = 2`. The Deployment reaches a peak of eleven active consumers, with an average of approximately 6.9 active consumers and 6.3 busy consumers during the workload processing window. The peak backlog is reduced to about 57 ready messages compared with the scale-to-zero scenario, while the workload is completed in approximately 80 seconds.

Figure 19.17 summarizes the main consumer efficiency and queue processing metrics for the KEDA warm-start configuration. The reported values show

that maintaining two consumer replicas permanently active significantly reduces the initial backlog compared with the scale-to-zero scenario, while still requiring substantially fewer consumer replicas on average than the static overprovisioned deployment. The *Peak Active Consumers* metric reaches **11**, matching the maximum number of replicas reported by the HPA timeline discussed in the previous section. This agreement confirms the consistency between the consumer metrics collected through Grafana and the replica decisions taken by Kubernetes during the autoscaling process. The *Avg Active Consumers* metric is equal to **6.9**, considerably lower than the fifteen replicas maintained throughout the static overprovisioned scenario. At the same time, the *Avg Busy Consumers* metric reaches **6.3**, indicating that the vast majority of the allocated consumer replicas actively contribute to message processing during the workload execution. The *Avg Idle Consumers* metric is equal to only **0.6**. Although this value is slightly higher than the **0.4** observed in the scale-to-zero scenario, it remains dramatically lower than the average of **5.0** idle consumers measured in the static overprovisioned configuration. Consequently, the warm-start strategy introduces only a modest increase in idle capacity while providing a faster response when new workload arrives. From the queue perspective, the *Peak Ready Messages* metric reaches approximately **57**. This value is substantially lower than the peak backlog of approximately **73 ready messages** measured in the scale-to-zero scenario. The reduction confirms the benefit of maintaining two consumer replicas permanently active: incoming messages begin to be processed immediately, limiting the amount of backlog accumulated before the autoscaler has sufficient time to complete the scale-out process. The *Backlog Duration* is approximately **75 seconds**, whereas the *Load Duration* is approximately **80 seconds**. As observed in the previous scenarios, the difference between these two metrics corresponds to the final processing tail, during which no messages remain in the *ready* state while some messages are still being processed in the *unacknowledged* state. Finally, the *Peak Busy Consumers* metric reaches **11**. This value is only slightly higher than the theoretical steady-state requirement of ten busy consumers. As discussed for the previous KEDA scenario, this behaviour is expected because the autoscaler must not only sustain the incoming workload, but also recover the backlog accumulated during the initial phase of the experiment.

19.12.3 Message Rate Analysis

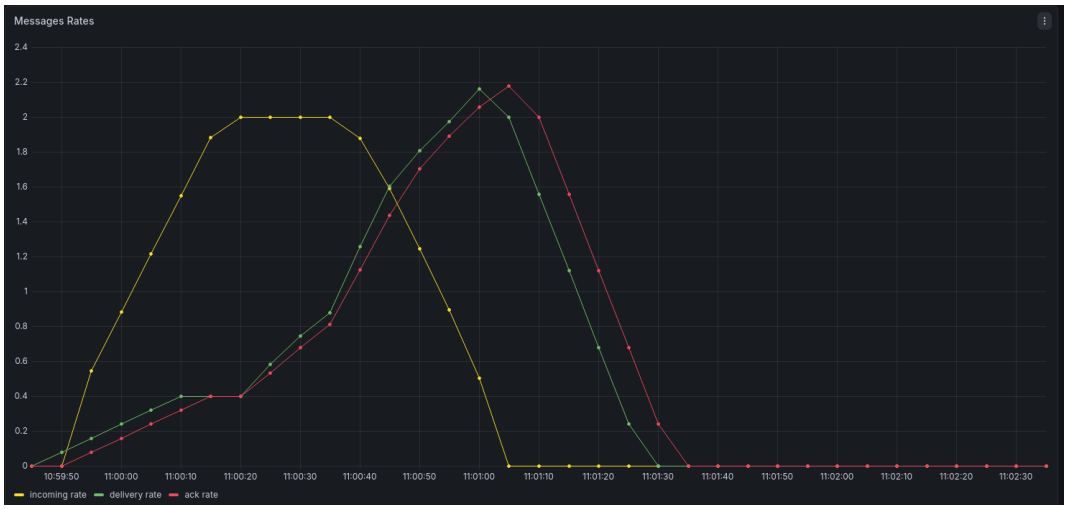


Figure 19.18: Message rates in the KEDA warm-start scenario. The incoming rate reaches the configured value of approximately 2 messages per second during the workload generation phase. The delivery and acknowledgement rates initially remain below the incoming rate because only two consumer replicas are active. As KEDA scales out the Deployment, both rates increase and eventually exceed the incoming rate, allowing the accumulated backlog to be recovered.

Figure 19.18 illustrates the evolution of the message rates during the warm-start autoscaling experiment. As in the previous scenarios, the *Incoming Rate* follows the configured workload pattern, reaching approximately **2 messages/s** throughout the traffic generation phase. At the beginning of the experiment, however, the *Delivery Rate* and the *Ack Rate* remain below the incoming rate because the Deployment starts with only two active consumer replicas. Their initial processing capacity is:

$$2 \text{ consumers} \times 0.2 \text{ messages/s} = 0.4 \text{ messages/s},$$

which is insufficient to sustain the incoming workload. Consequently, messages begin to accumulate in the RabbitMQ queue during the initial phase of the experiment. As the number of ready messages increases, KEDA detects the external metric exposed through Prometheus and the HPA progressively scales

the Deployment out. This behaviour is reflected in the graph, where both the delivery and acknowledgement rates increase after the initial phase and eventually exceed the incoming rate. This behaviour is expected because, once a sufficient number of consumer replicas has been deployed, the system must not only sustain the incoming workload but also recover the backlog accumulated during the initial scale-out phase. The *Ack Rate* remains slightly delayed with respect to the *Delivery Rate*, since each message is acknowledged only after the corresponding consumer has completed its 5-second processing time. This temporal offset is therefore an inherent consequence of the sequential processing model adopted by the consumers. Compared with the scale-to-zero configuration, the warm-start strategy noticeably reduces the initial delay because two consumer replicas are immediately available when the workload arrives. Nevertheless, the Deployment still requires some time to scale from its minimum size to the number of replicas required to eliminate the accumulated backlog. As a result, the initial backlog is reduced, but not completely eliminated.

