Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea Magistrale in Informatica

Exploring the Effectiveness of AWS Lambda and Knative in a Serverless Web Crawler: A Comparative Study

Relatore:
Chiar.mo Prof.
Gianluigi Zavattaro

Correlatori:
Ing. Emanuele Casadio
Dott. Matteo Trentin

Presentata da:
Davide Pruscini

sessione III - terzo appello
Anno Accademico 2022-2023

```
Here is my source code.
Run it on the Cloud for me.
I do not care how.
```

Onsi Haiku Test by
Onsi Fakhouri

Abstract

The Internet has become a key resource for accessing and sharing information. However, not all content found on it can be considered legitimate, and using tools such as web crawlers can help search for violations. In this thesis, carried out in collaboration with Kopjra, we aim to develop a web crawler application capable of automatically visiting a website, extracting URLs and indexing the HTML documents of its web pages, so as to enable keyword searches. We decided to compare two serverless implementations based on AWS Lamba and Knative, with a third microservice-based one that exploits the resources made available by Kubernetes. It is also possible to choose between two search methodologies: HTTP requests or Browser automation. To support the application, two microservices were developed, comprising the backend and frontend, as well as the deployment of an Elasticsearch cluster, which is necessary for proper ingestion of the content of web pages. Thanks to a series of tests, it is possible to compare the different implementations and understand the critical issues of each.

Keywords: *Cloud Computing, FaaS, Serverless, Kubernetes, AWS Lambda, AWS SQS, AWS SNS, Knative, RabbitMQ, CloudEvents, Web Crawler, Browser Automation, Puppeteer, Elasticsearch, Amazon CloudWatch, Prometheus, Grafana, InfluxDB, Telegraf.*

Sommario

Questa tesi descrive il lavoro svolto durante il tirocinio presso Kopjra Srl [63], società bolognese che si occupa di investigazioni online, OSINT e network forensics. L'attività è stata svolta con la supervisione dell' Ing. Emanuele Casadio, CTO e cofondatore dell'azienda, e la collaborazione del Dott. Matteo Trentin, studente Unibo al secondo anno di dottorato in "Computer Science and Engineering".

L'elaborato mira a valutare l'efficacia dell'approccio serverless per un applicativo web crawler.

L'idea nasce dalla necessità dell'azienda di raccogliere informazioni su siti web che potrebbero contenere violazioni della reputazione, della proprietà intellettuale o industriale. La visita di pagine web è quindi volta all'estrazione di URL e all'indicizzazione dell'HTML contenuto in esse, così da poter effettuare, in un secondo momento, ricerche per parole chiave.

Nel corso del tirocinio, ho approfondito la letteratura relativa al web crawling e valutato le diverse metodologie adottate per la sua realizzazione. Al fine di garantire costi d'infrastruttura minimi e massima scalabilità, si è scelto di utilizzare il paradigma serverless, già consolidato all'interno dell'azienda per altri prodotti. Sono state quindi confrontate le implementazioni basate su AWS Lambda e Knative, con la rispettiva implementazione a microservizio che sfrutta l'API di Kubernetes. È inoltre possibile scegliere tra due modalità di ricerca, l'automazione del browser, dove si interagisce direttamente con il DOM, oppure l'invio di richieste HTTP, più efficiente in termini di risorse e velocità.

A supporto dell'applicativo, ho sviluppato due microservizi: uno per la gestione delle ricerche (backend) e uno mirato a migliorare l'usabilità dell'applicazione, consentendo agli utenti di interagire in modo intuitivo con l'interfaccia (frontend). Inoltre, ho effettuato il deployment di un cluster Elasticsearch mediante l'apposito operatore di Kubernetes e creato un indice personalizzato per garantire una corretta elaborazione dei documenti HTML.

La validità delle soluzioni proposte è supportata da una serie di test che hanno permesso la raccolta di varie metriche, facilitando un confronto tra di esse. Infine, questa analisi ha permesso di rilevare i vantaggi e gli svantaggi di ciascuna variante, individuando contestualmente le relative limitazioni e le aree che richiedono miglioramenti.

Contents

List of Figures

Chapter 1

Introduction

This chapter provides an overview of the thesis and the topics covered, from the origin of the term cloud computing to modern-day cloud computing, focusing on the serverless paradigm. Also the web crawling process will be analysed, and the ingestion phase will be studied concerning the chosen search engine.

Section 1.1 defines the cloud and describes its five essential characteristics; some service models that have emerged over time are also included. In Section 1.2, the serverless paradigm will be analysed, considering its characteristics, advantages and issues. Section 1.3 explains a general web crawler architecture and how it can be implemented according to requirements. It uses browser automation to interact with the web browser and exploits Elasticsearch [23] as a search engine to index the scraped HTML documents. Section 1.4 presents the objectives of this thesis and Section 1.5 explains how it is structured.

1.1 The Cloud Era

The idea of computing in a "cloud" traces back to the origins of utility computing, a concept publicly proposed in 1961 by John McCarthy, a computer scientist considered a pioneer in the field of AI [2]. He compared future computers with the telephone system. He said that someday the computing process might be organised as a public utility, where users share resources and have access to different services [44].

After a few decades, that the computational model had existed, it was named cloud computing. It revolutionised the world from a technological and commercial point of view, enabling companies to move away from the need for large centralised mainframes and to have on-demand computing resources made available via the Internet with consumption-based pricing. Cloud computing has become an industry game-changer, and people realise the potential of combining and sharing computing resources instead of building and maintaining them [44].

Nowadays, one of the most widely used definitions of cloud computing is given by NIST [28] that provides five essential characteristics:

- *On-demand self-service*: a consumer can automatically access computing resources without needing human interaction with the service provider;
- *Broad network access*: capabilities are network-accessible through standard mechanisms, facilitating use on various client platforms;
- *Resource pooling*: provider pools computing resources, dynamically allocates them according to demand and offers location-independent access (e.g. storage, processing, memory and network bandwidth);
- *Rapid elasticity*: capabilities can be dynamically allocated and deallocated to match demand, often seeming limitless to the consumer;
- *Measured service*: cloud systems automatically manage and optimise the use of resources, offering transparency to service providers and consumers.

Interest in this topic has grown over the years, and the trend shown in Figure 1.1 demonstrates this. It is a new technology that has established itself in modern society and contributes benefits to other fields of computer science.

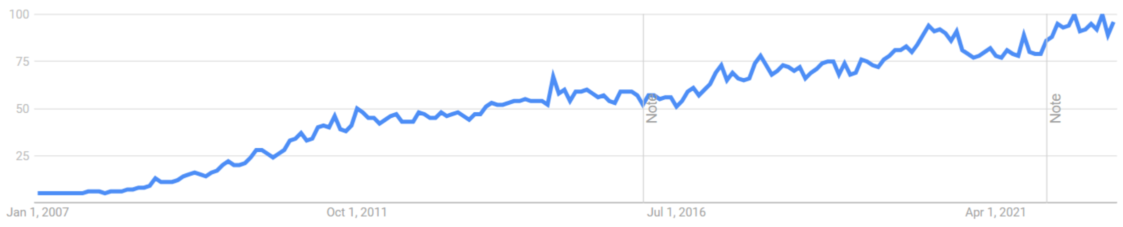


Figure 1.1: Interest over time for "Cloud Computing" searches on Google, from 2007 to 2023. The scale is normalised from 0 to 100, with 0 representing the least popular search term and 100 representing the most popular search term. Picture from Google Trends.

1.1.1 Cloud Service Models

Numerous cloud services are now available, each providing varying computing resources and capabilities, as shown in Figure 1.2. The three fundamental models are discussed by NIST in [28] and then clarified in [60]. In addition, new models have been introduced in recent years, which will be analysed.

Infrastructure as a Service

The first abstraction layer gives customers instant access to cloud-based computing infrastructure such as servers, storage capacity, and networking resources. Users can configure and use them just like they would with on-premises hardware. The main difference is that the Cloud Service Provider (CSP) is responsible for hosting, managing, and maintaining the data centre's hardware and computing resources. IaaS customers can access the hardware through an internet connection and pay for their usage via a subscription or pay-as-you-go model.

Typical end users include developers and other IT professionals who require direct control over their computational resources. Some examples of IaaS are Elastic Compute Cloud from Amazon Web Services (AWS) and Compute Engine from Google Cloud Platform (GCP).

Platform as a Service

The second abstraction layer is built over IaaS and gives customers a ready-to-use cloud platform to develop and deploy software without the complexities of managing the underlying infrastructure. Users can create, run and manage their applications using a software platform provided by another party. The provider has to take care not only of their physical resources but also of middleware, database systems, operating systems, and other supplementary services essential for the consumer application, leaving the end user in sole control of the deployed applications and their data.

Typical professional roles include developers who want to make their apps easily available. Some examples of PaaS are Elastic Beanstalk from AWS, App Engine from GCP and OpenShift from RedHat.

Software as a Service

The final fundamental layer at the top of the pyramid is SaaS, which provides access to applications running on a cloud infrastructure. These applications are usually accessible from various client devices using a program interface or a web browser. They can be intended as computer programs that enable the user to perform coordinated functions, tasks, or activities.

The last few decades have seen an increase in SaaS applications, such as Google Workspace (Gmail, Drive, Calendar, etc.), Slack and Netflix, which are increasingly becoming part of our daily lives.

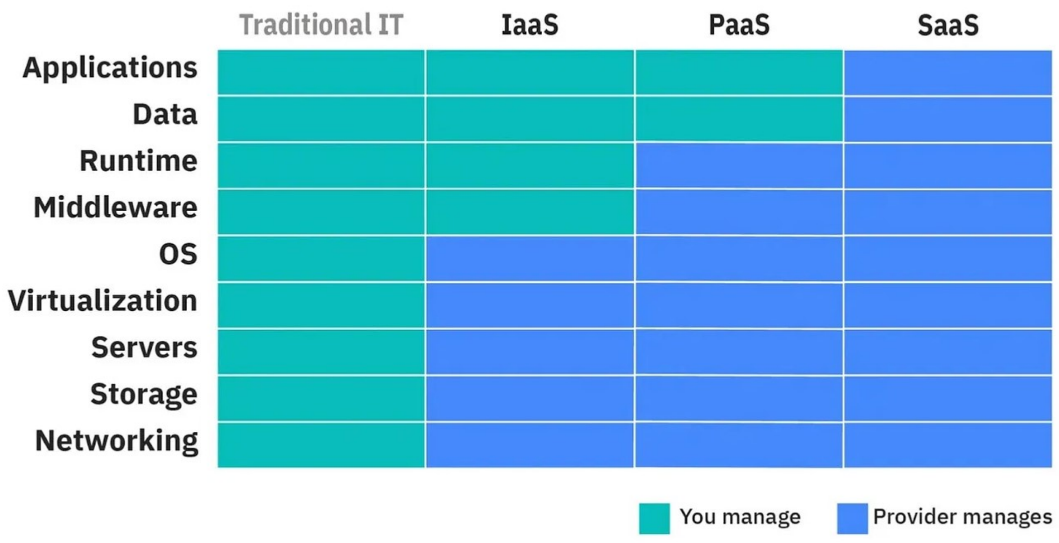


Figure 1.2: A diagram of the different levels of abstraction in cloud service models, depicting how IaaS, PaaS, and SaaS shift management and operation obligations from the end user to the cloud provider. Picture from ByteByteGo.

The following three service models aren't described by NIST in [60]; they emerged in later years and helped address specific challenges and trends in software development and deployment. Cloud Native Computing Foundation (CNCF) introduces these trends in [29] to better understand them and all their components.

Container as a Service

The CaaS model allows users to maintain complete control over infrastructure and get maximum portability. They can utilise container orchestration tools like Kubernetes [9, 69], Docker Swarm and Apache Mesos to develop and launch portable and easily-configured applications. These tools provide flexibility and control over the app's configuration, allowing it to run on various environments without needing reconfiguration or redeployment.

Thanks to its less-opinionated application deployment model, users can enjoy maximum control, flexibility, re-usability, and ease of bringing containerised apps into the cloud. In comparison, developers have significantly more responsibility for the operating systems, load balancing, capacity management, scaling, logging and monitoring.

Backend as a Service

The BaaS model provides similar services to PaaS but also handles the typical components of an application. It simplifies the management of servers, setting up authentication, data storage, APIs and other backend components by abstracting their complexities. The developers must primarily focus on frontend development and application logic: the framework manages all third-party services.

Two of the most used BaaS platforms are Firebase from Google and Amplify from AWS.

Function as a Service

According to [68], the FaaS model differs from fundamentals such as IaaS and PaaS, and has significant differences to more recent models such as microservices.

It is closely related to the notion of cloud function execution, defined in [66] as "a small, stateless, on-demand service with a single functional responsibility that implements specific business logic, depending on the goal of the application". They also identify three main characteristics for them:

- *Short-lived*: each function takes in a typically small input and, after a typically short amount of time, produces the output;
- *Devoid of operational logic*: the platform layer takes care of all operational matters, which allows the cloud functions to be platform-independent;
- *Context-agnostic*: is unaware of how or why it is used.

The Function as a Service model exploits these cloud functions to enable developers with minimal experience to create, monitor and invoke them. This paradigm can be seen as serverless computing in which the cloud provider manages the resources, lifecycle, and event-driven execution of user-provided functions [36].

In Section 1.2, the characteristics of this emerging paradigm will be explored in detail. Some examples of FaaS are Lambda from AWS, Cloud Functions from GCP and Knative, initially created by Google with contributions from over 50 different companies [42].

1.2 Serverless Computing

The traditional cloud computing service models have pros, but each has some cons that a software developer can perceive as complex. To ease this complexity, the cloud providers introduced a paradigm similar to PaaS, named serverless, hiding all management tasks about underlying servers for developers and allowing them more control over applications [70].

Serverless computing describes a finer-grained deployment model that bundles applications into one or more cloud functions and uploads them to a platform which deals with the execution, scaling and billing in response to the requested demand.

With this new paradigm, consumers don't have to spend time and resources on server provisioning, maintenance, updates, scaling and capacity planning: a serverless platform takes care of all the necessary tasks and capabilities rather than being handled by individual developers or IT/operations teams. This abstraction allows developers to concentrate on writing their application's business logic while freeing up operations engineers to focus on more critical tasks for business [29].

There are two principal actors in the serverless scenario:

- *Developer*: writes cloud functions and benefits from the advantages of a serverless platform;
- *Provider*: deploys the serverless platform for an internal or external client.

Although "serverless" implies that there is no need for servers, servers are still required to run a serverless platform. The platform provider manages the servers, be it virtual machines or Containers. It's worth noting that the provider will incur some cost for the running of the platform, even if there is no activity. Additionally, a self-hosted system can still be considered serverless. In this case, one team will act as the provider, while another will serve as the developer.

Usually, serverless is seen as cloud function execution, packaged as FaaS, representing the core of serverless computing. Moreover, cloud platforms also provide specialised serverless frameworks that meet specific application requirements, such as BaaS. In simple terms $serverless = FaaS + BaaS$ [50].

As shown in Figure 1.3, serverless functions are designed to be triggered by specific events like HTTP requests, data updates in cloud-based storage or simple notifications. These requirements concern developers who can define the rules for binding serverless functions to events. When events occur, serverless functions are triggered, and the serverless platform will automatically prepare the runtime environments. The instance initialisation, application transmission, and application code loading are all preparation processes in the runtime environments called function instances. At the end of the execution, the serverless platform automatically recycles function instances and releases corresponding resources.

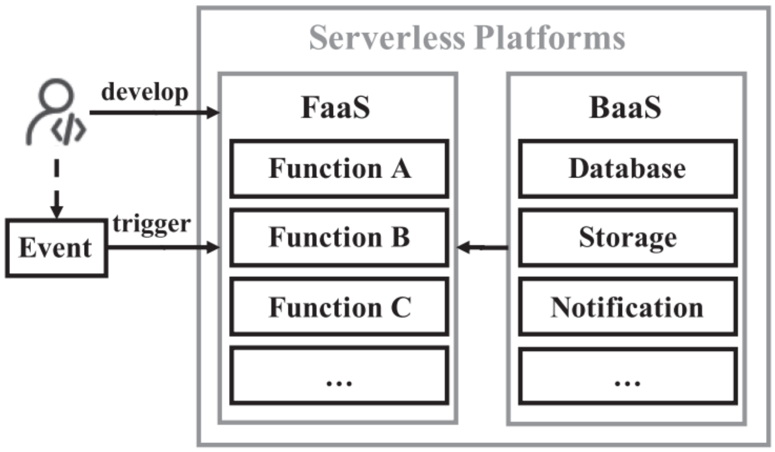


Figure 1.3: A diagram illustrating the development process within a serverless computing platform. Picture from [70].

To better understand serverless computing, the main features shown by the authors of "Rise of the Planet of Serverless Computing: A Systematic Review" [70] are described below.

- *Functionality and no operations*: developers can easily create applications on serverless platforms by choosing the language they are most familiar with and that suits the use case (e.g., Python, JavaScript, Java, etc.). When deploying serverless applications, developers only need to upload their application code or Container to the serverless platform, and the platform takes care of the rest;

- *Auto-scaling*: serverless platforms can automatically adjust the number of function instances depending on the application workload dynamics. This operation is called scaling, and it allows new function instances to be started up or running occurrences to be recycled. After a request is completed, the corresponding function instances and allocated resources are stored in the memory for a short time. This prepares them to be reused by subsequent requests of the same function. Without subsequent requests, the serverless platform will scale to zero by automatically recycling these instances and resources. However, scaling down to zero creates a problem called "cold start" for new incoming requests. This is because preparing the required runtime environments from scratch takes a long time;

- *Utilisation-based billing*: serverless computing is a pay-per-use model where software developers only pay for the resources allocated or consumed by the serverless application at the execution level. As we already said, serverless functions are event-driven and only run when triggered, allowing developers to avoid charges for idle resources. This is why the serverless paradigm is more cost-effective than traditional cloud computing, where resources need to be continuously rented;

- *Separation of computation and storage*: the separation mode of computation and storage is adopted by serverless computing to allow the auto-scaling and the effective handling of intensive workloads;
- *Additional limitations*: cloud providers impose additional restrictions on serverless functions to ensure the critical auto-scaling feature of serverless platforms. These limitations typically include function execution timeout, deployment package size, local disk size and maximum memory allocation. Moreover, different cloud providers have various rules concerning their serverless platforms.

In general, a serverless approach should be considered the best choice when there are processes that are easy to parallelize in independent work units and that have sporadic demand with significant, unpredictable variance in scaling requirements. Last but not least, the development time is significantly reduced to support one's evolving business.

Serverless architectures offer a new development and deployment option for cloud-native workloads that require careful consideration in the initial stages. Companies will have a high speed of change in terms of development and can handle unpredictable capacity and infrastructure requirements. Figure 1.4 reports an overview of the serverless landscape that gives us an idea of how many realities are available and ready to use today.

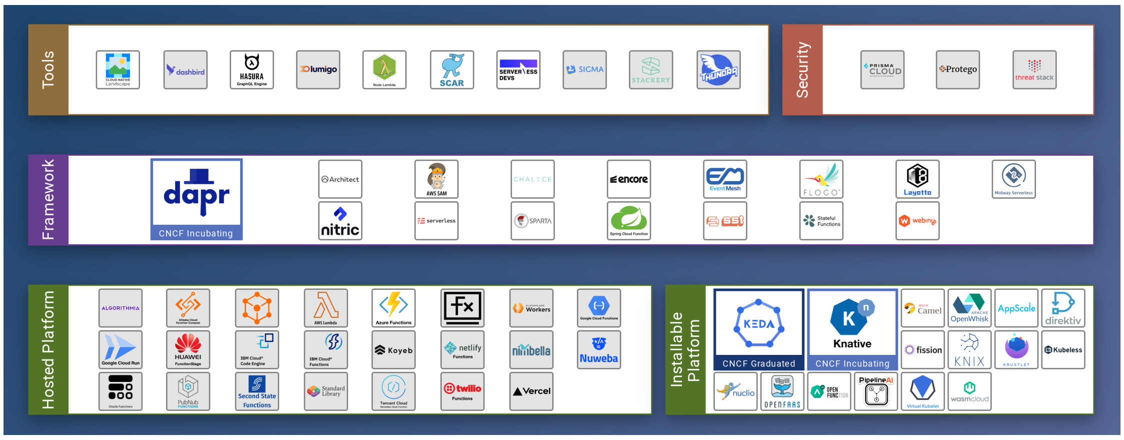


Figure 1.4: Overview of all serverless platforms, frameworks and tools categorised by Cloud Native Computing Foundation (CNCF). Grey-coloured logos indicate non-open-source projects. Picture from CNCF.

1.2.1 Open Source Solutions

Many CSPs offer event-driven serverless platforms on their public clouds, such as AWS Lambda [54]. These platforms allow deployment in any supported language and execute on-demand as Docker [46] Containers.

The authors of “Analyzing Open-Source Serverless Platforms: Characteristics and Performance” [40] analyse the problem of using public serverless platforms that may result in vendor lock-in risk. To avoid this risk, more and more open-source serverless platforms, which allow developers to deploy and manage functions on their self-hosted clouds, have emerged. Even so, building cloud functions requires a lot of expertise, and in order to not affect performance, an in-depth understanding of platform frameworks is necessary.

It becomes a challenge for a service developer to distinguish and select the appropriate serverless platform in different scenarios.

As shown in Figure 1.5, all these solutions are based on Kubernetes [9, 69], a portable and extensible open-source platform that allows for the automated deployment and management of containerised workloads through declarative configuration. Many open-source serverless platforms use it to orchestrate and manage function Pods, the atomic deployable units within Kubernetes [9, 69].

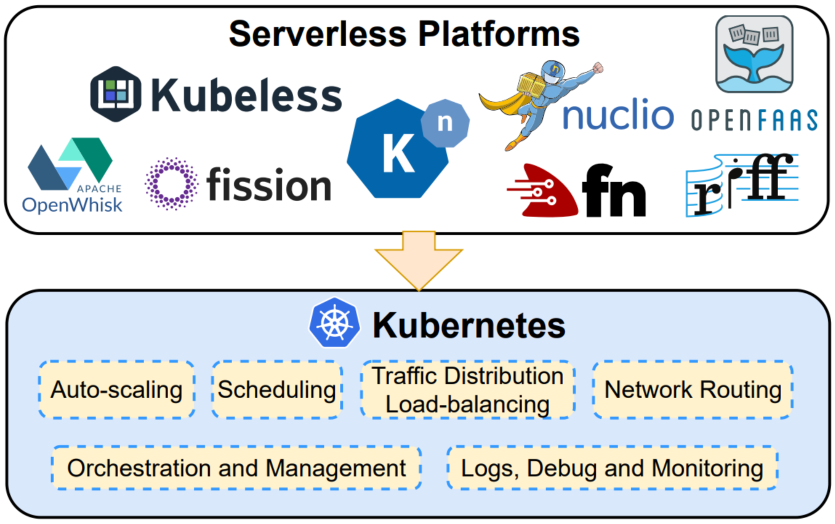


Figure 1.5: Open-source serverless frameworks and underlying Kubernetes services. Picture from [40].

The services in the blue box are necessary for configuration management, service discovery, auto-scaling, Pod scheduling, traffic load balancing, network routing, and service roll-out and roll-back.

1.3 Web Crawler

The World Wide Web (WWW) provides abundant data in the form of HTML documents. This data does not have a well-defined structure and, if site owners do not make APIs available, web scraping or web crawling are the fastest and most effective methods for collecting information [37, 59].

It is necessary to distinguish between the two processes:

- *Web crawling*: given a seed URL it downloads that web page and extracts all hyperlinks. They are needed to continue the crawling process and each visited web page is indexed to a search engine for future retrieves [61];
- *Web scraping*: it enables the extraction of unstructured data from websites and transforms it into structured data, suitable for storage in a database and further analysis [37].

Usually, they are used together to discover new web pages and extract information from them.

1.3.1 Web Crawling Strategies

According to requirements, there are different strategies for implementing a web crawler. In [19, 61], these strategies are described and highlights are provided for each of them:

- *General Purpose Crawling*: a web crawler fetches pages from a set of URLs and links. It can slow down the network speed as it fetches all pages;
- *Focused Crawling*: a focused crawler collects documents on a specific topic, reducing network traffic. It selectively looks for pages relevant to a predefined set of matters, leading to significant savings in resources;
- *Incremental Crawling*: an incremental crawler frequently refreshes the existing collection of pages based upon the estimate as to how often pages change. It replaces less important pages with new ones that are more important, conserving network bandwidth and enriching data;
- *Distributed Crawling*: a distributed crawler uses multiple processes to crawl and download web pages. It allows scalability based on demand;
- *Parallel Crawling*: a parallel crawler uses multiple processes in parallel to maximize the download rate, minimize parallelization overhead, and avoid repeated downloads.

All these strategies can be performed using both libraries that send HTTP requests (e.g. curl, wget, etc.) or full-featured web browser (e.g. Chrome, Firefox, etc.) that interacts directly with the DOM.

1.3.2 Legal Concerns

The process of automatically extracting data from websites has become increasingly popular in both corporate and academic research projects. Web crawling and web scraping have become more accessible to perform thanks to the development of various tools and technologies. However, it is crucial to consider the legal and ethical implications of using these tools for data collection, which are rarely respected.

In "Web Scraping or Web Crawling: State of Art, Techniques, Approaches and Application" [37], a review of the legal literature is conducted, as well as the literature on ethics and privacy, to identify areas of concern and specific issues that scholars and practitioners should address. Reflecting on these issues and concerns can help researchers reduce the risk of ethical and legal conflicts.

