ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Department of Computer Science and Engineering

Second Cycle Degree in
Computer Engineering

Dynamic and Integrated HPC Cluster Provisioning with Cloud Native Technologies

Dissertation in Mobile Systems

Supervisor
Prof. Paolo Bellavista

Co-supervisor
Eng. Michele Gazzetti

Presented by
Riccardo Bovinelli

in collaboration with
IBM Research Ireland

Session March 2026

Academic Year 2024/2025

Contents

List of Figures **vii**

Abstract **ix**

Introduction **1**

Thesis Organization . . . 2

1 Motivations and Contributions **5**

1.1 Motivations . . . 5
1.2 Contributions . . . 7

2 Related Work **9**

2.1 Background . . . 9
2.1.1 HPC . . . 9
2.1.2 Cloud computing . . . 12
2.1.3 Converged Computing . . . 15
2.2 State of the Art . . . 17

3 Employed Technologies **21**

3.1 Kubernetes . . . 21
3.2 Openshift . . . 25
3.3 Slurm . . . 27
3.4 Slinky . . . 31
3.5 Cluster API . . . 33
3.6 Metal^3 . . . 35
3.6.1 metal3-dev-env . . . 37
3.7 cluster-api-bootstrap-virtual-kubelet . . . 38
3.7.1 Slurm detach handler . . . 39
3.8 Additional Technologies . . . 40
3.8.1 Cadvisor . . . 40

3.8.2 Prometheus 41
3.9 Selected Benchmarks 42
3.9.1 HPL 42
3.9.2 iPerf 43
3.9.3 OSU Micro Benchmarks 45

4 SliMe Architecture 47
4.1 Preliminary Evaluation 48
4.1.1 Proposed solution 48
4.2 Injection Webhook - Integration with Slinky 51
4.2.1 Kubernetes Admission Webhooks 52
4.2.2 Cert-manager 54
4.2.3 Proposed solution 56
4.3 Scaling Webhook - Autoscaling integration 58
4.3.1 Autoscalers comparison 58
4.3.2 Proposed solution 63
4.3.3 Refined architecture overview 70

5 SliMe Implementation and Deployment 77
5.1 Preliminary Evaluation 78
5.1.1 Deploying Slinky on OpenShift 78
5.1.2 Deploying the proposed architecture 80
5.1.3 Enabling hybrid architecture 88
5.2 Injection Webhook - Integration with Slinky 91
5.2.1 Implementation details 91
5.2.2 Deployment 93
5.3 Scaling Webhook - Autoscaling integration 97
5.3.1 Refined architecture implementation 98

6 Performance Evaluation 101
6.1 Comparing bare-metal nodes and Slinky pods 101
6.1.1 HPL 101
6.1.2 iPerf 107
6.1.3 OSU Micro Benchmarks 112
6.2 Injection Webhook 115
6.2.1 Performance evaluation 115
6.3 Scaling Webhook 115
6.3.1 Performance evaluation 116
6.3.2 A scaling up example 118

Conclusions 121

Future Work 122

Bibliography **125**

A BIND configuration **133**

A.1 Configuration files 133

A.2 Iptables 135

List of Figures

1.1 Visualization of the *control knob* 7

3.1 Main components of a Kubernetes cluster 22
3.2 Main components of a Slurm cluster 28
3.3 Components involved in a Slurm cluster deployed by Slinky 31

4.1 Employed architectures: bare metal vs. pods 48
4.2 Simplified flow for the admission requests and admission webhooks . . 52
4.3 Simplified schema for the injection webhook workflow 57
4.4 KEDA architecture 60
4.5 Effect of the cycle on the pending job number 66
4.6 Effect on nodes and jobs with different job lengths 67
4.7 Effect of the refined prometheus query 68
4.8 Simplified architecture for autoscaling integration with CAPI 69
4.9 Refined architecture overview 75

5.1 Relationships between entities in preliminary architecture 87
5.2 Hybrid architecture in Slinky 89
5.3 Ownership reference chain for Injection Webhook 92
5.4 In-cluster PKI infrastructure 95

6.1 HPL performance for different problem sizes on pod workers 102
6.2 Pod workers: total CPU usage on a node and CPU usage by HPL job 103
6.3 Detailed view of the results in Figure 6.2 104
6.4 Bare-metal: total CPU usage on a node and CPU usage by HPL job . 105
6.5 Detailed view of the results in Figure 6.2 and Figure 6.4 106
6.6 HPL result comparison between pod workers and bare-metal workers 107
6.7 iPerf results for different number of parallel processes 108
6.8 iPerf results for pod workers with high number of parallel processes . 110
6.9 iPerf results when using `hostNetwork: true` 111
6.10 Latency measured with the `alltoall` OSU benchmark 112

6.11 Latencies in μs for the `alltoall` OSU benchmark 113
6.12 Latencies in μs for the `allreduce` OSU benchmark 114
6.13 Distribution of individual call latencies for the Injection Webhook . . 116
6.14 Distribution of individual call latencies for the Scaling Webhook . . . 117
6.15 Total time required for a scaling up instance 118
6.16 Detailed view of the total time required for a scaling up instance . . . 119

Abstract

High Performance Computing (HPC) systems are traditionally deployed as clusters of dedicated machines managed by specialized workload schedulers. On the other hand, cloud native platforms based on container orchestration technologies provide automated deployment and dynamic infrastructure management. The increasing interest in integrating these two paradigms has motivated research on architectures that combine HPC workload managers with cloud-native orchestration environments, and several challenges remain open.

This thesis investigates the feasibility and performance side effects of deploying an HPC cluster whose control plane runs inside a Slinky instrumented Kubernetes environment and worker nodes are dynamically provisioned as bare metal machines using the Kubernetes Cluster API. The work aims to determine whether such an architecture can manage HPC workloads while enabling automated deployment and dynamic scaling mechanisms.

An experimental architecture was designed to enable the aforementioned environment, using the Metal^3 infrastructure provider for Cluster API and an original solution, which is central to this thesis' work. It is called SliMe and addresses the needed technological integrations together with the enabling of dynamic provisioning. This architecture represents another step towards the convergence between traditional HPC systems and cloud computing.

Performance was evaluated both for latency introduced by SliMe and through a set of benchmarks measuring computational performance, network bandwidth and MPI communication latency in the proposed architecture.

Introduction

High Performance Computing (HPC) systems are designed to execute workloads that require large amounts of processing power and efficient communication between multiple concurrent computing nodes. These systems are organized as *clusters* composed of several machines interconnected through high-speed networks and coordinated by specialized workload managers. In this context, performance is influenced not only by the computational capabilities of the processors but also by the efficiency of the underlying communication infrastructure and by the mechanisms used to allocate resources to running jobs. The management of HPC clusters is traditionally performed through Resource and Job Management Systems (RJMSs) such as Slurm. These systems are responsible for scheduling jobs, allocating resources and managing the job queues. A workload manager maintains a queue of submitted jobs and assigns resources to them according to scheduling policies that maximize resource efficiency while considering fairness. In a typical environment, there is a central controller that assigns jobs to nodes, while worker nodes execute the tasks assigned to them.

Instead, cloud computing infrastructures have evolved with a different operational model. Instead of relying on HPC-like static clusters, cloud platforms are designed to dynamically provision resources according to demand. Cloud systems nowadays rely on containerization technologies, which package applications together with their runtime environment into isolated and self-contained environments that enable execution on heterogeneous contexts. Container orchestration platforms are the lifecycle managers of containers, coordinating their execution across clusters of machines and providing mechanisms for networking and for scaling applications according to workload conditions. Kubernetes is one of the most widely adopted container orchestration platforms. Applications are deployed in the form of pods and *controllers* manage the creation, scaling and deletion of such pods.

HPC and cloud computing have historically developed as separate paradigms, but recent developments, such as the increasing complexity of the scientific workloads and the advent of AI, have led to increasing interest in integrating their environments

to obtain a "best of both worlds" scenario, referred to as *convergence computing* [72]. The combination of these two approaches raises technical questions related to the interaction between dynamic, containerized environments and HPC applications that are sensitive to performance. In particular, container orchestration systems introduce additional software layers that manage networking, resource isolation and scheduling, which may influence the performance results of workloads executed on the underlying infrastructure.

A relevant aspect that is central for the following work concerns the difference in management of cluster resources. Traditional HPC clusters are generally composed of a fixed set of compute nodes that are permanently allocated to the system. In contrast, cloud infrastructures allow resources to be provisioned and released dynamically. Integrating these different paradigms requires solutions that enable HPC clusters to operate in environments where the set of available nodes changes dynamically over time, following a cloud-like philosophy. Such mechanisms must ensure that the workload manager remains able to see a consistent cluster state while external infrastructure components are responsible for provisioning and managing the lifecycle of nodes.

The interaction between HPC workload managers and container orchestration systems represents an architectural challenge that involves multiple components at different levels of software abstraction. Scheduling mechanisms, networking infrastructures, and resource provisioning systems need to work in a coordinated way. Investigating how these components interact in integrated environments such as the one proposed in this thesis is of interest to understand if cloud native technologies can be adopted together with HPC contexts, adding value to the requirements of modern scientific computing workloads.

Thesis Organization

This work follows a structure organized into several chapters targeting different aspects of the problem domain.

- Chapter 1 presents the motivations behind the research, defines the research questions that led the development of the presented work and outlines the specific contributions of this thesis.
- Chapter 2 provides a background on the central topics of this thesis, which are HPC and cloud computing, presenting the concept of convergence computing and the state of the art in this field.
- Chapter 3 describes the main technologies that are used during the design,

implementation and evaluation phases of the presented work, giving the necessary context to understand the architectural choices.

- Chapter 4 describes the architecture of the proposed solution, SliMe, in an incremental fashion following the research questions.
- Chapter 5 describes the implementation choices behind the components of SliMe and the deployment process and necessities that enabled it to function.
- Chapter 6 presents the results of the conducted experimental evaluations, first a comparison between the performance of HPC workloads running on bare metal nodes and Kubernetes pods, then an evaluation of the performance of the core components of SliMe.

Chapter 1

Motivations and Contributions

1.1 Motivations

This thesis has the fundamental aim of investigating the increase of flexibility in the management of Slurm clusters utilizing cloud native technologies, keeping a particular focus on Kubernetes, with the outcome of allowing for a further convergence between HPC and Cloud in the field of Convergence Computing.

In particular, the motivation behind this work is to explore the potential of Kubernetes as a platform for deploying and managing HPC clusters on bare-metal nodes, and to provide a solution that can enable HPC users to leverage the benefits of Kubernetes in terms of operational flexibility while still achieving the performance given by the bare-metal nodes. Solutions that allow running HPC workloads on Kubernetes are already available, e.g. Slinky [68], but they are designed to run HPC workloads on top of Kubernetes, thus inside containers.

The adoption of containerization in HPC has been relatively slow over the years [35], both due to potentially flawed perceptions of cloud computing [43] and for a variety of technical reasons. They can be security-related, such as containers being children of a root process in older versions of Docker and thus contradicting HPC center policies [1][83], vulnerabilities in the use of the `user` namespaces [83]. Also, they can be compatibility-related, since Application Binary Interfaces (ABIs) of the running containers must be ensured, otherwise they can affect scientific results [56]. Futhermore, applications written in languages still widely used in HPC, such as Fortran [34] are known to lack ABI compatibility [17]. Hardware access is another important aspect, since HPC workloads often require direct access to hardware resources such as GPUs and high-speed interconnects, containerization may introduce problems in accessing these resources [56][7].

For the aforementioned reasons, an exploration towards the integration of HPC with cloud native technologies that can still provide the bare metal properties and performance of traditional HPC clusters, but with the flexibility and the unification of control planes that Kubernetes can provide, could be of interest for the HPC community.

To achieve this goal, the technology of choice to be integrated during this thesis is the aforementioned Slinky. This is because it offers the possibility to run the typical Slurm architectural workload inside pods, not only the controller, but even the accounting system. Implementing this kind of convergence at node management and possibly at workload management can allow to be more efficient in terms of number of nodes used. It is indeed possible to run the Slurm head of the Slurm cluster, i.e. the node in charge of running `slurmctld`, inside a container belonging to a pod, without needing to use a whole node to run it, and instead reusing that node as an additional Slurm worker one. And, in a similar fashion, this holds for all the other infrastructural components that do not need to be run on a whole node, thus increasing the overall amount of available resources for the workloads.

In addition, one of the main problems that arises with current solutions is that resource contention is very likely to happen if the scheduling of the pods is left to the standard Kubernetes scheduler and no careful precautions are taken. In fact, Slurm nodes in the form of pods may be scheduled on the same node: this means that they are sharing the same physical resources. Therefore, when a Slurm job is submitted, the requested resources are allocated on the Slurm nodes by the Slurm workload manager, but if those containerized nodes share the same underlying physical resources then there is no more exclusivity in the said resource allocation, even if from the Slurm workload manager perspective every task in the job has its own reserved resources. This means that the different pods may start to compete to obtain the resources that should have been univocally allocated to them and this clearly poses a problem in terms of performance.

Another potential aspect of interest to investigate is the possibility to introduce an autoscaling solution to better handle heavy and non-constant workloads, Slinky already proposes a solution based on KEDA and HPA that is able to scale workers but just in the form of pods and, furthermore, node repurposing is not allowed since Slinky does not work with the concept of physical (Slurm) nodes. Instead with the introduction of autoscaling, nodes can be dynamically attached to Kubernetes or Slurm clusters on the basis of the computational needs that the current workload introduces, saving a considerable amount of time to system administrators.

Moreover, it can be interesting to explore the possibility of enabling an environment in which both bare metal nodes and nodes hosted inside pods are present, also called

a *hybrid* configuration. In fact, it is possible that certain workload configurations have better performance if they are run on bare metal nodes, while other configurations may be more efficient if run on Kubernetes pods, or benefit from a mixture of bare metal nodes and Kubernetes pods. The current available solutions do not allow for this behavior to happen from an environment exclusively based on Kubernetes, so it can be interesting to enable a solution that provides a sort of *control knob* (see Figure 1.1) that enables to fine-tuning the amount of pod worker nodes and bare metal worker nodes to have in a certain moment. This would allow easy deployment for these kinds of experiments and explore the performance of different workload combinations on different configurations of nodes.

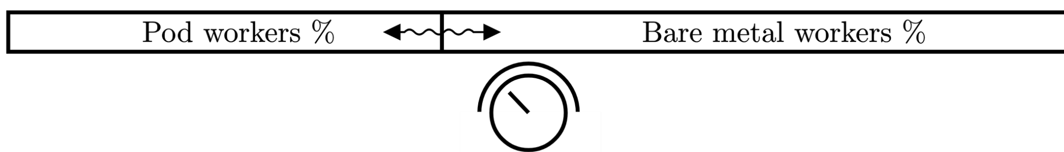


Figure 1.1: Visualization of the *control knob*

So, in order to explore the motivations described above, the following Research Questions (RQ) are defined:

RQ1 Is there any effective advantage in terms of performance in running HPC workloads on bare-metal nodes instead of Kubernetes pods?

RQ2 Is it possible to run the controller of the cluster inside a pod and connecting a set of bare metal worker nodes to it? Is Slinky a good candidate for this kind of integration?

RQ3 Is it possible to automate the deployment of bare metal worker nodes with a pod-based controller?

RQ4 Is it possible to make it a dynamically provisioned cluster, i.e. integrating an autoscaling solution that chooses an adequate configuration?

1.2 Contributions

From the investigation on all the posed **RQ**s, the contributions of this thesis can be summarized as the following:

- A comparative evaluation of the performance of Slurm clusters presenting different configurations of worker nodes, i.e. bare metal nodes and Kubernetes pods.

- The original SliMe (a **Sli**nky-bare**Me**tal connector), a new system that allows deploying Slurm clusters having the controller running inside a Kubernetes pod and the worker nodes running on bare metal, with the possibility to dynamically change the number of worker nodes (see Section 2.1.3). SliMe is enabled by two main components:
  - Our original *Injection Webhook*, a solution to deploy Slurm bare metal nodes that automatically join a Slurm cluster whose controller runs inside a Kubernetes pod, deployed by Slinky.
  - Our original *Scaling Webhook*, which allows dynamic provisioning and deprovisioning of bare metal nodes based on the workload needs, enabling the possibility of autoscaling for the aforementioned cluster.

Chapter 2

Related Work

2.1 Background

To correctly lead the reader to the topic of this thesis, it is necessary to introduce the main background topics that are relevant to the work. In particular, the two paradigms that are the foundations for this dissertation, which are the concepts of High Performance Computing and Cloud Computing, will be introduced. The interactions between these two seemingly discordant paradigms are the main focus of this thesis and thus it is important to understand the characteristics of each of them to better grasp the challenges and benefits that their convergence can bring.

2.1.1 HPC

HPC, or High Performance Computing is a field of endeavor that relates to all facets of technology, methodology and application associated with achieving the greatest computing capability possible at any time and technology [74]. It is a field of computing that focuses on the development and use of powerful computational systems, typically with the objective of solving complex problems that require significant computational resources. The performance of HPC systems is often measured in terms of FLOPS (Floating Point Operations Per Second). HPC systems are used in a wide range of computationally intensive scientific applications such as climate modeling, fluid dynamics and, in the recent years, machine learning and artificial intelligence.

An HPC system is typically composed of a large number of compute nodes, which are interconnected by typically high-speed network, assuming the name of HPC *cluster*. Each compute node is a powerful computer in its own right, often equipped with multiple CPUs, large amounts of memory, and especially in recent years, specialized

hardware accelerators such as GPUs. The nodes work together to perform typically parallel computations at a high scale. CPUs in HPC systems have a great number of cores for this reason, together with large caches and support for vector instruction sets (such as AVX on Intel and AMD x86 processors). The interconnection network is a critical component of an HPC system, as it enables the compute nodes to communicate and share data efficiently. High-speed networks such as InfiniBand and high-performance Ethernet technologies are commonly used in HPC clusters to minimize latency and maximize bandwidth. HPC systems also typically include a storage subsystem, with high-performance to handle high-scale data access, comprising distributed and parallel file systems, such as Lustre, and high speed storage devices (e.g. NVMe SSDs).