19.12.4 Queue State and Backlog Analysis

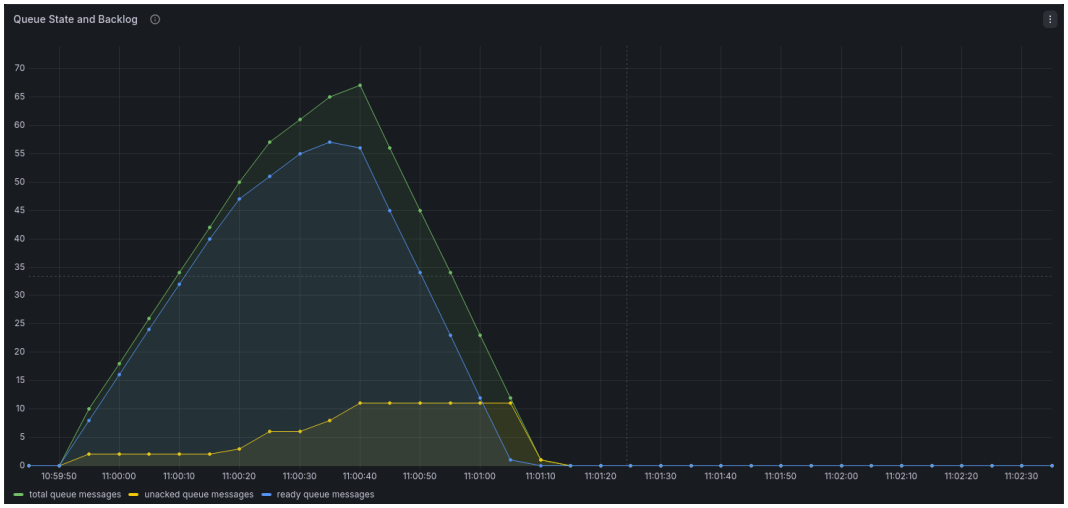


Figure 19.19: Queue state and backlog in the KEDA warm-start scenario. The *Ready Queue Messages* increase during the initial phase because the two baseline consumer replicas cannot sustain the incoming workload. As KEDA progressively scales out the Deployment, the backlog reaches a peak of approximately 57 ready messages before being rapidly drained by the additional consumer replicas.

Figure 19.19 illustrates the evolution of the RabbitMQ queue throughout the warm-start autoscaling experiment. The *Ready Queue Messages* initially increase because the Deployment starts with only two active consumer replicas. As discussed previously, their combined processing capacity is:

$$2 \text{ consumers} \times 0.2 \text{ messages/s} = 0.4 \text{ messages/s},$$

which is considerably lower than the producer rate of **2 messages/s**. Consequently, messages temporarily accumulate in the *ready* state while the autoscaler reacts to the increasing workload. Nevertheless, the peak backlog is significantly lower than that observed in the scale-to-zero scenario. In the warm-start configuration, the *Peak Ready Messages* metric reaches approximately **57**, whereas the scale-to-zero configuration produced a peak backlog of approximately **73 ready messages**. This improvement is a direct consequence

of maintaining two consumer replicas permanently active. Even before the autoscaler completes the scale-out process, these baseline consumers immediately begin processing incoming messages, partially compensating for the incoming traffic and limiting the backlog accumulation. The HPA timeline also explains the shape of the queue curves. During the initial phase, the Deployment remains at two replicas while the external metric progressively increases. As the number of ready messages exceeds the configured threshold, the HPA scales the Deployment first to six replicas and subsequently to eleven. This increase in processing capacity is reflected in the *Ready Queue Messages* curve: after an initial growth phase, the backlog reaches its maximum value and then progressively decreases as additional consumers become available. The *Total Queue Messages* curve remains above the *Ready Queue Messages* curve because it includes both ready and unacknowledged messages. At the same time, the *Unacked Queue Messages* metric reaches a peak value of approximately **11**, matching the maximum number of busy consumers observed during the experiment. This confirms that the additional replicas created during the scale-out phase are effectively used for message processing rather than remaining idle. The measured *Backlog Duration* is approximately **75 seconds**. Although this value is comparable to that observed in the scale-to-zero scenario, the peak backlog is noticeably lower. This result shows that the warm-start configuration does not completely eliminate backlog accumulation, but it significantly reduces its maximum intensity by providing an immediate baseline processing capacity before the autoscaler completes the scale-out process.

19.12.5 Consumer State Analysis

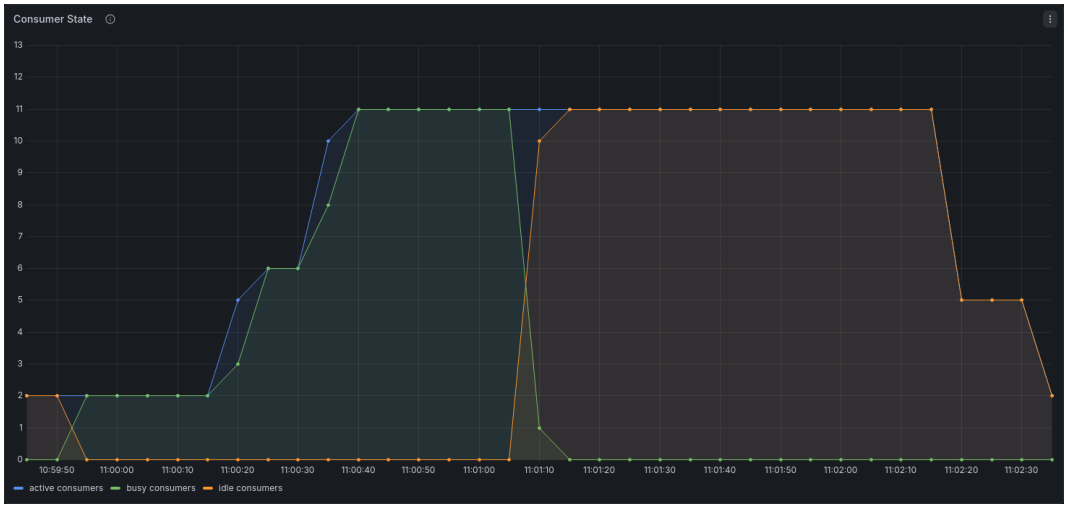


Figure 19.20: Consumer state in the KEDA warm-start scenario. The consumer Deployment starts with two active replicas, scales out to a peak of eleven consumers during backlog recovery, and then gradually scales back down to the minimum value of two replicas. Unlike the scale-to-zero configuration, the Deployment never returns to zero after the workload has been completed.

Figure 19.20 illustrates the evolution of the consumer Deployment throughout the warm-start autoscaling experiment. At the beginning of the experiment, the *Active Consumers* metric is already equal to **2**. These two replicas correspond to the configured `minReplicaCount` and provide an initial processing capacity immediately available when the first messages arrive. Unlike the scale-to-zero configuration, the system therefore does not require an initial activation phase before processing can begin. As the backlog increases, the number of active consumers progressively grows. The evolution observed in Grafana is fully consistent with the HPA timeline discussed previously, where the Deployment scales from two to six replicas and finally reaches a peak of eleven. During this phase, the *Busy Consumers* metric closely follows the *Active Consumers* curve, indicating that the additional replicas created by the autoscaler are effectively utilized for message processing rather than remaining idle. Once the queue has been completely drained, the number of busy consumers rapidly falls to zero. However, the number of active consumers

remains equal to eleven for a short interval before gradually decreasing. As in the previous KEDA scenario, this behaviour is expected and results from the configured HPA scale-down stabilization window, which intentionally delays replica reductions after the external metric reaches zero. The subsequent scale-in phase follows the sequence observed in the HPA timeline, reducing the Deployment first to five replicas and eventually back to the configured minimum of two. The final number of active consumers therefore remains equal to **2**, rather than returning to zero. This represents the fundamental difference with respect to the scale-to-zero configuration. By maintaining two consumer replicas permanently active, the warm-start strategy avoids the complete cold-start behaviour observed previously, while accepting a small amount of permanently allocated capacity during idle periods. As a consequence, the *Idle Consumers* metric increases after the workload has been completed, since the two baseline replicas remain active even though no messages are left to process. As discussed in the previous KEDA scenario, these post-processing idle intervals are intentionally excluded from the computation of the average consumer metrics reported in the summary dashboard. The *Avg Active Consumers*, *Avg Busy Consumers*, and *Avg Idle Consumers* metrics are evaluated exclusively over the workload processing window, ensuring a fair comparison with the static and scale-to-zero configurations.