The legality of web crawling and scraping is still a developing area, and courts are only now beginning to handle disputes that arise from them for analytic purposes. Additionally, deciding whether web crawling or scraping for analytics raises legal issues is a highly specific determination that depends on the facts of each case. Other matters concerning the extraction of data from websites that should be addressed are described below:

- It is essential to understand the language used in the service agreement or terms of use before using a website. It is important to check whether the terms allow automated access to the website, the usage of any data collected through such means, and the use of the website for purposes other than non-commercial and personal use;
- To use a website, users must agree to its terms. The enforceability of these terms depends on how they are presented, either through a click-wrap agreement or a prominent link to the terms-of-use page on each website;
- To prevent unauthorized web scraping and establish crawl rates, the robots exclusion standard protocol should be used;
- If the website content data is protected by copyright or not;
- If the website owner intends to allow or license the use of the content.

Given the rapid technological advancements in web scraping and crawling, website owners and users of these technologies must stay up-to-date regarding the law in this area.

1.4 Objective of the Thesis

The primary purpose of this thesis is to develop a web crawler that indexes HTML documents. The company Kopjra Srl [63], where the author did his internship, was interested in collecting website information that might contain violations of reputation, intellectual or industrial property. One interesting implementation that matches both the company's and the Professor's consensus was the serverless paradigm. For this reason, testing the effectiveness of this paradigm on a web crawler application was decided upon.

Two different serverless platforms were analysed and compared: AWS Lambda [54] and Knative [15]. The first is provided by AWS while the latter is an open-source alternative trusted by various companies like RedHat, Google, VMWare, IBM, etc. Thanks to the horizontal scalability of this paradigm, it should be possible to reduce the execution time of each search, which is split into several concurrent cloud functions.

The third implementation follows the microservices pattern and uses the Kubernetes API [9, 69] to create a Job that deploys a Pod for each new search. In this case, the cloud function is represented by the Container inside the Pod resource and manages the web crawling from start to end.

Following these implementations, the author exported metrics from each platform to facilitate performance comparison.

1.4.1 Contributions

The thesis outlines its main contributions in Chapter 4, which are summarised as follows:

- Three web crawlers implementations that support both browser automation and HTTP requests;
- The deployment of an Elasticsearch [21, 23] cluster and the configuration of an index for the ingestion of web pages to enable keyword searches;
- The implementation of backend and frontend services enabling search management (i.e. creation, deletion, visualization, query);
- The definition of a test suite to compare the two serverless platforms and the implementation exploiting Kubernetes [9, 69] resources.

The latest implementation allows greater flexibility than the others, so the additional feature to perform web crawling on The Onion Router (TOR) network [32] has only been developed in this one.

1.5 Structure of the Thesis

The first chapter was necessary to introduce serverless computing and web crawling, the two central topics of this thesis.

Chapter 2 illustrates the technologies required to realise the application at both the architectural and development levels.

Chapter 3 presents a review of web crawling applications implemented using the serverless paradigm, referring mainly to AWS Lambda [54] and Knative [15] platforms.

Chapter 4 presents the contributions of this thesis: the three proposed implementations will be analysed in detail, as well as the document ingestion, backend, and frontend services.

Chapter 5 illustrates the design of the test suite for each implementation and explains which metrics were collected. These metrics made it possible to compare results and draw performance conclusions.

The final chapter includes the conclusions of the thesis and potential future works, aiming to add new features and improve the performance side.

Chapter 2

Background

This chapter provides an explanation of the main technologies involved in the thesis. It illustrates the chosen serverless solutions, the NodeJS library for headless browser automation and the platform to perform data ingestion and searching. A note should be made on containerization and orchestration processes performed with Docker and Kubernetes, respectively. Due to the popularity of these two tools, it has been decided to not describe them in this chapter and to consider them as already known.

Section 2.1 describes the serverless computing platform provided by AWS, Section 2.2 analyzes the open-source serverless application layer incubated and supported by CNCF. Section 2.3 explains how the browser automation happens and what can be achieved using the library. Finally, Section 2.4 introduces some history of the tool chosen to perform data ingestion and searching, from its origins to its more technical aspects such as an installation example.

2.1 AWS Lambda

AWS Lambda is a serverless compute service provided by Amazon Web Services which allows code to be executed without provisioning or managing servers. All of the administration of the compute resources, including server and operating system maintenance, capacity provisioning, automatic scaling, and logging is performed on a high-availability compute infrastructure. As a serverless service, Lambda only runs functions when an event occurs with a pay-as-you-go billing model [55]. It integrates seamlessly with other AWS services to invoke functions or perform other actions, exploiting previously configured triggers or event source mapping. Some examples of well-connected services are API Gateway, S3, SQS, SNS, Dynamo DB, etc.

Lambda uses a secure and isolated execution environment that manages the resources required to execute the function, the latter also provides lifecycle support for the function's runtime and any external extensions associated with it. Figure 2.1 and 2.2 show the components of this isolated environment, and illustrate the runtime phases of the function.

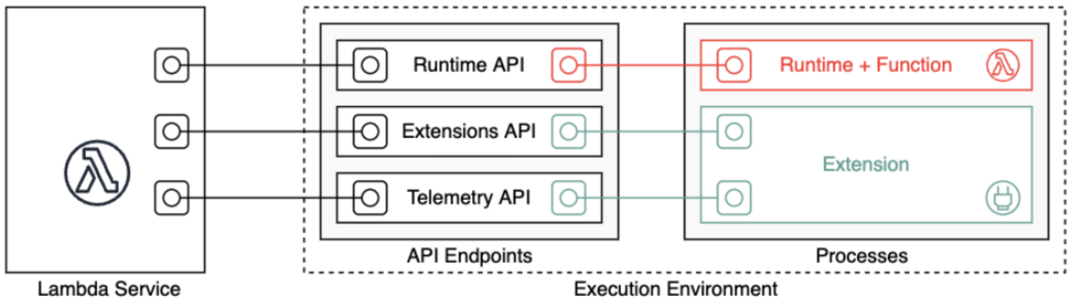


Figure 2.1: The function's runtime communicates with Lambda using the Runtime API. Extensions communicate with Lambda using the Extensions API and they can also receive log messages and other telemetry from the function by using the Telemetry API. Picture from AWS Lambda.

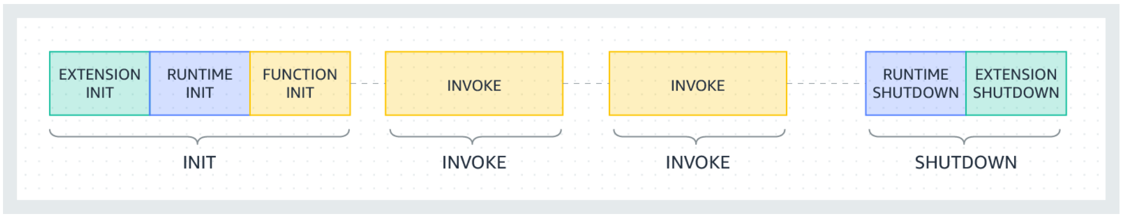


Figure 2.2: The Lambda execution environment lifecycle is composed of three phases: Init, Invoke and Shutdown. During the Init phase, the environment is prepared for the invocation of the Lambda function (download code, init extensions and runtime environments, load code). The Invoke phase occurs when the function is invoked, and the Shutdown phase freezes the execution environment when the runtime and extensions have completed their tasks. Picture from AWS Lambda.

The code downloaded and executed inside a function is composed of scripts or compiled programs with their dependencies. There are two available options that Lambda supports to deploy these files:

- Container images, includes the base operating system, the runtime, Lambda extensions, application code and its dependencies;
- .zip archives, include an application code and its dependencies; it is created by default when a Lambda console or a toolkit is used.

In our case, the Container image was chosen as the deployment package because the dependencies exceeded the size allowed by the zip archive. In addition, using the Container image enables a static vulnerability analysis to be performed on the code.

2.2 Knative

Knative is a layer over Kubernetes that solves common problems of deploying, upgrading and observing software, connecting disparate systems together, routing traffic, and scaling automatically [15].

Its implementation was originally started by Google but is maintained by the community, which includes people from companies like Red Hat, Google, IBM and VMware. As mentioned, this project was accepted by the CNCF at the incubation maturity level on 2 March 2022 and recently, a graduation proposal was made by Knative Steering Committee [38].

In short, it can be defined as a platform-agnostic solution for running serverless deployments or, in more detail with the following quote: "Knative is a developer-focused serverless application layer which is a great complement to the existing Kubernetes application constructs. Knative consists of three components: an HTTP-triggered autoscaling container runtime called Knative Serving, a CloudEvents-over-HTTP asynchronous routing layer called Knative Eventing, and a developer-focused function framework which leverages the Serving and Eventing components, called Knative Functions" [8].

There are several methods to install Knative in a cluster, and each was adapted to specific use cases and implementation scenarios.

- quickstart plugin: an easy-to-use plugin in which you can specify Minikube or Kind[1] to create a local Kubernetes cluster, which includes the installation of a simplified version of Knative, only for development purposes;
- YAML-based installation: a more comprehensive option for deploying Knative in production environments, which involves applying YAML files using `kubectl` CLI to establish Knative's main components and extensions;
- operator: an automated and managed installation process for the Knative components in a production environment.

In our case, the `quickstart` plugin was used during the development, and the YAML-based installation was chosen in the production environment.

At this point, it is possible to interact with the cluster and Knative components in a classical way using `kubectl`. Another method is to use `kn`[13] CLI, which provides a simple and fast interface for creating Knative resources and simplifies completing otherwise complex procedures such as autoscaling and traffic splitting.

2.2.1 Serving

Knative Serving defines a set of objects, such as Kubernetes Custom Resource Definitions, to control the behaviour of serverless workloads on the cluster. Figure 2.3 shows how these resources interact with each other, providing many simplifications for the user, when managing deployment or maintenance. A description of each is given below:

[1]Minikube and Kind are two tools that allow the creation of Kubernetes clusters locally, so you can have a local development environment. See https://kubernetes.io/docs/tasks/tools/ for more details.

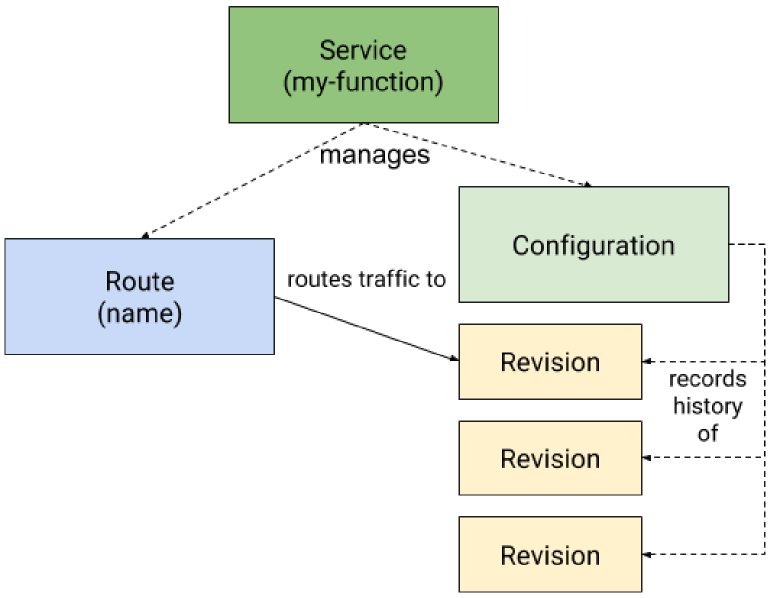


Figure 2.3: Diagram showing the primary resources of Knative Serving API and how they interact with each other. Picture from Knative.

`Service` Automatically manages the entire workload lifecycle and checks if all the necessary resources (route, configuration, revision) are up and running without problems. It is possible to define to which revision the traffic should be directed.

`Route` Maps a network endpoint to one or more revisions, making it possible to split traffic and create named routes.

`Configuration` Maintains the desired state of the deployment and provides a clean separation between code and configuration, following the Twelve-Factor App[2] methodology. When a configuration is changed, a new revision is automatically created.

`Revision` Represents an immutable snapshot of the application code and configuration. It can be scaled up and down automatically according to incoming traffic and enables progressive roll-out and roll-back of application changes. If idle for a certain period of time, it is automatically cleaned up by garbage collection, thus freeing cluster resources.

The logical components involved in creating a Knative Service were presented and illustrated. Figure 2.4 is intended to describe all components of the architecture that manage the serverless workflow.

[2]The Twelve-Factor App methodology is a set of best practices that can help in building modern, cloud-native applications that are scalable, reliable, and maintainable. See https://12factor.net/ for more details.

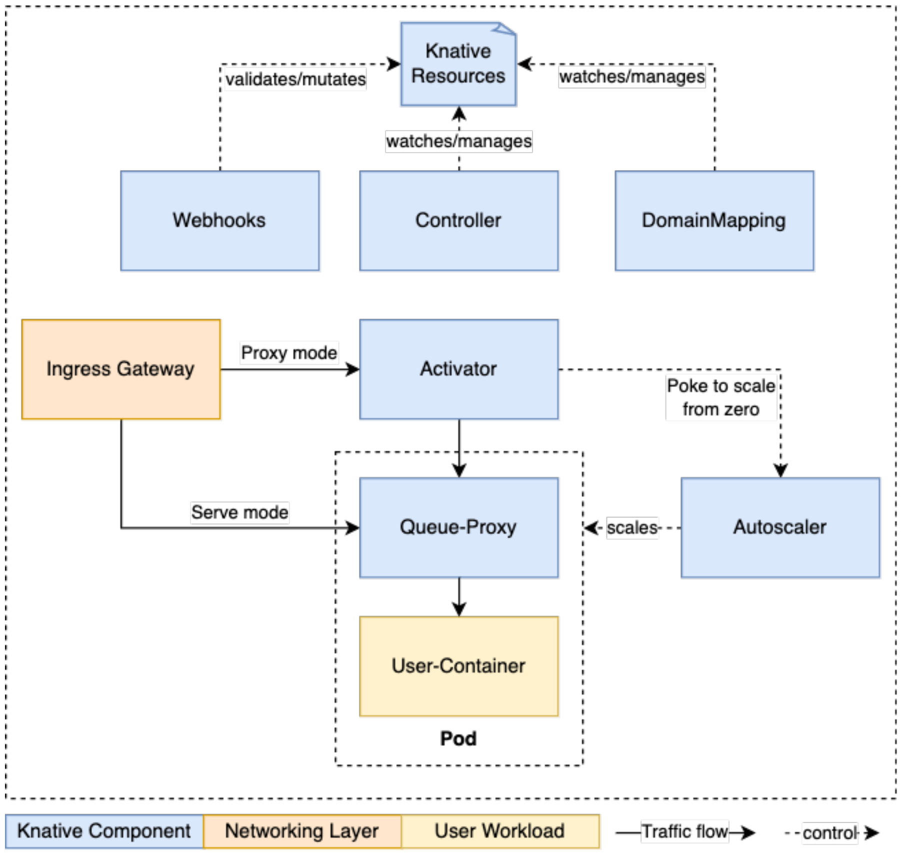


Figure 2.4: High-level architecture components of Knative Serving. Picture from Knative.

`Activator` Is responsible for queueing incoming requests when a Knative Service is scaled to zero. It communicates directly with the autoscaler to reactivate services scaled to zero and forward queued requests. It can also act as a request buffer to handle traffic bursts.

`Autoscaler` Receives the request metrics and enables a suitable number of Pods for the correct handling of this load.

`Controller` Watches the state of Knative resources within the cluster, manages the lifecycle of dependent resources, and updates the resource state.

`Queue-Proxy` Is a side-car Container in front of the user-container, collecting metrics and enforcing the desired concurrency when forwarding requests. When necessary, it can also act as a queue, similar to the `Activator`.

`Webhooks` They are responsible for the validation and mutation of Knative Resources.

A note must be made about the `Ingress` component. It does not refer to the Kubernetes Ingress Resource but to the concept of exposing external access to a resource on the cluster. This is possible thanks to the `Ingress` resource and the three layers of network abstraction available and supported by the community: Kourier, Contour and Istio.

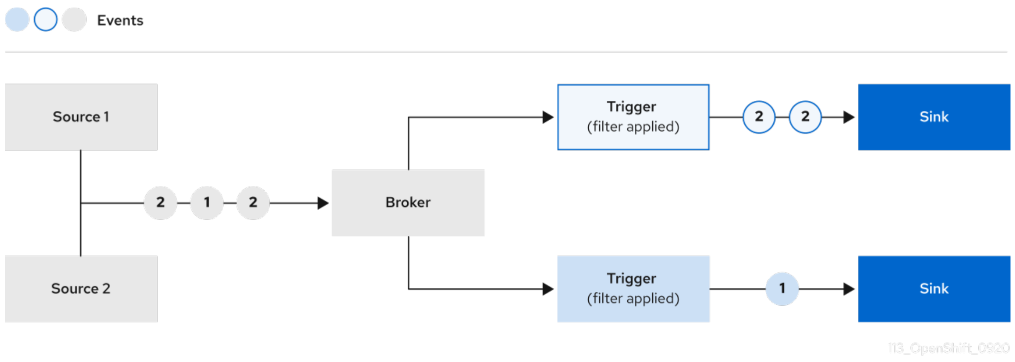
Knative Pod Autoscaler (KPA)

The Knative Serving module provides an automatic scaling of revisions to meet incoming demand. By default, the KPA is active and supports different metrics such as concurrency, requests-per-second, CPU and memory. In particular, the concurrency metric allows you to specify a soft constraint and a hard constraint, so you have more control over the number of running Pods based on the use case.

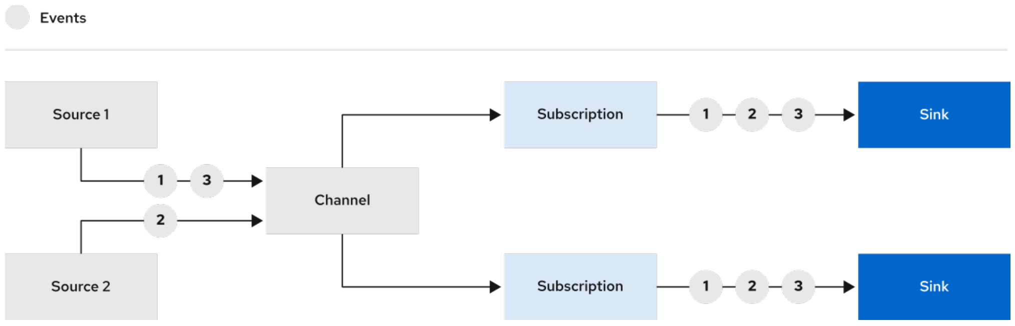
The Horizontal Pod Autoscaler (HPA) provided by Kubernetes is not part of the Knative Serving core. If you wish to use it, you must add it as an extension after installing Knative Serving.

2.2.2 Eventing

Knative Eventing is a collection of APIs that enables developers to easily follow an event-driven architecture with their applications. The Eventing module components allow events to be routed from producers (referred to as `Sources`) to consumers (referred to as `Sinks`) without worrying about the event format because it is consistent with the CloudEvents specification, a useful CNCF project to standardize communication, which will be described below.



(a) Event producers send events to a `Broker` by POSTing the event, then the `Broker` uses `Triggers` for event delivery.



(b) Event producers exploit `Channel` to fan out received events via `Subscriptions`.

Figure 2.5: Examples of event-driven workflow using different architectures and resources. Pictures from Knative (a)(b).

Diagrams of possible applications using Knative Eventing can be seen in Figure 2.5. The main components are described below.

`Sources` Any Kubernetes object that generates or imports an event and transmits it to another endpoint on the cluster via CloudEvents. The resource that receives the event is an `Addressable` (e.g. `Sink`, `Broker`, etc.). Only a few of the various types available are listed:

- *APIServerSource*, produces a new event each time a Kubernetes resource is created, updated or deleted, bringing the Kubernetes API into Knative;
- *PinSource*, allows events production with fixed payload based on a specified Cron[3] schedule;
- *GitHub*, produces a new event for selected GitHub event types, bringing GitHub events into Knative;
- *RabbitMQ*, brings RabbitMQ messages into Knative.

`Broker` Accumulates a pool of events defining an event mesh. It uses `Triggers` for event delivery and manages the delivery failures.

`Triggers` When an event is taken into account by the `Broker`, it can be forwarded to subscribers using this resource. They allow events to be filtered by attributes so that events with particular attributes can be sent to `Subscribers` that have registered interest in events with those attributes.

`Subscribers` Represents any URL or `Addressable` resources. They can also reply to an active request from the `Broker` and can respond with a new event that is sent back to the `Broker`.

`Channel` Defines a single event forwarding and persistence layer. There are various types available:

- *InMemoryChannel*, the best effort was provided using an in-memory channel, not suitable for a production environment;
- *KafkaChannel*, backed by Apache Kafka topics;
- *NatssChannel*, backed by NATSS Streaming.

`Sinks` It is an `Addressable` or a `Callable` resource that can receive incoming events from other resources.

- `Addressable` Receives and acknowledges an event delivered over HTTP to an address defined in their `status.address.url` field. As a special case, the core Kubernetes Service object also fulfils the `Addressable` interface.
- `Callable` Receives an event sent via HTTP and processes it, returning 0 or 1 new event in the HTTP response. This returned event can be further processed in the same way as events from an external event source.

All these resources make it possible to achieve a mixed infrastructure, with different `Sources` producing events which are processed and forwarded, as shown in Figure 2.6.

[3]Cron is a daemon software that runs continuously in the background and wakes up to handle periodic service requests when required. See https://en.wikipedia.org/wiki/Cron for more details.

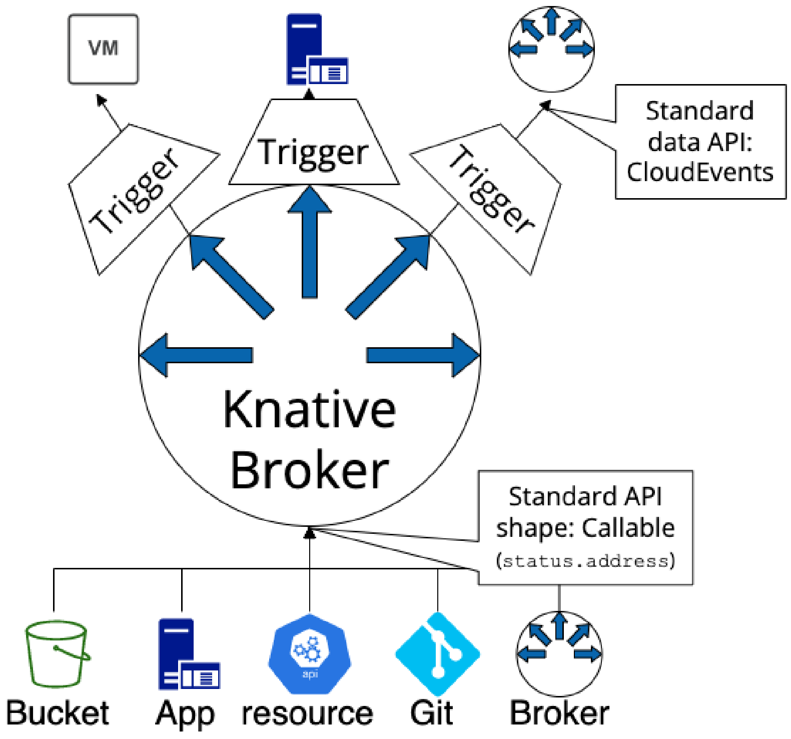


Figure 2.6: Diagram showing event mesh mechanism, defined by `Broker` and `Triggers` APIs. Picture from Knative.

CloudEvents

Nowadays, there isn't a common way of describing events; developers have to write new event-handling logic for each event source. This lack of a common event format also results in the absence of uniform libraries, tools, and infrastructure for transmitting event data across different environments.

CloudEvents [12] is a specification for describing event data in common formats in order to provide interoperability across services, platforms and systems and wants to fill this gap by offering SDKs for various programming languages (e.g. Go, JavaScript, Java, C#, Ruby, PHP, PowerShell, Rust, and Python). These libraries can be used to build event routes, tracing systems, and other related tools, thus addressing the challenges posed by the absence of a common event format.

The Listing 2.1 shows a CloudEvent example, where `id`, `source`, `specversion` and `type` are required fields; meanwhile, the others are optional[4].

```
{
    "specversion" : "1.0",
    "type" : "com.example.type",
    "source" : "/example/source",
```

[4]See https://github.com/cloudevents/spec/blob/v1.0.2/cloudevents/spec.md for more details.

```
    "id" : "82C32673-0C78",
    "time" : "2020-04-10T01:00:05+00:00",
    "datacontenttype" : "application/json",
    "data" : {
        "foo": "foo"
    }
}
```

Listing 2.1: Example of a CloudEvent in JSON format, with all necessary fields and the payload, defined by the `data` object.

2.2.3 Function

Knative Function offers a simple programming model for using functions on Knative, eliminating the need for in-depth knowledge of Knative, Kubernetes, Containers or Dockerfiles. Developers can easily create, build, and deploy stateless, event-driven functions as Knative Services by using the `func` [13] CLI.

When a function is built or executed, it automatically generates a Container image in Open Container Initiative (OCI) [26] format, which is stored in a container registry[5]. Subsequently, each time the code is updated and the function is executed or deployed, the Container image is updated to reflect the changes. It also simplifies the project creation by providing templates for different languages (e.g. Python, Go, Java, TypeScript, etc.) and invocation formats (HTTP or CloudEvent).