The history of HPC begins with vector supercomputers. Seymour Cray's CDC 6600 (1964) was the first supercomputer in history, followed by the Cray-1 in 1976. The architectural template of supercomputers, until the 80's, consisted of a small number of fast processors operating on vectors of data, also known as vector processors, that could operate on a shared memory, with IPC (Inter Process Communication) leveraging it [27]. During the 80's the attention for supercomputer architecture shifted towards distributed computing [21], and in the 90's the trend to move away from expensive and specialized proprietary parallel supercomputers towards networks of workstations was initiated, due to the availability of commodity components with high performance, and the concept of *commodity supercomputing* appeared [6]. The concept of having many interconnected nodes was consolidated during the 2000s, with the performance of supercomputers spiking to break the barrier of 1 PetaFLOP [21] in 2008 and of the exaFLOP in 2022, with the Frontier supercomputer [10]. In the last 20 years, as aforementioned, special-purpose hardware accelerators (such as GPUs) were added to the HPC landscape to offload some of the computation, increasing the performance.

The interconnection network is undoubtedly one of the most critical components of HPC clusters from the performance perspectives. One of the dominant interconnect technologies in modern HPC is InfiniBand. It provides low latency and high bandwidth communication through a switched fabric topology. A key feature of InfiniBand is its native support for Remote Direct Memory Access (RDMA). RDMA allows a process on one node to read from or write to the memory of a process on another node without involving either the remote or local CPU, since the operations get offloaded to the NICs, nor the operating system kernels [4].

The way in which an HPC cluster is interconnected defines a topology. The network topology of a cluster is an important thing to take into consideration since it determines the network performance of a cluster, which influence the scalability and performance of applications that relies heavily on communication. Common

topologies include fat-tree, used in most InfiniBand deployments, dragonfly, used in Cray systems such as the Cray XC series, and torus networks (originally used in IBM Blue Gene systems) [31].

Due to the parallel nature of current HPC systems, a core aspect is parallel programming. For this purpose, there are some well established paradigms, of which the Message Passing Interface (MPI) is the de-facto standard for distributed memory parallelism, it allows process to communicate with other processes by explicitly sending and receiving messages, both inside and outside the running node. MPI provides both point-to-point operations (e.g. `MPI_Send`, `MPI_Recv`) and collective operations (e.g. `MPI_Allreduce`, `MPI_Alltoall`) that specify data movement across groups of processes. There are different implementations of MPI, among which there are OpenMPI and MVAPICH, that also offer support for RDMA backed messaging operations. Regarding shared-memory parallelism within a single node, OpenMP provides a model based on directives: it is possible to annotate loops and specific code regions with compiler directives (`#pragma`) that tell the compiler to parallelize them across different threads that can be executed in parallel. Since HPC nodes contain even hundreds of CPU cores, hybrid programs that leverage both MPI and OpenMP programs are common: MPI handles inter-node communication while OpenMP manages intra-node parallelism [64]. This hybrid approach reduces the number of MPI processes, and thus the volume of inter-process messages, while still using all the available resources.

The increasing importance of hardware accelerators on modern HPC clusters, especially GPUs, has led to the development of dedicated programming models, for example CUDA [59] for NVIDIA GPUs, which provide control over accelerator execution. Higher level approaches, also based on directives such as OpenACC and the OpenMP target offloading, allow developers to annotate existing code for GPU execution with minimal refactoring. These models allow HPC applications to leverage the high parallelism of GPU architectures to increase throughput for specific computationally intensive tasks.

Another key concept in HPC is the workload manager. It is a tool responsible for allocating compute resources to user-submitted jobs, enforcing fair share policies while scheduling and maximizing cluster utilization to reduce resource waste and increase throughput. Users submit job scripts that specify resource requirements (number of nodes, CPUs, memory, wall time, needed hardware, ...) and the workload manager queues and dispatches them according to scheduling policies. Examples of workload managers are Torque and Slurm (see Section 3.3). The latter, formerly known as Simple Linux Utility for Resource Management, is the most widely deployed workload manager in modern HPC, used on the majority of the systems in the TOP500 list [15]. The workload manager is a critical piece of the HPC software stack because

it has the delicate task of being the intermediate between user requirements and the available physical resources. The scheduling decisions directly affect job completion time, overall cluster utilization, and fairness across submissions.

2.1.2 Cloud computing

Cloud computing is "a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction" as defined by NIST [48]. The concept of cloud computing is not a novel one, it traces back to the 1960s [75] with the vision of providing computing resources as a utility for the public, similarly to the telephone system. During the 70's and the 80's, distributed computing paradigms started to arise. During the 1990s the first instances of SaaS were introduced, but the real turning point for cloud computing was the introduction of Amazon Web Services (AWS) in 2006 that started the commercial availability of IaaS at a global scale [26]. Since then, the cloud computing market has grown and many providers have entered the market, such as Microsoft Azure and Google Cloud Platform, with the cloud market revenues projected to exceed US$ 1.1 trillion by 2027 [72].

In cloud computing, the typical offerings made by cloud providers are typically organized into a layered stack of service models, each of which abstracts away a different portion of the infrastructure. The three main paradigms [82] are, in order of increasing provided abstraction: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). Moreover, a fourth model, called Function as a Service (FaaS), has grown significantly in popularity in recent years.

Infrastructure as a Service provides only the lowest-level abstractions: typically virtual machines with on-demand computing resource amounts, raw storage, and networking capabilities. The customer has to manage all the above software stack, from the operating system to the running applications. Instead, the main task of the provider is to manage their hardware, virtualization, containerization and physical networking. In IaaS, the user configures or accepts a specific remote configuration to be used, with the freedom of either accepting default (recommendation by the provider) or defining completely a custom configuration. IaaS is appropriate when a customer needs precise control over its environment, for example, to run a specific OS version or install specific software not available in higher-level offerings. Another possible example of applicability for IaaS is a company that wants to avoid purchasing physical hardware for a transient phase, but still needs to control its entire software stack.

At a higher level of infrastructure abstraction, Platform as a Service adds a managed runtime and development environment on top of IaaS. The user is able to directly deploy application code without having to manage the OS or being forced to work on server configuration. Using a Platform as a Service means having a whole software framework available for remote execution with user control over the deployed applications and, optionally, configuration settings for the application-hosting environment. The cloud provider has the responsibility of maintaining all hardware layer akin to IaaS, but also to ensure the availability of the offered software platforms. The Google App Engine and Heroku are typical examples of PaaS. This paradigm removes unneeded configurational overhead and accelerates the deployment phase, at the cost of limiting the underlying configuration of the available execution environment. A typical use case for PaaS is for those user that only care about the application code, for example for quickly deploying a web application leveraging the cloud-provided web server and database services without having to manually install them.

The canonically highest level of abstraction is provided by Software as a Service, which delivers complete and functional applications in a remote way. The provider has to manage every layer of the stack: infrastructure, platforms and the software on top of them. The user accesses the provided application through a web browser or API calls and has no visibility or say into the underlying structure. So, using Software as a Service means being able to access resources, provided by an external provider, via remote Web access. Customers do not have to implement anything significant and have no responsibility on the control of the running applications. Classical examples of SaaS are email services (such as Gmail and Outlook 365) and collaboration tools (such as Google docs and Microsoft Office 365). The typical customers of a SaaS service are users that are interested in working with the software rather than building it, SaaS, in fact, is the easiest way to access remote applications from a customer perspective.

Finally, Function as a Service is a more recent paradigm [70], which shifts the stacking paradigm. The only thing that the customer provides are individual functions (in form of code) and the platform executes them in response to certain triggering events, such as HTTP requests or messages from a queue. The provider provisions resources as *ephemeral containers* for the duration of the function call and can release them immediately after [32]. AWS Lambda, Google Cloud Run Functions and Azure Functions are the principal offerings. FaaS eliminates the concept of a running server entirely from the developer's perspective, indeed the customer ignores the fact that there are processors, hardware, and software resources behind the functions. The billing is enforced on function execution time, it is thus cost-efficient for workloads that are reasonably shorter than their invocation frequency (e.g. runtime

of 200ms every 5 minutes [2]), since no resources are consumed when the function is idle. However, FaaS introduces its own constraints: functions must be stateless, execution duration is typically capped [70], and cold-start latency can be a significant factor for increased prices [46].

Another interesting aspect of Cloud Computing, is the deployment model. Cloud infrastructure can be deployed in different configurations: public, private, and hybrid [45]. In a *public cloud*, all infrastructure is owned and operated by a third-party provider and provisioned for open use by the public. Each customer's workloads are isolated from others through virtualization and access controls, but the underlying hardware can be shared. Public cloud is the most cost-effective option for most organizations because the provider achieves economies of scale by serving many customers from the same data centers and the costs for companies to own an maintain an entire infrastructure tailored to handle the workload peak would be greater. A *private cloud* is instead dedicated to a single organization. It may be hosted on-premise in the organization's own data center, or leased from a provider but isolated from other customers. Private clouds give organizations full control over their infrastructure and are used where issues like data privacy concerns arise or are enforced by law. The trade-off is that the organization bears bigger costs than using a public cloud. *Hybrid clouds*, instead, combine both approaches: some workloads run on private infrastructure, others on public cloud. Sensitive data or regulated systems may remain on-premise, while less sensitive workloads are destined to use the public cloud for reasons of cost and even possibly better scalability. Another deployment model is the *community cloud*: the cloud infrastructure is provisioned for exclusive use by a specific community of consumers from organizations that have shared requirements. It may be owned, managed, and operated by one or more of the organizations in the community, a third party, or some combination of them, and it may exist on or off premises.

Typical applications in the cloud are deployed in a microservices architecture, which is a software design pattern that structures an application as a collection of loosely coupled services that communicate via mechanisms such as RESTful API or asynchronous pub/sub patterns. Each service is responsible for a specific functionality and can be deployed and scaled independently, this adheres perfectly to the cloud philosophy. The microservice approach contrasts with the traditional monolithic architecture, where all components of an application are tightly integrated and run as a single unit. This latter architecture can be easier to develop and deploy for small applications, but it can become challenging to maintain and scale as the application grows in complexity. Microservices, instead, allow for greater flexibility and scalability and allow for the maintainers to focus explicitly on their reduced business logic, without having to worry about the rest of the application. However, microservices

introduce significant operational complexity. More services mean more deployment units, more network calls between components, more complex configuration and communication topologies [28].

Microservices should run within a context that can provide the entire necessary environment for their execution. Hence, microservices can be run within *containers*, which are suitable tools for microservice execution. These tools can also be used to ease the addition or removal of instances of microservices when necessary, following the scalability requirements. Containers are a form of virtualization technology that can enclose an application together with all of its dependencies, libraries, and configuration files into one single entity called container image. Containers differ from virtual machines in various ways, such as the fact that virtual machines run on a virtualized operating system, which requires a hypervisor to gain access to the physical hardware, while containers run on the same kernel as the host operating system. The kernel realizes container isolation through kernel *namespaces*, which provide each container with its own view of process IDs, network interfaces, file systems, and users. Another feature employed by containerization platforms are *cgroups*, which allow to enforce limits on CPU, memory, and I/O for the executing container. The difference in the architectural design between containers and virtual machines leads to profound divergence in terms of performance. Containers can be initialized in a very short amount of time, while virtual machines can take up to minutes for initiation, due to the necessary boot process. This improvement is mainly due to the absence of the duplicated operating system, which leads to the reduction of the amount of used computational resources, also allowing for increased quantities of workload on the same physical hardware.

Docker is one example of widespread container runtime and it provides the necessary tools for building and running containers. *Docker container images*, also called Docker images are built using a declarative syntax in a file called *Dockerfile*. While Docker addresses the problem of building and running a single container, managing a sheer number of containers (running, for example, microservices) across a set of machines requires a so called orchestration layer. Kubernetes (see Section 3.1) is the dominant container orchestration platform, it allows to deploy and manage containerized applications at scale across a set of machines, called Kubernetes cluster.

2.1.3 Converged Computing

High performance computing and cloud computing developed independently, under different economic constraints and with different design priorities. HPC was built around the requirements of scientific simulation: high FLOPS, extremely fast interconnects and centralized workload management. Cloud computing, instead, was built around elasticity, fault tolerance through redundancy and per-utilization

costs. Moreover, the combination of containers and orchestration platforms such as Kubernetes has become the standard deployment model for cloud applications. However, this cloud-oriented environment has historically evolved independently from the High Performance Computing world, which has its own established tools, schedulers, and programming models.

In the recent years, scientific workflows have grown in complexity and heterogeneity to the point where the strict models of HPC became not sufficient to support their increasing scalability requirements. An environment that combines performance and efficiency of HPC with the resiliency, dynamism and automation of Cloud is needed. This environment, together with the communities of both paradigms, is called *converged computing* [72].

One of the early steps towards convergence was the introduction of containerization in HPC. Containerization is a key technology for cloud environments and it has been progressively adopted in the HPC world for their benefits introduced in the challenges of software deployment and scientific studies reproducibility [40]. By packaging applications and their dependencies into containers, researchers can ensure that their software runs consistently across different HPC systems, which often have varying configurations and software stacks. For this reason, solutions for containers in HPC have been developed, such as Apptainer (formerly Singularity) [40]. Early version of Docker (up until version 19.03) required a privileged daemon process, which is not suitable for shared HPC clusters since security problems could arise from the execution of arbitrary submitted code in a container which is spawned as a child of a root owned process. Apptainer instead does not allow to escalate privileges inside containers [40]. Moreover, it has native support for interconnection technologies as Infiniband and parallel file systems like Lustre. There are other solutions for containerization in HPC, such as Charliecloud [62] and Shifter [25].

One of the most relevant fields of convergence computing is the integration of the Kubernetes container orchestrator with HPC [52]. There are different approaches to the integration of Kubernetes with HPC, among which figure convergence at the workload management level and convergence at the node management level. A non-convergent approach, which is still common, is to keep the two environments separate, with a Kubernetes cluster for cloud-like workloads and an HPC cluster for HPC workloads, with statically assigned nodes and two separate control planes that operate on their respective nodes. This approach is simple to implement, but it does not allow for dynamic resource management and can lead to underutilization of resources. On the other hand, convergence at the workload management level allows to run one control plane, typically the HPC workload scheduler, as an application inside the other cluster, typically the Kubernetes cluster, allowing for more flexibility in configuration and resource management, an example of a technology of this type

is [68]. But since both workloads are running inside the same physical node set, there can be potential resource contention issues if not enough measures are taken to coordinate the resource allocation. Finally, convergence at the node management level, such as [76], allows to have a single unified control plane for managing the cluster lifecycle, with the possibility to dynamically assign nodes to either the HPC cluster or the Kubernetes cluster, depending on the current needs. This approach allows for maximum flexibility and potentially optimized resource utilization, due to the possibility of repurposing one cluster's idle nodes to the other. Examples for tools that enable different levels of convergence will be presented in Section 2.2. Moreover, the integration of Kubernetes with HPC can also introduce challenges related to the performance of HPC workloads, since Kubernetes is not designed for the high-performance requirements of HPC applications. For example, it is not always the case that the chosen containerization tool has the native support for high-speed interconnects, which are critical for HPC workloads. Therefore, additional work is needed to ensure that HPC workloads can run appropriately on a Kubernetes cluster.

Finally, SliMe, the original solution presented in this thesis, aims to integrate Slinky with [76]: it joins the two approaches, the first being convergence at the workload management level and the second being convergence at the node management level. Giving the possibility of having an HPC workload manager running inside a Kubernetes cluster, converging at the workload management, with the possibility of administrate the nodes purpose and lifecycle, converging at the node management level. This allows to further increase the unification of the two paradigms and to enable the possibility of dynamic resource management for HPC workloads, with the flexibility of Kubernetes and the benefits of bare metal nodes (see Section 6.1).

2.2 State of the Art

In the literature, many solutions that integrated cloud technologies and HPC tools have been proposed, with different levels of convergence. For example, Fluence [52] (formerly known as KubeFlux [53]) is a Kubernetes plugin scheduler that uses the scheduling engine of Fluxion [61], the graph-based scheduling component of the Flux workload manager, which is a Resource and Job Management Software (RJMS) for HPC systems that overcomes the novel problems given by the increasing complexity and heterogeneity of scientific workflows [3]. Fluence allows defining logical groups of pods, and its internal scheduling component optimizes the resource allocation for the entire group (which can compose an entire application), rather than the single pods as the default Kubernetes scheduler does. This is a typical example of convergence at the workload management level.

Other solutions offer similar concepts of convergence at the workload management level, the Torque-operator [84] allows to submit Torque RJMS jobs from Kubernetes through the Kubernetes API. Two clusters are present at the same time: a Torque cluster and a Kubernetes cluster. The Torque-operator acts as a bridge between the two, acting as an enabler for a unified workload submission interface. Another similar, but more general solution is the Bridge-operator [42], which supports submitting jobs from Kubernetes to a certain set of supported external workload managers (among which figure Slurm and IBM Spectrum LSF) through HTTP/HTTPS requests, that for example, can interact with `slurmrestd` and, furthermore, allow for monitoring the HPC jobs status directly from the Kubernetes API. The workflow is similar to the Torque-operator, in which a Kubernetes CR is defined and objects of that type are created to submit jobs to the external workload manager.

Another project that allows for visibility of the current workloads is Supernetes [47], which allows for a comprehensive visibility of Slurm nodes and workloads from the Kubernetes perspective. There is a 1:1 mapping between Slurm nodes and Kubernetes Virtual Kubelets and a representation of Slurm jobs with a coherent number of unschedulable Kubernetes pods. It is composed by a controller running in a Kubernetes cluster and an *agent* running inside the HPC cluster. The agent is responsible for monitoring the existing Slurm nodes and jobs, updating the Kubernetes controller that creates virtual kubelets instances and pods accordingly. In the same way, Kubernetes workloads that are submitted to the virtual nodes are sent to the agent, that dispatches them as Slurm jobs utilizing HPC containerization solutions such as the aforementioned Apptainer.

A substantially different approach is instead taken by High Performance Kubernetes (HPK) [14], which is a solution that co-locates HPC and cloud-native technologies on the same hardware, overlapping them on the same physical platform. In particular it allows launching Kubernetes workloads on top of an existing HPC cluster, with the HPK system translating the YAML Kubernetes workload into Slurm scripts and then submitting them to the Slurm workload manager. Another solution that runs Kubernetes workloads leveraging HPC tools is Kind Slurm Integration (KSI) [18], it has a different approach since it completely encapsulates a Kubernetes cluster inside a Slurm job, which is then submitted to the Slurm workload manager. In this way, a temporary KIND Kubernetes cluster is dynamically created and destroyed as needed and it can be used to run cloud-native workloads on top of an HPC cluster, clearly with the overhead of having to create and destroy the Kubernetes cluster for each job. This latter solution is made possible since Docker (and therefore KIND) nowadays supports rootless operations and can be launched on HPC systems with such limitations.