19.12.6 Idle CPU Usage Estimation

Using the idle CPU reference value computed previously, the CPU usage associated with idle consumers during this scenario can be estimated as:

$$\text{Estimated idle CPU usage} = \text{Avg Idle Consumers} \times 0.0413 \text{ mCPU}.$$

Since the average number of idle consumers during the workload processing window is equal to **0.6**,

$$\text{Estimated idle CPU usage} = 0.6 \times 0.0413 \text{ mCPU} = 0.02478 \text{ mCPU}.$$

Expressed in CPU cores,

$$0.02478 \text{ mCPU} = 0.00002478 \text{ cores}.$$

The corresponding cumulative idle CPU time over the measured workload duration is:

$$\text{Estimated idle CPU-time} = 0.00002478 \text{ cores} \times 80 \text{ s} = 0.0019824 \text{ core} \cdot \text{s}.$$

This value is extremely small, confirming that the actual CPU consumed by idle consumers during the workload processing window is negligible. Nevertheless, unlike the scale-to-zero configuration, the warm-start strategy permanently maintains two consumer replicas even after the queue has been completely drained. Although this post-processing idle period is intentionally excluded from the workload-window average metrics, it remains relevant from an operational perspective because the Deployment never returns to zero replicas during idle periods.

19.12.7 Engineering Analysis

The KEDA warm-start scenario represents a compromise between static provisioning and full scale-to-zero. By configuring `minReplicaCount = 2`, the system permanently maintains a small amount of processing capacity, allowing incoming messages to be processed immediately without waiting for the Deployment to be activated from zero. The benefit of this approach is clearly visible in the queue behaviour. The peak backlog reaches approximately **57 ready messages**, compared with approximately **73 ready messages** in the scale-to-zero configuration. This reduction confirms that the two baseline consumer replicas immediately contribute to processing incoming messages while the autoscaler completes the scale-out process. At the same time, the average number of active consumers remains equal to only **6.9**, substantially lower than the **15** replicas permanently allocated in the static overprovisioned scenario. Similarly, the average number of busy consumers is **6.3**, whereas the average number of idle consumers is only **0.6**. These values indicate that the warm-start configuration maintains a high level of resource efficiency while providing a significantly faster initial response than the scale-to-zero configuration. The HPA timeline further confirms that the autoscaler behaves consistently with the external Prometheus metric. Starting from a baseline of two replicas, the Deployment progressively scales to six and then eleven replicas as the backlog increases, before gradually returning to the configured minimum once the workload has been processed. Unlike the previous scenario, no final transition to zero replicas occurs, as expected given the configured `minReplicaCount`. The main trade-off is therefore between maintaining a small amount of permanently allocated capacity and reducing the cold-start penalty. Compared with the scale-to-zero configuration, the warm-start strategy consumes slightly more resources during idle periods because two consumer replicas always re-

main active. In return, it substantially reduces the initial backlog and provides immediate processing capacity whenever new workload arrives. Compared with the static overprovisioned deployment, it achieves a similar processing behavior while requiring far fewer consumer replicas on average. Overall, the warm-start configuration represents an effective balance between resource efficiency and responsiveness. It combines most of the resource-saving benefits of event-driven autoscaling with a significantly smaller startup penalty than full scale-to-zero, making it particularly suitable for workloads where a modest amount of always-on capacity is acceptable in exchange for faster reaction to incoming traffic.

19.13 Final Comparison and Trade-off Analysis

The four scenarios analyzed throughout this chapter highlight different trade-offs between processing performance, resource efficiency and autoscaling responsiveness. Although the workload remains identical in all experiments, the overall system behavior changes significantly depending on how consumer replicas are provisioned and managed.

Scenario	**Main Advantages**	**Main Limitations**	**Best Suited For**
Static underprovisioned deployment	Very low number of allocated replicas; no idle consumers during processing; simple configuration	Insufficient processing capacity; large backlog; long load duration; high queueing delay	Lower-bound baseline; not suitable for the considered workload
Static overprovisioned deployment	Immediate message processing; no ready backlog; minimum queueing delay; predictable behavior	Many replicas remain idle; resources allocated even when unnecessary; inefficient during inactive periods	Latency-sensitive workloads where performance is more important than resource efficiency
KEDA with `minReplicaCount = 0`	Maximum idle-time efficiency; scale-to-zero capability; very low average idle consumers during workload	Cold-start effect; initial backlog accumulation; slower reaction to new traffic	Intermittent or bursty workloads where eliminating idle resources is the primary objective
KEDA with `minReplicaCount = 2`	Warm-start behavior; lower peak backlog than scale-to-zero; high resource efficiency; faster initial response	Small baseline resource allocation; cannot completely eliminate idle consumers	Intermittent workloads where reduced startup latency justifies a small always-on capacity

Table 19.3: Comparison of the four deployment strategies evaluated in this chapter.

The static underprovisioned deployment represents the least effective configuration from a performance perspective. Although it minimizes the number of allocated consumer replicas and completely avoids idle consumers during workload processing, its processing capacity is substantially lower than the incoming traffic rate. Consequently, the queue accumulates a large backlog and the workload completion time becomes considerably longer. This configuration is therefore efficient only in terms of consumer utilization, but it is unable to satisfy the performance requirements of the considered workload. The statically overprovisioned deployment represents the opposite extreme. By maintaining fifteen consumer replicas permanently active, the system provides sufficient processing capacity to handle incoming messages immediately, completely preventing backlog accumulation. This configuration achieves the best performance in terms of queueing delay and workload completion time. However, this performance comes at the cost of significant overprovisioning. During workload execution, only about ten consumers are effectively busy, while approximately five remain idle on average. Furthermore, outside the workload period all fifteen replicas remain allocated regardless of whether messages are present. Consequently, this approach maximizes responsiveness but sacrifices resource efficiency. The two KEDA-based configurations provide a considerably more flexible behavior by dynamically adapting the number of consumer replicas to the queue state. Instead of relying on a fixed replica count, KEDA continuously monitors the external Prometheus metric representing the number of RabbitMQ ready messages and delegates replica management to the HPA. As a result, consumer capacity increases only when backlog begins to accumulate and decreases once the workload has been processed, leading to a substantially more efficient allocation of resources than static overprovisioning. The behavior of the KEDA scenarios is also influenced by the selected Prometheus scaler threshold. In these experiments, the threshold was set to **5 ready messages**. This value should not be interpreted as globally optimal, since proving optimality would require a dedicated sensitivity analysis with multiple threshold values. However, for the workload considered in this chapter, it proved to be a balanced choice. The autoscaler was able to scale out sufficiently to drain the accumulated backlog, while the average number of idle consumers remained very low in both KEDA scenarios. A lower threshold would likely have produced a more aggressive scale-out, reducing backlog more rapidly but increasing the risk of temporary overprovisioning. Conversely, a higher threshold would have reduced the number of allocated