The Listing 2.2 shows how to install the `func` plugin and how the creation, execution and deployment of a function example works.

```bash
#!/bin/bash

# Install func plugin
brew tap knative-extensions/kn-plugins
brew install func

# Create hello function
func create --language go --template cloudevents hello
cd hello

# Build and push
func build --registry <registry> --push

# Deploy on the current context
func deploy
```

Listing 2.2: An example of how to install the `func` plugin and use it to create, build, push and deploy a simple function.

[5]A container registry is a repository or collection of repositories, used to store and access container images. See https://www.redhat.com/en/topics/cloud-native-apps/what-is-a-container-registry for more details.

2.3 Puppeteer

Puppeteer is a Node library which provides a high-level API to control headless Chrome or Chromium over the DevTools Protocol [43, 65], which enables instrumentation, inspection, debugging, and profiling in some Blink-based[6] browsers. The instrumentation is categorized into different domains (e.g., DOM, Debugger, Network, etc.), each defining supported commands and generated events, both of which are serialized as fixed-structure JSON objects.

The headless attribute specifies a way to run the Chrome browser in a headless environment, essentially without its graphical user interface. All the modern web platform features provided by Chromium and the Blink rendering engine are available by CLI and are useful for different web automation tasks, while also reducing resource usage. In December 2023, Chrome's Headless mode was updated due to the old implementation that was separated from the Headfull one and shipped as part of the same Chrome binary [39]. It was challenging to keep both implementations up-to-date, each with its own bugs and features; for these reasons, as shown in Figure 2.7, Chrome developers unified Headless and Headfull browsers in one code-base only.

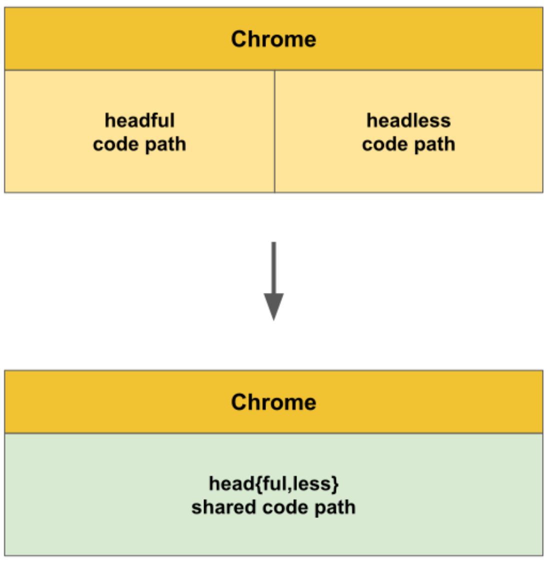


Figure 2.7: Schema of the new Headless Chrome implementation. The old Headless Chrome was unified with the Headfull one. Picture from Chrome for developers.

Puppeteer was quickly updated to support this new Headless mode and in addition to the features it introduced, there are a lot of actions that can be performed using this library, for example:

- generate screenshots and PDFs of pages;
- crawl a Single-page Application and generate pre-rendered content;

[6]Blink is a browser engine forked from the WebCore component of WebKit. See https://www.chromium.org/blink/ for more details.

- automate actions such as form submission, UI testing and keyboard input;
- create an up-to-date, automated testing environment to directly run tests in the latest version of Chrome using the latest JavaScript and browser features;
- capture a timeline trace of your site to help diagnose performance issues;
- intercept and manipulate network requests;
- test Chrome extensions.

The Chrome Browser Automation team oversees maintenance, but being open-source actively encourages and welcomes community support and contributions. Users can choose which of the two available packages to install: `puppeteer` or `puppeteer-core`. The first includes, by default, the download of a recent version of Chrome for Testing[7] while the latter doesn't.

Figure 2.8 shows an overview of the library architecture with all the basic components that make the operations described above possible.

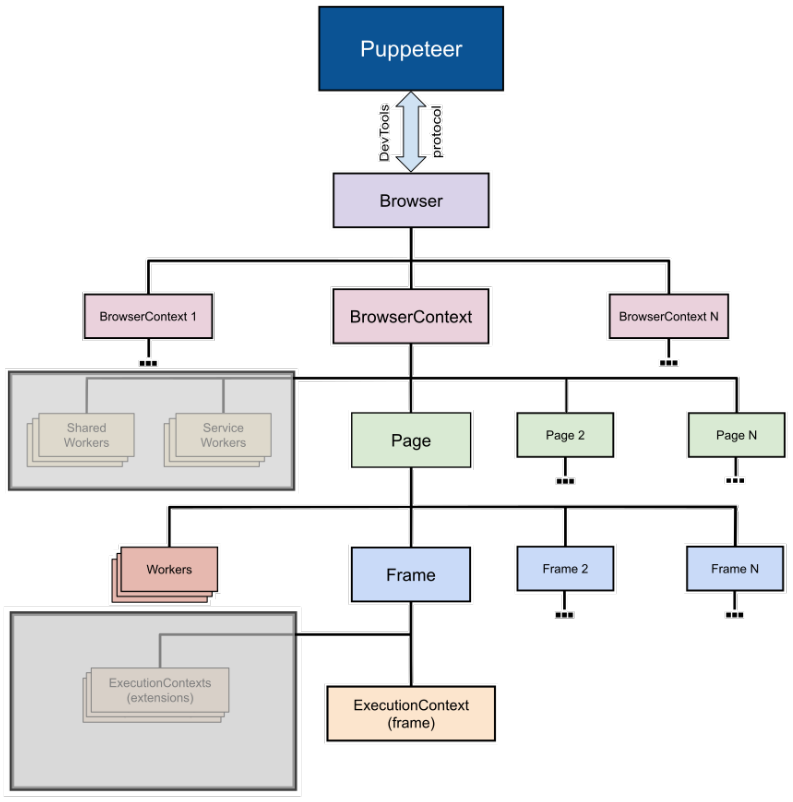


Figure 2.8: Overview of the Puppeteer library architecture. Picture from GitHub.

[7]Chrome for Testing has been created purely for browser automation and testing purposes and is not suitable for daily browsing. See https://developer.chrome.com/blog/chrome-for-testing/ for more details.

2.3.1 Stealth plugin

Although Puppeteer is a robust library, it has limitations regarding extensibility and customization. For these reasons, a German developer named berstend on GitHub created a modular plugin framework built on top of Puppeteer called Puppeteer-Extra [5].

One of the most widely used is `puppeteer-extra-plugin-stealth` [6], which deals with applying various evasion techniques to make the detection of puppeteers harder. It's probably impossible to prevent all ways to detect Headless Chromium, but it should be possible to make it so complicated that it becomes cost-prohibitive or triggers too many false positives to be feasible. Other relevant plugins are:

- `puppeteer-extra-plugin-recaptcha`, solves reCAPTCHAs[8] automatically, using a single line of code;
- `puppeteer-extra-plugin-adblocker`, very fast and efficient blocker for ads and trackers that reduces bandwidth and load times;
- `puppeteer-extra-plugin-anonymize-ua`, anonymizes the user-agent on all pages using dynamic replacing, so the browser version stays intact and recent.

2.4 Elasticsearch

Elasticsearch is a distributed search and analysis engine at the heart of Elastic Stack. Together with Kibana, Beats and Logstash, they form the ELK Stack: a product that allows reliable and secure data to be taken from any source and any format, with the aim of searching, analyzing and visualizing it.

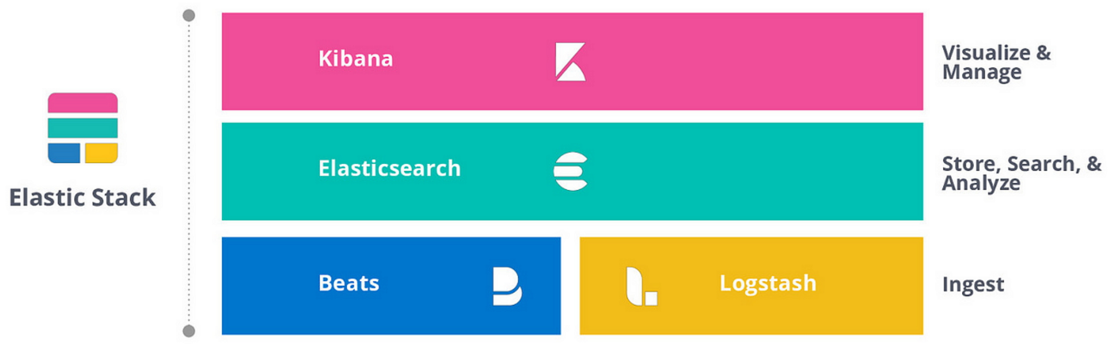


Figure 2.9: Overview of the Elastic Stack and its components. Picture from Medium.

As briefly illustrated in Figure 2.9, each module has its own purpose and can be used separately. In particular, the Elasticsearch tool is built on top of Apache Lucene, a historical library with the same objective. It was developed due to the complexity of Lucene and it hides the hard Java integration behind a simple and coherent RESTful API [71].

[8]reCAPTCHA is a free service from Google that helps protect websites from spam and abuse. See https://www.google.com/recaptcha/about/ for more details.

Since its first release, it has been distributed with open-source Apache License 2.0. However, in January 2021, with the release of Elasticsearch 7.11, the Elastic company decided to change the licence of Elasticsearch and Kibana to a dual, not open-source license: Elastic License and Server Side Public License (SSPL). This choice was made to prevent Amazon, and other companies, from providing Elasticsearch and Kibana as a service without collaborating with Elastic [4, 24].

To understand the functionality of Elasticsearch better and to discover its benefits, it is useful to describe what an index is. The index concept in Elasticsearch can be compared to an optimized collection of documents, each document is a compilation of fields represented as key-value pairs. By default, Elasticsearch indexes all data in each field and each indexed field has a dedicated and optimized data structure (e.g. text fields are stored in inverted indices, numeric and geoinformation fields are stored in BKD trees, etc.). This approach enables the efficient and specialized handling of different types of data, contributing to the platform's effectiveness in managing and retrieving various forms of information in near real-time [20]. When the document is actually stored, searches can be directly performed using the REST API or the Elasticsearch client available in different programming languages; both support structured queries, full-text queries and complex queries combining the two.

In addition to the index concept, it is essential to understand the fundamental components that make up the Elasticsearch backend. These components are depicted in Figure 2.10 and include:

- a cluster, defined as a group of one or more node instances that are connected together;

- one or more nodes, each representing a single server that stores data and participates in the cluster's indexing and search capabilities;

- shards, each representing a subset of the data stored in an index, are extremely useful for distributing the workload across multiple nodes;

- documents, as described above, are the basic unit of information and consist of a JSON object with key-value pairs that represent the data to be stored and indexed.

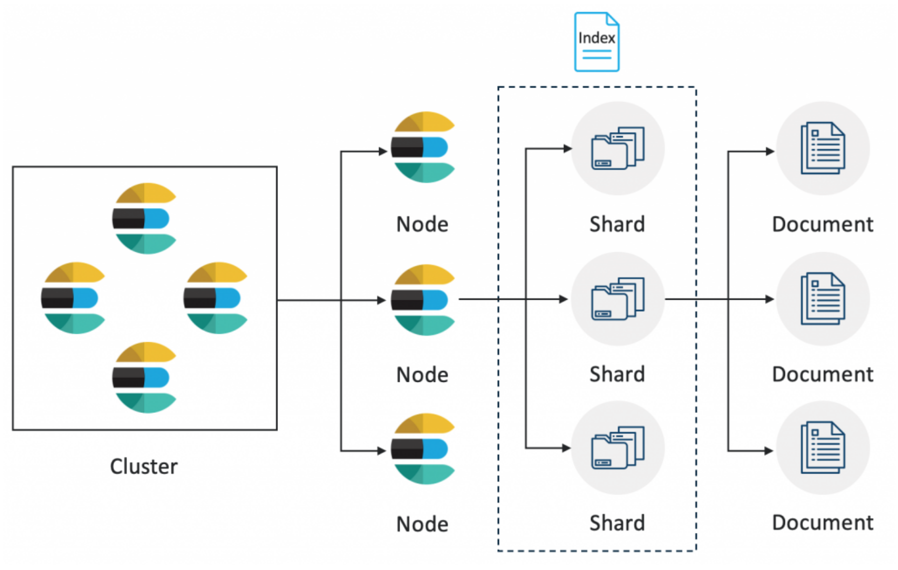


Figure 2.10: Architecture of an Elasticsearch installation, with all key components. Picture from Miracle.

It is important to specify that there are different types of nodes, such as master nodes, data nodes, ingest nodes and others, each with a specific role in the cluster. Furthermore, it is necessary to have at least one master node and three master-eligible nodes to keep the cluster healthy and enable the platform to operate properly, even in the event of failures.

2.4.1 Elastic Cloud on Kubernetes

There are different ways to install Elasticsearch. You can use the hosted service, manually install it, use package managers, run it on Containers with Docker, deploy it with Helm charts, or use Elastic Cloud on Kubernetes (ECK). The latter, in our scenario, is the most interesting way.

In May 2019, Elastic announced this new orchestration product based on the Kubernetes Operator pattern[9] that allows users to make available, manage and operate Elasticsearch clusters on Kubernetes [22]. It was called ECK and focuses not only on simplifying the task of deploying Elasticsearch and Kibana but also on streamlining critical operations such as managing and monitoring multiple clusters, upgrading to new stack versions, scaling cluster capacity, changing cluster configuration, dynamically scaling local storage, and scheduling backups.

A step-by-step guide for installing the ECK operator and deploying an Elasticsearch cluster is given in Listing 2.3 and 2.4.

```
#!/bin/bash
ECK="https://download.elastic.co/downloads/eck/2.11.0"
```

[9]Operators are software extensions to Kubernetes that make use of custom resources to manage applications and their components following Kubernetes principles. See https://kubernetes.io/docs/concepts/extend-kubernetes/operator/ for more details.

```
# Install custom resource definitions (CRDs)
kubectl create -f "${ECK}/crds.yaml"

# Install the operator with RBAC rules
kubectl apply -f "${ECK}/operator.yaml"
```

Listing 2.3: Install ECK operator with its CRDs and RBAC rules.

```
apiVersion: elasticsearch.k8s.elastic.co/v1
kind: Elasticsearch
metadata:
  name: quickstart
spec:
  version: 8.12.0
  nodeSets:
  - name: default
    count: 1
    config:
      node.store.allow_mmap: false
```

Listing 2.4: Elasticsearch cluster specification to deploy one Elasticsearch node.

Chapter 3

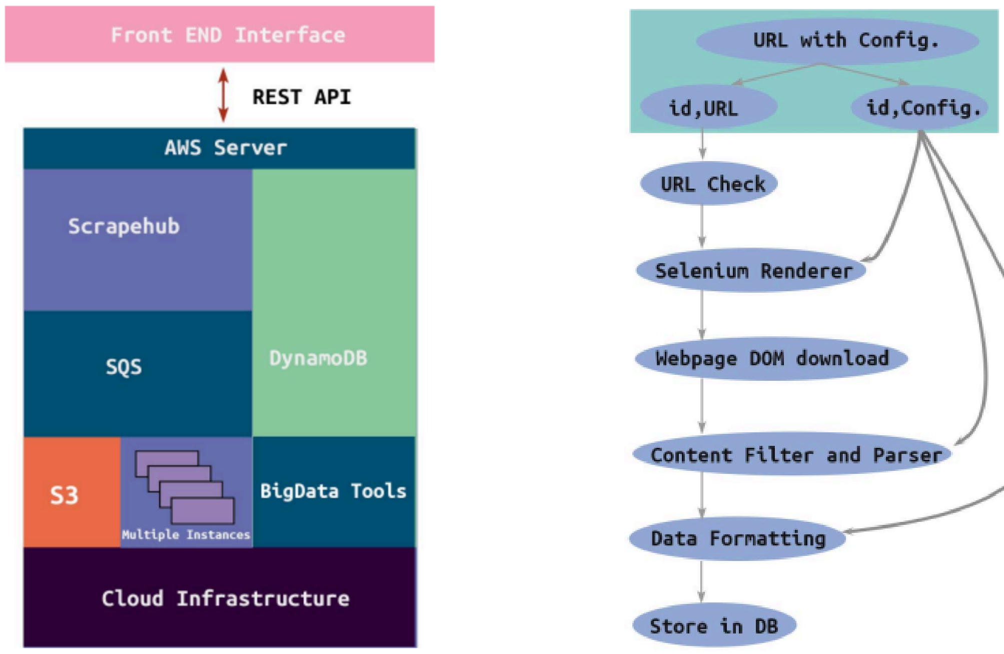
Literature Review

Serverless frameworks and the web crawling process are continuously growing areas of research. The latter is a more mature field: the first web crawlers appeared in 1994 [49], and they have taken advantage of the latest technologies. On the other hand, the serverless paradigm is a relatively younger technology: the first serverless platform was introduced at the end of 2014 by AWS and it was called AWS Lambda [58]. Both fields are characterized by many publications documenting issues, challenges and innovations. This section will describe some examples of publications about these two topics.

Hongfei Yan et al. designed and evaluated an efficient web crawling system [41] where they implemented a fully distributed web crawler, a URL allocation algorithm and a method to ensure the scalability of the system. It includes a WebGather Server Registry (WSR) that manages several main controllers; each main controller coordinates its own collector, which performs the crawling process. To improve communication performance, only URLs are shared. Another similar system is shown in [67], where the authors implemented a scalable and fully distributed web crawler using the Java programming language. Thanks to this choice, the software can be platform-independent. It consists of multiple identically programmed agents that communicate with each other to crawl the web, ensuring dynamic and decentralized coordination between them. To avoid overloading individual agents, the amount of URL to be handled by each of them is equally distributed.

Two cloud-based web crawlers are discussed in [72] and [59], where services provided by Azure and AWS are used respectively. The first exploits the MapReduce programming paradigm and uses distributed agents, each of which stores what it finds on Azure Tables or Blobs[1]. The second makes use of various AWS services such as EC2, Dynamo DB, SQS and S3 to perform on-demand web scraping for big data applications. To understand better how this last solution works, the architecture is shown in Figure 3.1.

[1]Azure Table is a NoSQL store of key-values for rapid development while Azure Blob allows massive storage of unstructured data. Both are storage services provided by Azure. See https://azure.microsoft.com/en-us/products/storage/tables and https://azure.microsoft.com/en-us/products/storage/blobs for more details.



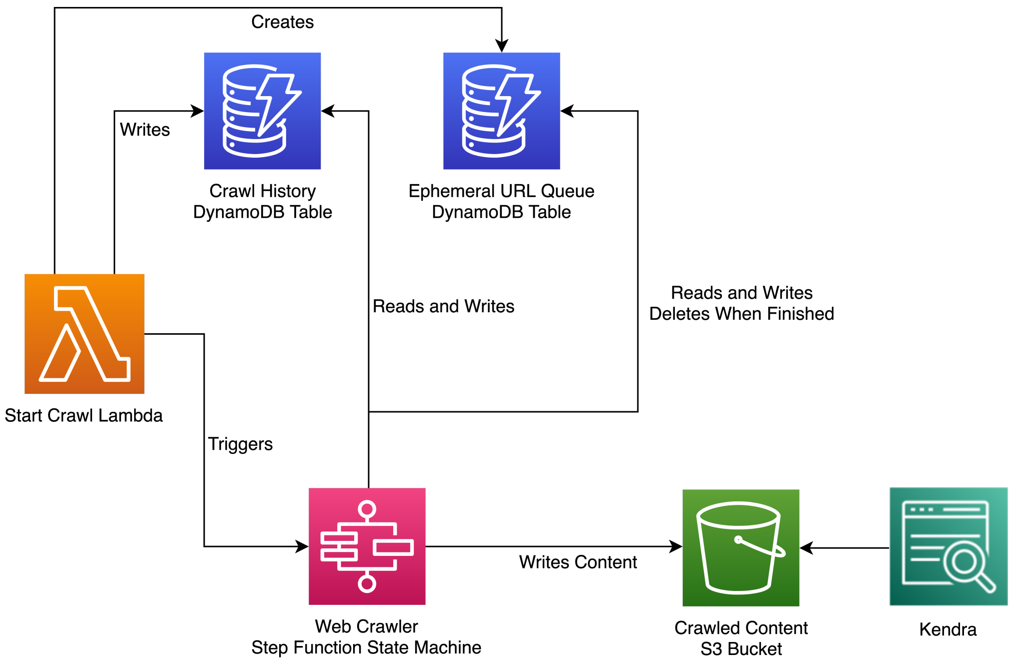
(a) Ecosystem. **(b)** Engine.

Figure 3.1: Diagrams show the architecture and logic workflow of a cloud-based web scraper. Users request the scraping service through a UI, providing URLs and configurations. These are stored in a database in JSON format, and a request is generated in SQS by Scrapehub. The engine reads the URLs and configuration from the JSON file, checks the validity of the URLs, renders the valid ones using a Selenium Renderer, and stores the extracted content in the database after parsing and filtering. Pictures from "Cloud Based Web Scraping for Big Data Applications" [59].

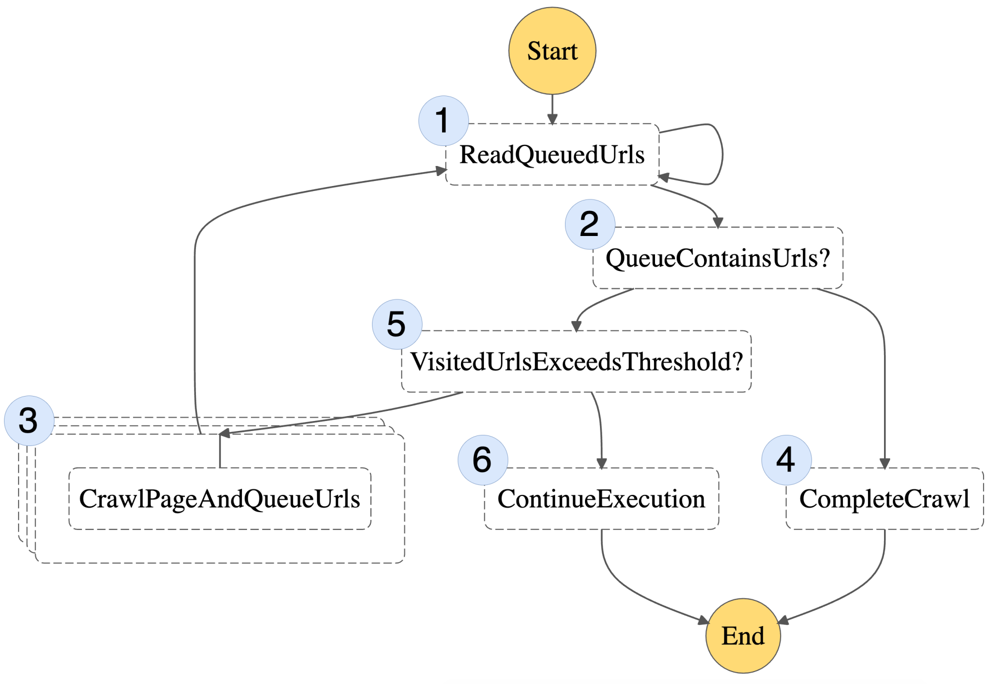
Other solutions that aim to implement a web crawler/scraper exploiting AWS Lambda serverless service are illustrated in [45] and [64]. In the first, some considerations are made regarding the different implementations that can be put in place to implement a web scraper, from an on-premises-like solution to a serverless solution.

- Use an EC2 instance to have full control of the infrastructure. It requires manual operations, such as setting up the environment, completing security tasks and monitoring the health status over time;
- Containerise the application and deploy it on Elastic Container Service. The biggest advantage of this is the platform independence;
- Use a Lambda serverless service, which allows a very lean infrastructure to be created on demand and scales continuously, with a generous free monthly tier. The main constraint of this is that the execution time of each individual function is limited to 15 minutes.

In the second, a serverless web crawler with a search engine was developed not only using AWS Lambda but also Dynamo DB, Step Functions, S3 and Kendra services. Figure 3.2 shows how these components interact, making it possible to run several crawlers simultaneously that store the title text and HTML text of the processed pages, with the aim of performing keyword searches on Kendra.


Creates
Writes
Crawl History DynamoDB Table
Ephemeral URL Queue DynamoDB Table
Reads and Writes
Reads and Writes Deletes When Finished
Start Crawl Lambda
Triggers
Writes Content
Web Crawler Step Function State Machine
Crawled Content S3 Bucket
Kendra

(a) Architecture.



(b) State machine defining web crawler algorithm.

Figure 3.2: Diagrams show the architecture and logic workflow of a serverless web crawler with a search engine. It is important to note in (b) how each step can be executed in a Lambda function dealing with a specific task so as to bypass the Lambda timeout of 15 minutes. Pictures from *Scaling up a Serverless Web Crawler and Search Engine* [64].

Several articles from the literature were illustrated, starting with classical web crawler solutions and ending with serverless web crawlers. Looking at Knative, there is no implementation of this type of application; the few articles mainly deal with modifying the KPA autoscaling algorithm and reducing the cold start time [27, 31].

Chapter 4

Methodology

This chapter introduces the main contributions of the thesis, first describing the components that constitute the solution, common to all implementations. Since the final objective of this work is to evaluate three different web crawler implementations and to understand whether it is appropriate to use the serverless paradigm in this use case, the chapter is divided into two main sections:

- Section 4.1 describes the main components common to all implementations, it analyses the operations performed by each component and illustrates the technology stack of our applications;
- Section 4.2 comprises three subsections, each of which aims to describe in depth the underlying infrastructure of each implementation; in addition, the changes made to the code bases are explained.