Cérin et al. [12] proposed a solution that introduced the deployment in form of

pods of a Slurm cluster's nodes via the containerization of Slurm heads (running `slurmctld`) and Slurm workers (running `slurmd`). Those components then can be deployed as needed on the Kubernetes cluster. This approach to convergence at the workload management level is very similar but less feature-rich than Slinky [68] (for more details see Section 3.4), since, for example, it does not provide flexible configuration through the Kubernetes API such as Slinky does (see Section 3.4), but instead bakes a static configuration inside the container image.

In light of the advancements in artificial intelligence and machine learning, both the HPC and the cloud computing worlds are increasingly involved in the provision of support for this field. HPC is traditionally being used for training large machine learning models, while cloud computing has been used for deploying and serving inference workload at the necessary scale. Typically, separate infrastructures and workload managers are used to support these kinds of processes, limiting the flexibility for resource management and requesting additional involvement from the system administrators. The convergence of HPC and cloud computing can enable new possibilities for such workloads, for example being able to train large models on HPC clusters and then deploy them on the cloud for inference, choosing which available nodes should be part of the HPC cluster or the Kubernetes cluster by using a single unified interface [76], realizing convergence at the node management level. In this way, node repurposing is facilitated, and the same physical resources can be used for both HPC and cloud workloads, depending on the current needs, with an emphasis on dynamicity and flexibility of the cluster configurations.

Chapter 3

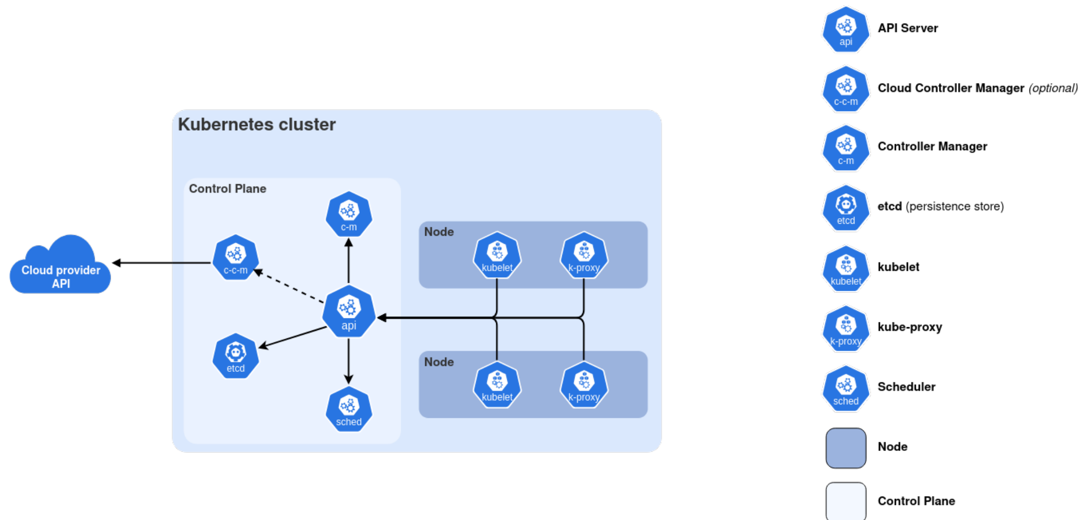
Employed Technologies

This chapter provides an overview of the software components and tools used during the design, implementation and evaluation phases of this thesis. Each of the following sections introduces a specific technology, describing its purpose and the role it assumes in the proposed architectures, if not immediate. The chapter covers container orchestration platforms, workload managers, bare-metal provisioning frameworks, monitoring systems and the benchmarking utilities selected for the experimental evaluation, giving the reader the necessary context to follow the architectural choices and results presented in the subsequent chapters.

3.1 Kubernetes

Kubernetes [36] is a container orchestration system, originally developed at Google and released as open source in 2014. Its fundamental goal is to manage the lifecycle of containers across a cluster of machines, along with their deployment and networking. The orchestration system is based on a declarative model, in fact users can submit manifests containing desired states for Kubernetes *objects* to the API and then a set of *controllers* continuously reconcile the observed state of the cluster with that user-specified desired state, achieving consistency like so.

A Kubernetes cluster is composed of a control plane and a set of worker nodes. The control plane is mainly responsible for: exposing the API server, scheduling workloads, persisting cluster state for recovery and running the aforementioned reconciliation controllers, see Figure 3.1 to have an overview of the main components. The central component of the control plane is the **API server** (`kube-apiserver`), which is responsible to take care of all the requests for operations to be performed on the cluster. Every interaction with the cluster, whether from a user (typically with



source: `https://kubernetes.io/images/docs/components-of-kubernetes.svg`

Figure 3.1: Main components of a Kubernetes cluster

the `kubectl` command) or an automated entity, is realized through API calls via RESTful HTTP requests. Upon request reception, the API server applies admission control (a chain of controllers that can mutate or reject requests before they are persisted, see Section 4.2.1 for more details) and then triggers the persistence of the object. The admission control mechanism is extensible: Kubernetes supports *Mutating Admission Webhooks* and *Validating Admission Webhooks*, which allow ad-hoc services to intercept and eventually modify and validate *admission requests* before they reach `etcd`. This extensibility capability is used by the *Injection Webhook* described in Section 4.2 and by the *Scaling Webhook*, see Section 4.3.

The cluster state is persisted in `etcd`, a consistent and distributed key-value store that implements the Raft consensus algorithm [60] to maintain consistency across replicas. etcd is the single reference for all Kubernetes objects and they are all stored as key-value pairs. The API server is the only component that can communicates with etcd directly and all other components must interact with etcd indirectly by passing through the API server. If etcd becomes unavailable, the control plane cannot process any state changes, but at the same time, already running workloads on worker nodes can continue to execute.

The next component in the control plane is the *scheduler* (`kube-scheduler`), it watches the API server for newly created pods that have not yet been assigned to a node. For each unscheduled pod, the scheduler chooses a suitable node analyzing all of them through a process composed of two phases: the first one is the *filtering* phase, that eliminates nodes that do not satisfy the pod's hard constraints (e.g. resource

requests, node selectors, taints and tolerations), and the second one is the *scoring* phase, that ranks the remaining nodes according to certain priority functions, which are also configurable. The pod is then scheduled onto the node that obtained the highest score.

Another component is the *controller manager* (`kube-controller-manager`), that runs a collection of controllers that implement reconciliation loops. For example, the ReplicaSet controller watches ReplicaSet objects and ensures that the actual number of running pods matches the declared replica count. The node controller monitors node health and marks nodes as unreachable, if it is the case. Another example is the EndpointSlice controller that populates the EndpointSlice objects associated with services by seeking for pods whose labels match the selector specified in the service. Each controller operates independently, watching a subset of the API objects and writing back corrections when the observed state diverges from the desired state.

The *Worker nodes* are the cluster elements that physically run the actual workloads. On each worker node three main components run: the `kubelet` is an entity that ensures that the containers are running and healthy on the local node, possibly by interacting with the container runtime (e.g. `containerd` or CRI-O). The other component is the `kube-proxy`, it maintains the networking rules on nodes and implements a part of Kubernetes Services: it takes care of the packet forwarding rules on the node, using the OS capabilities or by itself if there is not such availability. Finally, the *container runtime* is the component responsible for managing the lifecycle of containers (start, stop, restart, ...), from pulling container images from a registry to creating container processes with the appropriate namespace and cgroup isolation.

The smallest deployable unit in Kubernetes is the `Pod`, which is a logical group of one or more containers that share network and storage resources and are colocated on the same node by definition. Pods can share volumes to have shared storage (and to have a certain degree of persistence). Containers in a pod share the network namespace, the IP and the port space, they can communicate through `localhost` or with standard IPC. They are rarely created directly by users, usually, other controllers manage their lifecycle according to the declared desired state. An example can be `Deployments`, that are the standard objects for stateless applications and are backed by the Deployment controller: a `Deployment` declares the desired state of a set of identical pods through a pod template and a replica count, and manages a subordinate ReplicaSet object that maintains the specified number of running replicas. Pods managed by a Deployment are stateless and thus interchangeable: if a pod fails, it is replaced by a new instance without preserving any state. Instead, `StatefulSets` address workloads that need state. They assign to each pod a persistent ordinal index and are started and stopped in a deterministic way. Each

pod can be associated with a dedicated PersistentVolume through corresponding PersistentVolumeClaim, and this binding is preserved across pod rescheduling. An example of a `StatefulSet` in this thesis is the Slurm control plane in Slinky, that gets examined by the Injection Webhook (see Section 4.2). A core component to enable communication are the `Services`, they provide a network abstraction over an identifiable dynamic set of pods. A `Service` object uses label selectors to identify its backend pods and exposes them under a virtual IP address, which is called ClusterIP[1] and a DNS name resolved by the cluster's internal DNS server (CoreDNS in standard installations). This abstraction allows to decouple the provision of a service from the lifecycle of individual pods: the control plane dynamically updates the service's endpoint list (with the aforementioned controller), and kube-proxy manages the forwarding rules on each node, realizing the virtual IP mechanism and ensuring that traffic directed to the virtual IP reaches the backend. Different service types control the reachability level for the pods. A `ClusterIP` service is accessible only from within the cluster network. A `NodePort` service allocates a static port on every node in the cluster and forwards traffic arriving on that port to the service's backends, depending on the configuration only on the local node or cluster-wise. A `LoadBalancer` service relies on an external load balancer from the infrastructure provider, typically a cloud environment or from an in-cluster implementation such as MetalLB [51] in bare-metal environments, as is the case in this thesis and routes external traffic through the provisioned load balancer to the backend. `Services` are a fundamental part in the realization of the proposed solutions, since they are the components that allow to expose the Slurm control plane and allow the bare metal nodes to join the cluster and communicate with the controller.

A core feature of Kubernetes is its resource extensibility through Custom Resource Definitions (CRDs) and the *operator* pattern. A CRD allows users to define new object types in the Kubernetes API. Once a CRD is installed, instances of the new type can be managed in the same fashion as standard Kubernetes resources. An *operator* is a controller that watches Custom Resources and implements the related specific reconciliation logic and reacts to changes to the resources that it watches. Several of the technologies used in this thesis follow the operator pattern: Cluster API (Section 3.5) defines CRDs such as `Machine` and `MachineDeployment` to represent infrastructure resources, Slinky (Section 3.4) contains the `slurm-operator` that allows to deploy Slurm clusters. In each case, a dedicated controller watches the respective Custom Resources and reconciles the cluster state toward the declared specification.

Kubernetes has a central role in this thesis, as it is the main platform for workload orchestration and management that is considered. The proposed architectures are

[1] `https://kubernetes.io/docs/reference/networking/virtual-ips/`

designed to leverage Kubernetes characteristics for enabling a flexible and time-efficient management of HPC clusters.

3.2 Openshift

OpenShift [30] is a Kubernetes distribution by Red Hat [55] with additional components (such as platform operators like MCO and CVO, subsequently described) and stricter default configurations, having as a target production deployments in regulated or multi-tenant environments. The entire software stack is called OpenShift Container Platform (OCP).

OpenShift replaces several default Kubernetes components with its own implementations. The default container runtime is CRI-O rather than containerd, that is the default choice in Amazon AWS, Microsoft Azure and Google Cloud Platform. Moreover, regarding networking choices the default CNI (Container Network Interface) plugin is OVN-Kubernetes rather than a plugin selected by the Kubernetes cluster administrator. OVN-Kubernetes implements the Kubernetes networking model by encapsulating traffic exchanged by pods inside GENEVE tunnels between nodes. On each node, an Open vSwitch (OVS) bridge handles packet classification and forwarding using OpenFlow rules established by the OVN controller. In Kubernetes, desired policy for network management at OSI level 3 and 4 are specified through objects called `NetworkPolicy`, these objects allow, for example, to allow only inbound connections that comes from a specific CIDR (Classless Inter-Domain Routing). Not all the Kubernetes compatible CNIs support enforcing of Network Policies, one example is Flannel [22]. In Openshift, network policy enforcement is also performed within OVS: `NetworkPolicy` objects are translated into OpenFlow entries that are used in the OVS pipelines. This architecture routes all pod traffic through the OVS software switch, which introduces per-packet processing overhead (flow-table lookup, GENEVE tunnelling, double traversal of the kernel network stack through a `veth` pair) that is absent in bare-metal networking. The performance impact of this overhead is quantified in Section 6.1.2, together with a more comprehensive overview on the interactions for the Openshift CNI.

One of the most significant differences between OpenShift and standard Kubernetes concerns the security model applied to workloads. Standalone Kubernetes enforces pod security through Pod Security Admission (PSA) [2], which has the granularity of namespace level: a namespace can be labelled with one policy level defined in the Pod Security Standards which can be `Privileged`, `Baseline`, `Restricted`. Then the PSA admission controller checks the pods that are being created in the namespace

[2] https://kubernetes.io/docs/concepts/security/pod-security-admission/

and acts following an user specified mode, which can reject pods whose security context violates the selected one or just trigger a warning. PSA is coarse-grained: the three profiles are fixed and cannot be customized without introducing other external components (such as admission webhooks), and enforcement granularity is limited to namespaces. OpenShift implements a more fine-grained mechanism called Security Context Constraints[3] (SCCs). An SCC is a cluster-scoped object that defines a precise set of conditions a pod must satisfy to be admitted, among them figure: which Linux capabilities it may request or must drop, whether it may run as root or must run with a UID from a certain range defined in the namespace, whether it may access the host network, which volume types (emptyDir, hostPath, persistentVolumeClaim, ...) it may mount, whether it must run with a read-only root filesystem. When a pod creation request is issued, the OpenShift SCC admission controller evaluates the pod's requested security context against all SCCs that are bound (through RBAC bindings, specifying the verb `use`) to: the pod's service account, the issuing user's group and the issuing user itself. After gathering all the SCC, they get evaluated in priority order (an integer specified in the SCC yaml) and if the priority is the same, in a decreasing level of restrictiveness. The first SCC that allows the Security Context of the pod is selected and eventual missing fields in the pod's Security Context specification are filled with compliant values. If no SCC matches, the pod is rejected. The default SCC applied to the pods is `restricted-v2`, which is the most restrictive of the SCCs that are included with Openshift. It imposes a certain number of restrictions, some of them include the denying of privileged execution, mountings of host directory volumes, dropping of all Linux capabilities except `NET_BIND_SERVICE` (if explicitly stated) and enforcing a non-root UID from a namespace-specific range. This default is stricter than the upstream Kubernetes `Restricted` PSA profile (which, for instance, does not enforce UID range allocation) and has practical consequences for deploying software that assumes root privileges or hardcoded UIDs, as discussed in Section 5.1.1 in the context of deploying Slinky on OpenShift. Workloads that require elevated privileges must be explicitly granted a less restrictive SCC through an RBAC binding to the workload's service account.

OpenShift also introduces `Routes` as its native mechanism for exposing HTTP and HTTPS services to external clients, they are functionally similar to Kubernetes `Ingresses` but they were introduced earlier. A `Route` object associates an external hostname with a Kubernetes Service and configures the OpenShift router, which is an HAProxy instance, to accept traffic on that hostname and forward it to the service's backend pods. The DNS resolution is not handled by `Routes`, only the routing part from the Openshift routers. There are three possible TLS modes specifiable

[3] `https://docs.redhat.com/en/documentation/openshift_container_platform/4.13/html/authentication_and_authorization/managing-pod-security-policies`

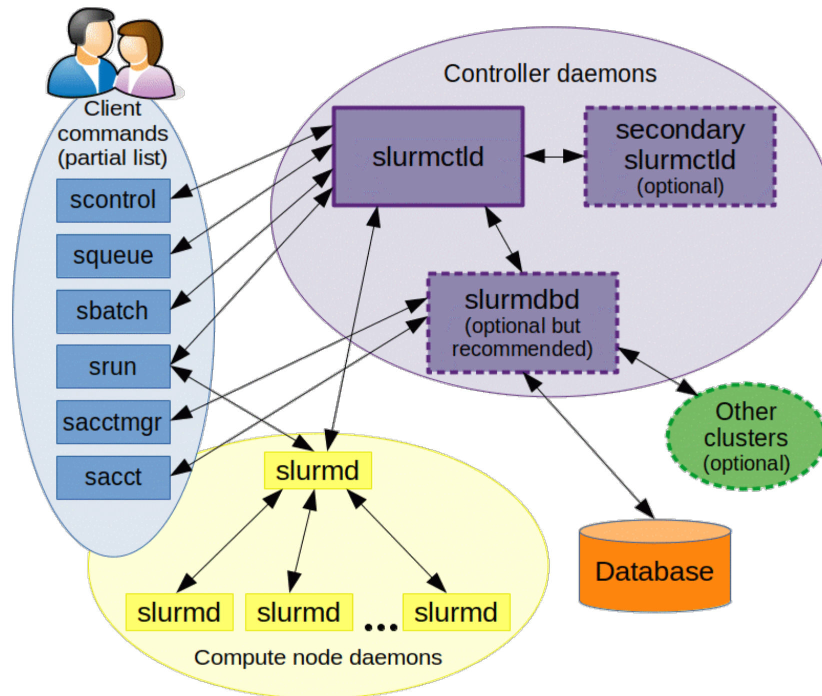
through the annotation `route.openshift.io/termination: edge`, which means that TLS is terminated at the router and traffic between the router and the backend remains unencrypted, `reencrypt`, where there are two different TLS sessions, one between the client and the router and another between the router and the service backend and *passthrough*, where the router forwards the encrypted TLS connection directly to the backend pod without interfering with it. The Route mechanism predates the standard Kubernetes `Ingress` resource and remains the primary method in OpenShift systems, indeed they support also the standard `Ingress` object by internally converting it to a `Route` resource. Standard Kubernetes `Ingress` resources have no direct support for TLS passthrough and require the use of an Ingress Controller that supports it, such as NGINX, and the configuration requires annotations to specify the TLS handling behavior.