consumers, but could have slowed backlog absorption and increased queueing delay. Therefore, the selected threshold contributed to the observed trade-off between responsiveness and resource efficiency. Among the two autoscaling strategies, the configuration with `minReplicaCount = 0` provides the highest level of resource efficiency during idle periods. When the queue becomes empty, the consumer Deployment can be completely scaled down to zero replicas, eliminating idle consumers altogether. This capability represents one of the main advantages of KEDA compared with conventional CPU-based HPA autoscaling. Nevertheless, this benefit introduces the well-known cold-start effect. When new traffic arrives, KEDA must first detect the external metric, activate the Deployment, create the initial consumer pod and subsequently scale out according to the queue backlog. During this interval, messages accumulate in the ready queue, resulting in a larger peak backlog than in the warm-start configuration. The warm-start configuration, obtained by setting `minReplicaCount = 2`, represents an intermediate solution between full scale-to-zero and static overprovisioning. Since two consumer replicas are always available, incoming messages can be processed immediately, significantly reducing the cold-start penalty and lowering the peak backlog compared with the scale-to-zero scenario. On the other hand, the Deployment never reaches zero replicas, meaning that a small amount of baseline resource consumption is permanently maintained. In practice, this configuration exchanges part of the idle-time efficiency of scale-to-zero for improved responsiveness when a new workload arrives. Another important aspect concerns the scale-down mechanisms adopted by KEDA and the HPA. The HPA scale-down stabilization window and the KEDA cooldown period intentionally delay replica removal after the external metric reaches zero. Although this behavior temporarily keeps consumers active after the workload has been completed, it prevents unnecessary oscillations and improves robustness when workloads arrive in bursts or at short time intervals. If new messages appear shortly after the previous workload has been processed, the already running consumer replicas can immediately resume processing without requiring a complete scale-up cycle. The role of the cooldown period also depends on the configured minimum replica count. In the scale-to-zero scenario, it directly determines how long the system remains partially active before KEDA performs the final transition from one replica to zero. Conversely, when `minReplicaCount = 2`, the Deployment cannot be reduced below two replicas. In this case, the cooldown period no longer controls a final scale-to-zero transition, because the system is

constrained to maintain the warm baseline capacity defined by the minimum replica count. Overall, the two KEDA-based configurations provide the best compromise among the solutions evaluated in this chapter. They successfully overcome the poor performance of the underprovisioned deployment while significantly reducing the resource waste associated with static overprovisioning. The most appropriate configuration ultimately depends on the characteristics of the target workload. If maximizing resource savings during prolonged idle periods is the primary objective, the scale-to-zero configuration is the preferred solution. Conversely, if reducing startup latency is more important while still preserving most of the benefits of event-driven autoscaling, the warm-start configuration represents the more suitable choice. Within the experimental workload considered in this thesis, the warm-start KEDA configuration can be regarded as the most balanced solution. It substantially reduces the peak backlog compared with the scale-to-zero approach while requiring, on average, far fewer consumer replicas than the static overprovisioned deployment. At the same time, the scale-to-zero configuration remains the preferred option for applications characterized by long inactive periods, where eliminating idle resources is more important than minimizing the initial backlog generated when new traffic arrives.

19.14 Chapter Summary

This chapter presented a comprehensive experimental comparison of four RabbitMQ consumer deployment strategies: static underprovisioning, static overprovisioning, KEDA scale-to-zero and KEDA warm-start. A common workload, together with Prometheus metrics and a dedicated Grafana dashboard, was used to evaluate both processing performance and resource efficiency through the analysis of queue evolution, consumer utilization and idle CPU consumption. The results showed that static underprovisioning provides insufficient processing capacity, whereas static overprovisioning minimizes queueing delay at the cost of significant idle resource allocation. The KEDA-based configurations achieved a more balanced behavior by dynamically adapting the number of consumer replicas to the workload, with the warm-start configuration reducing the cold-start effect while preserving most of the resource-efficiency benefits of scale-to-zero.

Chapter 20

Conclusions and Future Work

20.1 Conclusions

This thesis investigated Kubernetes autoscaling mechanisms for containerized applications, with a particular focus on event-driven autoscaling through KEDA. After introducing the main Kubernetes concepts and the native Horizontal Pod Autoscaler, the thesis analyzed how KEDA extends the standard autoscaling model by enabling scaling decisions based on external events and application-level metrics. The experimental evaluation proceeded progressively. Initial scenarios based on simpler KEDA triggers, such as the CPU scaler and the Cron scaler, were used to validate the interaction between KEDA, the generated HPA and the target Deployment. These experiments made it possible to observe the basic autoscaling workflow in controlled conditions, before moving to a more application-oriented queue-based workload. The final part of the thesis focused on a RabbitMQ scenario in which consumer replicas process messages produced at a controlled rate. This case study was particularly suitable for evaluating event-driven autoscaling, since the number of messages waiting in the queue directly represents the amount of work still to be processed. The RabbitMQ ready-message metric, collected via Prometheus and used by KEDA, enabled the autoscaler to respond to backlog accumulation and dynamically adjust the number of consumers. The comparison between static and KEDA-based scaling strategies highlighted the main trade-offs involved in replica management. The static underprovisioned configuration minimized the number of allocated consumers but could not process the workload fast enough, resulting in a large backlog and long completion times. The static

overprovisioned configuration removed the backlog almost entirely, but at the cost of keeping more consumers active than necessary. The KEDA-based configurations provided a more adaptive behavior, reducing unused capacity while still scaling out when the queue required additional processing power. Among the KEDA scenarios, the scale-to-zero configuration provided the strongest idle-time resource saving, since the consumer Deployment could be reduced to zero replicas when no messages were present. However, this introduced a cold-start penalty when new traffic arrived. The warm-start configuration, based on a positive minReplicaCount, represented a more balanced solution. It kept a small amount of processing capacity always available, reducing the initial backlog while still avoiding the waste of a fully overprovisioned static deployment. The results confirmed that KEDA is effective for queue-based workloads, but also showed that its behavior depends strongly on parameter tuning. The scaling threshold, minReplicaCount, cooldownPeriod, polling interval and HPA stabilization windows all influence the balance between responsiveness and resource efficiency. In the experiments, the selected threshold value of 5 ready messages provided a reasonable compromise for the considered workload, enabling effective scale-out without producing excessive idle consumers. However, this value should not be considered globally optimal, since the best configuration depends on the workload rate, processing time and acceptable queueing delay. The experiments also showed that scale-down mechanisms do more than delay resource release; they also improve stability and readiness for subsequent bursts. The HPA stabilization window and the KEDA cooldown period can temporarily keep replicas active after the workload has completed, increasing short-term resource usage but improving stability and readiness for possible subsequent bursts. Therefore, autoscaling must be interpreted as a trade-off between fast resource saving and the ability to react quickly to new traffic. Finally, the estimation of idle CPU usage provided a first quantitative indication of the cost of keeping inactive consumers allocated. The measured CPU usage of a generic idle consumer was very low, approximately 0.0413 mCPU. Nevertheless, idle replicas still represent an operational cost, because they consume memory, maintain broker connections and may reserve schedulable resources through Kubernetes requests. For this reason, reducing unnecessary replicas remains relevant, especially in larger or multi-tenant environments. Overall, the thesis showed that KEDA can effectively support application-aware autoscaling in Kubernetes. In the RabbitMQ case study, backlog-based scaling proved more meaningful than relying only

on generic resource metrics. The best strategy depends on the application requirements, scale-to-zero is preferable when idle-time efficiency is the priority, while warm-start configurations are more suitable when faster reaction to new traffic is required.