4.1 System Architecture

As mentioned earlier, it is necessary to describe the architecture used at a high level, which underlies the three web crawling applications, in order to understand how the framework works. Figure 4.1 illustrates the essential modules and the two types of memorization used to store search information and HTML documents.

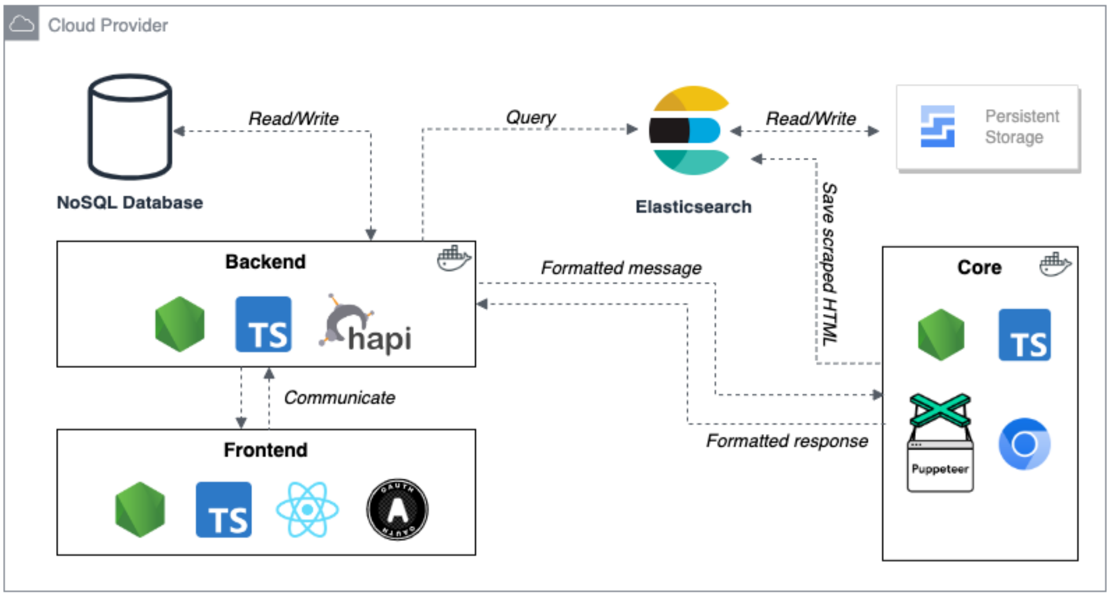


Figure 4.1: High-level architecture common to all web crawler implementations. Each component is shown, and the technology stack used is specified for our modules.

The frontend module provides a UI for users to interact with. It incorporates Open Authorization (OAuth 2.0) to ensure secure access to the platform and allows logged-in users to perform actions such as creating, analysing and deleting crawl searches. In order to start the web crawling process, users must provide a seed URL for the specific site they wish to explore, specify the number of pages to crawl, name the search with a brief description, and decide whether to use browser automation or HTTP requests.

The backend makes sure that the data entered are valid before creating a new document inherent to the search within the NoSQL database. It also initiates the search by forwarding the seed URL and a unique identifier to the core module.

The latter loads the web page according to the chosen method, scrapes all the links in it, filters and validates them, and extracts the HTML body to load the document into the Elasticsearch index. As a response, it sends the filtered links in batch to the backend, which is responsible for continuing the search or stopping it, based on the current number of visited URLs that is tracked in the database.

4.1.1 Indexing

One key aspect of this application is the ability to index the HTML body of visited web pages, so as to enable keyword searching. In order to create a high-availability cluster suitable for production, the smallest possible configuration comprising two nodes with a tiebreaker was chosen [25]: a particular node that resolves deadlocks or ties when decisions require a majority vote in Quorum-based System. As explained in 2.4, it was decided to deploy the ECK operator to enable easy installation of the Elasticsearch module. Figure 4.2 illustrates some of the resources required for proper operation.

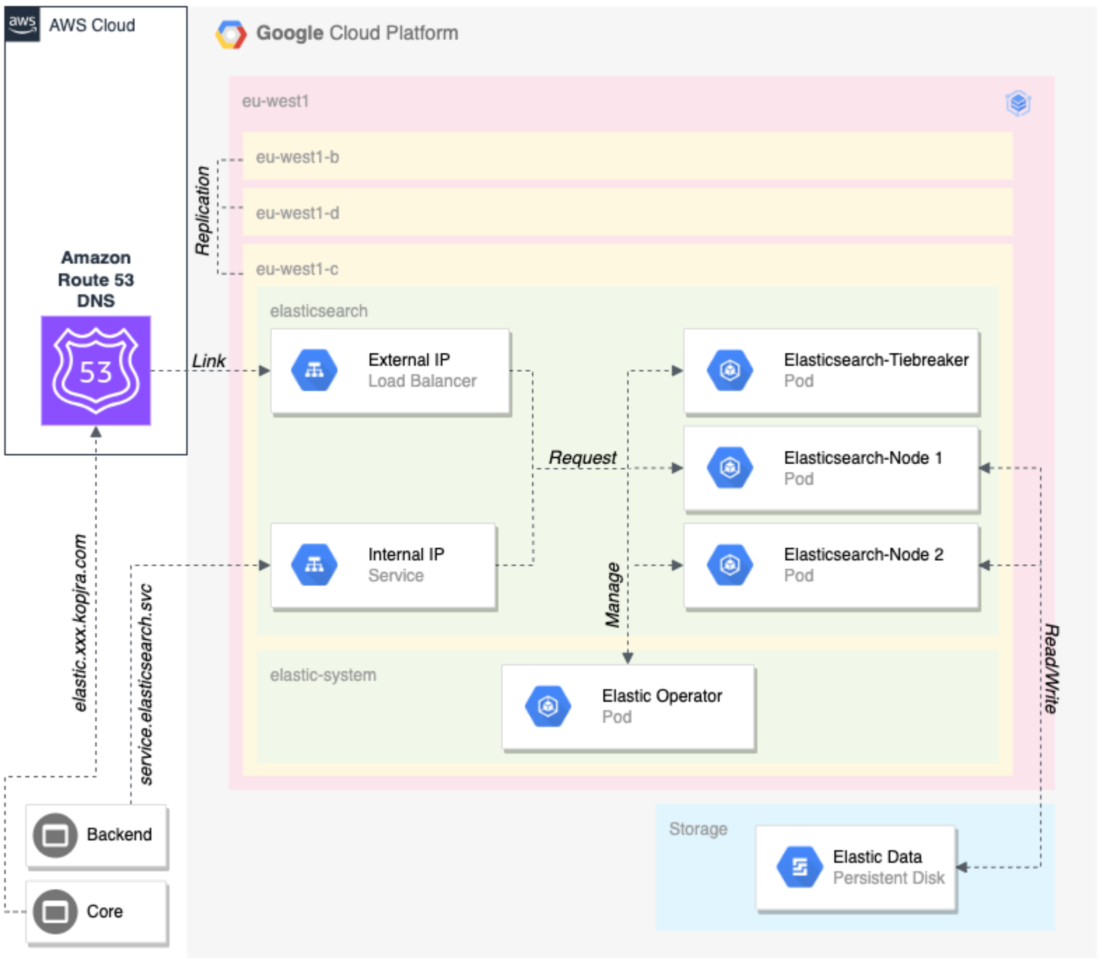


Figure 4.2: The smallest possible Elasticsearch cluster configuration that is suitable for production deployments. It ensures high availability and resiliency to the failure of an individual node.

The three Pods inside the elasticsearch Namespace compose the cluster[1], each node has one or more roles to play during the execution, in particular:

- *Node 1* and *Node 2* have `master` and `data` roles, they interact directly with the persistent storage to index and query the uploaded documents. Since Elasticsearch is a Quorum-based System, it is necessary for there to be multiple `master` or `voting_only` nodes within the cluster;
- *Tiebreaker*, has only `master` and `voting_only` roles, it is necessary to perform election also with one node failure and it doesn't interact with the persistent storage. It intervenes when the leading nodes are evenly divided or cannot reach a consensus due to network problems.

The installation includes an internal Service by default, and with a simple

[1]Any time that you start an instance of Elasticsearch, you are starting a node. A collection of connected nodes is called a cluster. See https://www.elastic.co/guide/en/elasticsearch/reference/current/modules-node.html for more details.

change to the configuration YAML file, an external Service can be added. The latter exposes a virtual IP address that was linked to a DNS name through Amazon Route 53. In both internal and external connections, the Elasticsearch module requires API token authentication.

Once the deployment is done, it is possible to create a new index that will store the scraped HTML body of the visited web pages. Thanks to the text analysis process, Elasticsearch provide a simple way of converting unstructured text into a structured format that is optimized for search. In our case, the documents indexed by the platform include three fields: `crawl_search_id`, `url` and `html`. The first is necessary to understand what search that page refers to; the others contain actual information about the web page. During the ingestion or search phase, tokenization and normalization are essential, together with the analyzer package, they define the rules to perform a full-text search. Specifically, the two custom analyzers `html_analyzer` and `html_search` were defined, which perform respectively the following operations:

1. strips HTML elements, removes punctuation and white spaces, casts all to lowercase, converts outlier characters to their ASCII equivalent when possible and removes stop words of different alphabets (ingestion phase);
2. the same above, without the strip of HTML elements (search phase).

Once the index is populated, it will be possible to search one or more keywords: the response will show all URLs that in the `html` field contain those words or, depending on the parameters chosen in the query, at least one of them.

4.1.2 Core

The core module is the beating heart of the application. It receives messages from the backend and starts the web crawling process, which is divided into several stages. Both implementations that exploit the serverless paradigm operate following this workflow; on the other hand, the microservices variant will be analyzed separately in Subsection 4.2.3.

An example of an initial message is shown in Listing 4.1, you can specify an optional fourth field within the JSON object called `credentials`, which is needed in case you have to perform automatic login to the site you want to crawl (n.b. it is a limited feature, working only in certain types of sites that contain standard authentication forms). Figure 4.3 illustrates all the intermediate steps of a search, according to the method chosen by the user.

```
[
   {
      "id":"65d5f46d7a848e6e8bf41bb3",
      "url":"https://tg24.sky.it/",
      "browserAutomation":"true"
   }
]
```

Listing 4.1: Example of an initial message received by the core module.

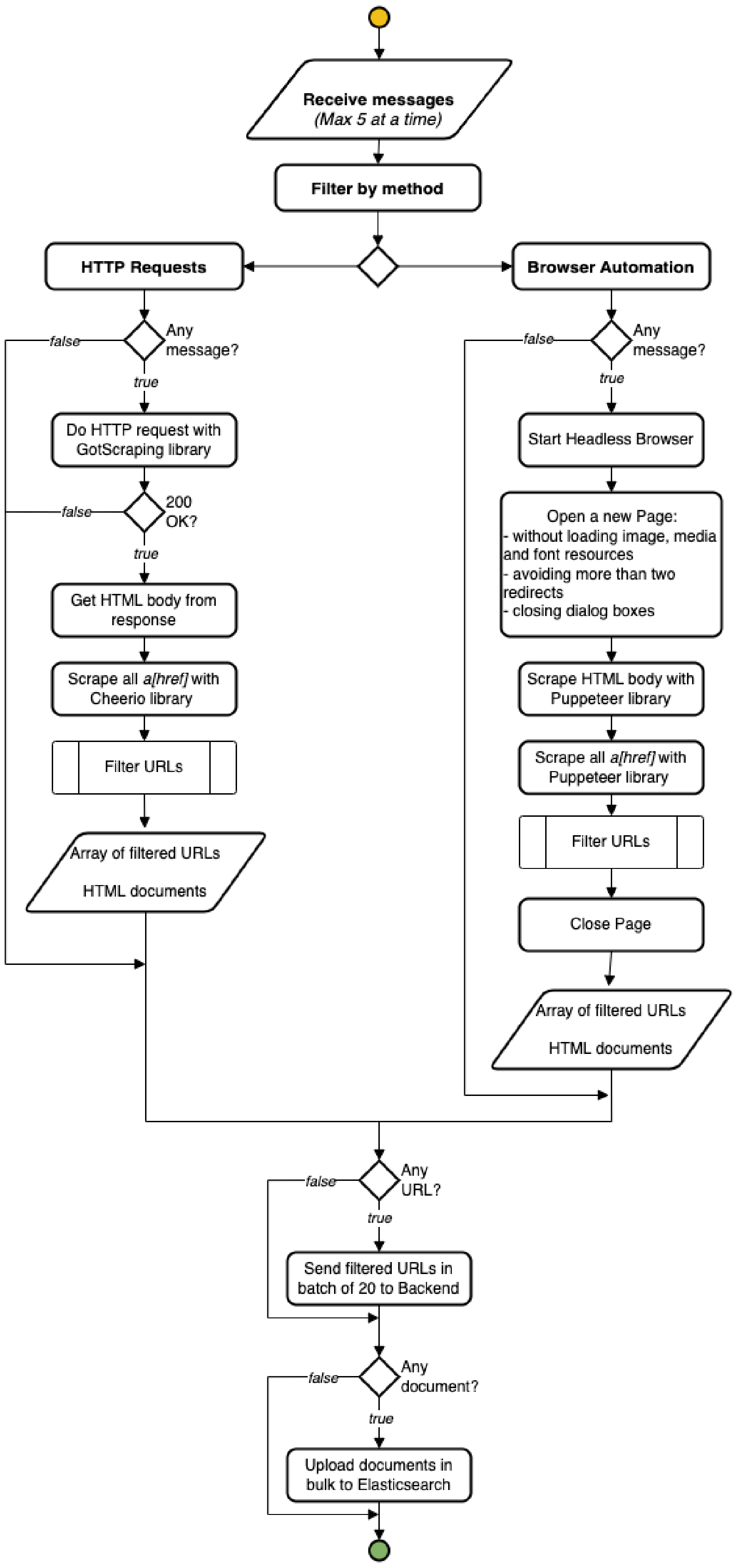

Receive messages
(Max 5 at a time)
Filter by method
HTTP Requests
Browser Automation
Any message?
false
true
Do HTTP request with GotScraping library
200 OK?
Get HTML body from response
Scrape all a[href] with Cheerio library
Filter URLs
Array of filtered URLs
HTML documents
Start Headless Browser
Open a new Page:
- without loading image, media and font resources
- avoiding more than two redirects
- closing dialog boxes
Scrape HTML body with Puppeteer library
Scrape all a[href] with Puppeteer library
Close Page
Any URL?
Send filtered URLs in batch of 20 to Backend
Any document?
Upload documents in bulk to Elasticsearch

Figure 4.3: Diagram describing the operations performed by the core module, in the two serverless implementations.

Functions can process a maximum of 5 messages at a time. After receiving an array in input, it is filtered to separate messages according to the `browserAutomation` field. The differences between the two methods will now be analyzed.

`HTTP request` For each message inside the filtered array with the `browserAutomation` field equal to *false*, get the `url` field and perform an HTTP request using the Got Scraping library [3]. If the response status code is *200 OK*, retrieve the HTML body of the web page and scrape all the *a[href]* selectors using Cheerio library [7]. Once the *href* attributes have been extracted from the *a* selectors and a set is created with them, the process of filtering URLs is carried out.

`Browser automation` If there is any message inside the filtered array with the `browserAutomation` field equal to *true*, start the Chromium browser in headless mode, get the `url` field and open a new tab (n.b. Puppeteer calls it page) for each of them. In these new tabs, the request interceptor is enabled so that image, media and font resources are not loaded and redirects do not exceed two-hop limits. Also, if dialog boxes are present, they are closed. When the DOMContentLoaded event is fired, it is possible to retrieve the HTML body of the web page and scrape all the *a[href]* selectors using directly Puppeteer. Once the *href* attributes have been extracted from the *a* selectors and a set is created with them, the same URL filtering process used in the previous method and illustrated in Figure 4.4 is performed.

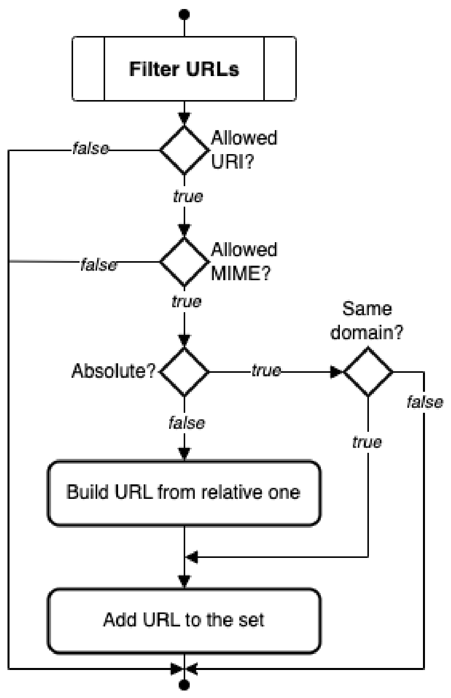


Figure 4.4: Diagram describing the steps performed to filter URLs.

Given the array containing all the raw links, it is necessary to filter them as follows:

1. Check the type of URI, discarding those not of interest with a regex (e.g.

mailto,*tel*, *javascript*, etc.);

2. Check the MIME resource type according to the allowed types (e.g. *text*, *html*, *application/x-httpd-php* etc.);

3. If both conditions are met, check whether the URL is absolute or relative. In the first case, it will be sufficient to verify that the host of the seed URL is contained therein; in the latter, it is necessary to reconstruct the absolute URL;

4. The filtered URL can be added to the set, which will then be output by the function in the form of an array.

Finally, the filtered URLs inside the array are sent to the backend in batches of 20 as shown in Listing 4.2 and the HTML documents are uploaded in bulk to the Elasticsearch index along with the source `url` and the `id` to link the search.

```
{
   "id":"65d5f46d7a848e6e8bf41bb3",
   "batch":[
      "https://tg24.sky.it/edizioni-locali",
      "https://tg24.sky.it/salute-e-benessere/alimentazione",
      etc.
   ]
}
```

Listing 4.2: Example of a response message sent by the core.

With these two search methods available, we offer users the flexibility to choose according to their needs, allowing each person to decide which method to use: more realistic browsing that takes advantage of browser automation and loads the DOM of each web page, or using HTTP requests that are certainly faster but with possible limitations imposed by web site administrators.

4.1.3 Backend

The backend module is needed to manage the creation, interaction, and deletion of searches and, in particular, allows the web crawling process to continue once it has started. It was developed according to the principles of Hexagonal Architecture, taking advantage of a scaffolder used in Kopjra as a basic project structure generator.

It connects to a MongoDB NoSQL Database and interacts with three collections defined as follows:

- `crawlsearches`, where all the information about searches is stored, it is necessary to keep track of the crawling status and the number of URLs visited so far, two useful timestamps related to creation and last update are also tracked. Some important fields in this collection are *seed*, *visitedUrls*, *maxVisitedUrls*, and *status*;

- `results`, stores all URLs obtained through the core module and keeps track of those visited; a timestamp related to the creation is also tracked. Some important fields in this collection are *url*, *visited*, and *crawlSearchId*;
- `users`, manages all stuff related to the user, such as credentials, OAuth 2.0 token, etc. It is fundamental during the authentication stage.

As a backend application, it offers all CRUD operations on the first two collections. Still, we will focus mainly on the paths of interest that manage the flow of creating a search up to its completion and allow you to search in the results obtained.

`POST /crawls` Listing 4.3 shows an example of a valid payload to use on this route. Once the body has been validated, an object within the `crawlsearches` and `results` collections is created, where the *visited* field is set to true for the *seed* URL. Based on the implementation, a first message is sent to the core module through SQS or RabbitMQ broker.

```
{
   "seed":"https://tg24.sky.it/",
   "name":"tg24 sky news",
   "maxVisitedUrls":100,
   "browserAutomation":true
}
```

Listing 4.3: Example of a valid payload message to create a new search.

`POST /internal/results` The same payload illustrated in Listing 4.2 is used on this route. When the core module starts sending data, the data does not go directly to the backend; instead, SNS or RabbitMQ is exploited, depending on the implementation. Once the message is forwarded by one of the previous technologies and received by the backend, it creates many objects inside the `results` collection based on the number of URLs in the *batch* field, setting them all to unvisited. Now, the current crawl search identified by the *id* can continue or not, there may be several scenarios:

- the search has already reached the finished *status* due to previous calls on the same internal route;
- the search has not yet reached the finished *status*, so the backend needs to check if the current number of *visitedUrls* has not exceeded the *maxVisitedUrls*. If this condition is verified, it increments the number of *visitedUrls*, retrieves a new unvisited URL from the `results` collection and sends it to SQS;
- the search has not yet reached the finished *status* but the current number of *visitedUrls* has reached the *maxVisitedUrls* value, so the search *status* is updated to finished;
- the search has not yet reached the finished *status* nor the *maxVisitedUrls* value, but there are no more visitable URLs within the `results` collection, so the search *status* is updated to finished.

`GET /crawls/crawlSearchId/results` During the crawling process, as we already said, the HTML body of each visited URL is indexed. This route allows keyword searches within the URL string and inside the index managed by Elasticsearch, which contains the HTML body of the web pages visited. Just specify the *url* parameter for the first type of search, and the *htmlQuery* and *strictHtmlQuery* parameters for the second. In particular, *strictHtmlQuery* is a boolean that specifies whether the keywords must all be present in the document or at least one is sufficient.

Having explained the backend's role in creating and completing a search as well as keyword searching in the results, let us now turn to its architecture, illustrated in Figure 4.5.

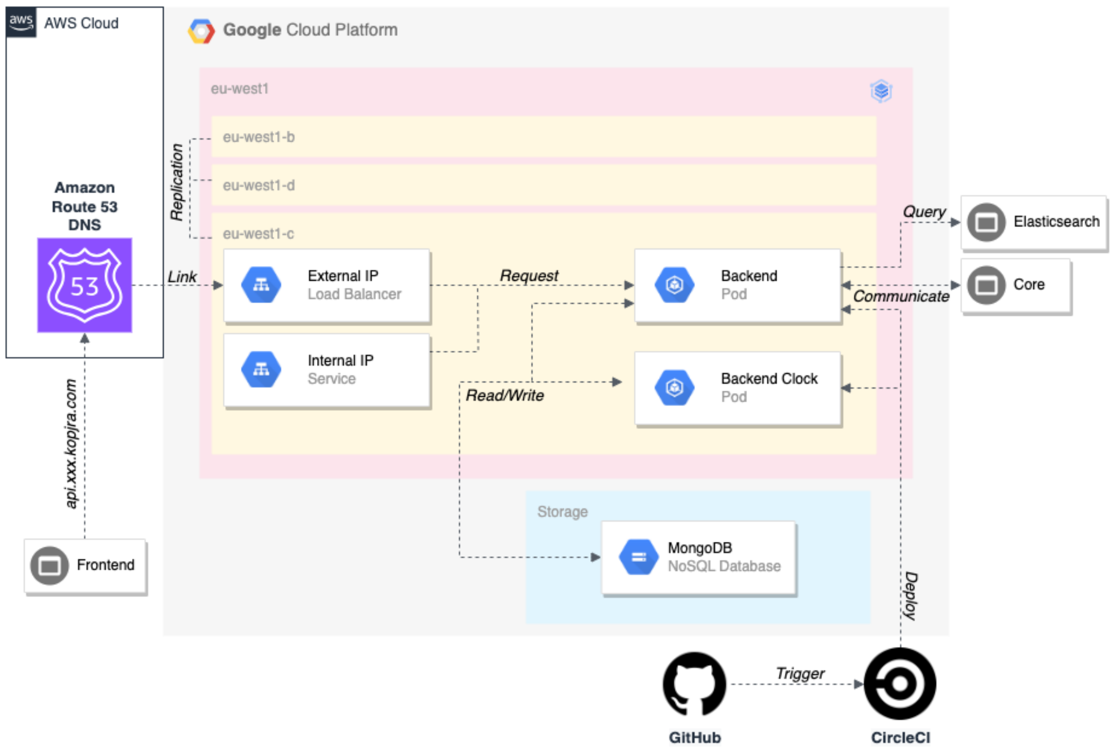


Figure 4.5: Backend architecture.

Docker was used to define a multi-stage build, which, together with a `node:alpine` base image, allowed us to optimize our Container. The CircleCI tool builds and pushes the Docker image, and then leverages the Kustomize configuration files to deploy in a standard way our backend application on Google Kubernetes Engine (GKE). Several resources have been defined in the Kubernetes files, such as ConfigMap, Secret, Deployment, Service and Ingress. The latter manages the external access to the backend so that the frontend can query it directly from the DNS name specified in Amazon Route 53 service.

A note on why a clock Pod was also deployed: it checks every 10 minutes if, for some reason, a search crashes and does not reach the finished state, updating it to a general error state if the last change was within 5 minutes.

4.1.4 Frontend

The frontend module makes life easier for users who want to use this application. It is a SPA written in React and deployed inside an S3 bucket, according to [30]. Again, a scaffolder provided by Kopjra was used to create a base project structure and start the development.

The architecture is illustrated in Figure 4.6, mainly leveraging AWS services such as S3, Cloudfront, ACM and Route 53. In addition, the deployment is performed using CircleCI's `aws-s3` orb[2], which allows us to synchronize and copy all files inside our repository to an S3 bucket.