Finally, another core difference between OpenShift and Kubernetes is that, while Kubernetes does not impose OS limitations on its cluster nodes, Openshift supports on its control nodes only the operating system Red Hat Enterprise Linux CoreOS (RHCOS), a mainly immutable, Red Hat Enterprise Linux (RHEL) based OS, which natively supports containers. RHCOS nodes are configured through the Machine Config Operator (MCO) that applies declarative configurations in form of `MachineConfig` objects to nodes and performs reboots when the configuration changes. This approach ensures that all nodes in an OpenShift cluster run an identical, operating system configuration, reducing configuration drift. The configuration drift happens when there is a node whose configuration differs from the desired one, in OpenShift there are mechanisms to detect this situation realized into the Machine Config Daemon (MCD). The OS configuration is not the only component of a cluster that gets checked for consistency, also the underlying platform operators are subjects to a similar mechanism: the Cluster Version Operator (CVO) continuously checks that the installed operators and their versions match the desired version specified in the *release payload*, which is an image containing the desired manifests for all the operators, and reconciles the cluster state accordingly. The CVO is the component responsible for managing OpenShift updates, it ensures that all components are updated in a compatible way by applying the updates specified in the release payload.

3.3 Slurm

Slurm [81] (formerly an acronym for Simple Linux Utility for Resource Management) is an open source workload manager for Linux clusters, widely adopted in the HPC community. It is the most used workload manager in the TOP500 list [15] making it the dominant job scheduler in production supercomputing environments. Slurm

is responsible for three core functions: allocating compute resources (nodes, CPUs, GPUs, etc.) to user-submitted jobs, support the lifecycle, monitor and the execution of those jobs across the allocated resources, and managing a queue of pending jobs according to configurable scheduling policies. Slurm is developed and maintained by SchedMD, which also provides commercial support.



source: https://slurm.schedmd.com/arch.gif

Figure 3.2: Main components of a Slurm cluster

As can be seen in Figure 3.2, a Slurm cluster is composed of a control plane that runs controller daemon and a set of compute nodes, on which compute daemons resides. The control plane runs the Slurm controller daemon, or `slurmctld`, which is the central management daemon responsible for accepting job submissions, monitoring the cluster state and scheduling jobs. The controller daemon can be deployed in a configuration with an active and a secondary backup instance, where the backup takes over if the active instance fails. The cluster state managed by `slurmctld` can be persisted to disk for recovery after a restart, moreover Slurm supports the use of an external database (such as MySQL or MariaDB) through the Slurm database daemon, or `slurmdbd` (Slurm database daemon), to persist accounting information. Each compute node in a Slurm cluster runs the `slurmd` daemon, which acts as a remote execution agent. When `slurmctld` assigns a job to a node, it sends execution instructions to the corresponding `slurmd` instance, which spawns the required

processes to execute the job and reports the job status back to the controller. The slurmd daemon also periodically sends heartbeat messages to slurmctld to report the health status of the node. If slurmctld does not receive heartbeats from a node within a configurable timeout, it marks the node as DOWN and jobs that happen to be running on the node get rescheduled or marked as failed, in a similar scenario as Section 4.3.2.1. Worker nodes can be logically grouped in Partitions that share common scheduling constraints, access controls and resource limits, they serve a similar role to job queues with uniform characteristics.

Users can interact with Slurm workload manager through a set of command line utilities. The sbatch command is one of the most important ones, since it is used to submit a batch job to the Slurm queue. These batch jobs are specified by a job script that define the commands to execute and a set of parameters for that job, that can be: the number of nodes, the number of tasks per node, the number of CPUs per task, the maximum amount of memory, the maximum allowed walltime, the target partition, the environment variables, etc. Another command to run a workload is the srun command, that launches tasks on one or multiple nodes and is capable to specify resource requirements. There is also the squeue command, that displays the current state of the job queue, scontrol is the main command used to administrate the cluster state and sinfo reports the state of the cluster nodes and partitions.

The configuration of a Slurm cluster is mainly contained in the main configuration file: slurm.conf. It specifies a wide variety of settings, among which figures the cluster name, the address and port of the controller daemon, possibly the list of compute nodes (including their hostnames, IP addresses, CPU numbers, memory sizes and also GPUs), the partition definitions and the scheduling parameters. Every node in the cluster, both the controller and each of the compute nodes, must have access to the same slurm.conf file, as both slurmctld and slurmd read it at startup to work correctly. There are other important configuration files, such as cgroup.conf, which controls how Slurm uses Linux cgroups to constrain and track the resource usage of jobs, if Slurm plugins that enable the use of cgroups are employed. Through cgroups, Slurm can enforce fine grained limits on CPU time, memory, and device accesses, preventing jobs from consuming more resources than they were allowed and also provide accounting of actual resource usage. Slurm supports both cgroups v1 and v2, but never a combination of the two, which is called a *hybrid* cgroup configuration. This configuration may happen when a containerized version of Slurm that uses cgroups v2, which is a required condition [66], is deployed on a host that uses also cgroups v1. This prevents the correct startup of the Slurm daemons.

In Slurm, the authentication mechanism is handled through plugins, there are two

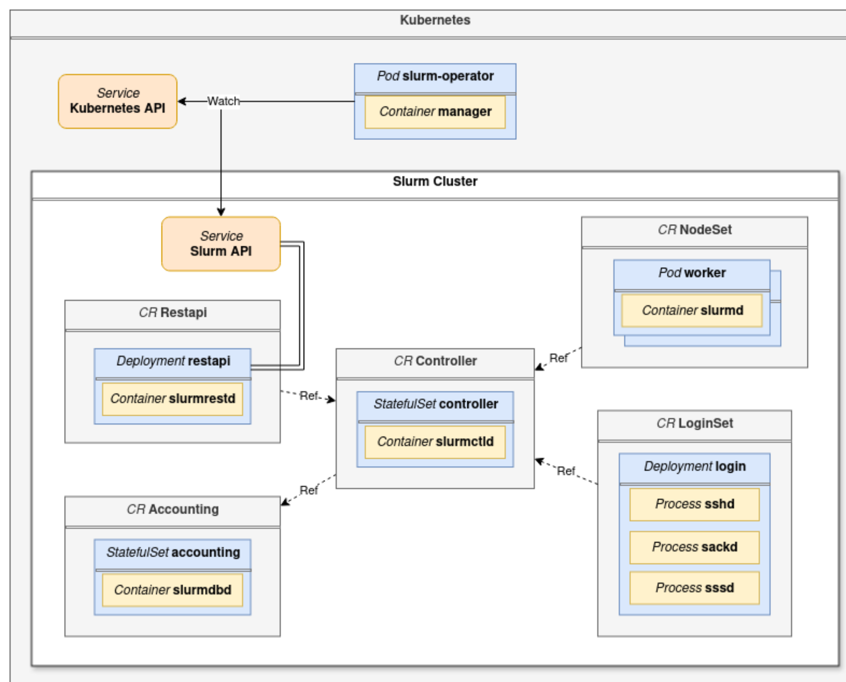
main plugins: `auth/munge` and `auth/slurm`. The default and historically most common method is MUNGE [57], a symmetric key authentication service designed for HPC environments that allows to create and validate credentials. All nodes in the cluster share a secret key file, called `munge.key` and every message exchanged between Slurm components is authenticated by coupling it with a MUNGE credential that includes a timestamp and the sender's UID/GID. For example, when a job gets submitted, the receiving daemon verifies the credential using the shared key, trough the `munged` daemon, ensuring that only authorized nodes and users can submit jobs. Starting from Slurm 23.11, SchedMD introduced the `auth/slurm` plugin as an alternative to MUNGE. This plugin implements authentication natively within Slurm itself, eliminating the dependency on the external `munged` daemon. The `auth/slurm` plugin provides a native authentication layer for the cluster, utilizing a JSON Web Token (JWT) based format for its internal credentials. It utilizes the Slurm Auth and Cred Kiosk (SACK) framework, which adopts a JSON Web Token (JWT) data format for internal credentials. Each Slurm component uses a shared symmetric secret, `slurm.key`, to sign and verify the identity of the tokens. The key material is managed in a similar way than with MUNGE, since a mechanism to distribute it across all nodes is needed. The `auth/slurm` plugin is the authentication method used by Slinky (see Section 3.4) in its default deployment configuration. The requirement for a shared `slurm.key` across all nodes has a profound impact on the architecture of SliMe proposed in this thesis: when bare metal nodes provisioned through Cluster API must join a Slinky-managed Slurm cluster, the key used by the controller pod must be distributed to the bare metal nodes during their provisioning process, as described in Section 4.2.

Slurm's architecture historically assumes a static and pre-configured set of compute nodes: the list of nodes, their hostnames, and their resources are declared in `slurm.conf` at cluster startup time. Adding or removing nodes requires modifying `slurm.conf` and restarting the controller daemon. Starting from Slurm 22.05 *dynamic nodes* were introduced, which allow to add and remove nodes at runtime without editing `slurm.conf`. While in traditional static node configuration the data needed for communication (NodeAddr and NodeHostName) is retrieved from the configuration file, in the dynamic node model, the controller daemon retrieves it during the registration phase. Moreover, Slurm has a feature called *configless*. Nodes that operate in that mode do not need to have a pre-distributed `slurm.conf` configuration file, but they can retrieve it from `slurmctld`, removing the necessity to distribute the configuration file. It is sufficient to enable the mode inside the controller through the standard Slurm configuration file and calling `slurmd` with the appropriate flags from the worker nodes. The combination of those two last features, `dynamic nodes` and `configless` Slurm is particularly well suited for environments where nodes are provisioned on demand, such as Slinky. Moreover, it is

particularly useful for the work in this thesis, since it allows adding bare metal nodes provisioned through Cluster API without neither having to change the configuration file, nor distribute it.

3.4 Slinky

Slinky [68] is a technology developed by SchedMD that comprises two sub-components: the `slurm-operator` and the `slurm-bridge`. The `slurm-operator` is a Kubernetes operator that allows to deploy and manage Slurm clusters on Kubernetes and the `slurm-bridge` which contains a Kubernetes scheduler that allows to manage Slurm workloads from Kubernetes, enabling the execution of both different workloads on the same cluster, reducing operating costs due to shared hardware[4]. While the `slurm-bridge` is an interesting tool towards HPC and cloud convergence, it is not of particular use for the work presented in this thesis, its description is thus beyond the scope of this section.



source: https://slinky.schedmd.com/projects/slurm-operator/en/release-1.0/_static/images/architecture-operator.svg

Figure 3.3: Components involved in a Slurm cluster deployed by Slinky

The `slurm-operator` has a fundamental role in this thesis, since it is the main tool for deploying and managing Slurm clusters on Kubernetes. Its main feature is being

[4] https://slinky.schedmd.com/projects/slurm-bridge/en/release-1.0/

able to deploy a Slurm cluster in a containerized form (see Figure 3.3), where the `slurmctld` daemon runs in a pod and communicates with the worker nodes, which are also deployed in form of pods running the `slurmd` daemon. The concepts behind standard components of a traditional Slurm cluster are encapsulated in a number of ad-hoc Kubernetes CRDs: there is the `Controller` CRD, that represents the Slurm controller daemon and deploys a `StatefulSet`, which manages the aforementioned pods running `slurmctld`. Another example is the `NodeSet` CRD, that represents a set of homogeneous worker nodes and deploys a set of equivalent pods running `slurmd`. From a Slurm partition perspective, it is a sort of "atomic unit", since a `NodeSet` cannot spawn worker nodes belonging to different partitions (without manually intervening on the Slurm cluster in a subsequent moment), but the opposite is true: it is indeed possible that different `NodeSets` belong to the same Slurm partition. Moreover, this resource supports the `/scale` subresource, making it suitable for autoscaling purposes, for example with KEDA (see Section 4.3.2.1).

If desired, it is also possible to create an object from the CRD `LoginSet`, whose operator creates a `Deployment` that runs a set of pods acting as login nodes for the Slurm cluster. Each login pod runs three processes: `sshd`, which is the OpenSSH daemon that handles SSH connections, `sackd`, the aforementioned Slurm Auth and Cred Kiosk daemon that handles authentication token management on behalf of the login node and `sssd`, the System Security Services Daemon, that provides remote identity data retrieval and authentication (using services such as LDAP or Active Directory), enabling centralized user account resolution inside the pod. There are also CRDs for `accounting`, which deploy pods running `slurmdbd` that can interact with an external database to persist accounting information and another CRD for Slurm REST API, `Restapi`, that deploys pods running the `slurmrestd` daemon, which provides a RESTful API for interacting with the Slurm cluster, allowing to submit jobs and query the cluster state without using the traditional Slurm command line utilities.

Slinky uses the plugin `auth/slurm` to manage the authentications of the nodes, hence MUNGE is not employed. The distribution of the key happens during the creation of the worker pods: during the reconciliation phase, the nodeset controller checks if there are inconsistencies with the desired state, and if new worker pods needs to be created, it calls a `WorkerBuilder` that generate the pod template. Inside this template, among all the necessary specifications, there is a VolumeMount that mounts the Kubernetes Secret containing the `slurm.key` file, which is referenced by the `Controller` CR.

As previously mentioned in Section 3.3 the combination of the `dynamic nodes` and `configless` features of Slurm is particularly well suited for environments such as Slinky. Indeed, these features are employed on the pod worker nodes. Again, during

the creation of the workers, the `WorkerBuilder` injects the args for the `slurmd` container, putting `-Z` to enable dynamic nodes and `--conf-server` followed by the address and the port retrieved from the `Controller` object.

3.5 Cluster API

Cluster API [41] is a Kubernetes sub-project that extends the Kubernetes API with abstractions in form of Kubernetes objects that are agnostic to the chosen infrastructure and that allow provisioning and managing the lifecycle of Kubernetes clusters. Rather than relying on external tooling, Cluster API manages the desired state of infrastructure resources (clusters, sets of machines, single machines, etc.) through Kubernetes Custom Resource Definitions. A set of ad-hoc controllers reconcile that desired state against the actual available infrastructure by interacting with provider-specific backends (called *Infrastructure Providers*). Cluster API applies the same reconciliation pattern used by Kubernetes for workloads (described in Section 3.1) to the infrastructure layer itself, for example: the user declares how many machines of which type should exist, and the controllers act accordingly to satisfy the request. Cluster API is another core technology in the context of this thesis: it is the mechanism through which bare metal machines are provisioned and then joined to Slinky-managed Slurm clusters.

Cluster API introduces the concept of *Provider*, there are different types of them, with the main three being: the *Bootstrap Provider*, the *Control Plane Provider* and the *Infrastructure Provider*. The *Bootstrap Provider* is responsible for turning a server into a Kubernetes node, generating the necessary data to initialize it. The *Control Plane Provider* is responsible for provisioning and managing the control plane of a Kubernetes cluster. The *Infrastructure Provider* is responsible for provisioning and managing the underlying infrastructure resources, such as virtual machines in the cloud or bare metal hosts, as in the case analyzed in this thesis. Providers decouple the core logic of Cluster API from the specifics of how machines are provisioned and configured. Cluster API makes an important distinction between two types of clusters: the *management cluster* and the *workload cluster*. The management cluster is a Kubernetes cluster that runs the Cluster API controllers and manages the lifecycle of one or more workload clusters. The workload cluster is the Kubernetes cluster that is being provisioned and managed by Cluster API; it can be running on any supported infrastructure and is not required to have Cluster API installed. In this thesis, the management cluster is a KIND cluster running the Cluster API controllers, while the workload clusters are those that run Slinky clusters with bare metal nodes.

The Cluster API project defines a number of core Custom Resource Definitions.

The Cluster resource represents a complete Kubernetes cluster, including its control plane specification (controlPlaneRef), networking configuration (e.g. the pods CIDR) and a reference to an infrastructure-specific resource (infrastructureRef) that contains the platform-level parameters for provisioning. A fundamental resource is the Machine, which represents a single machine (physical or virtual) that belongs to a Cluster. A Machine object contains a reference to a resource exposed by the *infrastructure provider* and a reference to a *bootstrap* resource, which describes how to configure the machine system once it has been provisioned. The Cluster API Machine controller watches Machine objects and manages their lifecycle by delegating infrastructure provisioning to the infrastructure provider controller and bootstrap configuration to the bootstrap provider controller. When a Machine is created, the bootstrap provider generates the configuration data, stores it in a Kubernetes Secret, and marks the bootstrap resource as ready. The infrastructure provider then provisions the underlying resource (e.g. powers on a bare metal host or creates a VM), injects the bootstrap configuration, and starts the machine. The machine controller monitors the provisioning progress until the infrastructure provider sets the providerID field on the Machine object, at which point the machine transitions to the Running phase. Deletion follows the reverse path: the machine controller triggers the infrastructure provider to deprovision the machine, and once the infrastructure resource is deleted, the Machine object is removed.

The MachineDeployment manages groups of identical Machines, analogous to how a Kubernetes Deployment manages groups of identical pods. A MachineDeployment declares a desired replica count and a machine template that specifies the infrastructure and bootstrap references. The MachineDeployment controller then manages a subordinate MachineSet object whose controller is responsible for maintaining the specified number of Machine replicas. When the replica count changes (either through manual editing, failure, or through an external scaling mechanism), the MachineSet controller creates or deletes Machine objects to match the desired count, in the same way as a ReplicaSet controller would have done in standard Kubernetes. Rolling updates are also supported: when the machine template in a MachineDeployment changes, the controller creates a new MachineSet with the updated template and progressively scales it up, starting the scaling down of the old MachineSet when the new one has a sufficient number of ready Machines, ensuring to have a minimum total number of available entities to avoid service disruption. The MachineDeployment resource is central to the work of this thesis since it is the object that manages the Slurm bare metal workers and motivates the choices for the autoscaling solution proposed in this thesis (see Section 4.3.2.2), where the replica count of a MachineDeployment is adjusted in response to Slurm job queue length. However, in contrast to Slinky, MachineDeployments do not manage pods like Kubernetes Deployments do, which prevents KEDA from targeting them directly and

motivates the measures taken and described in that section.

To facilitate the management of the Cluster API ecosystem, the project also provides a dedicated CLI tool named `clusterctl`. Standard Kubernetes tools like `kubectl` are sufficient for interacting with the Custom Resources once the system is running and `clusterctl` is the primary mechanism for bootstrapping the management cluster and managing the lifecycle of the provider components in an initial phase. Its most critical function is the initialization phase enabled by the `clusterctl init` command, which fetches the necessary controller manifests and Custom Resource Definitions from provider repositories and installs them into the management cluster. It automatically handles the necessary dependencies, such as ensuring `cert-manager` (see Section 4.2.2) is present for webhook certificate management. Beyond installation, through the `clusterctl generate cluster` command, it is able to produce a fully formed YAML manifests sufficient to provision a cluster, even by using different providers. For example:

```
clusterctl generate cluster <test-cluster>
↪  --control-plane-machine-count=2 --worker-machine-count=5
↪  --infrastructure aws > <test-cluster>.yaml
```

generates the YAML manifests for a cluster named `test-cluster` with 2 control plane machines and 5 worker machines, using the AWS infrastructure provider. The generated YAML can then be applied to the management cluster to create the workload cluster.