20.2 Future Work

Several extensions could further improve and generalize the results obtained in this thesis. A first direction is the development of a more complete energy consumption model. In this work, idle CPU usage was used as a first approximation of the cost associated with idle consumers. However, CPU usage alone is not sufficient to estimate the real energy impact of autoscaling decisions. A more accurate model should also include memory usage, network activity, container overhead, broker connection costs, node-level power measurements and the effect of CPU and memory requests on schedulable capacity. A second possible extension is a systematic sensitivity analysis of KEDA parameters. The experiments used fixed values for the scaling threshold, cooldown period and stabilization windows. Future work could repeat the same workloads with different parameter combinations in order to quantify their impact on peak backlog, load duration, average active consumers and idle resource usage. This would make it possible to identify more precise tuning rules for different workload profiles. Another relevant direction is the evaluation of more realistic traffic patterns. The workloads used in this thesis were intentionally controlled and reproducible, with fixed message count and generation rate. Real systems, however, often experience irregular, bursty or time-varying traffic. Testing KEDA with non-periodic bursts, variable interarrival times or real traces would provide a more complete view of its behavior in production-like conditions. Future work could also investigate alternative or combined scaling metrics. In this thesis, RabbitMQ ready messages were used as the main scaling signal because they directly represent queued work. Other metrics, such as message arrival rate, acknowledgment rate, message age, processing latency, or consumer utilization, could provide additional information. Combining multiple metrics may lead to more robust autoscaling decisions, especially in workloads where backlog alone is not sufficient to describe system state. Another possible improvement is the integration of predictive or proactive scaling strategies. KEDA reacts after a metric changes,

which can introduce delays, especially when scaling from zero. Predictive approaches based on historical patterns, scheduled events or traffic forecasts could prepare the system before workload arrival, reducing cold-start effects while preserving resource efficiency. Finally, the same methodology could be applied to more realistic telecommunications and edge-computing workloads. For example, messages could represent monitoring events, logs or data streams generated by network functions or 5G applications. In such scenarios, KEDA could be used to dynamically allocate processing components according to the actual workload intensity, reducing idle resources while preserving responsiveness to traffic bursts. In summary, the results of this thesis provide a basis for further work on event-driven autoscaling in Kubernetes. KEDA proved to be a flexible and effective tool, but its practical benefits depend on workload characteristics, metric selection and parameter tuning. Future developments should therefore focus on energy-aware modeling, broader experimental validation and more advanced autoscaling strategies.

Acknowledgements

First and foremost, I would like to sincerely thank all the professors of the Department of Electrical, Electronic and Information Engineering "Guglielmo Marconi" (DEI) at the University of Bologna for their teaching, dedication and guidance throughout my academic journey. I am grateful for having studied in a stimulating academic environment that encouraged curiosity, critical thinking and continuous learning. Their contribution has played a fundamental role in my education and has provided the knowledge and skills that made this work possible.

My sincere gratitude also goes to my supervisor, Professor **Davide Borsatti**, for his availability, guidance and continuous support throughout the development of this thesis. His expertise, constructive feedback and encouragement have been invaluable to the completion of this work and have provided a solid foundation for my future professional career.

A special thank you goes to my **parents**, whose love, patience, and unwavering support have always been my greatest source of strength. Even during the most challenging moments of my academic journey, when an exam did not go as I had hoped or when the road ahead seemed steeper than ever, they never made me feel discouraged. Instead, they continued to encourage me and believe in my abilities, giving me the confidence to keep moving forward. This achievement belongs to you as much as it belongs to me.

My heartfelt gratitude goes to my **grandparents** for their constant love, support, and confidence in me. They taught me the value of hard work, humility, and perseverance, always encouraging me to believe in my goals, even when they seemed distant or even impossible.

Among them, a special thought is dedicated to my beloved **grandfather Al-**

berto, who passed away just over two years ago. His love, encouragement, and the confidence he always placed in me continue to accompany me every day. I know how much he wished to be here to witness this milestone, and I like to think that, somehow, he is watching over me today. I hope you are proud of the person I have become and of the milestone I have achieved.

I would also like to thank my lifelong friends: **Checco, Francesco Dinatolo, Francesco Maida, Giovanni, and Bruno**. Our friendship has stood the test of time and, despite the commitments, distances, and changes that life inevitably brings, we have always managed to find time to be together, share laughter, and enjoy carefree moments, always with mutual respect. I sincerely hope that this bond will continue to accompany us for many years to come.

A special thank you goes to **Andrea**. Over the past few years, our friendship has grown into something truly special, becoming a relationship built on trust, mutual support, and honest conversations. Having someone with whom I could share doubts, achievements, and challenges has made this journey much easier and reminded me how valuable it is to have a true friend to rely on, especially during the most difficult moments.

And how could I not dedicate a special thought to **Laura, Martina, and Martin**. You came into my life in the most spontaneous, unexpected, and natural way, and somehow managed to turn one of the most demanding periods of my life into a time filled with memories that I will always cherish. Our wonderful trip to **Venice**, together with the many moments we have shared over the past few years, will forever remain among the happiest memories of my university journey, made even more unforgettable by all the funny and unexpected situations we experienced along the way. Without even realizing it, you helped lift the weight of exams and academic stress, making this journey far more enjoyable and serene. **Laura and Martina**, through your spontaneity, energy, and remarkable ability to make everyone feel included, you helped me rediscover a lighter and more carefree side of myself, reminding me how wonderful it is to simply be yourself, free from the pressures and pace of everyday life. I truly hope that time will preserve the spontaneity with which everything began and continue to gift us many more unforgettable moments together.

A special thought also goes to **Nicola, Alessandro**, and to **all the friends**

who have shared part of this journey with me. It would be impossible to mention each of you by name, but every one of you has contributed, in your own way, to making these years truly special. The days we spent together, the dinners, the spontaneous outings, the laughter, and all the moments we shared provided a precious balance between university life and everyday life, making this journey richer in experiences and memories that I will always treasure.

I would also like to express my sincere gratitude to my roommates, **Alessandro Jr, Filippo, and Luca**, who shared with me these unforgettable years in Bologna. Living together gave us countless moments of laughter, meaningful conversations, and everyday adventures that I will always carry with me. Thank you for patiently putting up with my unusual routines, my habits, and my little quirks, making Bologna truly feel like a second home.