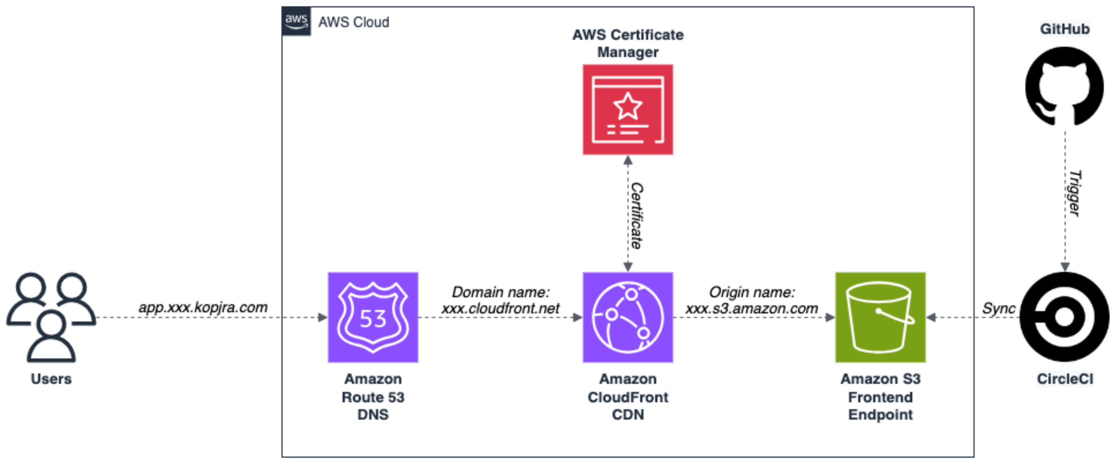


Figure 4.6: Frontend architecture.

The ability to authenticate with your Google account by leveraging OAuth 2.0 is given, so one does not have to create an additional user account.

Some screenshots are given in Appendix A so that the various scenarios can be better understood.

4.2 Implementations

The various modules that make up the web crawling application were described so as to get an overview of the behaviour of each of them. In this section, we want to take an in-depth look at the three proposed implementations, dwelling on the configuration of the core module of each. Similar architectures to the previous ones will then be shown, with some variations highlighted in orange.

4.2.1 `krawler` on AWS Lambda

In this scenario, the core module is deployed in a Lambda function, and the I/O communications are managed by SQS and SNS services respectively. The first challenge was to install a browser inside the latest NodeJS image[3] provided by

[2]See https://circleci.com/developer/orbs/orb/circleci/aws-s3 for more details.
[3]See https://gallery.ecr.aws/lambda/nodejs for more details.

AWS. One simple method would be installing Google Chrome directly, as explained in [53], but unfortunately, it significantly enlarges image size. Using `chrome-aws-lambda` [1] could have been an option thanks to the small size of the binary and the updates made a few days after Puppeteer's, but it is no longer maintained. The solution chosen involves the use of a fork of the latter, called `chromium` [62], which is no longer tied directly to Puppeteer but works perfectly with it and weighs a few megabytes more. It was then installed as a dependency along with `puppeteer-core`, without additional instructions within the Dockerfile. Again, a multi-stage build was defined to optimize our Container. Figure 4.7 illustrates the specific architecture of this implementation.

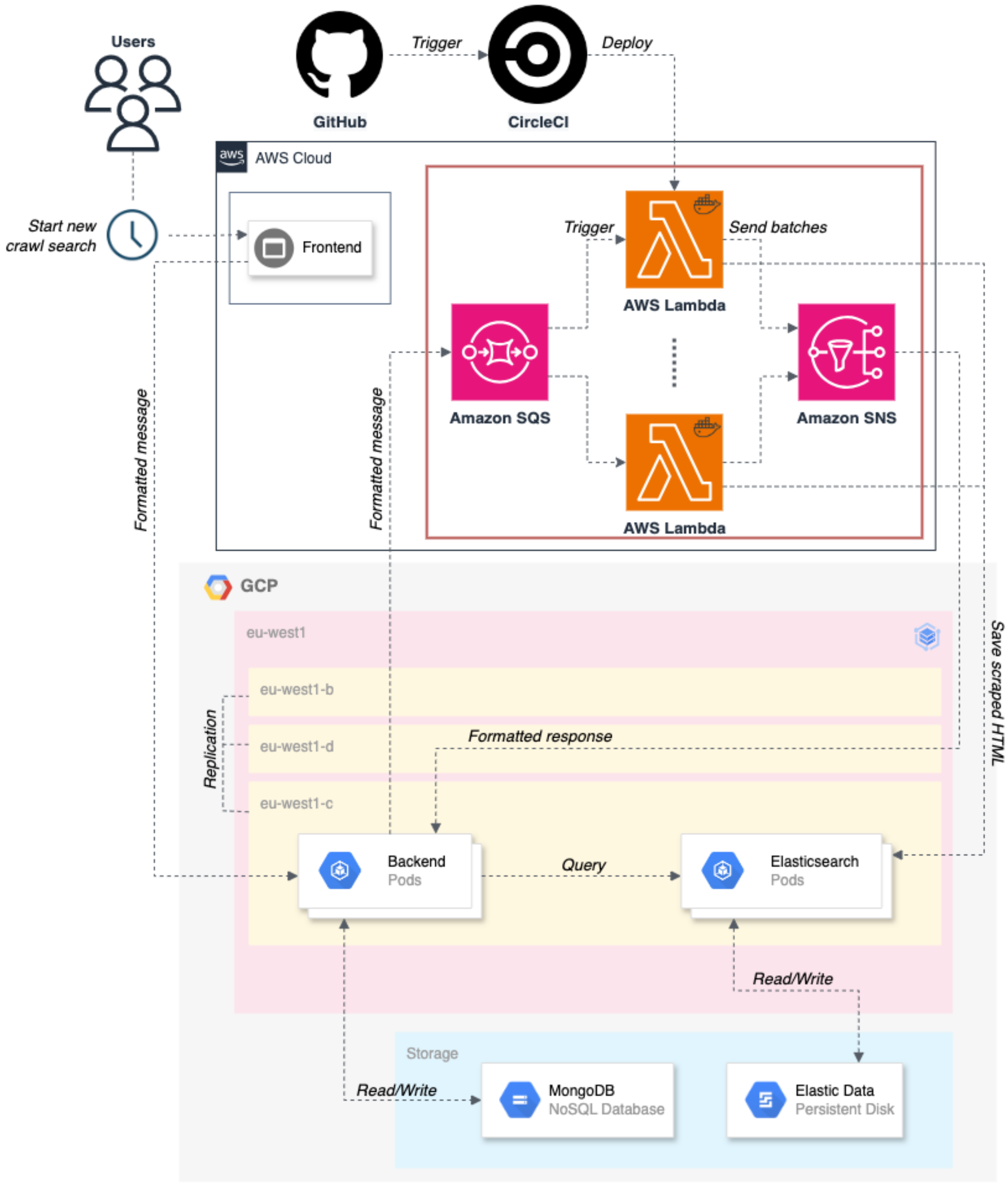


Figure 4.7: Application on AWS.

The deployment was managed by CircleCI, which handles the image building process, pushes the Container image into a registry, and uses the AWS CLI to update the Lambda function, previously configured with 1280MB of memory and 60s of timeout. It is not possible to explicitly state the computational capacity because it varies according to the allocated memory (1769MB of memory has the equivalent of one vCPU [57]) and the timeout was useful in case some web pages took too long to load.

SQS is used as a trigger for our Lambda function. Through it, we were able to choose the number of messages contained in each batch, and processed by the function at each invocation. The parameters set are described below with their corresponding value :

- *Batch size* defines the maximum number of messages to send to the function and was set to 5;
- *Batch window* specifies the maximum amount of time to gather messages before invoking the function. It was set to 20 seconds;
- *Maximum concurrency* represents the maximum number of concurrent functions that the event source can invoke. Since we had to perform a series of tests with the other implementations as well, it was decided to set it to 100.

During its execution, the core sends batch results to SNS, which acts as a deliverer and is responsible for notifying the registered endpoint. The latter points to the `/results/internal` route of the backend, made accessible by the DNS name previously registered in Route 53.

To recap, the backend module receives a new crawl search request and forwards it to SQS using the specific TypeScript SDK. Next, SQS sends this message to Lambda which is responsible for processing them and sends batches of URLs to SNS and HTML documents to Elasticsearch. Finally, SNS forwards the backend module with the results, which may or may not continue the crawling process.

4.2.2 `kn-krawler` on Knative

In this scenario, the core module and backend have been deployed using Knative, while I/O communications are handled by a RabbitMQ broker [17] that is responsible for routing all the events, represented by the CloudEvents [12] specification. The choice was made to use RabbitMQ [51] instead of Apache Kafka as the message deliverer because it is lighter and requires fewer computational resources; in addition, when this thesis work was started the integration of Apache Kafka within the Knative Eventing ecosystem was still in beta. In this case, the browser installation was easier, the latest NodeJS Alpine image was used, and thanks to its package manager, it was possible to install Chromium from the community repository. The `faas-js-runtime` framework [52] was used to handle requests on the core module and a multi-stage build was defined to further optimize the Container image, ensuring the highest efficiency level.

Some preliminary steps to install Knative components and RabbitMQ operator are given in Listing B.1. Then, you must deploy a RabbitMQ cluster as defined in Listing B.2, allowing the broker we will configure in Listing B.3 to function properly. Figure 4.8 shows the specific architecture of this implementation.

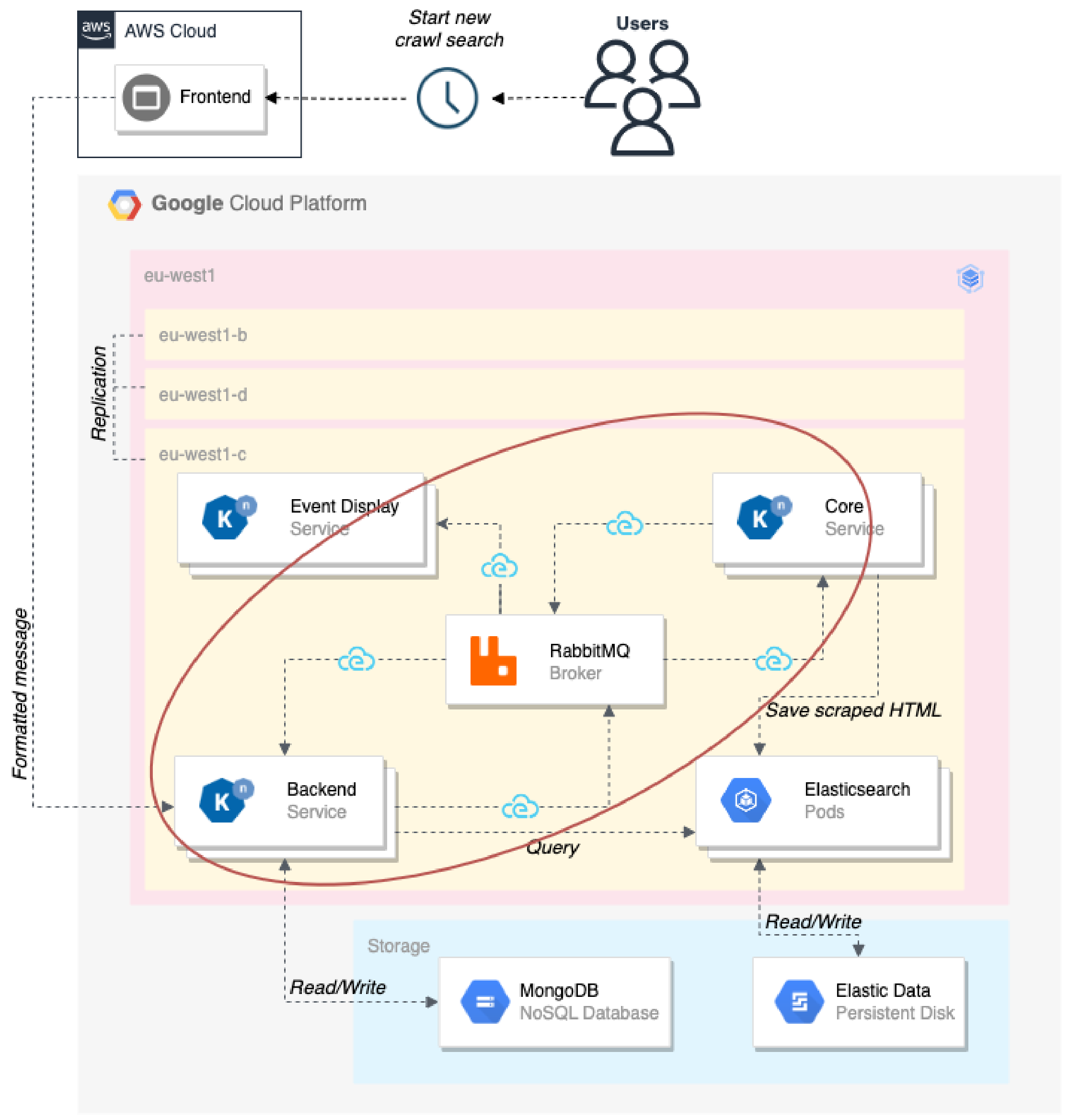


Figure 4.8: Application on Knative.

You can immediately notice an additional Pod called *Event Display*. It is used as a message viewer for the Dead Letter Sink (DLS) of the broker and for general logging, it scales up from zero when there is any message. Its Knative deployment is described by Listing B.4.

The backend module was also deployed using Knative as described in Listing B.5, with a minimum scale attribute equal to 1. It communicates with the broker leveraging the CloudEvents SDK [10] for TypeScript and receives CloudEvents on the internal route. One lacking feature, compared to the previous implementation, is the ability to specify a *batch_size*; in fact, changes were made by the backend to send, within a single CloudEvent, a list of messages of maximum length 5. An example of a message it sends is shown in Listing 4.4.

```
{
   "type":"dev.kapturer.crawls",
   "source":"kn-kapturer-be",
   "datacontenttype":"application/json",
   "data":[
      {
         "id":"65d5f46d7a848e6e8bf41bb3",
         "url":"https://tg24.sky.it/",
         "browserAutomation":false
      }
   ]
}
```

Listing 4.4: Example of search start message on Knative from backend module. The mandatory fields that CloudEvents must have were not reported.

The core module takes advantage of the scale-to-zero feature, enabling it to automatically terminate idle replicas that have remained inactive for a designated duration. The Knative deployment configuration for this service is detailed in Listing B.6. The various autoscaling annotations show that the metric chosen to scale up this service is concurrency, which determines the number of simultaneous requests that can be processed by each replica of an application at any given time. A cap of 100 concurrent functions has been set, which controls the maximum number of replicas that each revision should have. Knative will attempt to never have more than this number of replicas running or in the process of being created at any one point in time. Also, it is possible to specify a soft or hard limit, which represents a targeted threshold and an enforced upper bound on concurrency, respectively. The `containerConcurrency` field defines a hard limit of 1; this value defines the exact number of requests that can be sent to a replica at any time. The concurrency value can be further adjusted using the `target-utilization-percentage` attribute, which explains the percentage of the previously specified target that the autoscaler must actually achieve. In other words, it will create a new replica as soon as any concurrent request is detected. This means that the autoscaler will initiate a new replica to handle potential additional requests even with just one concurrent request.

When the RabbitMQ broker receives an event, it then takes care of routing it according to the subscribers to that event type. Therefore, communications to the backend and core modules, occur via two triggers that filter events based on the *source* and *type* fields of the CloudEvent, as can be seen from their definition in Listing B.7. The `parallelism` attribute helps to define the number of workers the trigger creates to consume messages off the queue and dispatch them to the sink.

4.2.3 `krawler-pod` on Kubernetes

The previous implementations follow the serverless paradigm, either delivered via service from a cloud provider such as AWS or managed in-house through the deployment of Knative, an open-source serverless solution. In this last implementation, we want to use the resources provided by Kubernetes, leveraging the microservices paradigm.

A lot of responsibilities regarding search management are taken away from the backend module. It still handles CRUD operations, but it is no longer concerned with the termination of searches. As can be seen from Figure 4.9, when the backend receives a request to create a search, a new Job resource is added to the cluster. The latter creates a Pod that handles the entire search to completion, visiting the number of web pages specified by the *maxVisitedUrls* field.

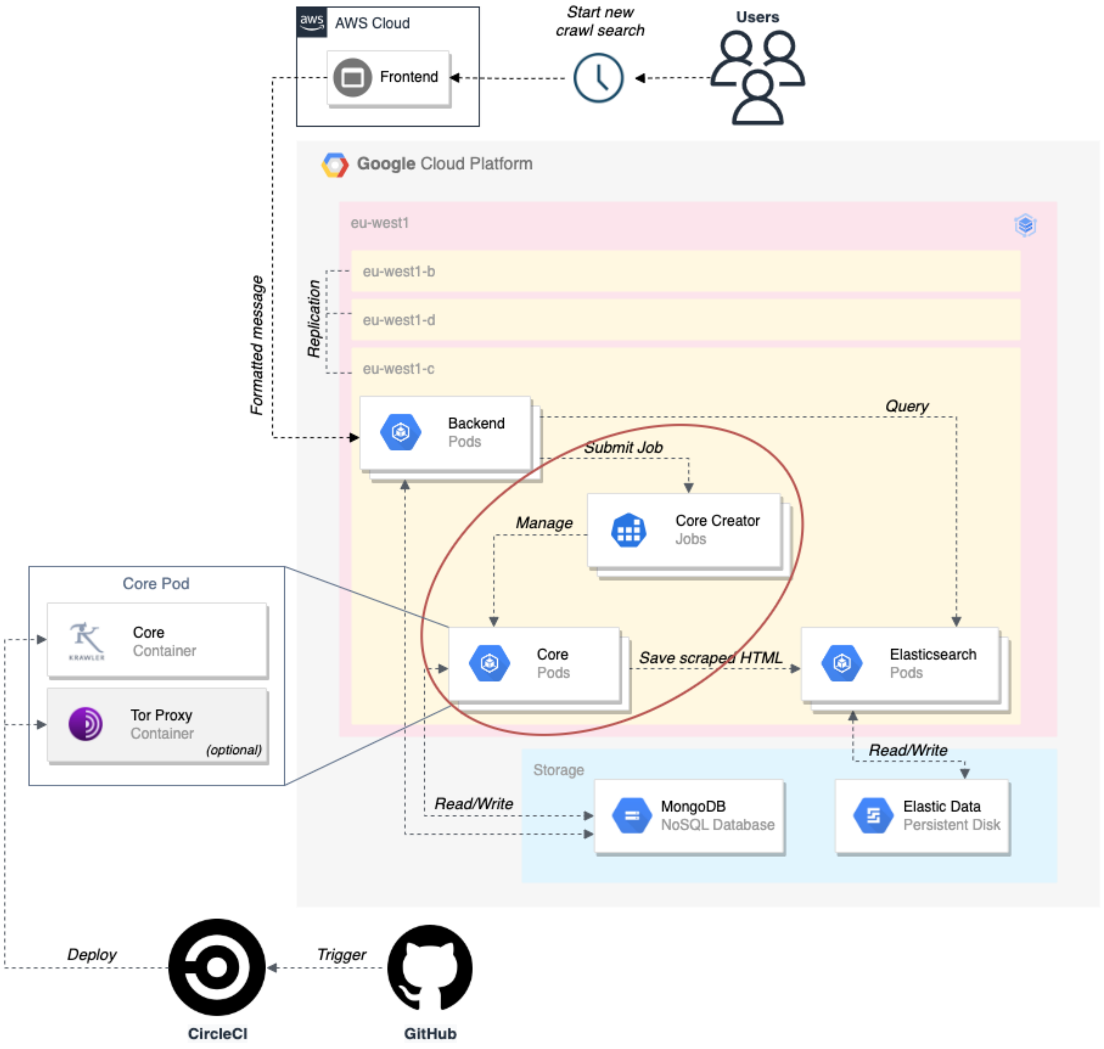


Figure 4.9: Application on Kubernetes.

In more detail, the backend after creating the search document within the `crawlsearch` collection, leverages the Kubernetes API client [11] to programmatically define a comprehensive set of resources, including Container, Pod and Job. These resources are encapsulated sequentially, akin to a matryoshka doll, and they compose the object that will be submitted to the cluster. The operation of submitting a Job without permission cannot be performed, which is why a Role-based access control (RBAC) policy was added through the definition of ServiceAccount, Role and RoleBinding resources.

All useful search information is passed to the core via environment variables injected into the Container so that it can handle the search autonomously since it connects directly to the database. The definition of Dockerfile is the same as that used in the Knative core, so the browser was installed the same way using APK. Due to the fact that this implementation has no concurrency, it was decided to visit 10 URLs at a time instead of 5 to speed up searches, slightly increasing the computational capacity and leaving the memory limit at 1280MB. In addition, thanks to the behaviour of this architecture, it was possible to add two interesting features:

1. stop and resume searches that have been started but not yet finished;
2. possibility of performing web crawling on *.onion* sites when the browser automation method is chosen, thanks to the optional use of a Sidecar Container that acts as a TOR proxy.

Analyzing the serverless implementations, surely stopping a search for which invocation requests have already been sent is not feasible by paradigm, since there is no storage of information and the overall status of the search is unknown. Also, injecting a Sidecar Container inside a function would only slow its execution, considering the proxy tor startup time, which is on the order of a few minutes. A microservices-oriented implementation strategy involves maintaining a continuously running Pod serving as a proxy tor. In the context of Knative, individual concurrent functions can specify the Pod's cluster domain name as proxy URL to perform web crawling on *.onion* websites.

Both Container's sources have been placed for convenience in the same GitHub repository, in different folders. For each commit, the eventual build and submission image to the registry is handled by CircleCI using `path-filtering` orb[4]. Changing even a single file within the folder will determine whether or not the pipeline will continue in that folder.

[4]See https://circleci.com/developer/orbs/orb/circleci/path-filtering for more details.

Chapter 5

Performance Analysis

This chapter describes the tests carried out to evaluate the effectiveness of the different web crawler implementations deployed for this thesis. The first section introduces the monitoring stack that allowed the gathering, visualization and storage of the metrics needed to compare the performance of the proposed architectures. The second section illustrates these tests, the motivation behind the choice of target URL and the cluster configuration for each variant. Finally, the results are reported in the last section and significant plots are shown.

5.1 Monitoring Stack

The usage of auxiliary technologies was necessary to collect, visualize and store the metrics exposed during the test executions. For these reasons, we based our monitoring stack on well-known tools such as Prometheus Stack [18], InfluxDB [33], Telegraf [35] and CloudWatch [56], which was provided by AWS and only useful for the implementation that exploits Lambda service. All other tools are open-source and have been deployed using Helm Charts.

`Prometheus Stack`, also known as `kube-prometheus stack`, is a collection of Kubernetes manifests, Grafana dashboards and Prometheus rules, as well as documentation and scripts, which enables easy end-to-end monitoring of the Kubernetes cluster using *Prometheus Operator*.

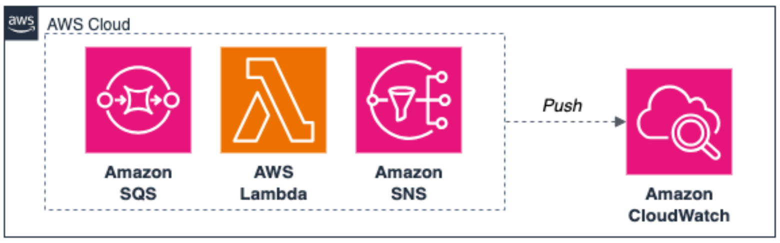
- *Prometheus* is the heart of the stack and provides a monitoring platform that collects metrics from the monitored targets by scraping the HTTP endpoints made available on those targets;
- *Grafana* enables the analysis and visualization of metrics from various sources, including databases, applications and monitoring systems such as Prometheus;
- *kube-state-metrics* listens to the Kubernetes API server and generates metrics about the state of the objects (e.g. Deployments, Pods, Jobs, etc.);
- Other components included in the stack but not useful for our tests are *node-exporter* and *Alertmanager*.

InfluxDB is a time series database designed to handle high write and query loads for efficient monitoring and analysis of time data such as metrics and events.