Finally, Cluster API makes use of Kubernetes annotations to influence the behavior of its controllers. These annotations provide a mechanism for administrators to signal specific intents to the reconciliation processes that watches the cluster's resources. For instance, the `cluster.x-k8s.io/paused` annotation can be applied to any Cluster API resource to completely halt its reconciliation. Another annotation is `cluster.x-k8s.io/delete-machine`, which allows an operator to target a specific `Machine` for deletion when scaling down a `MachineDeployment`, giving to it more priority during the deletion phase. Additionally, there are annotations like `machine.cluster.x-k8s.io/exclude-node-draining`, that can be used to bypass entire lifecycle phases, in this case skipping the node draining during the deletion of a machine. Another example is `pre-drain.delete.hook.machine.cluster.x-k8s-.io`, whose implications are described in Section 3.7.

3.6 Metal^3

Metal^3 [49] (pronounced "Metal Kubed") is a Cluster API Infrastructure Provider that enables the declarative management of bare metal machines through the Ku-

bernetes API. It integrates two main components: the Cluster API Provider Metal^3 (CAPM3), which implements the Cluster API infrastructure provider contract, and the Bare Metal Operator (BMO), which manages the lifecycle of individual physical machines through their Baseboard Management Controllers (BMCs). The BMCs are chips contained in motherboards that allow to monitor, power-cycle and configure hardware. Together, these components allow bare metal hosts to be provisioned, configured, and deprovisioned using the same declarative model that Cluster API uses for virtual machines in cloud environments.

Metal^3 relies on OpenStack Ironic as its provisioning engine, which is a bare metal provisioning service originally developed as part of the OpenStack project. In a Metal^3 deployment, Ironic runs as a set of containers on the management cluster. The Bare Metal Operator communicates with Ironic through its REST API to perform all hardware management operations.

The central abstraction introduced by Metal^3 is the `BareMetalHost` or `bmh` Custom Resource. A `BareMetalHost` object represents a single physical server and contains the information necessary to manage it, among them figure: the BMC address (typically an IPMI or Redfish endpoint), the reference to the Kubernetes secret containing the BMC credentials, the desired boot mode (UEFI or BIOS), and a reference to a *metaData* secret, that contains a set of information on the host. This secret is a key object for the work of this thesis (see Section 4.2). The Bare Metal Operator watches `BareMetalHost` objects and manages their lifecycle through different phases. During the provisioning of a new `bmh` it briefly go through the `creating` state, subsequently, if the `bmh` does not contain BMC info, it transition to the state `unmanaged`, otherwise it enters the `registering` state, until the BMC data gets validated and the connectivity ensured. Afterwards the host passes into state `inspecting` and the BMO contacts the host BMC, retrieving hardware information. The process of retrieving info is the following: the operator powers the machine on via the BMC and boots it into an ephemeral ramdisk called Ironic Python Agent (IPA). IPA runs in memory and performs a hardware inspection: it reports the CPU info, memory size, network interfaces details and other hardware info, which stores them as properties of the object. After inspection completes, the `bmh` transitions to the state `preparing`, during which some configuration is made on the HW to make it ready for provisioning (e.g. BIOS setup). Finally, the host transitions to the `Available` state, meaning it is ready to be provisioned, and the machine gets powered off again, waiting to be selected for provisioning during a `Machine` creation.

On the CAPM3 side, the provider defines two CRDs that are related to Cluster API `Machines`: the first one is `Metal3Machine`, which maps a Cluster API `Machine` to a `BareMetalHost` in a 1:1 ratio and contains the Metal^3-specific parameters to allow provisioning, such as the image, the host selector, a reference to the `metaData` and

`networkData` secrets. The second CRD is `Metal3MachineTemplate`, which serves as a template for creating `Metal3Machine` objects when needed.

When a Cluster API `Machine` is created, a `Metal3Machine` is created from the `Metal3MachineTemplate` specified. CAPM3 selects an available `BareMetalHost` for the new object, sets the `image` field on the `BareMetalHost` (pointing to a URL of the OS image and its checksum), writes the bootstrap user-data, which was generated by the bootstrap provider, into the host specification, and marks the host for provisioning, the `bmh` passes into state `provisioning`. The BMO then instructs Ironic to power the machine on, boot it again into the IPA ramdisk, and write the specified OS image to the machine's disk, then injects the bootstrap userdata into e.g. the image's cloud-init and reboots the machine into the newly installed operating system, at this point the `bmh` is in state `provisioned`. Once the machine boots into the installed OS, the bootstrap process starts. In the case of this thesis the node is configured with the necessary Slurm binaries and `slurmd` is started. When the bootstrap process is terminated, a `Node` object gets created in the Kubernetes cluster, which has a certain `providerID` (which is the UUID of the bare metal node). The `machine` that during the provisioning process received the same `providerID` from the `Metal3Machine` is then reported as `running` by the Machine operator.

3.6.1 metal3-dev-env

`metal3-dev-env` [50] is a tool that allows to deploy a Cluster API environment with the Metal^3 infrastructure provider on a single machine, which can be either a physical machine or a VM, and it is designed for development and testing purposes. It provides a set of scripts and tools that automate the deployment of the Cluster API components, the Metal^3 components and the provisioning of bare metal machines as Cluster API `Machines`. Those scripts set up the management cluster (a KIND cluster running inside Docker), install the Cluster API core controllers, the Metal^3 infrastructure provider (CAPM3), the Bare Metal Operator (BMO), and the Ironic provisioning services all running as containers on the host. To simulate bare-metal hosts without requiring physical hardware, `metal3-dev-env` uses `libvirt/KVM` to create virtual machines that behave like bare metal nodes. Each VM is equipped with a VirtualBMC (VBMC), allowing BMO and Ironic to manage the VM power state and perform their operations as if it were a physical machine. Moreover, the scripts create a dedicated network that interconnects all the VMs and the physical host.

The behavior of `metal3-dev-env` is controlled through a set of environment variables defined in configuration files (`config_$USER`). These variables control various parameters such as the number of bare-metal hosts to create (`NUM_NODES`), the number of CPUs and amount of memory allocated to each VM, the Kubernetes version

to install on the workload cluster, the OS image to use for provisioning (between Ubuntu and CentOS), the IP address ranges for the aforementioned networks and the choice of bootstrap provider. The environment also configures the Ironic services with the appropriate network parameters and the BMC credentials for the VBMC instances, storing them in Kubernetes Secrets that the Bare Metal Operator references when creating `BareMetalHost` objects. While `metal3-dev-env` is designed for development and testing rather than production use, it provides a suitable environment for validating the provisioning workflows and the integration between Cluster API and Slinky, since the provisioning lifecycle is virtually identical to that of an actual bare metal cluster.

3.7 cluster-api-bootstrap-virtual-kubelet

`cluster-api-bootstrap-virtual-kubelet` [29] (CABPV) is a Cluster API Bootstrap Provider that enables the provisioning and the lifecycle management of non-Kubernetes workloads running the Virtual Kubelet [79]. The Virtual Kubelet is an implementation of the Kubernetes Kubelet API that is able to disguise itself as an actual `kubelet` and allow connecting Kubernetes with other APIs. It presents an architecture with provider interfaces, enabling the integration with various backend systems, some of the providers are part of the project, but it is possible to write custom ones. It connects to the Kubernetes API server registering a `Node` resource and it delegates pod lifecycle management to the pluggable provider interface. This architecture allows to bring non-Kubernetes APIs or infrastructure into the Kubernetes control plane.

The CABPV is designed to work in conjunction with the Cluster API Infrastructure Provider Metal^3 (see Section 3.6), which provisions the underlying bare metal hosts. During the provisioning process, CABPV generates cloud-init bootstrap configuration that gets injected into the `userData` secret referenced by the `Machine` object and contained in the `Metal3Machine`. The provider defines two CRDs: `VirtualKubeletConfig` and `VirtualKubeletConfigTemplate` which are the bootstrap counterparts for the infrastructure components `Metal3Machine` and `Metal3-MachineTemplate` and are referred by `Machines` and `MachineDeployments`, respectively.

When a `MachineDeployment` that contains a reference to a `VirtualKubeletConfigTemplate` triggers the creation of a new `Machine`, a new `VirtualKubeletConfig` is created from the template and the reconciliation loop of the associated operator is triggered. The operator retrieves certain necessary configuration files from the management cluster, such as the TLS certificates and prepares a service unit file to run the Virtual Kubelet on the target node together with the commands to execute to

download the Virtual Kubelet binary from a user-specified URL derived from the `VirtualKubeletConfigTemplate`. Subsequently, the core operation is performed: a cloud-init configuration is generated starting from a jinja template, containing *Commands*, *Files* and *Users*. The *Commands* section contains the aforementioned commands relative to the Virtual Kubelet binary, but also the commands that are defined inside the `VirtualKubeletConfig.spec.commands` field. The *Files* section contains the cluster-related files, together with the user-defined files specified in the `VirtualKubeletConfig.spec.files` field, which specifies also the path and the permission modes for each file. The same thing happens for *Users*, contained in the eponymous field of `spec`, that allow for creating user accounts on the target node and also specify their SSH keys, in a cloud-init syntax. After having generated the cloud-init YAML configuration, a secret is created and its `userData` field is populated with it. Then the `VirtualKubeletConfig` is marked as ready, which allows the provisioning process to proceed, since the `Machine` that triggered the creation of the `VirtualKubeletConfig` has a reference to it in its `bootstrap.configRef` field. The configuration data is then passed to the infrastructure provider that uses it during the bootstrap phase of the new bare metal node, installing the Virtual Kubelet and configuring it to run the user-defined commands. In the context of this thesis, the user-defined commands are those necessary to install the Slurm worker daemon (`slurmd`) and configure it to connect to the Slurm controller deployed by Slinky, allowing the bare metal node to join the Slurm cluster as a worker.

3.7.1 Slurm detach handler

The slurm detach handler was presented in [76] to address a specific problem: when a bare metal node is deprovisioned, it is necessary to ensure that the Slurm controller is aware of the change and that the node is removed from the cluster. By default there is no mechanism to achieve this, neither in Metal[3] nor in the Cluster API ecosystem, and obviously it is outside of the scope of `cluster-api-bootstrap-virtual-kubelet`. The detach handler is a rather concise controller that watches for `Machine` objects that are in phase `provisioning` and, as anticipated in Section 3.5, annotates them with `pre-drain.delete.hook.machine.cluster.x-k8s.io`, that prevents the draining and deletion of the node associated with the annotated `Machine` until the annotation is removed. When the controller notices a `Machine` with a non-empty `deletionTimeStamp`, it retrieves the hostname of the node associated to the machine and makes an HTTP request to a custom ad-hoc HTTP server that gets deployed together with the Slurm bare metal worker. Upon receiving a request to the `/detach` endpoint, the server triggers the execution of the command `sudo scontrol update NodeName=<node_hostname> State=DOWN Reason=deprovision` on the node to detach, effectively notifying the `slurmctld` that the node is no longer available. After performing the HTTP call successfully, the annotation is removed and the deprovi-

sioning process can proceed as normal.

3.8 Additional Technologies

3.8.1 Cadvisor

cAdvisor (Container Advisor) is a monitoring tool that collects and exports information about resource usage and performance of running containers. It works as a daemon that runs in each node of the cluster and it interfaces directly with the Linux kernel control groups (cgroups) and network namespaces to retrieve resource usage statistics, such as CPU time and used memory as well as network traffic.

The architecture of cAdvisor is designed around a *Manager* component that is responsible for the discovery of active containers and the lifecycle of data collection. Upon startup, cAdvisor traverses the cgroup hierarchy to identify existing containers and watches for the creation of new cgroups to detect container starts. It supports both cgroup v1 and cgroup v2 hierarchies, making it compatible with virtually all container runtimes that use cgroups for resource isolation. Each cgroup is perceived as a container, and cAdvisor collects metrics for each of them. This mechanism allowed to retrieve resource usage metric for bare metal hosts in Chapter 6, even if no container was actually running on the node, but since Slurm was configured to spawn a cgroup for each job, cAdvisor was able to retrieve the resource usage of the job by monitoring the corresponding cgroup. CPU and memory statistics are collected by reading the relevant files in the cgroup filesystem, such as `<cgroup_dir>/cpu.stat` or `<cgroup_dir>/memory.stat`, which contain different counters that keep track of resource usage. cAdvisor periodically samples these files to create time series, providing useful data on the resource consumption of each container. Instead, since there is no mechanism to retrieve network traffic statistics from inside the cgroups FS to gather network telemetry, cAdvisor must know the PID of the container and read the interface statistics from `/proc/net/dev`. This allows cAdvisor to attribute network traffic to specific containers, being able to differentiate the activity on different network interfaces

In the context of Kubernetes, cAdvisor is not deployed as a Kubernetes workload by default, but it is instead baked into the `kubelet` binary. The `kubelet` utilizes the internal cAdvisor instance to monitor the state of the pods running on the cluster. This internal monitoring is critical for the cluster stability, the `kubelet` leverages the memory usage metrics provided by cAdvisor to make pod eviction decisions if a node runs out of memory. Additionally, the `kubelet` exposes these aggregated metrics via its own HTTP server at the `/metrics/cadvisor` endpoint using the Prometheus exposition format. This allows monitoring systems to scrape resource

statistics for all containers in a cluster via the `kubelet` API without requiring the installation of a separate monitoring agent on every node. It is possible to retrieve these metrics by deploying `metrics-server`, that also allows the use of `kubectl top` to query the resource usage of nodes and pods in real time. But this solution is rather limited, since it does not have a way to persist retrieved data, to have a more complete monitoring solution, it is common to deploy a Prometheus instance that scrapes the `kubelet` metrics and stores them in its database (see Section 3.8.2).

3.8.2 Prometheus

Prometheus is a monitoring and alerting system that is free and open source. Its fundamental constituent part is a database for time-series that stores numeric metrics identified by a metric name and a set of key-value pairs called labels. The primary mechanism through which Prometheus collects data is scraping: it periodically pulls metrics over HTTP from targets that expose them, rather than receiving metrics in a server-like paradigm. This pull based model distinguishes Prometheus from other monitoring systems that adopt a push model, like Graphite and InfluxDB when using Telegraf agents. Each monitored service exposes a `/metrics` HTTP endpoint that returns the current values of its internal metrics in a plain text format and the Prometheus server retrieves this data periodically, with a configurable interval, writing the resulting time series into its database. Tools like Grafana can be used to create dashboards to visualize the stored metrics, and the Prometheus Alertmanager can be configured to send notifications when certain conditions are met, based on the values of the metrics.

The scraping operations are managed by a configuration file for the server, that defines one or more scrape jobs. Each job comprises a set of targets sharing a common rationale, e.g. all instances of a given microservice. Targets can be specified statically by the admin or discovered dynamically through service discovery integrations, there is support for many platforms, among which figure Kubernetes, AWS, GCE, AWS EC2 and Azure. At each scrape interval, whose default is 15 seconds but is configurable with job granularity, Prometheus issues an HTTP GET request to the `/metrics` endpoint of each target. The response has to follow the Prometheus exposition format to be correctly interpreted by the server. It consists in a plain text response where each line contains: the metric's name, an optional set of labels, a numeric value (which is the metric's data), and an optional timestamp. There are four different supported metric types: Counters, which are values that monotonically increase (they can be seen as an integral over time, e.g. the number of bytes received on a certain interface), Gauges, which are values that can go up or down (e.g. the current total CPU utilization for a node), Histograms, which are observations that get bucketed by range, and Summary, which are similar to histograms,

but with bucketing calculations performed on the client application side. Scrape metadata is also kept track of, such as scrape duration and success status and is stored as separate metrics (e.g. `scrape_duration_seconds`).

Prometheus also exposes a functional query language called Prometheus Query Language (PromQL), which allows retrieval and computation over stored data. Queries rely on two fundamental data structures: the first being *instant vectors*, which are a set of time series, each with a single sample at a given timestamp, which is the same one for all of them and the second data structure being *range vectors*, which are a set of time series, each with a sequence of samples over a certain time interval. PromQL also supports arithmetic and logical operators, aggregation functions (e.g. `sum`, `avg`, `rate`, etc.), and filtering based on labels. The rate() function, for instance, computes the per-second average increase of a range over a specified time range, which is useful for observing behaviors such as the quantity of network traffic has been received from a pod in a second. Queries are handled through HTTP APIs and can be used directly by multiple entities, such as the Prometheus standard expression browser, Grafana dashboards and the alerting subsystem, which periodically checks the result of PromQL expressions against recording and alerting rules.

In the context of this thesis, Prometheus is used to collect and store metrics from the Slurm cluster for autoscaling reasons, primarily job queue length. It was also the key tool for evaluating the performance of the worker pods over the benchmarks described in Section 3.9, scraping its metrics from the `/metrics` endpoint of the cAdvisor instance running on the worker nodes.

3.9 Selected Benchmarks

In this section, the chosen benchmarks for evaluating the performance of the proposed solution are described in detail. The selection of benchmarks is motivated by the need to cover a range of workloads that are representative of typical use cases in high-performance computing environments. The benchmarks include HPL (High-Performance Linpack), which is a standard benchmark for measuring the floating-point performance of supercomputers, OSU Micro Benchmarks, which provide a suite of tests among which figure MPI communication performance tests and iPerf, a tool for measuring network bandwidth and performance. Each benchmark is described in terms of its purpose, the specific metrics it measures, and how it works.

3.9.1 HPL

HPL (High-Performance Linpack) [19] is a portable implementation of the Linpack benchmark for distributed-memory computers. Its primary goal is to evaluate the

floating point performance of a computing system by solving a randomly generated dense system of linear equations of the form $Ax = b$, where A is an $N \times N$ matrix of double precision floating point terms. The solution is obtained via LU factorization with partial row pivoting[5]. HPL is the benchmark used to produce the TOP500 list, which ranks the most powerful supercomputers worldwide by their obtained performance on the Linpack benchmark. The choice of HPL as a benchmark is motivated by its widespread adoption in the HPC community and its ability to stress the computational capabilities of a system, particularly its floating-point performance. Comparing results between different systems using HPL can provide information about the relative performance of different hardware architectures and software configurations.