Finally, I would like to thank all the **colleagues** I had the pleasure of meeting throughout my academic journey. Whether through a simple conversation, the exchange of lecture notes, a joke before an exam, or a shared moment between classes, each of you contributed, perhaps without even realizing it, to making this journey more enjoyable and to easing the challenges of lectures and exam sessions. It is often these small gestures that, taken together, make the greatest difference, and they too have played a part in helping me reach this milestone.

Matteo Pannia
Bologna, 20 July 2026

Appendix A

Grafana Dashboard PromQL Queries

A.1 Dashboard Overview

This appendix reports the complete set of PromQL queries used to construct the Grafana dashboard employed throughout the comparative evaluation presented in the experimental chapter 19. The queries are grouped according to the corresponding dashboard panels in order to facilitate the reproduction of the monitoring configuration and the interpretation of the collected metrics. The same dashboard and query definitions were used for all four deployment scenarios, while only the consumer provisioning and autoscaling strategy changed between experiments. The dashboard consists of five main panels:

- *Consumer Efficiency Summary*
- *Queue and Processing Summary*
- *Messages Rates*
- *Queue State and Backlog*
- *Consumer State*

Overall, these panels contain seventeen metrics. The complete PromQL query associated with each metric is reported in the following sections.

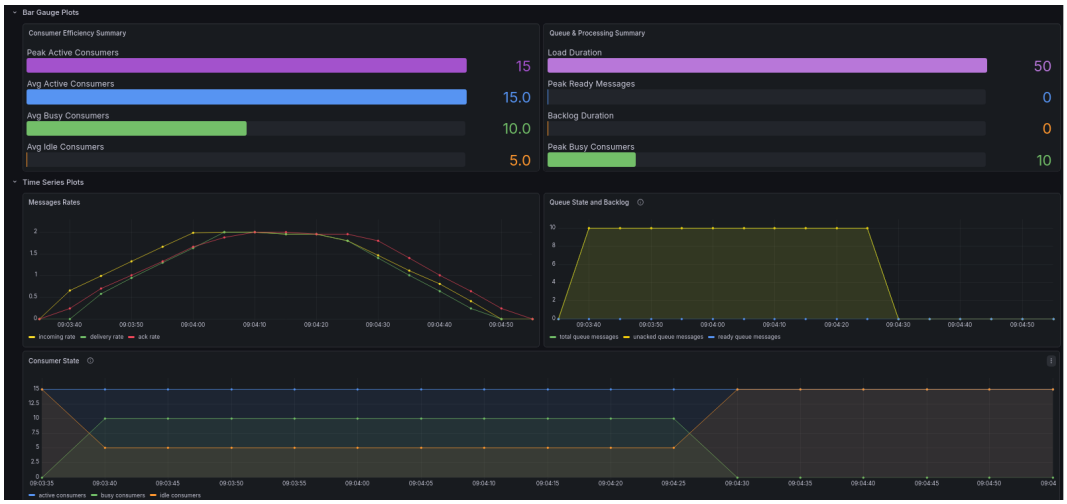


Figure A.1: Complete Grafana dashboard used during the comparative experimental evaluation. The values shown in the figure refer to the static overprovisioned scenario and are included as a representative example of the dashboard organization and the metrics displayed by its five panels.

Figure A.1 provides an overview of the complete monitoring dashboard. Although the displayed values refer to the static overprovisioned experiment, the same panel organization and PromQL queries were used throughout all four scenarios analysed in the thesis.

A.2 Consumer Efficiency Summary

The *Consumer Efficiency Summary* panel reports aggregate metrics related to the number of consumer replicas allocated and effectively used during the workload processing window.

A.2.1 Peak Active Consumers

```
1 max_over_time(rabbitmq_consumers[$__range])
```

Listing A.1: PromQL query used to compute the peak number of active consumers.

A.2.2 Average Active Consumers

```
(
  sum_over_time(
    (
      sum(rabbitmq_queue_consumers)
      *
      (sum(rabbitmq_queue_messages) > bool 0)
    )[$__range:5s]
  )
  /
  sum_over_time(
    (
      sum(rabbitmq_queue_messages) > bool 0
    )[$__range:5s]
  )
) or vector(0).
```

Listing A.2: PromQL query used to compute the average number of active consumers during the workload processing window.

A.2.3 Average Busy Consumers

```
(
  sum_over_time(
    (
      sum(rabbitmq_queue_messages_unacked)
      *
      (sum(rabbitmq_queue_messages) > bool 0)
    )[$__range:5s]
  )
  /
  sum_over_time(
    (
      sum(rabbitmq_queue_messages) > bool 0
    )[$__range:5s]
  )
```

```
15 ) or vector(0)
```

Listing A.3: PromQL query used to compute the average number of busy consumers during the workload processing window.

A.2.4 Average Idle Consumers

```
1  (
2    sum_over_time(
3      (
4        clamp_min(
5          sum(rabbitmq_queue_consumers) - sum(
       rabbitmq_queue_messages_unacked),
6          0
7        )
8        *
9        (sum(rabbitmq_queue_messages) > bool 0)
10     )[$__range:5s]
11   )
12   /
13   sum_over_time(
14     (
15       sum(rabbitmq_queue_messages) > bool 0
16     )[$__range:5s]
17   )
18 ) or vector(0)
```

Listing A.4: PromQL query used to compute the average number of idle consumers during the workload processing window.

A.3 Queue and Processing Summary

The *Queue and Processing Summary* panel reports aggregate indicators related to workload duration, backlog accumulation and peak consumer activity.

A.3.1 Load Duration

```
(
  sum_over_time(
    (
      sum(rabbitmq_queue_messages) > bool 0
    )[$__range:5s]
  ) * 5
) or vector(0)
```

Listing A.5: PromQL query used to compute the overall workload duration.

A.3.2 Peak Ready Messages

```
max_over_time(rabbitmq_queue_messages_ready[$__range])
```

Listing A.6: PromQL query used to compute the peak number of ready messages.

A.3.3 Backlog Duration

```
(
  sum_over_time(
    (
      sum(rabbitmq_queue_messages_ready) > bool 0
    )[$__range:5s]
  ) * 5
) or vector(0)
```

Listing A.7: PromQL query used to compute the duration of the ready-message backlog.

A.3.4 Peak Busy Consumers

```
max_over_time(
  (
    sum(rabbitmq_queue_messages_unacked)
  )[$__range:5s]
) or vector(0)
```

Listing A.8: PromQL query used to compute the peak number of busy consumers.

A.4 Message Rates

The *Messages Rates* panel shows the temporal evolution of message publication, delivery and acknowledgement rates.

A.4.1 Incoming Rate

```
sum(rate(rabbitmq_channel_messages_published_total[30s])) or vector
    (0)
```

Listing A.9: PromQL query used to visualize the incoming message rate.

A.4.2 Delivery Rate

```
sum(rate(rabbitmq_channel_messages_delivered_ack_total[30s])) or
    vector(0)
```

Listing A.10: PromQL query used to visualize the RabbitMQ message delivery rate.