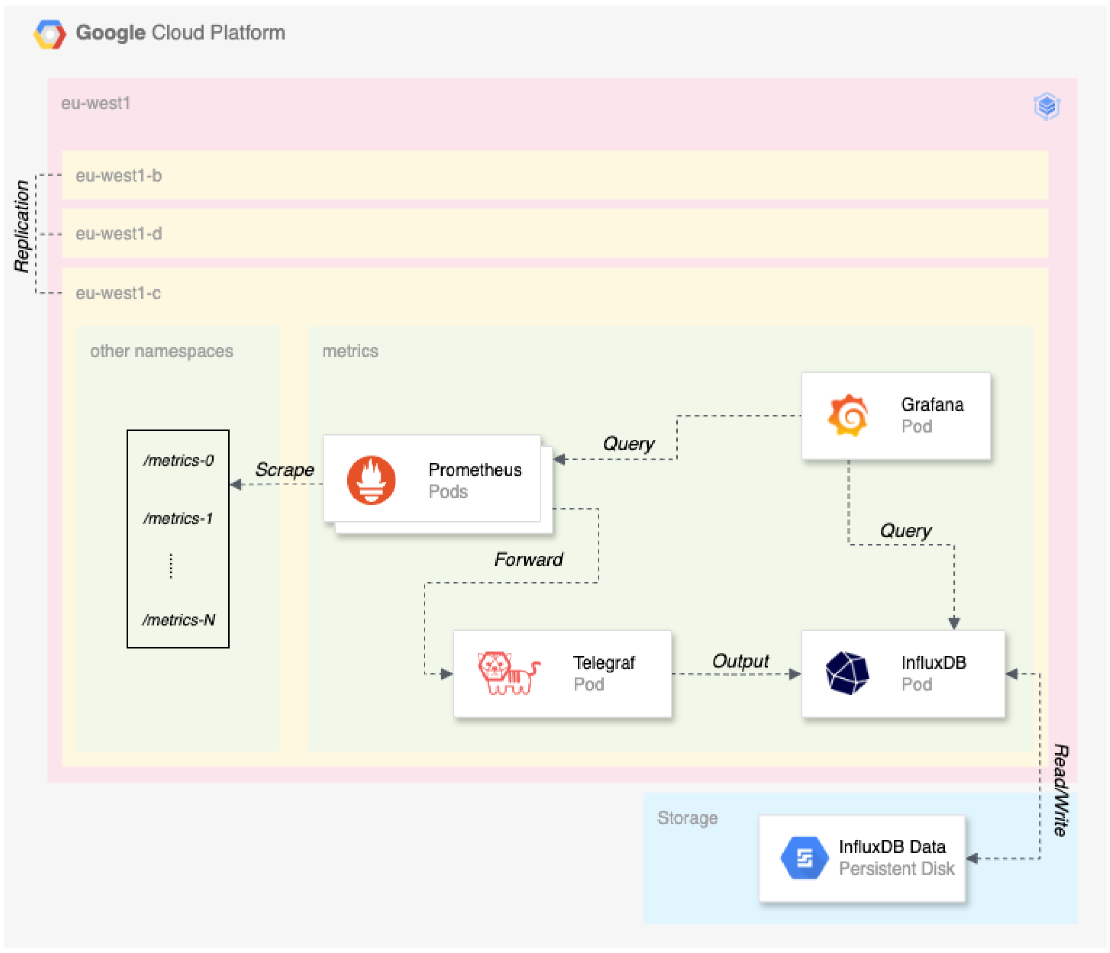
Telegraf is a plugin-driven server agent used to collect metrics from various sources and then send this data to various data stores or visualization platforms for monitoring and analysis.

CloudWatch is an AWS service that monitors and collects real-time metrics, log files, and events from AWS resources and on-premises servers. It enables users to set alarms, analyze data and create custom dashboards.

Figure 5.1 shows both architectures used by the different cloud providers.



(a) On AWS.



(b) On GKE.

Figure 5.1: Monitoring stack architectures.

Regarding AWS (Figure 5.1a), using CloudWatch doesn't require configurations; the services employed automatically send metrics to it. We had only to familiarise ourselves with the UI to easily view pre-defined dashboards based on the selected service and download data points in CSV format.

Instead, several steps were required to configure the chosen monitoring system on GKE. The two guides provided by Knative [14] and InfluxData [34] were followed and the setup configurations were reported on Section B.2. Summarizing, Prometheus collects all metrics from the monitored targets and forwards them to Telegraf using the `remote_write` API. Telegraf receives input metrics every 10 seconds and sends them as output to the designated bucket on InfluxDB for permanent storage. This allows us to analyse saved time series data directly from InfluxDB using the Flux[1] language. Alternatively, we can use Grafana, which provides pre-configured dashboards for Kubernetes, Knative and RabbitMQ.

5.2 Test Design

The test suite aims to analyse the performance of the three implementations, considering both the available web crawling methods, HTTP requests and browser automation. For every one of the six test configurations, we will vary the *maxVisitedUrls* parameter across values of $[250, 1000, 4000]$, determining the maximum number of web pages to be visited by the crawler.

Eighteen executions in total will therefore be carried out, each of which will be repeated only once, due to the high cost of the infrastructure for the Knative-based implementation.

The target URL chosen to run these tests was https://apkpure.net/it/, a website where you can download open-source Android applications that, however, does not have strict verification procedures like Google Play and could therefore contain malware. ApkPure is legal, but sometimes it's possible that the platform may host unauthorized content like cracked or pirated apps, meaning there is a risk of encountering illegal material if you decide to explore it [47, 48]. For this reason, it could be a specific use case for the application presented in this thesis.

5.2.1 Metrics

In all test configurations, the following metrics were considered:

- search **duration** (time between the start of a search and its completion with the last result, in the format *hh:mm:ss,ms*)
- cumulative **size of messages** in bytes (amount of data handled during communications)
- cumulative **error count**

[1]Flux is InfluxData's functional data scripting language designed for querying, analyzing, and acting on data. See https://docs.influxdata.com/influxdb/cloud/query-data/get-started/ for more details.

5.2.2 Cluster configurations

In order to execute the tests, we first set up the cluster for the AWS implementation. A three-node cluster was created on GKE with 6 vCPUs and 20 GB of memory, necessary to deploy the monitoring stack, Elasticsearch cluster and the backend's Deployments.

The second implementation based on Kubernetes used the same cluster; the backend's Deployments were updated and the necessary manifests to authenticate the Kubernetes client were added. There was no need to increase resources because only one search was carried out at a time.

The final implementation, using Knative, required more resources due to the additional components and the need to provide 100 concurrent functions. For these reasons, a new pool of three nodes with 92 vCPU and 384GB of memory was created to replace the previous one.

5.3 Experimental Results

The following sections contain the results of the different implementations. A table and a plot will be shown for the first two metrics, search duration and cumulative size of messages. Instead, the third metric will only be shown through a plot.

All of them require the rental of a cluster via a cloud provider, but the difference in resources required for Knative, compared to the managed version that follows the same serverless paradigm, provided by AWS Lambda, is considerable.

Figure 5.2 illustrates the search duration of each configuration. We can notice an increase in times as more URLs are visited, the Knative implementation is the one that performed best, followed by AWS and finally Kubernetes. These results were expected as the serverless paradigm is more performance-oriented due to more concurrent executions, with Knative having the advantage over AWS due to the ability to manage autoscaling as desired, as well as not having to communicate with services outside the cluster (e.g. SQS, SNS).

HTTP Request			
	Implementations		
Num. URLs	*Knative*	*AWS*	*Kubernetes*
250	00:00:20,103	00:02:44,066	00:01:04,241
1000	00:00:39,495	00:02:03,878	00:04:34,639
4000	00:02:02,620	00:05:33,763	00:14:45,958

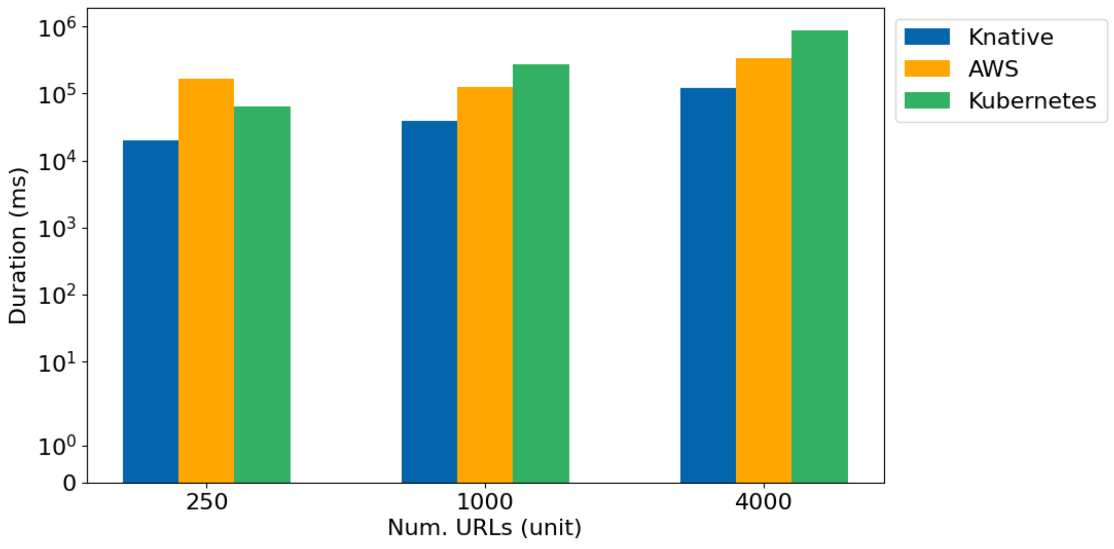


Figure 5.2: Results on search duration with HTTP request method.

Figure 5.3 shows the same trend as before, with obviously longer duration due to browser management. It is interesting to notice the last Kubernetes test with 4000 URLs, which reached a duration of over 4 hours. This could be due to using the same browser for a large number of web pages.

Browser Automation			
	Implementations		
Num. URLs	*Knative*	*AWS*	*Kubernetes*
250	00:01:42,815	00:04:35,153	00:14:41,554
1000	00:03:13,657	00:06:02,608	00:59:49,047
4000	00:08:08,082	00:10:39,026	04:10:34,170

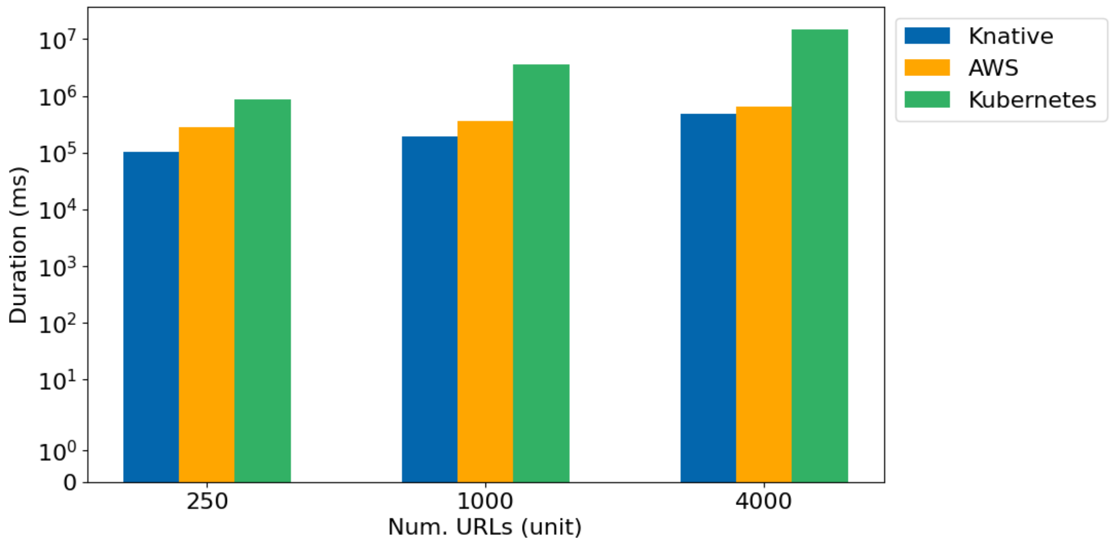


Figure 5.3: Results on search duration with Browser Automation method.

Figure 5.4 displays the results of the cumulative **size of messages** metric. As we saw in the previous chapter, the Kubernetes implementation does not exploit any queuing or notification mechanism, so we tracked the size of the response object that the client used to create a Job. It is invariant across the different numbers of URLs because it is taken directly from the database; we submitted the same object with the same environment variables. For these reasons, it is the implementation with the lowest value. AWS and Knative have similar results, with the latter remaining slightly lower but tending to increase probably for two reasons:

- the overhead generated by the return messages of the various functions, used for logging purposes;
- The higher throughput of HTTP requests compared to browser automation causes more CloudEvents to contain less than five URLs, as the requests do not get queued up. This increases the amount of messages sent when using HTTP request, while having no effect when using browser automation.

HTTP Request			
	Implementations		
Num. URLs	*Knative*	*AWS*	*Kubernetes*
250	49052	1642277	7708
1000	3571312	7080464	7708
4000	67007627	32316143	7708

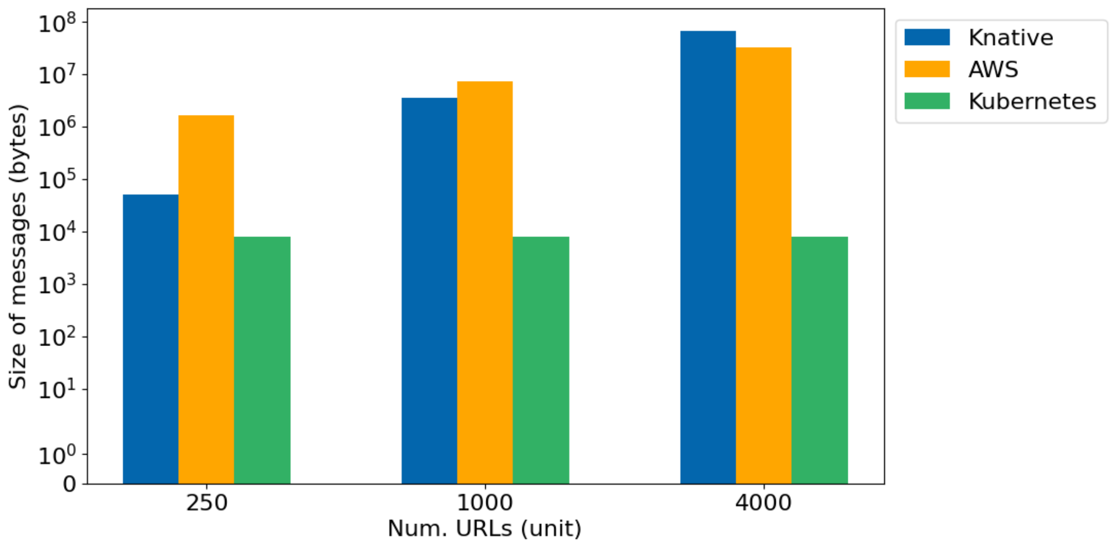


Figure 5.4: Results on cumulative size of messages with HTTP request method.

Figure 5.5 follows a similar pattern, the Knative implementation using the browser method can better manage the number of requests stored within each CloudEvents, which is why we have lower values than the other method.

Browser Automation			
	Implementations		
Num. URLs	*Knative*	*AWS*	*Kubernetes*
250	101059	1782075	7708
1000	678352	8624444	7708
4000	7755194	28512443	7708

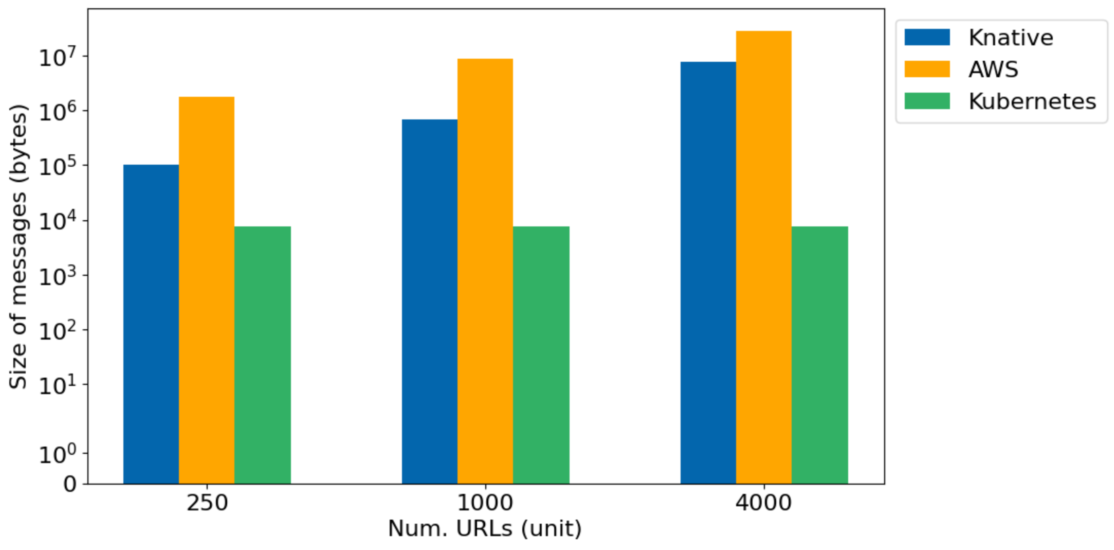


Figure 5.5: Results on cumulative size of messages with Browser Automation method.

Finally, in Figure 5.6 and 5.7, it can be seen that only the AWS implementation reported errors during testing. This could be due to the timeout set to 60s on the Lambda or some rate limit implementing the visited website for the AWS IPs.

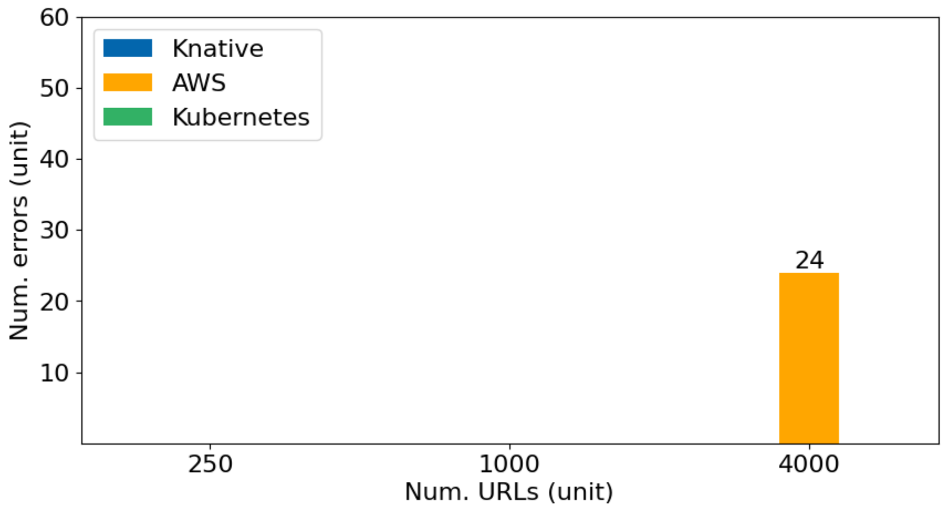


Figure 5.6: Results on error count with HTTP request method.

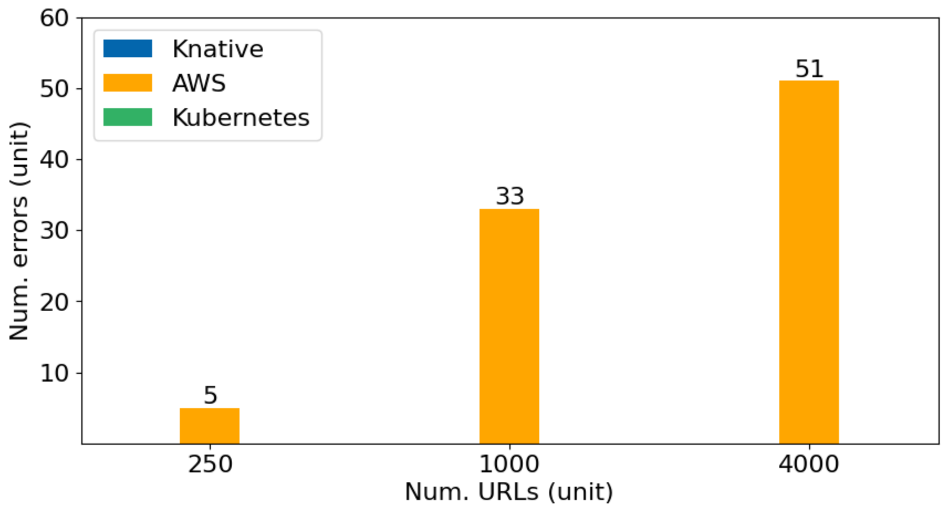


Figure 5.7: Results on error count with Browser Automation method.

Chapter 6

Conclusions

All three web crawler implementations presented in this thesis were able to collect and index information about web pages. The tests performed and the metrics recorded made it possible to understand the advantages and disadvantages of each platform, both from a performance and economic point of view.

As expected, the two platforms exploiting the serverless paradigm proved faster in executing the crawl due to the concurrent execution of multiple functions. Using queuing or notification services, the latter generated an average of 40MB for all 6 experiments performed on AWS and Knative respectively.

On the other hand, using Kubernetes resources to create a Job that deploys a Pod for each new search, doesn't introduce any overhead due to communication but has lower performance in terms of crawl duration, especially as the number of URLs increases.

A fundamental difference in AWS, but not in Knative, concerns the possibility of specifying the batch size. As mentioned, this lack has been fixed on the development side of the backend and core modules. The community has been asked about this issue, and for now, a GitHub issue [16] mentions the problem.

Cluster configurations already provide at first glance an idea of which implementation is the most expensive. Knative is the platform where tests performed best for this specific use case, but we must consider that to set up a cluster of that value and maintain it requires a considerable portfolio, especially if we assume a maximum limit of a thousand concurrent functions, so as to match normal AWS Lambda behaviour. Other implementations also require an underlying infrastructure, at a minimum for the backend, but we are talking about considerably fewer resources because the pod has a few more resources than a Knative function and handles all the search, whereas AWS manages Lambda, you are going to pay for just the actual usage.

6.1 Future works

The work discussed in this thesis can be improved from several points of view, which concern both the application and architectural side:

- the test results showed that the Kubernetes implementation could have some problems with the browser on searches with more than a thousand of URLs, so it would be interesting to investigate and repeat the tests;
- in AWS Lambda, it might be useful to increase the timeout to check whether errors remain by repeating the experiments;
- new features could be added to the general core implementation, such as automatic bypassing of reCAPTCHAs, the possibility of using residential proxies to avoid the rare problems with CDNs, and the possibility of automatic authentication even in those websites that do not use the classic forms;
- defines a hybrid architecture that uses AWS as main and, in case of saturated concurrent functions, routes subsequent messages to a Knative cluster;
- deploy a Pod that is always up and allows to service a TOR proxy so that serverless implementations can also use this functionality.

Bibliography

[2] Andresen, S.L. “John McCarthy: Father of AI”. In: *IEEE Intelligent Systems* 17.5 (2002), pp. 84–85. DOI: 10.1109/MIS.2002.1039837 (cit. on p. 1).

[19] Udapure, Trupti; Kale, Ravindra; Dharmik, Rajesh. “Study of Web Crawler and its Different Types”. In: *IOSR Journal of Computer Engineering* 16 (Jan. 2014), pp. 01–05. DOI: 10.9790/0661-16160105 (cit. on p. 9).

[27] Lin, Ping-Min; Glikson, Alex. “Mitigating Cold Starts in Serverless Platforms: A Pool-Based Approach”. In: *CoRR* abs/1903.12221 (2019). DOI: 10.48550/arXiv.1903.12221 (cit. on p. 32).

[28] Mell, Peter; Grance, Tim. “The NIST Definition of Cloud Computing”. In: *NIST Special Publications* 800.145 (2011). DOI: 10.6028/NIST.SP.800-145 (cit. on pp. 1, 2).

[31] Fan, Dayong; He, Dongzhi. “Knative Autoscaler Optimize Based on Double Exponential Smoothing”. In: *2020 IEEE 5th Information Technology and Mechatronics Engineering Conference (ITOEC)*. 2020, pp. 614–617. DOI: 10.1109/ITOEC49072.2020.9141858 (cit. on p. 32).

[36] Van Eyk, Erwin; Toader, Lucian; Talluri, Sacheendra; Versluis, Laurens; Uță, Alexandru; Iosup, Alexandru. “Serverless is More: From PaaS to Present Cloud Computing”. In: *IEEE Internet Computing* 22.5 (2018), pp. 8–17. DOI: 10.1109/MIC.2018.053681358 (cit. on p. 5).

[37] Khder, Moaiad. “Web Scraping or Web Crawling: State of Art, Techniques, Approaches and Application”. In: *International Journal of Advances in Soft Computing and its Applications* 13 (Dec. 2021), pp. 145–168. DOI: 10.15849/IJASCA.211128.11 (cit. on pp. 8, 9).

[40] Li, Junfeng; Kulkarni, Sameer G.; Ramakrishnan, K. K.; Li, Dan. “Analyzing Open-Source Serverless Platforms: Characteristics and Performance”. In: *Proceedings of the 33rd International Conference on Software Engineering and Knowledge Engineering*. KSI Research Inc., 2021. DOI: 10.18293/SEKE2021-129 (cit. on p. 8).

[41] Hongfei, Yan; Jianyong, Wang; Xiaoming, Li; Lin, Guo. “Architectural design and evaluation of an efficient Web-crawling system”. In: *Journal of Systems and Software* 60.3 (2002), pp. 185–193. ISSN: 0164-1212. DOI: 10.1016/S0164-1212(01)00091-7 (cit. on p. 29).

[44] Erl, Thomas; Puttini, Ricardo; Mahmood, Zaigham. *Cloud Computing: Concepts, Technology & Architecture*. 1st. Pearson Education, 2014. ISBN: 9789332535923. URL: https://dl.acm.org/doi/abs/10.5555/2500934 (cit. on p. 1).

[46] Merkel, Dirk. "Docker: Lightweight Linux Containers for Consistent Development and Deployment". In: *Linux J.* 2014.239 (2014). DOI: 10.5555/2600239.2600241 (cit. on p. 8).

[49] Mirtaheri, Seyed; Dinçktürk, Mustafa; Hooshmand, Salman; Bochmann, Gregor; Jourdan, Guy-Vincent; Onut, Iosif-Viorel. "A Brief History of Web Crawlers". In: (2014). DOI: 10.48550/arXiv.1405.0749 (cit. on p. 29).