The employed implementation[6] consists in an executable named `xhpl` (if the provided Makefiles are used). It is written in C and relies on two external libraries: an MPI library for inter-process communication across distributed nodes, in this case the software was dynamically compiled against `openmpi`, and a BLAS (Basic Linear Algebra Subprograms) library for performing the computationally intensive local matrix operations, in this case `openblas`. The build system is based on a provided set of `Make.<arch>` makefiles that handle the compilation process for a chosen architecture automatically, once the path of the dependencies is specified. Once compiled, xhpl can be launched as a standard MPI job, in this way `mpirun --npernode R ./xhpl`, where the total number of copies run $R \times num_nodes$ must be consistent with the process grid defined in the configuration file.

`xhpl` reads a single configuration file called `HPL.dat`, at startup, that controls the problem dimensions and the algorithmic choices applied during factorization. The main parameters are the following: N is the problem size, which is the order of the coefficient matrix A. It mainly determines the memory footprint and the total computational work and $P \times Q$ defines the two-dimensional MPI *process grid*, its choice should depend on the physical interconnection between nodes. There is no single configuration that is universally optimal, it is a matter of tuning those parameters to values that improve overall performance.

3.9.2 iPerf

iPerf [20] is a widely used network benchmarking tool designed to measure the network communication performance by generating and analyzing traffic exchanges between two endpoints. It is commonly used to evaluate network bandwidth, packet loss, and jitter. The reference implementation of iPerf is written in C and provides a

[5] `https://www.netlib.org/benchmark/hpl/algorithm.html`
[6] `https://www.netlib.org/benchmark/hpl/software.html`

command-line interface for configuring and executing tests, and it is called `iperf3`.

The fundamental mechanism behind iPerf is the client-server model in which one endpoint acts as a server that receives traffic, while the other endpoint generates a certain amount of traffic, which can be artificially upper bounded or left unrestricted. The server component listens on a certain port (default 5201) and waits for incoming connections. As in a standard client-server contract, the client initiates a session (not necessarily a connection at transport level) and sends traffic according to parameters specified by the user. During the test, the client transmits data packets continuously for a certain configurable period of time or until a specified amount of data has been transferred. The benchmark then measures the amount of data received and reports the results on both the endpoints.

Internally, iPerf can use standard transport protocols to perform its measurements, mainly TCP and UDP, but other options are available. When operating over TCP, the benchmark by default evaluates the maximum achievable bandwidth under the constraints of the reliability measures implemented by the protocol. Instead, in UDP mode, iPerf allows the user to specify a target data rate and transmits packets at that rate, obviously without relying on congestion control mechanisms. This allows to evaluate network characteristics such as packet loss and jitter, which can be of particular interested for those applications that have time constraints, such as video streaming and voice communication.

`iPerf3` is invoked from the command line. The server is started by imparting `iperf3 -s`, which starts the listening process on a certain port. The client connects with `iperf3 -c <server_address>`, and, if it is configured to initiate a test with default parameters, a single TCP stream with a 10 second measurement window is started with results reported every second.

`iPerf3` exposes a range of parameters that control the behavior of the test. The flag `-t` sets the test duration in seconds, alternatively `-n` can specify a total number of bytes to transfer. A very important flag for the employed setup is `-P`, which defines the number of parallel client streams. Multiple simultaneous TCP flows can better saturate high-bandwidth links where a single stream is limited by factors such as CPU processing power. Moreover, `-w` sets the socket buffer size, directly influencing the TCP window size and the maximum achievable bandwidth, for some large windows sizes it may be necessary to lessen the kernel limits with `sysctl`. Another interesting flag is `-u`, that switches the transport protocol to UDP.

iPerf was chosen as a benchmark of interest because it can measure the true performance limits of the interconnection mean, managing to saturate the link and providing insights on the network performance that can be achieved by the cluster, which is a critical aspect for HPC workloads that rely heavily on inter-node com-

munication. iPerf3 does not exactly represent a real application workload, besides certain heavy file transfer applications, but it is a useful tool for evaluating the performance of the cluster and allow to investigate on potential limitations given by the deployment mode of the Slurm worker nodes, as described in Chapter 6.

3.9.3 OSU Micro Benchmarks

OSU Micro Benchmarks [78] are a suite of benchmarks developed by the Ohio State University, designed to evaluate network communication performance, among which figures a collection of benchmarks dedicated to measure MPI performance. OMB comprises tests written in multiple languages, namely C, Java and Python, the chosen tests are the ones that are written in C. This suite is structured as a set of independent executables, each typically targeting a specific MPI operation or communication pattern. The suite covers point-to-point operations and collective operations. By examining the outcome of specific MPI operations, the benchmark suite can provide information about how efficiently a distributed system handles network data exchange between computing nodes when a certain communication library is used.

Point-to-point benchmarks, such as osu_latency and osu_bw, involve just 2 MPI processes. osu_latency measures the round-trip latency of small messages using a ping-pong exchange (similarly to the Intel MPI Benchmark PingPong) and reports half of the average round-trip time. osu_bw measures *unidirectional* bandwidth by sending a certain number of messages in a non-blocking way (MPI_Isend()), reporting the estimated transfer rate across a range of message sizes. These benchmarks iterate over different message sizes, reporting one measurement per size. Each measurement is obtained by averaging over a number of iterations. It is also possible to include a warm-up phase to allow the network to reach steady state before results are recorded.

The collective portion of OMB benchmarks includes tests for a wide range of collective MPI operations, performed on a certain number of MPI processes. Among these, osu_alltoall and osu_allreduce are of particular interest for this thesis, since they are representative of realistic communication patterns that can happen on a regular basis in HPC parallel workloads. osu_alltoall reports the latency of the MPI_Alltoall() operation, in which every process sends a message to every other process in the domain and then receives a message from each one of them. This operation generates a communication volume that scales quadratically with the number of involved processes. osu_allreduce measures the latency of the operation MPI_Allreduce(), which performs a reduction operation (a summation) utilizing data that come from every process and then distributes the final result to all of them. Both of the selected benchmarks return results for a range of message

sizes. The reported latency is averaged over a number of iterations and reported in microseconds.

The specific choice of `osu_alltoall` and `osu_allreduce` is motivated by the fact that they better represent regular workloads than iPerf, and it is interesting to see how the cluster performs under these communication patterns.

Chapter 4

SliMe Architecture

This chapter presents the solutions developed towards the realization of SliMe, with the aim of integrating instances of SchedMD's Slinky running on two container orchestration platforms, namely Kubernetes and OpenShift, with bare-metal provisioning through Cluster API and its Metal^3 infrastructure provider. It describes how those solutions were designed and implemented for the experiments reported in Chapter 6.

The chapter is organized as follows. Section 4.1 describes the preliminary evaluation, in which a manual deployment of the system was performed to assess the feasibility of connecting a Slinky-managed Slurm controller pod with bare-metal Slurm worker nodes, and presents the two architectures used to collect the comparative performance data between pod-based and bare-metal workers. Section 4.2 introduces the *Injection Webhook*, a Kubernetes Mutating Admission Webhook that automates the registration of bare metal nodes provisioned through Cluster API with the Slurm controller pod, removing the manual configuration steps required in the preliminary architecture. Subsequently, Section 4.3 addresses the integration of Slinky's proposed autoscaling mechanism with Cluster API `MachineDeployments`, highlighting the limitations that prevent a direct use of KEDA and HPA on `MachineDeployments` and the architecture used to overcome them, which is based on the *Scaling Webhook*. The same section also describes the refined architecture that introduces DNS-based service discovery to support and ease the management of multiple Slurm clusters from a single Kubernetes control plane.

4.1 Preliminary Evaluation

This section presents the architectures employed to conduct a comparative evaluation of the performance of Slurm worker nodes deployed on bare metal machines against Slurm worker nodes deployed as Kubernetes pods via Slinky, as shown in Figure 4.1. These architectures address the investigation for **RQ1**.

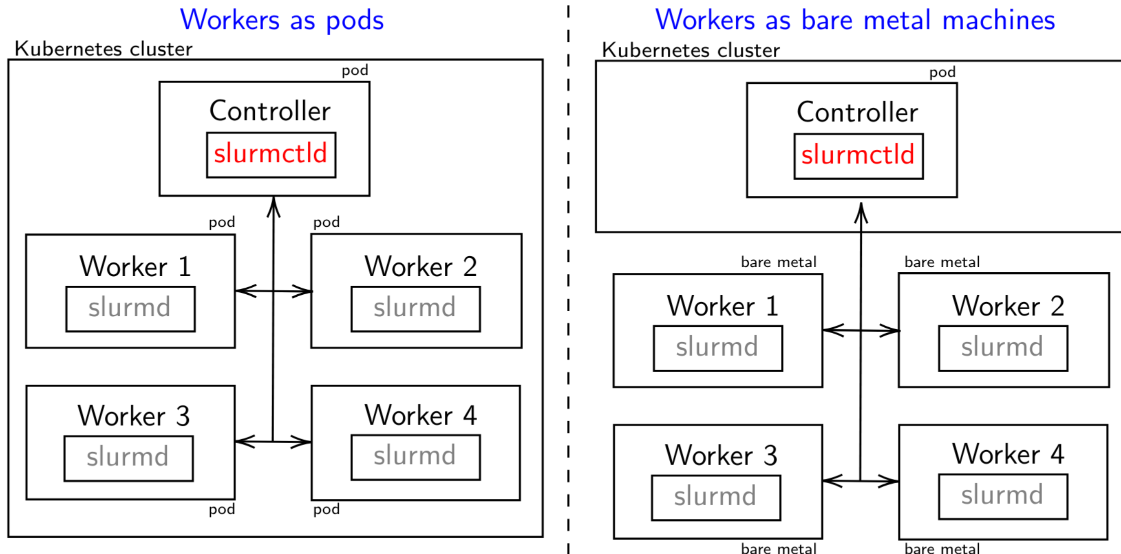


Figure 4.1: Employed architectures: bare metal vs. pods

First of all, by keeping in mind the employed technologies and the selected benchmarks for the comparative analysis, reported in Chapter 6, the section then presents an in-depth description of the proposed solution, its architecture and the rationale behind the main choices in the design. Finally, implementation and deployment details that can be required to reproduce the experiments are provided, accompanied by a number of notes on configuration and metrics collection used during the evaluation.

4.1.1 Proposed solution

The proposed solution has the goal of allowing for a non-automated integration of Slinky and an appropriately modified version of an "extension to the Kubernetes cluster life-cycle management tool Cluster API to automate the provisioning, bootstrapping, and graceful de-provisioning of HPC batch scheduling platforms" [76], which is set of tools that allows to provision a complete, standalone, bare metal Slurm cluster with Cluster API. The next sections will describe a more integrated system, it is based on the deployment of two separate entities: a stan-

dard Slinky installation that will provide the Slurm controller pod and a Cluster API `MachineDeployment` that will manage the bare metal Slurm worker nodes (see right-side of Figure 4.1).

Instead, regarding the left-side of Figure 4.1, the architecture is the standard Slinky deployment where both the Slurm controller and worker nodes are deployed as pods inside the same Kubernetes cluster. While the description of this particular configuration is beyond the scope of this section, it is worth noting that the tests on the standard Slinky installation were performed on a different orchestration platform than Kubernetes, namely OpenShift. This latter platform has different and stricter security policies that required some adjustments to the installation procedure, those changes, along with the chosen deployment steps, are described in Section 5.1.1.

4.1.1.1 Preliminary integration architecture

The preliminary architecture is the result of an initial exploration on the feasibility of integrating Slinky with Cluster API to have bare metal Slurm worker nodes and a Slinky-managed Slurm controller (`slurmctld`) pod with the objective of collecting results to be compared with an all-pod solution (i.e. both `slurmd` and `slurmctld` are hosted inside pods and thus, containers).

The first steps towards this architecture were taken by deploying a Slinky installation on OpenShift, to be more precise, a particular version called `slinky-on-openshift` was chosen (see Section 5.1.1). Then a set of 4 physical (bare metal) nodes were isolated and taken from the *Morrigan cluster* and connected to the Slinky-managed Slurm controller pod. The Morrigan Cluster is an HPC cluster for research purposes that is owned by IBM. It is used as a testbed for an OpenShift environment, an article that describes the testbed and that is written by the researchers who manages the aforementioned cluster is available on the web[1].

The Slinky installation runs on a generic OpenShift node, while the chosen physical nodes were isolated from the Morrigan cluster and did not run any non necessary workload besides the standard daemon `slurmd` and the core OS services, obviously no kind of Kubernetes/OpenShift related workload (e.g. the kubelet) was running, since the nodes were completely isolated and drained of all assigned workload. This was done to remove potential performance degradation due to the utilization of CPU by non required entities, that could have biased the final results in the experiments.

Concisely, the solution workflow was the following:

[1] `https://medium.com/@michele.gazzetti/morrigan-a-shape-shifting-testbed-for-next-gen-research-computing-c517b3cfeb19`

1. Isolate a set of physical nodes from the cluster and drain them from any unnecessary workload.
2. Deploy Slinky on OpenShift (possibly using the `slinky-on-openshift` repository).
3. Enable the communication between the Slurm controller pod and the physical nodes with the tools provided by OpenShift
4. Start the `slurmd` daemon on the physical nodes and configure them to register with the `slurmctld` pod

All the deployment steps and the configuration choices that were made to enable the interaction is described in Section 5.1.2, since they are beyond the scope of this section.

This initial configuration allowed to have a working architecture with a Slinky-managed Slurm controller and bare metal Slurm worker nodes and it is the architecture on which the performance tests were taken, since all the subsequent Cluster API experiments were performed inside `metal3-dev-env` that was entirely hosted on a single, isolated, Morrigan Cluster node and thus, in a reduced resource environment that was not suitable for comparative performance tests. All the metrics collected from the preliminary architecture, among which there are total CPU usage from a node, CPU usage for a particular cgroup to which a container refers, inbound and outbound network communication traffic, were taken from OpenShift's built-in Prometheus service that can scrape the metrics that the resource monitoring system Cadvisor exposes on the endpoint `/metrics` of each node. Those metrics were then compared with the ones collected from a standard Slinky installation where both the controller and worker nodes were hosted inside pods, as can be seen in Figure 4.1, with an analogous hardware environment.

This preliminary deployment therefore was the foundation to build a concrete baseline: it validated the feasibility of the integration, it enabled the possibility to have definite and quantitative comparisons between pod-based and bare-metal workers, and helped to define the deployment constraints (for example what type of service configuration was needed to enable a correct communication between nodes and pods) that led to the design of the final and automated solution described in the next sections.

4.1.1.2 Manual integration of Slinky with Cluster API

After having deployed the first test architecture and having collected performance data from the executed tests, the natural following step to answer **RQ2** was to assess the feasibility of connecting bare metal nodes managed by Cluster API with the

Slurm controller pod, this time using Kubernetes for availability reasons on the test environment. In the subsequent passage, the entities involved and their respective roles in the architecture that uses Cluster API are delineated, accompanied by notes on the required interactions and metrics retrieval.

As already mentioned, Slinky provides the in-cluster Slurm control plane, a pod called `slurm-controller`, and a set of useful services, among which there is one of type ClusterIP that exposes on the cluster network the port 6817, which is the default one for communications with the Slurm head. Instead, regarding the physical bare metal nodes, Cluster API infrastructure provider Metal[3] is used as the external provisioning layer that creates and manages bare-metal `Machines` which later become Slurm worker nodes via a configuration contained in a `VirtualKubeletConfigTemplate`, which is uniformly referred to by all the `Machines` in the `MachineDeployment`.

Concerning the interactions between the entities, the worker nodes running `slurmd` register with `slurmctld` over the shared network between the Kubernetes cluster and the bare metal nodes, which is not the cluster network itself, but a separate one that is dedicated to the deployed machines. If metrics are needed, it is possible to use an ad-hoc installation of Prometheus to scrape the `slurmctld` metrics endpoint (`slurm-controller:6817/metrics/...`), since there is no default metric retrieving service available.

To completely automate the integration between Slinky and Cluster API, there were some manual steps that were needed to be performed, such as retrieving the controller pod host IP and the authentication key `slurm.key` from the secret `auth-slurm`, and then configuring the `VirtualKubeletConfigTemplate` to use those data to configure the bare metal nodes to be able to register with the controller pod.

Refer to Section 5.1.2 for the deploying process and configuration notes.

4.2 Injection Webhook - Integration with Slinky

In this section, the solution for automating the integration between Slinky and bare metal workers provisioned with Cluster API is described, naturally following the investigation for **RQ3**. The main goal of this automation is to remove the manual steps that are needed to enable the communication between the controller pod and the worker nodes, which was still needed in the preliminary architecture described in Section 4.1.1.2, making the process of provisioning and managing the worker nodes more seamless and time-efficient. Moreover, the automation of the integration allows to have a more dynamic and flexible architecture, where entire

sets of worker nodes can be added and removed at runtime without needing to modify the configuration files or to perform manual steps, which is particularly useful in the context of a Kubernetes-centric environment where the number of worker nodes can be dynamically adjusted based on the workload and the available resources.

4.2.1 Kubernetes Admission Webhooks

Kubernetes Admission Webhooks are a mechanism that allow extending the validation step of the Kubernetes API server by allowing to execute custom logic in correspondence of certain API calls, namely *Admission Requests* which are HTTP requests that contain operations to perform on Kubernetes resources. The fundamental concept is that the validation step happens prior to the persistence of the object, but after the request authentication (see Figure 4.2). This means that, when an API call is made to the Kubernetes API server, the request goes through a series of steps before the object is actually created or modified in the cluster. The first step is the authentication, where the API server verifies the identity of the requester and checks if it has the necessary permissions to perform the requested operation. After that, the request goes through the admission control phase, where it is processed by a series of admission controllers, which are built-in inside Kubernetes API server, and optionally by custom admission webhooks, which are the point of extension of admission controllers. So, in a concise way, Admission Webhooks are HTTP callbacks that operate on Admission Requests in a certain way.