A.4.3 Acknowledgement Rate

```
sum(rate(rabbitmq_channel_messages_acked_total[30s])) or vector(0)
```

Listing A.11: PromQL query used to visualize the message acknowledgement rate.

A.5 Queue State and Backlog

The *Queue State and Backlog* panel distinguishes messages waiting in the queue from messages that have already been assigned to consumers but have not yet been acknowledged.

A.5.1 Total Queue Messages

```
1 rabbitmq_queue_messages
```

Listing A.12: PromQL query used to visualize the total number of messages present in the queue system.

A.5.2 Unacknowledged Queue Messages

```
1 rabbitmq_queue_messages_unacked
```

Listing A.13: PromQL query used to visualize the number of unacknowledged messages.

A.5.3 Ready Queue Messages

```
1 rabbitmq_queue_messages_ready
```

Listing A.14: PromQL query used to visualize the number of messages waiting in the ready state.

A.6 Consumer State

The *Consumer State* panel reports the temporal evolution of the active, busy and idle consumer replicas.

A.6.1 Active Consumers

```
sum(rabbitmq_consumers)
```

Listing A.15: PromQL query used to visualize the number of active consumers.

A.6.2 Busy Consumers

```
sum(rabbitmq_queue_messages_unacked)
```

Listing A.16: PromQL query used to visualize the number of consumers currently processing messages.

A.6.3 Idle Consumers

```
clamp_min(
  sum(rabbitmq_queue_consumers)
  -
  sum(rabbitmq_queue_messages_unacked),
  0
)
```

Listing A.17: PromQL query used to visualize the number of active consumers not currently processing messages.

Author's Notes

Software Versioning and Project Evolution

The technologies discussed throughout this thesis, including Kubernetes and KEDA, are open-source projects that are actively developed and continuously improved by large communities of contributors. As a consequence, APIs, component names, configuration parameters, supported features and implementation details may evolve over time as new software releases become available. All architectural descriptions, software configurations, experiments and implementation details presented in this work refer to the versions available during the development of the thesis. Readers should therefore be aware that some technical aspects described in this document may change in future releases or may no longer reflect the latest implementation of the corresponding projects. A representative example concerns the *scale-to-zero* capability. Throughout this thesis, scale-to-zero is presented as one of the distinguishing features provided by KEDA, since it represented the mature and production-ready solution adopted during the experimental evaluation. However, after the completion of the experimental activities described in this work, Kubernetes introduced an *alpha* implementation of native Horizontal Pod Autoscaler (HPA) scale-to-zero support for workloads using External and Object metrics through the `HPAScaleToZero` feature gate, first released in Kubernetes v1.36 [9]. At the time of writing, this functionality remains experimental, disabled by default and limited to specific autoscaling scenarios. Conversely, KEDA continues to provide a significantly broader event-driven autoscaling ecosystem, supporting a wide range of event sources, external systems and scaling triggers beyond the current native capabilities of the Kubernetes Horizontal Pod Autoscaler. For these reasons, the technical descriptions and comparisons presented in this thesis should be interpreted with reference to the software versions available

during the preparation of this work.

Use of Generative Artificial Intelligence

Generative Artificial Intelligence tools were used as supporting instruments during the preparation of this thesis. More specifically, these tools were employed to assist in:

- Improving the readability and overall fluency of selected technical descriptions
- Rephrasing portions of text derived from official documentation while preserving their original technical meaning
- Generating a limited number of illustrative figures and diagrams used to visualize the communication flow between software components and the interactions among the different elements of the proposed architecture

The use of these tools was limited to editorial and graphical support. All experimental activities, software implementations, Kubernetes configurations, measurements, analyses, architectural decisions, obtained results and final conclusions presented throughout this thesis were independently designed, implemented, executed and critically evaluated by the author. Every AI-assisted contribution included in this manuscript was carefully reviewed, verified against the official documentation and the experimental results, and revised whenever necessary before being incorporated into the final version of this thesis.

Bibliography

[1] Imesh Gunaratne. *A Reference Architecture for Deploying WSO2 Middleware on Kubernetes*. White Paper. WSO2, 2016. URL: https://wso2.com/whitepapers/a-reference-architecture-for-deploying-wso2-middleware-on-kubernetes/.

[2] Wim Van Deun. *Declarative vs Imperative Automation: Why It Matters for Infrastructure Success*. OpsMill. 2025. URL: https://opsmill.com/blog/declarative-vs-imperative-automation/.

[3] Jérôme Petazzoni and Contributors. *Container Training: Deploying and Scaling Microservices with Docker and Kubernetes*. https://container.training/. Open-source training material. 2026.

[4] Erwin Staal. *Event-driven Autoscaling on Kubernetes with KEDA and Azure Functions*. Presentation, DevOps Fest 2020. Available online. 2020. URL: https://www.youtube.com/watch?v=LmrEY_6raNQ.

[5] Always learning. *Autoscaling Explained — HPA Vs VPA — Kubernetes*. 2024. URL: https://ibrahims.medium.com/autoscaling-explained-hpa-vs-vpa-kubernetes-b9d6133d95b4.

[6] *Introduction to KEDA*. YouTube video. 2020. URL: https://www.youtube.com/watch?v=bBLnmKrWDHg.

[7] Microsoft. *Simplified Application Autoscaling with Kubernetes Event-Driven Autoscaling (KEDA) Add-on*. Microsoft Learn. 2025. URL: https://learn.microsoft.com/en-us/azure/aks/keda-about.

[8] Weeking and Gaby Fulchic. *Kubernetes Resource Optimization with Horizontal Pod Autoscaling via Custom Metrics and Prometheus Adapter*. Zwindler.fr. 2024. URL: https://blog.zwindler.fr/en/2024/10/11/kubernetes-resource-optimization-with-horizontal-pod-autoscaling-via-custom-metrics-and-prometheus-adapter/.

[9] *Kubernetes v1.36 Release Notes*. 2026. URL: https://kubernetes.io/blog/2026/04/22/kubernetes-v1-36-release/.

[10] The Kubernetes Authors. *Kubernetes Documentation*. 2026. URL: https://kubernetes.io/docs/.

[11] The Kubernetes Authors. *Kubernetes Concepts*. 2026. URL: https://kubernetes.io/docs/concepts/.

[12] The Kubernetes Authors. *Horizontal Pod Autoscaling*. 2026. URL: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.

[13] Kubernetes SIG Instrumentation. *Metrics Server*. 2026. URL: https://github.com/kubernetes-sigs/metrics-server.

[14] Kubernetes SIG Instrumentation. *Prometheus Adapter*. 2026. URL: https://github.com/kubernetes-sigs/prometheus-adapter.

[15] The KEDA Authors. *KEDA Documentation*. 2026. URL: https://keda.sh/docs/.

[16] The KEDA Authors. *ScaledObject Specification*. 2026. URL: https://keda.sh/docs/latest/reference/scaledobject-spec/.

[17] The KEDA Authors. *Prometheus Scaler*. 2026. URL: https://keda.sh/docs/latest/scalers/prometheus/.

[18] The KEDA Authors. *RabbitMQ Scaler*. 2026. URL: https://keda.sh/docs/latest/scalers/rabbitmq-queue/.