[50] Jonas, Eric; Schleier-Smith, Johann; Sreekanti, Vikram; Tsai, Chia-Che; Khandelwal, Anurag; Pu, Qifan; Shankar, Vaishaal; Carreira, Joao; Krauth, Karl; Yadwadkar, Neeraja; Gonzalez, Joseph; Popa, Raluca Ada; Stoica, Ion; Patterson, David. "Cloud Programming Simplified: A Berkeley View on Serverless Computing". In: (2019). DOI: https://doi.org/10.48550/arXiv.1902.03383 (cit. on p. 5).

[59] Chaulagain, Ram Sharan;Pandey, Santosh;Basnet, Sadhu Ram; Shakya, Subarna. "Cloud Based Web Scraping for Big Data Applications". In: *2017 IEEE International Conference on Smart Cloud*. 2017, pp. 138–143. DOI: 10.1109/SmartCloud.2017.28 (cit. on pp. 8, 29, 30).

[60] Simmon, Eric D. "Evaluation of Cloud Computing Services Based on NIST SP 800-145". In: *NIST Special Publications* 500.322 (2018). DOI: 10.6028/NIST.SP.500-322 (cit. on pp. 2, 4).

[61] Abu Kausar, Mohammad; Dhaka, Vijaypal; Singh, Sanjeev. "Web Crawler: A Review". In: *International Journal of Computer Applications* 63 (Feb. 2013), pp. 31–36. DOI: 10.5120/10440-5125 (cit. on p. 9).

[66] Van Eyk, Erwin; Iosup, Alexandru; Seif, Simon; Thömmes, Markus. "The SPEC cloud group's research vision on FaaS and serverless architectures". In: *Proceedings of the 2nd International Workshop on Serverless Computing* (2017). DOI: 10.1145/3154847.3154848 (cit. on p. 4).

[67] Boldi, Paolo; Codenotti, Bruno; Santini, Massimo; Vigna, Sebastiano. "UbiCrawler: A Scalable Fully Distributed Web Crawler". In: *Softw. Pract. Exper.* 34.8 (2004), pp. 711–726. ISSN: 0038-0644. DOI: 10.1002/spe.587 (cit. on p. 29).

[68] Shahrad, Mohammad; Balkind, Jonathan; Wentzlaff, David. "Architectural Implications of Function-as-a-Service Computing". In: *Proceedings of the 52Nd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '52. ACM, 2019, pp. 1063–1075. ISBN: 978-1-4503-6938-1. DOI: 10.1145/3352460.3358296 (cit. on p. 4).

[69] Brendan, Burns; Grant, Brian; Oppenheimer, David; Brewer, Eric; Wilkes, John. "Borg, Omega, and Kubernetes". In: *ACM Queue* 14 (2016), pp. 70–93. URL: http://queue.acm.org/detail.cfm?id=2898444 (cit. on pp. 4, 8, 10, 11).

[70] Jinfeng, Wen; Zhenpeng, Chen; Xin, Jin; Xuanzhe, Liu. "Rise of the Planet of Serverless Computing: A Systematic Review". In: *ACM Trans. Softw. Eng. Methodol.* 32.5 (2023). DOI: 10.1145/3579643 (cit. on pp. 5, 6).

[71] Clinton, Gormley; Zachary, Tong. *Elasticsearch: The Definitive Guide*. 1st. O'Reilly Media, Inc., 2015. ISBN: 1449358543. URL: https://stmarysguntur.com/wp-content/uploads/2019/04/1021302647.pdf (cit. on p. 24).

[72] Bahrami, Mehdi; Singhal, Mukesh; Zhuang, Zixuan. "A Cloud-based Web Crawler Architecture". In: *2015 18th International Conference on Intelligence*

in Next Generation Networks. 2015, pp. 216–223. DOI: 10.1109/ICIN.2015.7073834 (cit. on p. 29).

Sitography

[1] alixaxel. *Repository chrome-aws-lambda*. URL: https://github.com/alixaxel/chrome-aws-lambda (cit. on p. 43).

[3] Apify. *Repository got-scraping*. URL: https://github.com/apify/got-scraping (cit. on p. 38).

[4] Banon, Shay. *Amazon: NOT OK - why we had to change Elastic licensing*. URL: https://www.elastic.co/blog/why-license-change-aws (cit. on p. 25).

[5] berstend. *Puppeteer-Extra Wiki*. 2018. URL: https://github.com/berstend/puppeteer-extra/wiki (cit. on p. 24).

[6] berstend. *Repository puppeteer-extra-plugin-stealth*. 2018. URL: https://github.com/berstend/puppeteer-extra/tree/master/packages/puppeteer-extra-plugin-stealth (cit. on p. 24).

[7] CheerioJS. *Repository cheerio*. URL: https://github.com/cheeriojs/cheerio (cit. on p. 38).

[8] CNCF. *Knative*. URL: https://www.cncf.io/projects/knative/ (cit. on p. 15).

[9] CNCF. *Kubernetes*. URL: https://kubernetes.io/ (cit. on pp. 4, 8, 10, 11).

[10] CNCF. *Respository cloudevents/sdk-javascript*. URL: https://github.com/cloudevents/sdk-javascript (cit. on p. 45).

[11] CNCF. *Respository kubernetes-client/javascript*. URL: https://github.com/kubernetes-client/javascript (cit. on p. 47).

[12] CloudEvents Community. *CloudEvents*. URL: https://cloudevents.io/ (cit. on pp. 20, 44).

[13] Knative Community. *CLI tools*. URL: https://knative.dev/development/client/ (cit. on pp. 15, 21).

[14] Knative Community. *Collecting Metrics in Knative*. URL: https://knative.dev/docs/serving/observability/metrics/collecting-metrics/ (cit. on p. 51).

[15] Knative Community. *Knative*. URL: https://knative.dev/ (cit. on pp. 10, 11, 14).

[16] Knative Community. *QP request batcher*. 2023. URL: https://github.com/knative/serving/issues/13691 (cit. on p. 59).

[17] Knative Community. *Respository eventing-rabbimq*. URL: https://github.com/knative-extensions/eventing-rabbitmq (cit. on p. 44).

[18] Prometheus Community. *Kube Prometheus Stack*. URL: https://github.com/prometheus-community/helm-charts/tree/main/charts/kube-prometheus-stack (cit. on p. 49).

[20] Elastic. *Data in: documents and indices*. URL: https://www.elastic.co/guide/en/elasticsearch/reference/current/documents-indices.html (cit. on p. 25).

[21] Elastic. *Elastic Cloud on Kubernetes*. URL: https://www.elastic.co/guide/en/cloud-on-k8s/current/k8s-quickstart.html (cit. on p. 11).

[22] Elastic. *Elasticsearch on Kubernetes: A new chapter begins*. URL: https://www.elastic.co/blog/introducing-elastic-cloud-on-kubernetes-the-elasticsearch-operator-and-beyond (cit. on p. 26).

[23] Elastic. *Elasticsearch Platform*. URL: https://www.elastic.co/ (cit. on pp. 1, 11).

[24] Elastic. *FAQ on 2021 License Change*. URL: https://www.elastic.co/pricing/faq/licensing (cit. on p. 25).

[25] Elastic. *Resilience in small clusters*. URL: https://www.elastic.co/guide/en/elasticsearch/reference/current/high-availability-cluster-small-clusters.html (cit. on p. 34).

[26] Linux Foundation. *About the Open Container Initiative*. URL: https://opencontainers.org/about/overview/ (cit. on p. 21).

[29] CNCF Serverless Working Group. *CNCF Serverless Whitepaper*. 2018. URL: https://github.com/cncf/wg-serverless/blob/master/whitepapers/serverless-overview/cncf_serverless_whitepaper_v1.0.pdf (cit. on pp. 4, 5).

[30] Jean-Baptiste Guillois. *Deploy a React-based single-page application to Amazon S3 and CloudFront*. URL: https://docs.aws.amazon.com/prescriptive-guidance/latest/patterns/deploy-a-react-based-single-page-application-to-amazon-s3-and-cloudfront.html (cit. on p. 42).

[32] The Tor Project Inc. *Tor Project*. URL: https://www.torproject.org/ (cit. on p. 11).

[33] InfluxData. *InfluxDB v2*. URL: https://github.com/influxdata/helm-charts/tree/master/charts/influxdb2 (cit. on p. 49).

[34] InfluxData. *Prometheus Remote Write Support with InfluxDB 2.0*. URL: https://www.influxdata.com/blog/prometheus-remote-write-support-with-influxdb-2-0/ (cit. on p. 51).

[35] InfluxData. *Telegraf*. URL: https://github.com/influxdata/helm-charts/tree/master/charts/telegraf (cit. on p. 49).

[38] Knative Steering Committee. *Knative graduation proposal*. URL: https://github.com/cncf/toc/pull/1245 (cit. on p. 15).

[39] Bynens, Mathias; Kvitek, Peter. *Chrome's Headless mode gets an upgrade: introducing –headless=new*. 2023. URL: https://developer.chrome.com/docs/chromium/new-headless (cit. on p. 22).

[42] Google LLC. *Knative*. URL: https://cloud.google.com/knative (cit. on p. 5).

[43] Google LLC. *Overview of Puppeteer*. 2018. URL: https://developer.chrome.com/docs/puppeteer/ (cit. on p. 22).

[45] Martinaitis, Dzidas. *Serverless Architecture for a Web Scraping Solution*. 2020. URL: https://aws.amazon.com/blogs/architecture/serverless-architecture-for-a-web-scraping-solution/ (cit. on p. 30).

[47] NordVPN. *A Guide to APKPure: Is It Legal and Is It Safe?* 2023. URL: https://www.avast.com/c-is-apkpure-safe (cit. on p. 51).

[48] NordVPN. *Is APKPure safe to use?* 2022. URL: https://nordvpn.com/blog/is-apkpure-safe/ (cit. on p. 51).

[51] RabbbitMQ. *RabbitMQ 3.13*. URL: https://www.rabbitmq.com/ (cit. on p. 44).

[52] NodeShift by Red Hat. *Respository faas-js-runtime*. URL: https://github.com/nodeshift/faas-js-runtime (cit. on p. 44).

[53] Evan Sangaline. *Installing Google Chrome on CentOS, Amazon Linux, or RHEL*. URL: http://intoli.com/blog/installing-google-chrome-on-centos/ (cit. on p. 43).

[54] Amazon Web Services. *AWS Lambda*. URL: https://aws.amazon.com/lambda/ (cit. on pp. 7, 10, 11).

[55] Amazon Web Services. *AWS Lambda Documentation*. URL: https://docs.aws.amazon.com/lambda/ (cit. on p. 13).

[56] Amazon Web Services. *CloudWatch Documentation*. URL: https://docs.aws.amazon.com/cloudwatch/ (cit. on p. 49).

[57] Amazon Web Services. *Configuring Lambda function options*. URL: https://docs.aws.amazon.com/lambda/latest/dg/configuration-function-common.html#configuration-memory-console (cit. on p. 44).

[58] Amazon Web Services. *Introducing AWS Lambda*. 2014. URL: https://aws.amazon.com/about-aws/whats-new/2014/11/13/introducing-aws-lambda/ (cit. on p. 29).

[62] Sparticuz. *Repository chromium*. URL: https://github.com/Sparticuz/chromium (cit. on p. 43).

[63] Kopjra Srl. *Kopjra*. URL: https://www.kopjra.com/ (cit. on pp. vii, 10).

[64] Stevenson, Jack. *Scaling up a Serverless Web Crawler and Search Engine*. 2021. URL: https://aws.amazon.com/blogs/architecture/scaling-up-a-serverless-web-crawler-and-search-engine/ (cit. on pp. 30, 31).

[65] Chrome DevTools team. *Puppeteer*. URL: https://pptr.dev/ (cit. on p. 22).

Glossary

ACM It is a service that handles the complexity of creating, storing, and renewing public and private SSL/TLS X.509 certificates and keys that protect your AWS websites and applications. 77

AI It is the science and engineering of making intelligent machines, especially intelligent computer programs. It is related to the similar task of using computers to understand human intelligence. 77

API The set of defined rules that enable different applications to communicate with each other. 77

APK It is the package manager of the Alpine Linux distribution. It is used to manage the system packages and the primary method for installing additional software. 77

ASCII It is the most common character encoding format for text data in computers and on the internet. 77

AWS It is a comprehensive and widely-used cloud computing platform provided by Amazon. 77

CDN It is a geographically distributed group of servers that caches content close to end users. A CDN allows for the quick transfer of assets needed for loading Internet content, including HTML pages, JavaScript files, stylesheets, images, and videos. 77

Cheerio It is a fast, flexible, and elegant library for parsing and manipulating HTML and XML. 38, 69

CircleCI It is a continuous integration and delivery platform that helps the development teams to release code rapidly and automate the build, test, and deploy. 41, 42, 44, 48, 69

CLI It is a text-based interface where you can input commands that interact with a computer's software. 77

Cloudfront It is a web service that speeds up distribution of your static and dynamic web content, such as .html, .css, .js, and image files, to your users. 42, 69

CNCF It is an open-source software foundation that promotes the adoption of cloud-native computing. 77

ConfigMap It is an API object used to store non-confidential data in key-value pairs. Pods can consume ConfigMaps as environment variables, command-line arguments, or as configuration files in a volume. 41, 70

Container It is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. 5, 6, 8, 11, 14, 17, 21, 26, 41, 43, 44, 48, 70

CRD It enables the introduction of unique objects or types into Kubernetes clusters to meet the needs of developers. 77

CRUD It describes the four essential operations for creating and managing persistent data elements, mainly in relational and NoSQL databases. 77

CSP It is a third-party company that provides scalable computing resources that businesses can access on demand over a network, including cloud-based computing, storage, platform, and application services. 77

CSV It is a text file format that uses commas to separate values, and newlines to separate records. A CSV file stores tabular data (numbers and text) in plain text, where each line of the file typically represents one data record. Each record consists of the same number of fields, and these are separated by commas in the CSV file. 77

Deployment It provides declarative updates for Pods and ReplicaSets. 41, 49, 52, 70

DLS It is a Knative construct that allows the user to configure a destination for events that would otherwise be dropped due to some delivery failure. This is useful for scenarios where you want to ensure that events are not lost due to a failure in the underlying system. 77

DNS It translates human-readable domain names to machine-readable IP addresses. 77

Docker It is an open platform for developing, shipping, and running applications. Docker enables you to separate your applications from your infrastructure so you can deliver software quickly. 8, 13, 26, 41, 70

Dockerfile Docker can build images automatically by reading the instructions from a Dockerfile. It is a text document that contains all the commands a user could call on the command line to assemble an image. 21, 43, 48, 70

DOM The data representation of the objects that comprise the structure and content of a document on the web. 77

DOMContentLoaded It is an event fired when the HTML document has been fully parsed and all deferred scripts have been downloaded and executed. See PuppeteerLifeCycleEvent for more details. 38, 70

Dynamo DB It is a fully managed NoSQL database service that provides fast and predictable performance with seamless scalability. 13, 29, 30, 71

EC2 It provides on-demand, scalable computing capacity in the Amazon Web Services Cloud. 77

ECK It is the official operator by Elastic for automating the deployment, provisioning, management, and orchestration of Elasticsearch, Kibana, APM Server, Beats, Enterprise Search, Elastic Agent, Elastic Maps Server, and Logstash on Kubernetes. 77

ECS It is a fully managed container orchestration service that helps you easily deploy, manage, and scale containerized applications. 77

ELK It is a stack that comprises three popular projects: Elasticsearch, Logstash, and Kibana. Often referred to as Elasticsearch, the ELK stack gives you the ability to aggregate logs from all your systems and applications, analyze these logs, and create visualizations for application and infrastructure monitoring, faster troubleshooting, security analytics, and more. 77

GCP It is a suite of cloud computing services offered by Google. 77

GKE It is a Google-managed implementation of the Kubernetes open source container orchestration platform. 78

GotScraping It is a small but powerful got extension with the purpose of sending browser-like requests out of the box. This is very essential in the web scraping industry to blend in with the website traffic. 38, 71

Helm Chart It allows you to manage Kubernetes manifests without using the Kubernetes CLI or remembering complicated Kubernetes commands to control the cluster. 49, 71

Hexagonal Architecture It is an architectural pattern used in software design. It aims to create loosely coupled application components that can be easily connected to their software environment through ports and adapters. This makes components exchangeable at any level and facilitates test automation. 39, 71

HPA The default scaling method in Kubernetes cluster. 78

HTML The markup language for the web that defines the structure of web pages. 78

HTTP The communications protocol used to connect to web servers on the Internet. 78

Ingress It is an API object that manages external access to the services in a cluster, typically HTTP. It may provide load balancing, SSL termination and name-based virtual hosting. 41, 71

IP It is the unique identifying number assigned to every device connected to the internet. An IP address definition is a numeric label assigned to devices that use the internet to communicate. 78

Job It is a Kubernetes resource that creates one or more Pods and will continue to retry execution of the Pods until a specified number of them successfully terminate. As Pods successfully complete, the Job tracks the successful completions. When a specified number of successful completions is reached, the task (i.e. Job) is complete. Deleting a Job will clean up the Pods it created. Suspending a Job will delete its active Pods until the Job is resumed again. 10, 47, 48, 49, 55, 59, 72

JSON It is a standard text-based format for representing structured data based on JavaScript object syntax. 78

Kubernetes It is a portable, extensible, open-source platform for managing containerized workloads and services that facilitates both declarative configuration and automation. vii, 4, 8, 10, 11, 13, 14, 15, 17, 18, 19, 21, 26, 41, 47, 49, 51, 52, 53, 54, 55, 59, 60, 72

Apache Kafka It is an open-source distributed event streaming platform used by thousands of companies for high-performance data pipelines, streaming analytics, data integration, and mission-critical applications.. 44, 72

Kendra It is an intelligent search service that uses natural language processing and advanced machine learning algorithms to return specific answers to search questions from your data. 30, 31, 72

KPA The default automatic scaling method inside Knative Serving. It allows the scale to zero and it works with different metrics. 78

Kustomize It is a Kubernetes configuration transformation tool that enables you to customize untemplated YAML files, leaving the original files untouched. 41, 72

MIME It indicates the nature and format of a document, file, or assortment of bytes. See MIME types for more details. 78

NIST It is a United States government agency responsible for developing and promoting measurement standards and technology advancements to enhance innovation and industrial competitiveness. 78

NoSQL Database They are non-tabular databases and store data differently than relational tables. They come in a variety of types based on their data model. The main types are document, key-value, wide-column, and graph. They provide flexible schemas and scale easily with large amounts of data and high user loads. 39, 72

Namespace It provides a mechanism for isolating groups of resources within a single cluster. Names of resources need to be unique within a namespace, but not across namespaces. 35, 73

OAuth 2.0 It is a standard designed to allow a website or application to access resources hosted by other web apps on behalf of a user. 78

OCI It is an open governance structure for the express purpose of creating open industry standards around container formats and runtimes. 78

OSINT It is the act of gathering and analyzing publicly available data for intelligence purposes. 78

Pod It is the smallest deployable unit of computing that you can create and manage in Kubernetes. A Pod is a group of one or more containers, with shared storage and network resources, and a specification for how to run the containers. 8, 10, 11, 17, 18, 35, 41, 45, 47, 48, 49, 59, 60, 73

Quorum-based System It is a mechanism by which decisions are made regarding data consistency and availability in a distributed environment. A quorum represents the minimum number of nodes that must agree on an operation to be successful. This ensures that data operations maintain consistency and resilience, even in the presence of node failures or network partitions. 34, 35, 73

RabbitMQ It is an open-source message-broker software that originally implemented the Advanced Message Queuing Protocol (AMQP) and has since been extended with a plug-in architecture to support Streaming Text Oriented Messaging Protocol (STOMP), MQ Telemetry Transport (MQTT), and other protocols. 40, 44, 45, 46, 73

RBAC It is a method of regulating access to computer or network resources based on the roles of individual users within your organization. 78

reCAPTCHA It is a free service from Google that helps protect websites from spam and abuse. 60, 73

REST It is an architectural style for designing networked applications. It is based on a stateless, client-server communication protocol, typically the HTTP protocol and it is commonly used in web services development. 78

Role It is a collection of permissions that allow users to perform specific actions on a defined set of Kubernetes resource types. 48, 73

Role Binding It grants the permissions defined by roles to the relevant user or user group. 48, 73

Route 53 It is a highly available and scalable DNS web service which allows domain registration, DNS routing, and health checking. 36, 41, 42, 44, 73

S3 It is an object storage service provided by Amazon that offers industry-leading scalability, data availability, security, and performance. 78

SDK It is a tool for third-party developers to use in producing applications using a particular framework or platform. 78

Secret It is an object that contains a small amount of sensitive data such as a password, a token, or a key. Such information might otherwise be put in a Pod specification or in a container image.. 41, 74

Sidecar Container It is the secondary container that runs along with the main application container within the same Pod. These containers are used to enhance or extend the functionality of the main application container by providing additional services or functionality, such as logging, monitoring, security, or data synchronization, without directly altering the primary application code. 48, 74

SNS It is a managed service that provides message delivery from publishers to subscribers. Publishers communicate asynchronously with subscribers by sending messages to a topic, which is a logical access point and communication channel. Clients can subscribe to the SNS topic and receive published messages using a supported endpoint type, such as Amazon Kinesis Data Firehose, Amazon SQS, AWS Lambda, HTTP, email, mobile push notifications, and mobile text messages (SMS). 78

SPA It is a web application or website that interacts with the user by dynamically rewriting the current web page with new data from the web server, instead of the default method of a web browser loading entire new pages. 78

SQS It is a fully managed message queuing service that makes it easy to decouple and scale microservices, distributed systems, and serverless applications. It moves data between distributed application components and helps you decouple these components. 78

SSPL It is a source-available software license introduced by MongoDB Inc. in 2018. 78

Step Functions It is a serverless orchestration service that lets you integrate with AWS Lambda functions and other AWS services to build business-critical applications. 30, 74

Service It is a method for exposing a network application that is running as one or more Pods in your cluster. The three most important service types are ClusterIP, NodePort and LoadBalancer. 35, 36, 41, 74

Service Account It provides an identity for processes that run in a Pod. 48, 74

TOR It is free and open-source software for enabling anonymous communication and directs Internet traffic via a free, worldwide, volunteer overlay network comprising more than seven thousand relays. 78

UI It is the point of human-computer interaction and communication in a device. This can include display screens, keyboards, a mouse and the appearance of a desktop. 78

URI It is a character sequence that identifies a logical or physical resource, usually connected to the internet. It distinguishes one resource from another. 78

URL It is a unique identifier used to locate a resource on the Internet. It consists of multiple parts, including a protocol and domain name, that tell a web browser how and where to retrieve a resource. 78

WSR It is a coordinator module within the WebGather system. It serves as a central storage unit for essential information, including the IPs and ports of all registered main controllers in the system. 78

WWW It refers to all websites or public pages that users can access from their local computers and other devices via the Internet. These pages and documents are interconnected by means of hyperlinks that users click on to obtain information. This information can be in different formats, including text, images, audio and video. 79

YAML It is a human-readable data serialization language that is often used for writing configuration files. 79

Acronyms

ACM AWS Certificate Manager. 42, 69

AI Artificial Intelligence. 1, 69

API Application Programming Interface. vii, xi, 4, 8, 10, 14, 16, 18, 19, 20, 22, 24, 25, 36, 47, 49, 51, 69

APK Alpine Package Keeper. 48, 69

ASCII American Standard Code for Information Interchange. 36, 69

AWS Amazon Web Services. vii, 3, 4, 5, 7, 10, 11, 13, 29, 30, 42, 43, 44, 47, 49, 50, 51, 52, 53, 57, 59, 60, 69

CDN Content Delivery Network. 60, 69

CLI Command Line Interface. 15, 21, 22, 44, 69, 71

CNCF Cloud Native Computing Foundation. xi, 4, 7, 13, 15, 18, 70

CRD Custom Resource Definition. 15, 27, 70

CRUD CREATE, READ, UPDATE and DELETE. 40, 47, 70

CSP Cloud Service Provider. 2, 7, 70

CSV Comma-separated values. 51, 70

DLS Dead Letter Sink. 45, 70

DNS Domain Name System. 36, 41, 44, 70

DOM Document Object Model. vii, 9, 22, 39, 70

EC2 Elastic Compute Cloud. 2, 29, 30, 71

ECK Elastic Cloud on Kubernetes. 26, 34, 71

ECS Elastic Container Service. 30, 71

ELK Elasticsearch Logstash Kibana. xi, 71

GCP Google Cloud Platform. 3, 5, 71

GKE Google Kubernetes Engine. 41, 50, 51, 52, 71

HPA Horizontal Pod Autoscaler. 18, 71

HTML Hypertext Markup Language. vii, 8, 10, 31, 33, 34, 36, 38, 39, 41, 44, 71

HTTP Hypertext Transfer Protocol. vii, 5, 9, 11, 19, 21, 34, 38, 39, 49, 51, 53, 55, 57, 71, 73

IP Internet Protocol. 36, 57, 72

JSON JavaScript Object Notation. 22, 25, 30, 36, 72

KPA Knative Pod Autoscaler. 18, 32, 72

MIME Multipurpose Internet Mail Extensions. 39, 72

NIST National Institute of Standards and Technology. 1, 2, 4, 72

OAuth 2.0 Open Authorization. 34, 40, 42, 73

OCI Open Container Initiative. 21, 73

OSINT Open Source Intelligence. vii, 73

RBAC Role-based access control. 48, 73

REST Representational State Transfer. 24, 25, 73

S3 Simple Storage Service. 13, 29, 30, 42, 74

SDK Software Development Kit. 20, 44, 45, 74

SNS Simple Notification Service. 13, 40, 42, 44, 53, 74

SPA Single-page Application. 22, 42, 74

SQS Simple Queue Service. 13, 29, 30, 40, 42, 44, 53, 74

SSPL Server Side Public License. 25, 74

TOR The Onion Router. 11, 48, 60, 74

UI User Interface. 30, 34, 51, 75

URI Uniform Resource Identifier. 38, 75

URL Uniform Resource Locator. vii, xi, 9, 19, 29, 30, 34, 36, 38, 39, 40, 41, 44, 48, 49, 51, 53, 54, 55, 59, 60, 75

WSR WebGather Server Registry. 29, 75

WWW World Wide Web. 8, 75

YAML Yet Another Markup Language. 15, 36, 75, 83

Appendix A

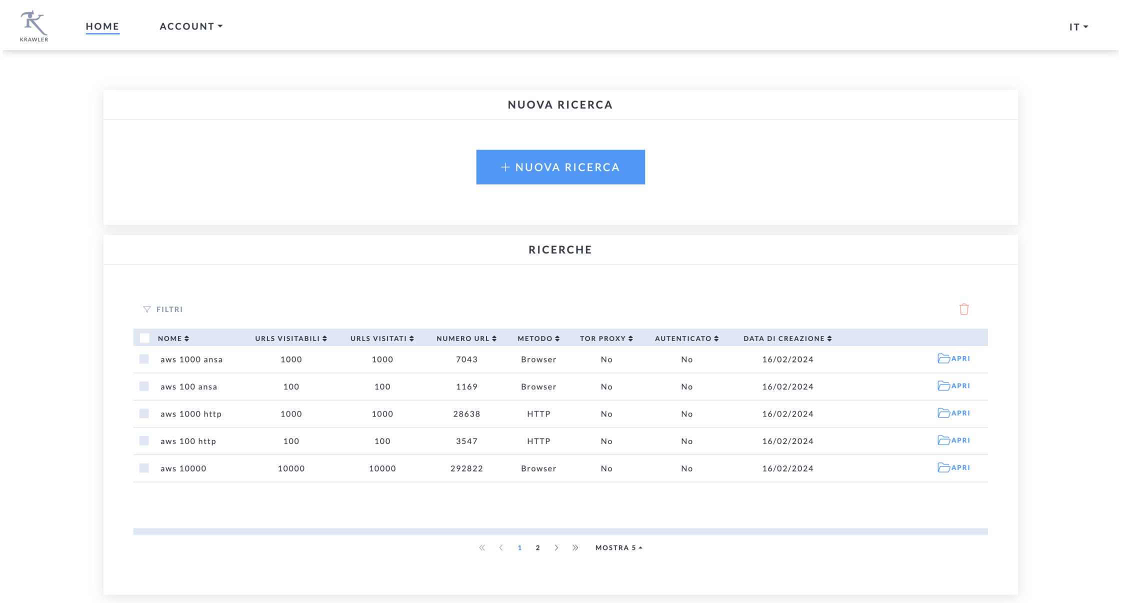
Screenshots

The frontend module provides three screens, each with specific tasks:

Figure A.1 allows the creation of new searches and the display of previous ones.