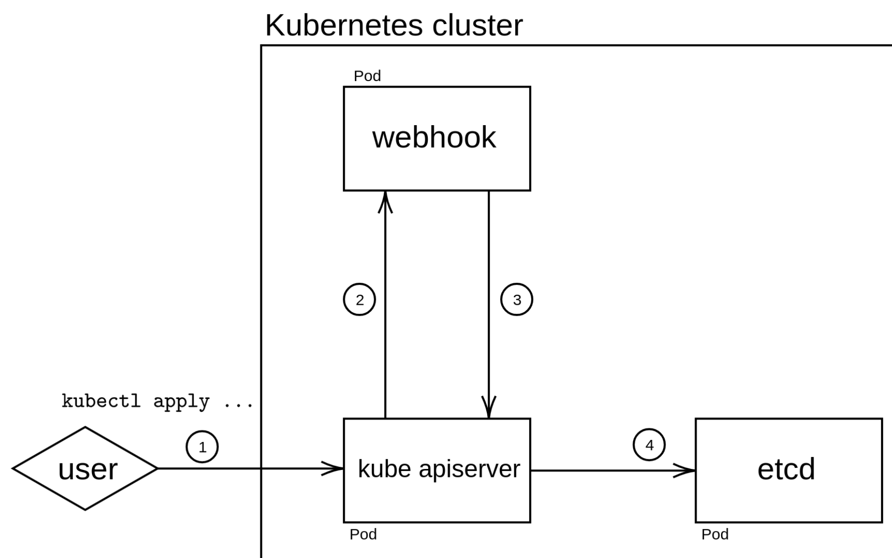


Figure 4.2: Simplified flow for the admission requests and admission webhooks

There is a distinction between two types of Admission Webhooks, that differs from

what kind of operation they can perform on the intercepted Admission Requests: Mutating Admission Webhooks and Validating Admission Webhooks. The latter are used to only admit or refuse the requests, while the former are used to modify them, typically to satisfy some policies, in addition they can also admit or refuse the requests, but their main purpose is to modify them, and it is best practice to let a validating webhook being the one that performs the decision on whether to admit or refuse the request. The main use case for Mutating Admission Webhooks is to modify the incoming request in a way that it satisfies certain policies, for example by adding certain labels or annotations to the object being created, or by modifying certain fields of the object to ensure that they are compliant with certain chosen standards. While Validating Admission Webhooks are typically used to enforce certain policies by admitting or refusing the incoming requests based on certain criteria, for example by checking if the object being created has certain required fields or has a certain format, and refusing the request if it does not satisfy those criteria.

Admission webhooks can be configured with objects called `MutatingWebhookConfiguration` and `ValidatingWebhookConfiguration`, depending on the type of webhook. Inside these objects, it is possible to specify the rules that determine which Admission Requests should be intercepted by the webhook, for example by specifying the API groups, versions, resources, and operations that should trigger it.

This mechanism of interception of requests is particularly useful in this thesis, since it is possible to inject data into objects before their effective persistence on the cluster, and, running inside the cluster, it can have access to all the resources if the correct permissions are set. In a simplified description, the process described in Figure 4.2 can be presented as follows:

1. The user submits a `kubectl apply ...`, sending an Admission Request to the API server (in this figure a user is depicted, but it can be any entity that can send a call to the Kubernetes API server)

2. The API server authenticates the request and sends it to the webhook (it can be either a mutating or a validating webhook, or both in a sequence)

3. The webhook returns the admission response, which can be positive or negative and can contain modifications to the original request in case of a Mutating webhook

4. The API server sends the request to be persisted on `etcd` if the admission response is positive, otherwise it returns an error

4.2.2 Cert-manager

Cert-manager is an open-source Kubernetes X.509 certificate controller that automates the management, issuance, and renewal of those certificates which are used in protocols like TLS, which is the base for HTTPS. Cert-manager also ensures that the certificates are valid and up-to-date and attempts to renew certificates at a configured time before their expiration [13].

The `Issuer` and the `ClusterIssuer` are fundamental components of cert-manager, they are custom Kubernetes resources that refer to a certificate authority (CA) and that can sign certificates upon certificate signing requests, the main difference between the two is that the former is namespace-scoped while the latter is cluster-scoped. The Issuer can be configured to use different types of CA, for example a self-signed CA, a CA that is managed by cert-manager itself and is typically used to build in-cluster trust, or an external CA that can be from many providers, such as AWS Certificate Authority or Microsoft Active Directory Certificate Service. The Issuer is used to issue certificates for the Kubernetes resources that require them, for example for the Admission Webhooks, which require TLS certificates to secure the communication between the API server and the webhook server.

With cert-manager, X.509 certificates are stored inside in-cluster secrets through the use of the `certificate` CRD, those secrets contain the certificates themselves and the private key. They can be used by other Kubernetes resources (e.g. Pods that contains webhooks servers) to be mounted as volumes. The secret containing the certificates, has three fields in its data, which are `tls.crt`, `tls.key` and `ca.crt`, the first two contain the certificate and the private key, while the last one contains the CA certificate that is used to sign the certificate, this latter field is particularly useful for the clients that need to trust the certificate, since they can use it to verify the authenticity of the certificate.

The `certificate` CRD represents a declarative specification of a desired certificate, i.e. a human readable definition of a certificate request. When a `certificate` object is created, cert-manager automatically creates a corresponding certificate signing request (CSR) and submits it to the configured Issuer for signing. Once the CSR is approved and signed by the Issuer, cert-manager retrieves the signed certificate and stores it in the specified secret. Inside the CRD, there are fields that allow specifying the desired duration and the renewal time for the certificate, so that cert-manager can automatically renew the certificate before it expires (the `renewBefore` field specifies the required amount of time before the expiration that triggers the certificate renewal request). This process allows to automate the management of TLS certificates for Kubernetes resources, ensuring that they are always valid and up-to-date without requiring manual intervention, anyway if certificate renewal is

desired, it can be invoked at any time with the `cmctl` CLI tool.

Another important service that is provided by cert-manager is the key rotation, which allows to automatically rotate the private keys of the certificates, this is particularly useful for security purposes, since it allows to reduce the risk related to the certificate key being previously compromised. Key rotation is performed upon certificate reissuing, which can be triggered by a renewal process or by inconsistencies such as the certificate secret missing, mismatches between the `certificate` specs and the private key, DNS names, IP addresses, URLS or email addresses. The key rotation policy can be set inside the `certificate` object itself and can be `always`, i.e. every certificate (re-)issue or `never`. The default value used to be `never`, but in the most recent versions of the software it was changed to `always`, which is a more secure default choice. This key rotation is important in such cases as the private key of a certificate being compromised, since in cert-manager there is no built-in method to check for certificate revocations (there is no support for both CRL [9], acronym for Certificate Revocation List, which is a list that gets periodically released by the CA containing the revoked certificates and the client should check against it during verifications, nor for OCSP [67], which is Online Certificate Status Protocol, that consist in a service that is provided from the CA that provides real-time protocol status upon request, this latter method is being made optional [23] and even discontinued by some major CA providers due to privacy concerns[2] [73]). Moreover, Kubernetes core components do not handle certificate revocation by themselves[3], hence the key rotation acquires additional value.

A number of Kubernetes objects, among which are Admission Webhooks use the field `caBundle` in `webhookConfiguration` objects to specify the CA certificate that is used by the Kubernetes API server to verify the authenticity of the TLS certificate used by the webhook server. Cert-manager provides a convenient way to automatically inject the CA certificate into the `caBundle` field of the webhook configuration objects through the use of the CA Injector, which is a component of cert-manager that watches for changes in the certificates and automatically updates the corresponding `caBundle` field in the webhook configuration objects. This allows to ensure that the API server can always verify the authenticity of the TLS certificates used by the webhook server, even in cases where the certificates are renewed or rotated, without requiring manual intervention to update the `caBundle` field. To enable this feature, it is sufficient to add the annotation `cert-manager.io/inject-ca-from` or `cert-manager.io/inject-ca-from-secret` which, respectively, specifies the name of the `certificate` object or the name of the `Secret` containing the certificate that should be injected into the `caBundle` field of the *Injectable Resource*, which can be

[2] https://letsencrypt.org/2024/12/05/ending-ocsp
[3] https://github.com/kubernetes/kubernetes/issues/18982

ValidatingWebhookConfiguration, MutatingWebhookConfiguration, CustomResourceDefinition or APIService.

4.2.3 Proposed solution

The natural subsequent step to the preliminary solution described in Section 4.1.1.2 is to automate the integration between Slinky and Cluster API by removing the manual steps needed to render the worker nodes to be able to register correctly with the controller pod. In particular, as previously introduced in the same Section, there was the need of retrieving the controller pod host IP and the authentication key slurm.key, both obtainable from in-cluster Kubernetes objects, respectively from the controller pod itself and from the secret auth-slurm. As conveniently introduced in Section 4.2.1, Kubernetes Admission Webhooks are an ideal candidate to serve this particular purpose, since they allow intercepting the creation of a particular object and modify it. For this exact reason, a Mutating Admission Webhook is introduced in the cluster, and it is configured to intercept the creation of a particular object of kind Secret.

When a Machine object is deployed, both as a part of a MachineDeployment or as a standalone object, and the infrastructure provider for Cluster API is set to be metal[3], it has a certain Metal3MachineTemplate inside its infrastructureRef field. This template contains a field called dataTemplate that can refer to a Metal3DataTemplate object. This object, likewise, contains another set of fields[4], among which there is the metaData field. This latter field is used to populate a Secret called <machine_name>-metadata, that can be used during the provisioning of the bare metal node to retrieve some useful data referring to the machine. In particular, the used bootstrap provider cluster-api-bootstrap-virtual-kubelet (see Section 3.7) allows to use some of the fields contained in the aforementioned secret to populate a template for a cloud-init configuration, which is leveraged to configure the bare metal node to make it ready for the environment in which it is provisioned.

For this reason, a Mutating Admission Webhook called *Injection Webhook* was developed and deployed in the cluster, intercepting the creation of Secret objects, with specific matchConditions that identifies secrets owned by a Metal3Data object and that contains a metaData field. The webhook, once it intercepts the admission request, retrieves the controller pod host IP from the controller pod and the authentication key slurm.key from the secret auth-slurm, then it injects this data into the metaData field of the intercepted secret, which will be used during

[4]https://book.metal3.io/capm3/data_sources.html#5-metal3datatemplate--metal3data

the provisioning of the bare metal node to configure it without needing any manual step. In this way, the integration between Slinky and Cluster API is automated, and it is possible to dynamically add and remove worker nodes at runtime without needing to modify any configuration file.

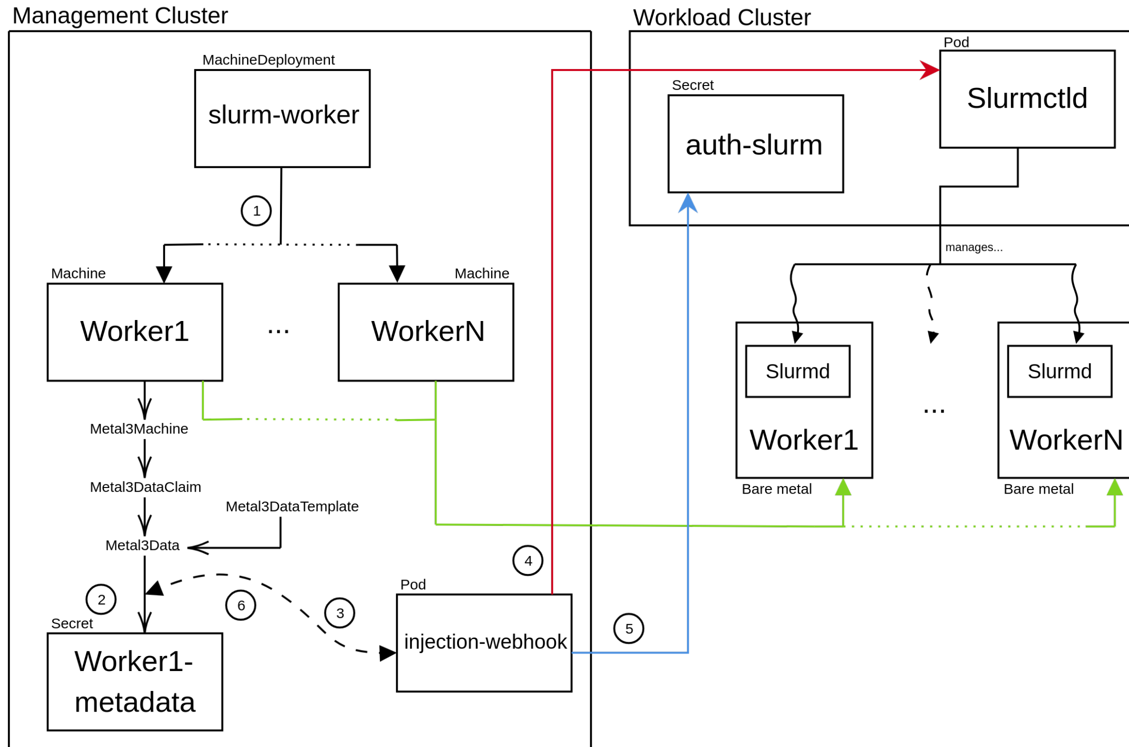


Figure 4.3: Simplified schema for the injection webhook workflow

Moreover, to have less configuration needed, the Slinky installation was moved from the management cluster to the workload cluster. This allows to avoid the need of deploying a set of iptables rules to forward the packets directed to the controller pod to the docker container(s) running the Kind cluster. In this way, the communication between the controller pod and the worker nodes is enabled without needing to have a manual step, since the Slinky installation will be hosted inside (virtual) bare metal nodes created with `metal3-dev-env` that are already connected and there are no intermediary layers such as in-docker Kubernetes clusters in between, like in the previous solution in which the bare metal nodes were not physically connected with the Kubernetes cluster network, since it was encapsuled in a Docker network. In addition to having the bare metal worker nodes directly connected with the controller pod host, moving Slinky to the workload cluster makes possible to obtain a configuration in which multiple Slinky installations are managed from the management cluster, allowing for a unified control plane that can manage all the

lifecycle of an HPC cluster completely via Kubernetes, this particular configuration will be further described in Section 4.3.3.

A visualization of the workflow can be seen in Figure 4.3, which can be described as:

1. The number of replicas in the `MachineDeployment` is increased, thus triggering the creation of a new `Machine` object
2. A chain of intermediate objects are created and shown in a reduced form for brevity, first the `Machine` object, then the `Metal3Machine`. Since the latter contains a reference to a `Metal3DataTemplate`, an object of kind `Metal3DataClaim` is created, this object commands for the creation of a `Metal3Data` for the `Machine`, the same controller that handles the resource `Metal3DataTemplate` also creates the final `Secrets`, among which there is the *metaData* one.
3. The Injection Webhook intercepts the creation of the `metaData` secret
4. It retrieves the host IP from the `slurmctl` pod
5. It retrieves the `slurm.key` from the `auth-slurm` secret
6. It emits a patch that modifies the *metaData* secret by adding the data inside its `metaData` field

The realization of this architecture resolved positively the **RQ3**, since it allows to have a completely automated deployment of the worker nodes with Cluster API, with them being able to connect to the controller pod in the same way as described in the preliminary architecture.

Refer to Section 5.2 for the implementation details and the deployment steps for the Injection Webhook.

4.3 Scaling Webhook - Autoscaling integration

In this section, a solution for integrating the autoscaling system proposed by the creators of Slinky [68] with the architecture designed in this thesis is presented. First, a brief comparison between two popular autoscalers is provided, then the proposed solution is described in detail.

4.3.1 Autoscalers comparison

In the following, two popular autoscalers for Kubernetes are presented and compared from both their features and limitations: KEDA and Cluster Autoscaler. This short

comparison focuses on aspects that matter for integrating autoscaling with the previously described Slurm deployment enabled by Cluster API, for example whether the scaler operates at pod or node level, the kinds of triggers and metrics it supports, compatibility with Custom Resources (CRs) (like `MachineDeployments`), and typical operational constraints. The main goal is to highlight practical limitations that motivate the autoscaling integration proposed later but not to provide excessive operational details on the said technologies.

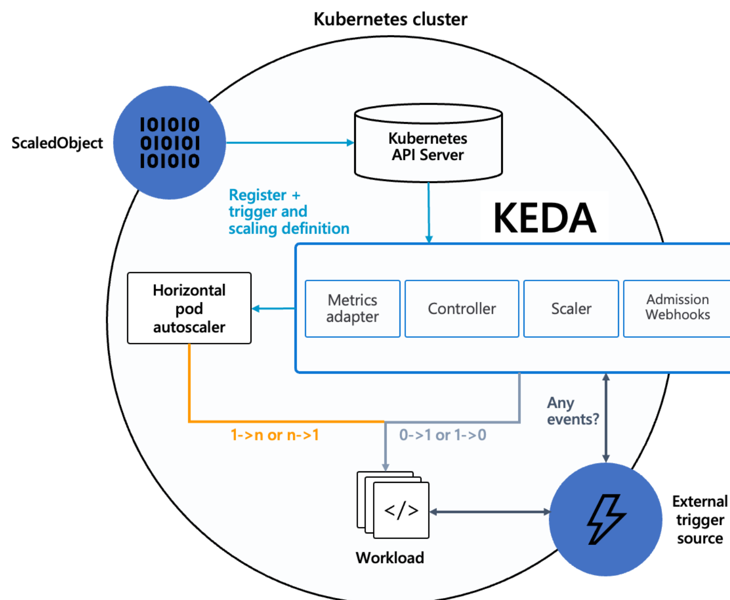
4.3.1.1 KEDA

KEDA (Kubernetes-based Event-Driven Autoscaler) [33] is an autoscaler that allows for fine-grained scaling of Kubernetes objects based on external event sources. It is designed to work with a variety of event sources, like message queues length, utilization of certain resources or even the time of the day.

Architecture and operational overview
The fundamental component of KEDA is the *Operator*, which continuously watches event sources and scales the targeted Kubernetes objects (e.g. Deployments) accordingly. In addition, KEDA introduces the concept of *Scaler*, which is an abstraction that allows connecting to an external event source, gather metrics from it and expose those metrics in a format that is understandable for the entities that regulate the autoscaling process.

There are many available built-in Scalers, each one designed to interact with a specific event source (e.g. Prometheus, Apache Kafka, RabbitMQ, MySQL, ...), but it is also possible to create custom user-defined Scalers if needed, it is sufficient to write a gRPC service that implements the Scaler interface as defined in KEDA documentation. This extensibility makes KEDA suitable for numerous use cases, from more typical cloud-native pod scaling, to less usual instances, such as the HPC provisioning environment considered in this thesis.

KEDA also defines various CRDs, one of the most relevant being the `ScaledObject`, which specifies how a given Kubernetes object (`target`) should be scaled using metrics gathered by one or more configured Scalers. A `ScaledObject` typically declares triggers (the event/metric sources), bounds such as `minReplicas` and `maxReplicas`, and timing parameters like `pollingInterval` and `cooldownPeriod`. Upon application of a `ScaledObject`, KEDA creates a Horizontal Pod Autoscaler (see Paragraph "Horizontal Pod Autoscaler (HPA)") attached to the target. Then, the HPA then uses metric values retrieved by KEDA to compute the desired replica count. It is worth noting that KEDA does not aim to replace HPA, but rather to add more possibilities to it. Indeed, KEDA is responsible for the metric retrieval



source: https://keda.sh/img/keda-arch.png

Figure 4.4: KEDA architecture

and for the scaling to/from zero logic, while HPA handles the actual computation of the desired number of replicas during the *Scaling Phase*.