[19] The KEDA Authors. *Cron Scaler*. 2026. URL: https://keda.sh/docs/latest/scalers/cron/.

[20] The KEDA Authors. *CPU Scaler*. 2026. URL: https://keda.sh/docs/latest/scalers/cpu/.

[21] Prometheus Authors. *Prometheus Documentation*. 2026. URL: https://prometheus.io/docs/.

[22] Prometheus Authors. *PromQL Documentation*. 2026. URL: https://prometheus.io/docs/prometheus/latest/querying/basics/.

[23] Prometheus Operator Authors. *Prometheus Operator Documentation*. 2026. URL: https://prometheus-operator.dev/docs/.

[24] Grafana Labs. *Grafana Documentation*. 2026. URL: https://grafana.com/docs/.

[25] RabbitMQ Team. *RabbitMQ Documentation*. 2026. URL: https://www.rabbitmq.com/docs.

[26] RabbitMQ Team. *RabbitMQ Prometheus Plugin*. 2026. URL: https://www.rabbitmq.com/prometheus.html.

[27] Helm Authors. *Helm Documentation*. 2026. URL: https://helm.sh/docs/.

[28] Docker Inc. *Docker Documentation*. 2026. URL: https://docs.docker.com/.

[29] NGINX Authors. *NGINX Prometheus Exporter*. 2026. URL: https://github.com/nginx/nginx-prometheus-exporter.

[30] Kubernetes SIG Instrumentation. *kube-state-metrics*. 2026. URL: https://github.com/kubernetes/kube-state-metrics.

List of Figures

1.1 Kubernetes cluster architecture 3
1.2 Physical and logical networking 7
1.3 Simplified communication model in Kubernetes. 9
1.4 Imperative vs declarative management 10
1.5 Kubernetes deployment lifecycle 13
1.6 Kubernetes Pod architecyure 14
1.7 Internal traffic flow for a ClusterIP Service. 16
1.8 ClusterIP Service communication 17
1.9 External traffic flow for a NodePort Service. 18
1.10 NodePort Service communication 19

2.1 Horizontal vs vertical scaling 23
2.2 Kubernetes autoscaling architecture 23

3.1 Keda high-level overview 28
3.2 Keda architecture 30
3.3 Simplified KEDA scaling workflow. 34

4.1 Native CPU autoscaling workflow 41
4.2 Simplified metric collection workflow. 43
4.3 Kubernetes API Aggregation Layer 47

5.1 kubectl top workflow 63

6.1 Preserved Native CPU metrics pipeline 86
6.2 KEDA CPU autoscaling workflow 91

7.1 Native CPU metrics pipeline 103

8.1 Workflow of the KEDA Cron scaler. 118

11.1 Prometheus autoscaling architecture without keda 174

13.1 Prometheus Adapter vs KEDA autoscaling architectures. 236
13.2 Internal metric polling workflow in KEDA. 241
13.3 Simplified KEDA autoscaling workflow. 245

15.1 Monitoring workflow for KEDA components. 268

16.1 RabbitMQ message lifecycle. 279
16.2 RabbitMQ-based autoscaling workflow with KEDA. 282

18.1 RabbitMQ Prometheus metric acquisition workflow. 327

19.1 Idle consumer CPU usage time series 364
19.2 Average idle consumer CPU usage 365
19.3 Reference idle consumer CPU usage 367
19.4 Idle CPU measurement dashboard 368
19.5 Static underprovisioned summary metrics 370
19.6 Static underprovisioned message rates 372
19.7 Static underprovisioned queue state 373
19.8 Static underprovisioned consumer state 375
19.9 Static overprovisioned summary metrics 379
19.10Static overprovisioned message rates 381
19.11Static overprovisioned queue state 382
19.12Static overprovisioned consumer state 384
19.13Scale to zero summary metrics 394
19.14Scale to zero message rates 396
19.15Scale to zero queue state 397
19.16Scale to zero consumer state 399
19.17Warm start summary metrics 406
19.18Warm start message rates 408
19.19Warm start queue state 410
19.20Warm start consumer state 412

A.1 Grafana dashboard overview 430

List of Tables

3.1 Main Keda components summary 37

4.1 CPU utilization example 49
4.2 CPU-based scaling modes 56

5.1 Resources involved in CPU-based autoscaling 60
5.2 Example of node resource utilization 63
5.3 Example of Pod resource utilization 64
5.4 Example kubectl get hpa 76
5.5 HPA status under low CPU utilization 77
5.6 Pod resource utilization under low CPU load 78

6.1 Kubernetes CPU autoscaling vs KEDA CPU autoscaling. 88

7.1 Resources involved in the KEDA CPU autoscaling 96
7.2 Native Kubernetes approach vs KEDA-based approach. 108

8.1 CPU-based vs Cron-based autoscaling. 113
8.2 HPA default stabilization windows 130
8.3 Example of scale-down recommendations 131
8.4 Example of scale-up recommendations 133
8.5 Stabilization window and scaling policies 134
8.6 HPA behavior and equivalent ScaledObject configuration. 137
8.7 HPA stabilization window vs KEDA cooldown period. 139

9.1 Resources used in the Cron scaler experiment. 146
9.2 kubectl get hpa -w Cron scaler simplified output 158
9.3 Initial state before the Cron trigger becomes active. 159
9.4 First sample after the Cron trigger becomes active. 159

9.5 First HPA evaluation after the initial scale-from-zero operation. 160
9.6 Second sample after applying the default scale-up policies. . . . 161
9.7 Final scale-up phase reaching the desired number of replicas. . . 162
9.8 First sample after the Cron trigger becomes inactive. 162
9.9 Activation of the stabilization window. 163
9.10 Samples first scale-down evaluation. 163
9.11 Recommendations first scale-down evaluation. 164
9.12 Observed result after the first scale-down evaluation. 164
9.13 Samples second scale-down evaluation. 164
9.14 Recommendations second scale-down evaluation. 165
9.15 Observed result after the second scale-down evaluation. 165
9.16 Samples considered third scale-down 165
9.17 Recommendations third scale-down evaluation. 165
9.18 Observed result after the third scale-down evaluation. 166
9.19 Final HPA sample 166

11.1 Metrics exposed by the NGINX Prometheus Exporter. 178
11.2 Prometheus components role separation 187

12.1 kubectl get hpa -w output 228

13.1 Prometheus Adapter vs KEDA architecture 242

15.1 Representative metrics exposed by the KEDA components. . . . 271

17.1 Deployment strategies: Producer vs consumer 301
17.2 Main KEDA autoscaling parameters in RabbitMQ setup. 306
17.3 Recommendations first stabilization window. 316
17.4 Second scale-down stabilization window. 317
17.5 Third scale-down stabilization window. 317
17.6 Fourth scale-down stabilization window. 318
17.7 Recommendations final stabilization window. 318
17.8 Final scale-down stabilization window. 319

18.1 Native RabbitMQ scaler vs KEDA Prometheus scaler. 336

19.1 Grafana dashboard metrics overview 361
19.2 Main KEDA autoscaling parameters 390
19.3 Deployment strategies comparison 416