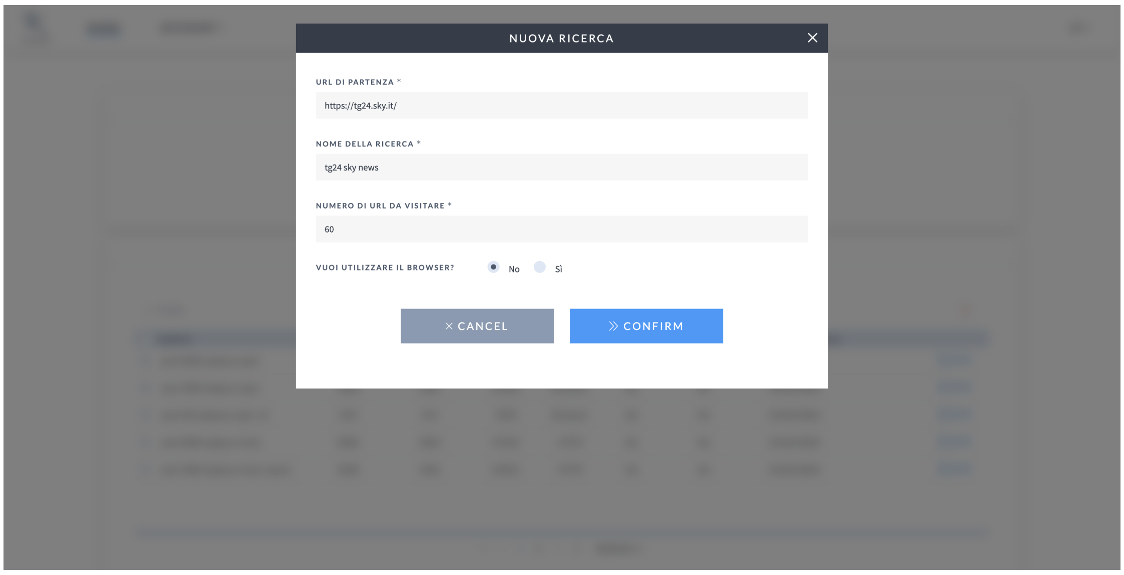
Figure A.2 shows an overlay card that allows parameters to be entered for a new search.

Figure A.3 allows all results of a specific search to be displayed and provides options for filtering them. As has already been mentioned, it will suffice to enter the keywords of interest within the HTML box to search them into the Elasticsearch index.



NOME	URLS VISITABILI	URLS VISITATI	NUMERO URL	METODO	TOR PROXY	AUTENTICATO	DATA DI CREAZIONE	
aws 1000 ansa	1000	1000	7043	Browser	No	No	16/02/2024	APRI
aws 100 ansa	100	100	1169	Browser	No	No	16/02/2024	APRI
aws 1000 http	1000	1000	28638	HTTP	No	No	16/02/2024	APRI
aws 100 http	100	100	3547	HTTP	No	No	16/02/2024	APRI
aws 10000	10000	10000	292822	Browser	No	No	16/02/2024	APRI

Figure A.1: Dashboard.



NUOVA RICERCA

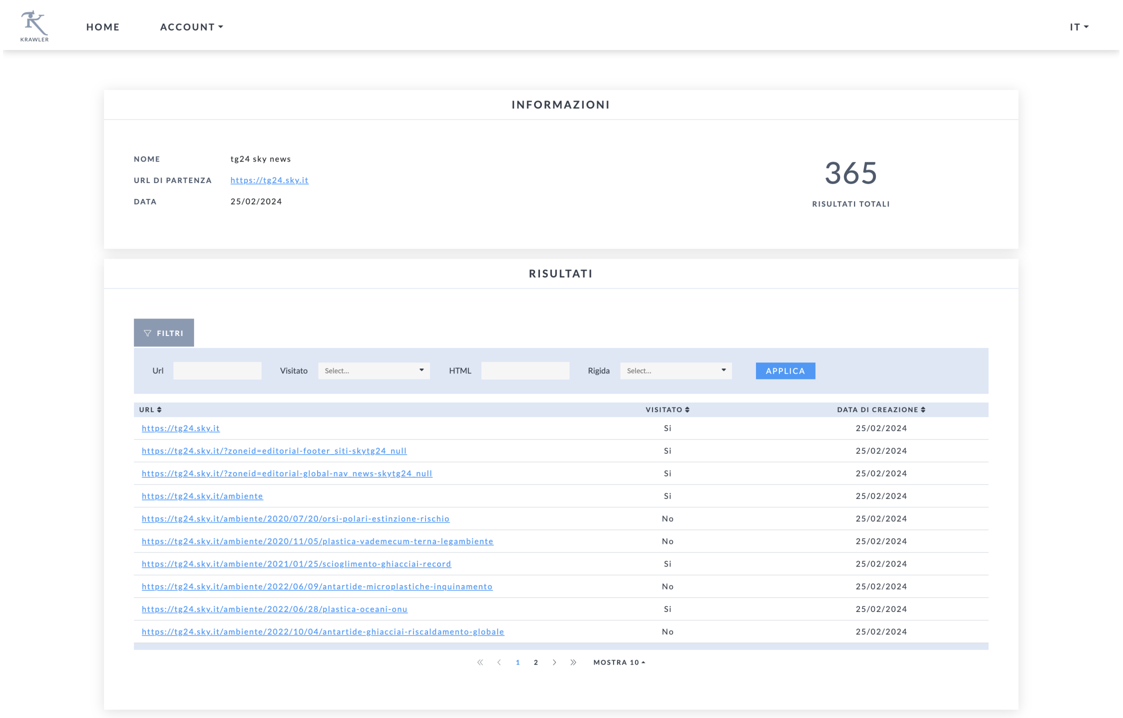
URL DI PARTENZA *
https://tg24.sky.it/

NOME DELLA RICERCA *
tg24 sky news

NUMERO DI URL DA VISITARE *
60

VUOI UTILIZZARE IL BROWSER? [✓ No] Sì

Figure A.2: Create a new search.



INFORMAZIONI

NOME	tg24 sky news
URL DI PARTENZA	https://tg24.sky.it
DATA	25/02/2024

365
RISULTATI TOTALI

RISULTATI

URL	VISITATO	DATA DI CREAZIONE
https://tg24.sky.it	Si	25/02/2024
https://tg24.sky.it/?zoneid=editorial-footer_siti-skytg24_null	Si	25/02/2024
https://tg24.sky.it/?zoneid=editorial-global-nav_news-skytg24_null	Si	25/02/2024
https://tg24.sky.it/ambiente	Si	25/02/2024
https://tg24.sky.it/ambiente/2020/07/20/orsi-polari-estinzione-rischio	No	25/02/2024
https://tg24.sky.it/ambiente/2020/11/05/plastica-vademecum-terna-legambiente	No	25/02/2024
https://tg24.sky.it/ambiente/2021/01/25/scioglimento-ghiacciai-record	Si	25/02/2024
https://tg24.sky.it/ambiente/2022/06/09/antartide-microplastiche-inquinamento	No	25/02/2024
https://tg24.sky.it/ambiente/2022/06/28/plastica-oceani-onu	Si	25/02/2024
https://tg24.sky.it/ambiente/2022/10/04/antartide-ghiacciai-riscaldamento-globale	No	25/02/2024

Figure A.3: View and filter results of a specified search.

Appendix B

Configurations

B.1 Knative

The Knative implementation includes many commands to run and resources to deploy; therefore, all have been reported in this chapter.

In the YAML files, a placeholder will be inserted when referring to the *namespace* field, which must be replaced at your convenience.

```
#!/bin/bash

export KNATIVE_VERSION="v1.13.1"
export KOURIER_VERSION="v1.13.0"
export BROKER_VERSION="v1.13.0"

echo "Setting up Serving"
kubectl apply -f https://github.com/knative/serving/releases/
   download/knative-${KNATIVE_VERSION}/serving-crds.yaml
kubectl apply -f https://github.com/knative/serving/releases/
   download/knative-${KNATIVE_VERSION}/serving-core.yaml
kubectl patch -n knative-serving configmap/config-autoscaler
   --type merge --patch '{"data":{"scale-to-zero-grace-period
   ":"10s", "allow-zero-initial-scale":"true"}}'

echo "Setting up Kourier"
kubectl apply -f https://github.com/knative/net-kourier/
   releases/download/knative-${KOURIER_VERSION}/kourier.yaml
kubectl patch configmap/config-network -n knative-serving --
   type merge --patch '{"data":{"ingress-class":"kourier.
   ingress.networking.knative.dev"}}'

echo "Setting up DNS"
kubectl apply -f https://github.com/knative/serving/releases/
   download/knative-${KNATIVE_VERSION}/serving-default-domain
   .yaml
```

```
echo "Setting up Eventing"
kubectl apply -f https://github.com/knative/eventing/releases
   /download/knative-${KNATIVE_VERSION}/eventing-crds.yaml
kubectl apply -f https://github.com/knative/eventing/releases
   /download/knative-${KNATIVE_VERSION}/eventing-core.yaml

echo "Setting up RabbitMQ"
kubectl apply -f https://github.com/cert-manager/cert-manager
   /releases/download/v1.5.4/cert-manager.yaml
echo "Wait for cert—manager"
sleep 120
kubectl apply -f https://github.com/rabbitmq/messaging-
   topology-operator/releases/latest/download/messaging-
   topology-operator-with-certmanager.yaml
kubectl apply -f https://github.com/rabbitmq/cluster-operator
   /releases/latest/download/cluster-operator.yml
kubectl apply -f https://github.com/knative-extensions/
   eventing-rabbitmq/releases/download/knative-${
   BROKER_VERSION}/rabbitmq-broker.yaml

echo "All done!"
```

Listing B.1: Bash script used to deploy Knative Serving, Eventing and RabbitMQ components.

```yaml
apiVersion: rabbitmq.com/v1beta1
kind: RabbitmqCluster
metadata:
  name: rabbitmq
  namespace: NAMESPACE
spec:
  replicas: 1
  resources:
    requests:
      cpu: 500m
      memory: 500Mi
    limits:
      cpu: 2000m
      memory: 2Gi
  override:
    statefulSet:
      spec:
        template:
          spec:
            containers:
            - name: rabbitmq
              env:
              - name: ERL_MAX_PORTS
                value: "4096"
```

Listing B.2: How to deploy a RabbitMQ cluster.

```
apiVersion: eventing.knative.dev/v1alpha1
kind: RabbitmqBrokerConfig
metadata:
  name: kopjra-config
  namespace: NAMESPACE
spec:
  rabbitmqClusterReference:
    name: rabbitmq
    namespace: NAMESPACE
  queueType: quorum
---
apiVersion: eventing.knative.dev/v1
kind: Broker
metadata:
  name: kopjra
  namespace: NAMESPACE
  annotations:
    eventing.knative.dev/broker.class: RabbitMQBroker
    rabbitmq.eventing.knative.dev/cpu-request: 500m
    rabbitmq.eventing.knative.dev/memory-request: 512Mi
    rabbitmq.eventing.knative.dev/cpu-limit: 1000m
    rabbitmq.eventing.knative.dev/memory-limit: 1024Mi
spec:
  config:
    apiVersion: eventing.knative.dev/v1alpha1
    kind: RabbitmqBrokerConfig
    name: kopjra-config
  delivery:
    deadLetterSink:
      ref:
        apiVersion: serving.knative.dev/v1
        kind: Service
        name: event-display
        namespace: NAMESPACE
    backoffDelay: PT1S
    backoffPolicy: exponential
    retry: 3
```

Listing B.3: How to deploy a RabbitMQ broker with its configuration.

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: event-display
  namespace: NAMESPACE
spec:
  template:
    metadata:
      annotations:
        autoscaling.knative.dev/initial-scale: "0"
    spec:
      containers:
        - image: gcr.io/knative-releases/knative.dev/eventing/
          cmd/event_display
          securityContext:
            allowPrivilegeEscalation: false
            readOnlyRootFilesystem: true
            runAsNonRoot: true
            capabilities:
              drop:
                - ALL
            seccompProfile:
              type: RuntimeDefault
```

Listing B.4: Knative Service of event-display utility.

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  labels:
    app: kn-kapturer-be
    env: dev
  name: kn-kapturer-be
  namespace: NAMESPACE
spec:
  template:
    metadata:
      annotations:
        autoscaling.knative.dev/min-scale: "1"
    spec:
      containers:
        - env:
          - name: ENTRYPOINT
            value: ./dist/index.js
          envFrom:
          - configMapRef:
              name: kabe-conf-dev-xxx
          - secretRef:
              name: kabe-secret-dev-xxx
          name: kn-kapturer-be
          image: eu.gcr.io/REGISTRY/kn-kapturer-backend:v1.0.0
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 8080
              protocol: TCP
          securityContext:
            seccompProfile:
              type: RuntimeDefault
            allowPrivilegeEscalation: false
            capabilities:
                drop:
                  - ALL
              readOnlyRootFilesystem: true
              runAsNonRoot: true
            resources:
              requests:
                memory: 500Mi
                cpu: 800m
              limits:
                memory: 1000Mi
                cpu: 1500m
            livenessProbe:
              httpGet:
                path: /v1
                port: 8080
              initialDelaySeconds: 30
              periodSeconds: 60
              failureThreshold: 5
            readinessProbe:
              httpGet:
                path: /v1
                port: 8080
              initialDelaySeconds: 30
              periodSeconds: 60
              failureThreshold: 5
---
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  labels:
    app: kn-kapturer-clk
    env: dev
  name: kn-kapturer-clk
  namespace: NAMESPACE
spec:
  template:
    metadata:
      annotations:
        autoscaling.knative.dev/min-scale: "1"
    spec:
      containers:
        - env:
          - name: ENTRYPOINT
            value: ./dist/index-clk.js
          envFrom:
          - configMapRef:
              name: kabe-conf-dev-xxx
          - secretRef:
              name: kabe-secret-dev-xxx
          name: kn-kapturer-clk
          image: eu.gcr.io/REGISTRY/kn-kapturer-backend:v1.0.0
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 8080
              protocol: TCP
          securityContext:
            seccompProfile:
              type: RuntimeDefault
            allowPrivilegeEscalation: false
            capabilities:
              drop:
                - ALL
            readOnlyRootFilesystem: true
            runAsNonRoot: true
          resources:
            requests:
              memory: 64Mi
              cpu: 50m
            limits:
              memory: 128Mi
              cpu: 100m
```

Listing B.5: Knative Services of the backend module.

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  labels:
    app: kn-krawler
    env: dev
  name: kn-krawler
  namespace: NAMESPACE
spec:
  template:
    metadata:
      annotations:
        autoscaling.knative.dev/target-utilization-percentage
           : "100"
        autoscaling.knative.dev/class: kpa.autoscaling.knative
           .dev
        autoscaling.knative.dev/metric: concurrency
        autoscaling.knative.dev/min-scale: "0"
        autoscaling.knative.dev/max-scale: "100"
    spec:
      containerConcurrency: 1
      containers:
        - name: kn-krawler
          image: eu.gcr.io/REGISTRY/kn-krawler:v1.0.1
          imagePullPolicy: IfNotPresent
          envFrom:
            - configMapRef:
                name: kabe-conf-dev-xxx
            - secretRef:
                name: kabe-secret-dev-xxx
          env:
            - name: FUNC_LOG_LEVEL
              value: "info"
            - name: CHROMIUM_PATH
              value: "/usr/bin/chromium-browser"
          securityContext:
            seccompProfile:
              type: RuntimeDefault
            allowPrivilegeEscalation: false
            capabilities:
              drop:
                - ALL
            readOnlyRootFilesystem: true
            runAsNonRoot: true
          resources:
            requests:
              memory: 1000Mi
              cpu: 500m
            limits:
              memory: 1280Mi
              cpu: 750m
```

Listing B.6: Knative Services of the core module.

```
apiVersion: eventing.knative.dev/v1
kind: Trigger
metadata:
  name: crawls-trigger
  namespace: NAMESPACE
  annotations:
    rabbitmq.eventing.knative.dev/cpu-request: 100m
    rabbitmq.eventing.knative.dev/memory-request: 100Mi
    rabbitmq.eventing.knative.dev/cpu-limit: 300m
    rabbitmq.eventing.knative.dev/memory-limit: 300Mi
    rabbitmq.eventing.knative.dev/parallelism: "100"
spec:
  broker: kopjra
  filter:
    attributes:
      type: dev.kapturer.crawls
      source: kn-kapturer-be
  subscriber:
    ref:
      apiVersion: serving.knative.dev/v1
      kind: Service
      name: kn-krawler
      namespace: NAMESPACE
---
apiVersion: eventing.knative.dev/v1
kind: Trigger
metadata:
  name: results-trigger
  namespace: NAMESPACE
  annotations:
    rabbitmq.eventing.knative.dev/cpu-request: 100m
    rabbitmq.eventing.knative.dev/memory-request: 100Mi
    rabbitmq.eventing.knative.dev/cpu-limit: 300m
    rabbitmq.eventing.knative.dev/memory-limit: 300Mi
    rabbitmq.eventing.knative.dev/parallelism: "100"
spec:
  broker: kopjra
  filter:
    attributes:
      type: dev.krawler.results
      source: kn-krawler
  subscriber:
    ref:
      apiVersion: serving.knative.dev/v1
      kind: Service
      name: kn-kapturer-be
      namespace: NAMESPACE
    uri: /v1/internal/results
---
apiVersion: eventing.knative.dev/v1
kind: Trigger
metadata:
  name: response-trigger
  namespace: NAMESPACE
  annotations:
    rabbitmq.eventing.knative.dev/cpu-request: 100m
    rabbitmq.eventing.knative.dev/memory-request: 100Mi
    rabbitmq.eventing.knative.dev/cpu-limit: 300m
    rabbitmq.eventing.knative.dev/memory-limit: 300Mi
    rabbitmq.eventing.knative.dev/parallelism: "100"
spec:
  broker: kopjra
  filter:
    attributes:
      type: dev.krawler.response
      source: kn-krawler
  subscriber:
    ref:
      apiVersion: serving.knative.dev/v1
      kind: Service
      name: event-display
      namespace: NAMESPACE
```

Listing B.7: Trigger definitions to subscribe event to a sink.

B.2 Monitoring

The collection and storage of metrics generated during the various tests was made possible by the monitoring platform, which includes several applications. The commands required for deploying and configuring the latter are listed below.

A complete reading of the configuration files is recommended before running any commands.

```
#!/bin/bash

kubectl create namespace metrics

# Deploy Prometheus Stack
helm repo add prometheus-community https://prometheus-
   community.github.io/helm-charts
helm repo update
helm install prometheus prometheus-community/kube-prometheus-
   stack -n metrics -f prom-values.yaml
## Expose the services
## kubectl port-forward -n metrics svc/prometheus-kube-
   prometheus-prometheus 9090:9090
## kubectl port-forward -n metrics svc/prometheus-grafana
   3000:80
## Grafana credentials: {"user":"admin", "password":"prom-
   operator"}

# Deploy Knative and RabbitMQ monitoring stuff
kubectl apply -f https://raw.githubusercontent.com/knative-
   extensions/monitoring/main/servicemonitor.yaml
kubectl apply -f https://raw.githubusercontent.com/knative-
   extensions/monitoring/main/grafana/dashboards.yaml
kubectl apply -f https://raw.githubusercontent.com/rabbitmq/
   cluster-operator/main/observability/prometheus/monitors/
   rabbitmq-servicemonitor.yml
kubectl apply -f https://raw.githubusercontent.com/rabbitmq/
   cluster-operator/main/observability/prometheus/monitors/
   rabbitmq-cluster-operator-podmonitor.yml
## Manual import of the following dashboards about RabbitMQ
## 10991
## 11340


# Deploy InfluxDB
helm repo add influxdata https://helm.influxdata.com/
helm upgrade -i influxdb influxdata/influxdb2 -n metrics -f
   inf-values.yaml
## Copy the password that the previous command provides
## InfluxDB credentials: {"user":"admin", "password":"the
   password copied before"}
## Expose the service
## kubectl port-forward -n metrics svc/influxdb-influxdb2
   8086:80
## Create a bucket
## Create a custom API Token following https://docs.
   influxdata.com/influxdb/cloud/admin/tokens/create-token/
   with WRITE to the created bucket and READ from telegraf
   agent permissions
## Create another API Token with only read access permission
   to all buckets
## Follow https://grafana.com/docs/grafana/latest/datasources
   /influxdb/ to create InfluxDB data source on Grafana

export API_TOKEN=""
export BUCKET_NAME=""

# Deploy Telegraf
helm repo add telegraf https://helm.influxdata.com/
helm upgrade -i telegraf influxdata/telegraf -n metrics -f
   tel-values.yaml
```

Listing B.8: How to deploy Prometheus Stack, InfluxDB and Telegraf.

```
kube-state-metrics:
  metricLabelsAllowlist:
   - jobs=[*]
   - pods=[*]
   - deployments=[app.kubernetes.io/name,app.kubernetes.io/
     component,app.kubernetes.io/instance]
prometheus:
  prometheusSpec:
    serviceMonitorSelectorNilUsesHelmValues: false
    podMonitorSelectorNilUsesHelmValues: false
    remoteWrite:
      - url: "http://telegraf.metrics.svc:8080/telegraf"
grafana:
  sidecar:
    dashboards:
      enabled: true
      searchNamespace: ALL
```

Listing B.9: prom-values.yaml

```
resources:
  limits:
    cpu: 4000m
    memory: 8Gi
  requests:
    cpu: 2000m
    memory: 6Gi
```

Listing B.10: inf-values.yaml

```
resources:
  requests:
    memory: 500Mi
    cpu: 500m
  limits:
    memory: 1Gi
    cpu: 1000m
config:
  agent:
    interval: "10s"
    round_interval: true
    metric_batch_size: 50000
    metric_buffer_limit: 100000
    collection_jitter: "0s"
    flush_interval: "10s"
    flush_jitter: "0s"
    precision: ""
    debug: false
    quiet: false
    logfile: ""
    hostname: "$HOSTNAME"
    omit_hostname: false
  inputs:
    - http_listener_v2:
        service_address: ":8080"
        paths: ["/telegraf"]
        data_format: "prometheusremotewrite"
  outputs:
    - influxdb_v2:
        urls:
          - "http://influxdb-influxdb2.metrics.svc:80/"
        token: "$API_TOKEN"
        bucket: "$BUCKET_NAME"
        organization: "influxdata"
        timeout: "5s"
        insecure_skip_verify: false
metrics:
  health:
    enabled: true
    service_address: "http://:8888"
    threshold: 5000
  internal:
    enabled: false
    collect_memstats: false
```

Listing B.11: tel-values.yaml