There can be two different phases during operation: the *Activation Phase* and the *Scaling Phase*. During the Activation Phase, the scalable Kubernetes resource is scaled only from 0 to 1 (and from 1 to 0) when a certain threshold is reached (called `activationThreshold` inside the `ScaledObject`). This phase is managed entirely by the KEDA Operator and does not involve HPA, since HPA does not natively support scaling to zero replicas. During the Scaling Phase instead, the scaling operations get delegated to the HPA, that is responsible for scaling the resource from 1 to N and vice versa, using the metrics that KEDA feeds into the Kubernetes Resource Metrics Pipeline.

Horizontal Pod Autoscaler (HPA)

An HPA [38] automatically scales horizontally (i.e. deploys more/fewer pods) a Kubernetes object such as a Deployment or a StatefulSet based on a certain metric. It is the dual of a Vertical Pod Autoscaler (VPA), which instead scales vertically (i.e. increases/decreases the resources assigned to a pod rather than replicating it). It operates by periodically querying the metrics server for the current value of the desired metric that influences the scaling decision (e.g. CPU utilization). The logic

behind the scaling decision is given by the following equation:

$$desiredReplicas = \left\lceil currentReplicas \cdot \left(\frac{currentMetricValue}{desiredMetricValue} \right) \right\rceil \tag{4.1}$$

This means that, if, for example it is desirable to maintain `500MB` of memory and the current memory usage is `1000MB`, the number of replicas will be doubled, vice versa will be halved if the current memory usage is `250MB`.

A HPA can be quickly created and attached to an existing Kubernetes object using the `kubectl autoscale` command, e.g.:

```
kubectl autoscale deployment <deployment-name> --min=2 --max=10
↪  --cpu-percent=75
```

This command will create a HPA that scales the `my-deployment` Deployment between 2 and 10 replicas, trying to maintain an average CPU utilization of 75%. Specifying bounds is not strictly necessary, but it is a good practice to avoid having a very high number of replicas, the minimum value for `minReplicas` is 1 since HPA does not natively support scaling to 0 replicas.

HPA is a native Kubernetes component that can be used independently from KEDA, but KEDA leverages it to perform the scaling operations during the Scaling Phase.

Limitations
By design, KEDA relies on HPA for the scaling phase (from 1 to N replicas), which introduces some limitations, the biggest of them is that it only supports objects exposing the `/scale` subresource (`kubectl get --subresource scale ...`) and also having a selector inside of it that allows to identify a certain group of pods (that typically are those that get actually scaled), this latter requirement is necessary to have a feedback on how many pods are ready after a scaling operation. This means that KEDA cannot scale objects like `DaemonSets`, since they do not expose the `/scale` subresource but also objects like Cluster API's `MachineDeployments`, since their subresource does not have a selector defined inside of it that returns a valid set of pods.

4.3.1.2 Cluster Autoscaler

Cluster Autoscaler [5] is an autoscaler that changes the size of a Kubernetes cluster by adding or removing nodes based only on pods' schedulability needs. It continuously monitors the cluster for pods that cannot be scheduled due to insufficient resources and for nodes that are underutilized.

Cluster Autoscaler works at the node level and support multiple cloud providers such as Amazon AWS, HuaweiCloud, Microsoft Azure, and others. It can also work with Cluster API as a provider to manage the scaling of custom Kubernetes clusters deployed on various infrastructures.

Operational overview

Cluster Autoscaler adjusts the size of the Kubernetes cluster when one of the following conditions is true:

- there are pods that failed to schedule on any of the current nodes due to insufficient resources.
- there are nodes in the cluster that have been underutilized for an extended period of time and their pods can be placed on other existing nodes.

A Cluster Autoscaler instance manages one or more *Node Groups*, that represent sets of uniform nodes with respect to available resources. When unschedulable pods are detected it requests the cloud provider to provision additional nodes to accommodate the pending workload. It does this by evaluating the resource requests of the unschedulable pods and determining how many additional nodes are needed to satisfy those requests, every Node Group contains the necessary information to determine how much resources a new provisioned node belonging to that group will offer. Node groups may contain different types of configurations such as minimum and maximum number of nodes or the amount of underutilization time needed for a node to be considered as unneeded (by using the annotation `cluster-autoscaler.kubernetes.io/scale-down-unneeded-time`).

This Autoscaler decreases the size of the cluster when some nodes are consistently unneeded for a significant amount of time by draining them and transferring their workload in another part of the cluster. A node is unneeded when it has low utilization and all of its important pods can be moved elsewhere[5].

Limitations

Cluster Autoscaler does not give fine-grained control over scaling operations, as it only adjusts the number of nodes in the cluster based on resource (in)utilization and pending pods, it does not offer the possibility of defining custom metrics in a KEDA-like fashion and this makes difficult to use it on its own in scenarios where more specific scaling policies are required, especially when dealing with resources that cannot be scaled at the pod level.

[5]`https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/FAQ.md#when-does-cluster-autoscaler-change-the-size-of-a-cluster`

4.3.1.3 Possible interactions between KEDA and Cluster Autoscaler

It is possible to use KEDA (and HPA) in conjunction with Cluster Autoscaler to achieve a sort of multi-level autoscaling [6]: KEDA/HPA can be used to scale the number of pods within a scalable object based on a specific metric, and if the current amount of pods exceeds what the current dimension of the cluster can handle, Cluster Autoscaler can intervene and add more nodes to the cluster to accommodate the increased workload.

In more detail, the typical interaction works as follows: KEDA observes an external metric and manages an HPA with the goal of adjusting the number of pod replicas for a given object. Then, if those additional pods cannot be scheduled because there are not enough node resources, Cluster Autoscaler detects the unschedulable pods and requests the underlying infrastructure, through the configured node group, to provision additional nodes. Once new nodes are provisioned and connected to the cluster, the pending pods can be scheduled and the workload can be correctly handled. This multi-layered approach thus provides pod-level scaling driven by metrics with KEDA/HPA together with cluster-level, node scaling driven by scheduling capacity with Cluster Autoscaler, that can be well compatible with standard Kubernetes workloads.

However, this combination can have important limitations that make it unsuitable for the specific case of scaling Cluster API's `MachineDeployments` used as Slurm worker machines, which is the case that is the most interesting in this work. Cluster Autoscaler and HPA are designed around the model where nodes host arbitrary Kubernetes pods and where scaling decisions are mainly driven by pod schedulability. In contrast, the `Machines` created by a `MachineDeployment` in our scenario are intended to become dedicated Slurm worker nodes that run `slurmd` and Slurm workload, rather than general-purpose Kubernetes node targets for the scaled pods. Because of this fundamental difference in the intended role of the provisioned machines and in the subresource/selector results required by HPA/KEDA to count the number of ready entities, the standard KEDA/HPA + Cluster Autoscaler multi-leveled integration cannot be used to directly scale up `MachineDeployments` for Slurm workers, which is the case explained in Section 4.3.2.

4.3.2 Proposed solution

In this section, the autoscaling solution that Slinky suggests is presented, along with its limitations that renders it natively incompatible with the architecture based on

[6] `https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/FAQ.md#how-does-horizontal-pod-autoscaler-work-with-cluster-autoscaler`

Cluster API objects presented in this thesis. Finally, the proposed architecture needed to make it functional and to formulate an answer for **RQ4** is described.

4.3.2.1 Slinky proposed autoscaling system

Inside Slinky's documentation [68], an autoscaling system based on KEDA is proposed to automatically scale the number of worker nodes in a Slurm cluster deployed on Kubernetes. The idea is to create a `ScaledObject` with a `NodeSet` as its target that monitors the length of the Slurm job queue using a Prometheus scaler and, when the number of pending jobs exceeds a certain threshold, it changes the number of worker nodes accordingly by scaling up or down the `NodeSet` that manages the worker nodes in form of pods.

In the following yaml snippet, a partial example of such `ScaledObject` is shown:

```yaml
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
spec:
  scaleTargetRef:
    apiVersion: slinky.slurm.net/v1beta1
    kind: NodeSet
    name: slurm-worker-nodeset
  idleReplicaCount: 0
  minReplicaCount: 1
  maxReplicaCount: 3
  triggers:
    - type: prometheus
      metricType: AverageValue
      metadata:
        serverAddress: <prometheus-server-address>
        query: slurm_partition_jobs_pending{partition="slinky"}
        threshold: '1'
```

With this configuration, when there is at least one pending job in the queue for the partition "`slinky`", KEDA will scale up the `NodeSet` that is identified by the name "`slurm-worker-nodeset`" from 0 to 1 (*activation phase* as described in Paragraph "Architecture and operational overview" inside Section 4.3.1.1), then transferring the control for $1 \leftrightarrow N$ scaling operations to the created HPA for the *scaling phase*. It is interesting to note that limits for the number of replicas are defined directly inside the `ScaledObject`, in correspondence of the fields `minReplicaCount` and `maxReplicaCount`, those values are then propagated to the HPA upon its creation.

The field `idleReplicaCount` defines the number of replicas to maintain when there are no pending jobs in the queue, with KEDA the only value supported is 0[7], meaning that all worker nodes managed by the `NodeSet` will be terminated when there are no pending jobs in queue for that specific partition. This is the default behavior, the minimum number of pending jobs for the scaler to be considered in idle state can be changed, in fact a custom value for the activation threshold can be defined as follows: `activationThreshold: '3'`, meaning that the scaling from 1 to 0 (0 to 1) replicas will happen only when there are less than (at least) 3 pending jobs in the queue.

Another interesting aspect is that the field `metricType: AverageValue` is used, meaning that the HPA will try to maintain an average value of pending jobs across all replicas equal to the defined threshold (in this case 1), this means that if there are 5 pending jobs in the queue, KEDA will try to scale up the `NodeSet` to 5 replicas (if allowed by the `maxReplicaCount` field). More formally, `metricType: AverageValue` will set the desired number of replicas according to this equation:

$$desiredReplicas = \left\lceil \frac{currentMetricValue}{threshold} \right\rceil \tag{4.2}$$

this means that each replica will handle approximately `threshold` jobs. In the case of `metricType: Value`, the scaling behavior will be the same described in Equation (4.1)

Issues

This solution presents a fundamental issue that make it not suitable for scaling the bare metal Slurm worker nodes managed with Cluster API, as described in Paragraph "Limitations" inside Section 4.3.1.1, if we try to adapt in the most straightforward way the above `ScaledObject` to target a `MachineDeployment` instead of a `NodeSet`, KEDA will be able to perform the *activation phase* and the $0 \leftrightarrow 1$ scaling, but the created HPA will error out when trying to compute the desired number of replicas giving: "`The HPA was unable to compute the replica count: unable to calculate ready pods: no pods returned by selector while calculating replica count`".

This means that for directly scaling `MachineDeployments`, the proposed solution based on KEDA is not a viable option and a workaround needs to be introduced (see Section 4.3.2.2).

Another problem may naturally arise from the particular choice of metric used for scaling: the number of pending jobs in the Slurm job queue for a specific partition

[7] `https://github.com/kedacore/keda/issues/2314`

(`slurm_partition_jobs_pending{partition="slinky"}`) and it is strictly correlated with the `coolDownPeriod`, i.e. the time interval in which the metric must remain under the value specified in the field `threshold` for KEDA to trigger the scaling from N currently deployed replicas to 0 (`idleReplicaCount`).

When $pendingJobNumber = 0 \land runningJobNumber \leq$ `maxReplicaCount` $\land\ runningJobNumber > 0$, i.e. when the last jobs of the queue are being processed, the number of pending jobs is below the threshold, thus the `coolDownPeriod` timeout is triggered, and if the remaining running jobs would take more time to complete than the defined `coolDownPeriod`, KEDA would scale down the worker nodes to 0 while there are still executing jobs, having as a side effect that those jobs would again transit to the pending state, retriggering the scaling up of the worker nodes after the first polling to the metrics server. When those new nodes will eventually come online, the same jobs will be rescheduled and begin execution again, with the effect of dropping the number of pending jobs to 0. This cycle will repeat itself endlessly, effectively throttling the job execution and triggering a frequent scaling up and down of nodes without completing any job.

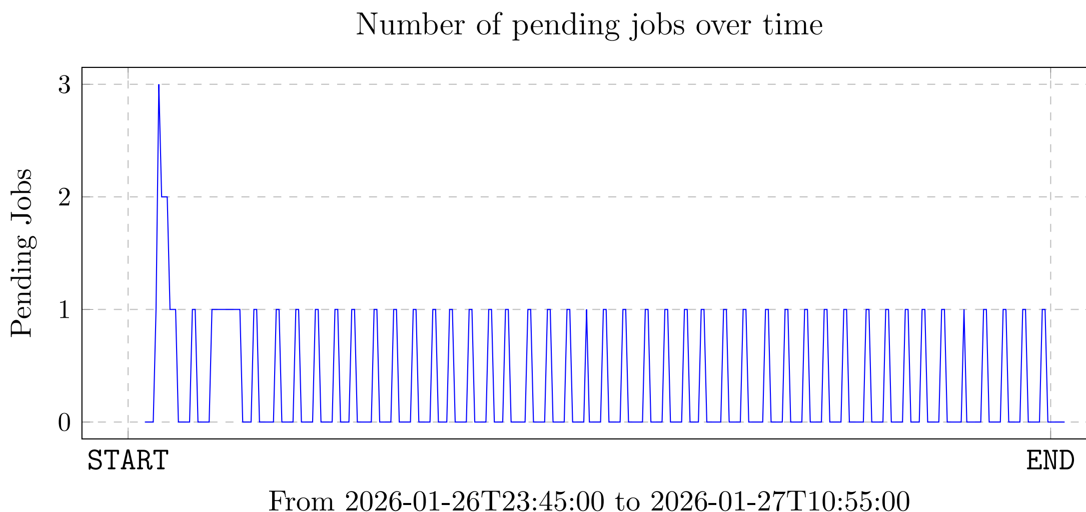


Figure 4.5: Effect of the cycle on the pending job number

In Figure 4.5, the effect of this cycle on the number of pending jobs is shown, where it is possible to see how the number of pending jobs oscillates between 0 and N, and then between 0 and 1, without ever reaching the completion of all jobs (the duration of the experiment being around 11 hours). Three jobs were submitted, two of them with an execution time smaller than the defined `coolDownPeriod`, which is set to 5 minutes, while the last one required more time to complete.

This “bouncing” effect is caused by the default policy of Slurm on job requeuing

(`JobRequeue = 1`), that allow for certain jobs to be requeued automatically when certain events occur, e.g. node failure. In this particular scenario from Slurm's perspective, a node is abruptly getting deprovisioned upon scaling to 0, thus all running jobs on that node are requeued and set back to the pending state. One may say that disabling this automatic job requeuing could solve the issue, but this is obviously not the case, since the jobs would simply fail and never complete, which is anyway not a desirable outcome.

Another similar experiment, shown in Figure 4.6, shows the effect of setting the execution time of the shorter job to exactly `coolDownPeriod`. Instead of observing the two shorter jobs getting completely executed as expected before the "bouncing" effect takes place (in Figure 4.5), we can see that the number of pending jobs and available nodes can oscillate between 0 and 3, because none of them is executed before the scaling down to 0 is triggered. The effect eventually reduces to the same 0 and 1 oscillation due to latencies in commanding the scaling to 0, giving time to the shorter jobs to terminate their execution successfully and not get requeued.

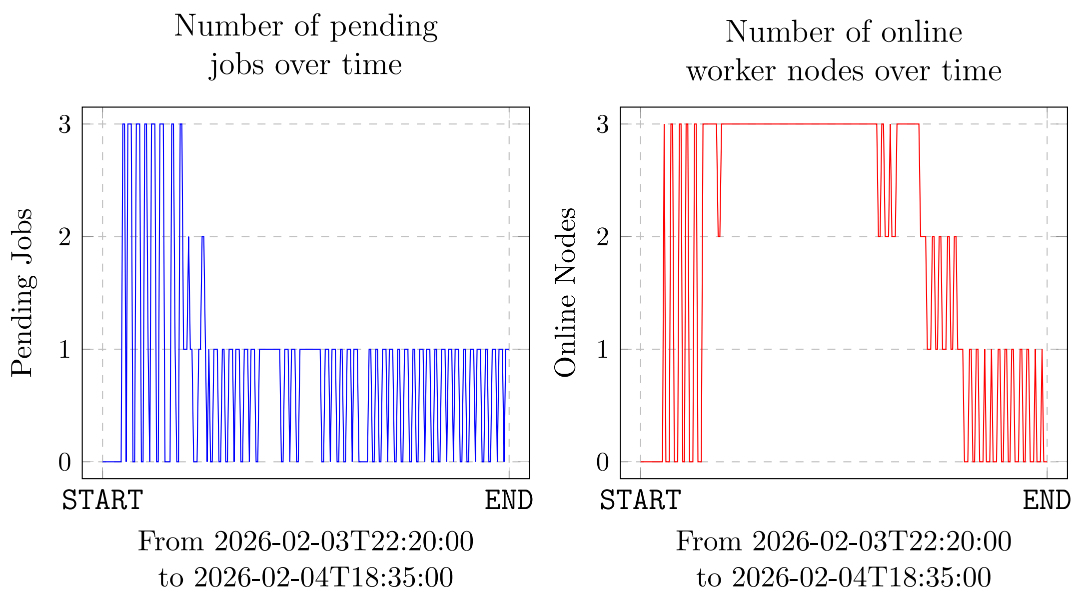


Figure 4.6: Effect on nodes and jobs with different job lengths

Finally, given that this issue is also inherent to the choice of metric used for scaling, to address it correctly should be sufficient to change the prometheus query to consider the number of running jobs when the number of pending jobs is equal or below the `threshold` specified in the `ScaledObject`, e.g.:

```
slurm_partition_jobs_pending{partition="slinky"} >= threshold or
↪  slurm_partition_jobs_running{partition="slinky"}
```

In this way, the scaling down to 0 will happen only when there are no pending jobs and no running jobs, thus avoiding the aforementioned bouncing effect. If we try again to launch the same jobs, we can get the expected outcome with all the jobs being completed and then the nodes getting deprovisioned, but it is also possible that we obtain a result similar to the experiment in Figure 4.7.

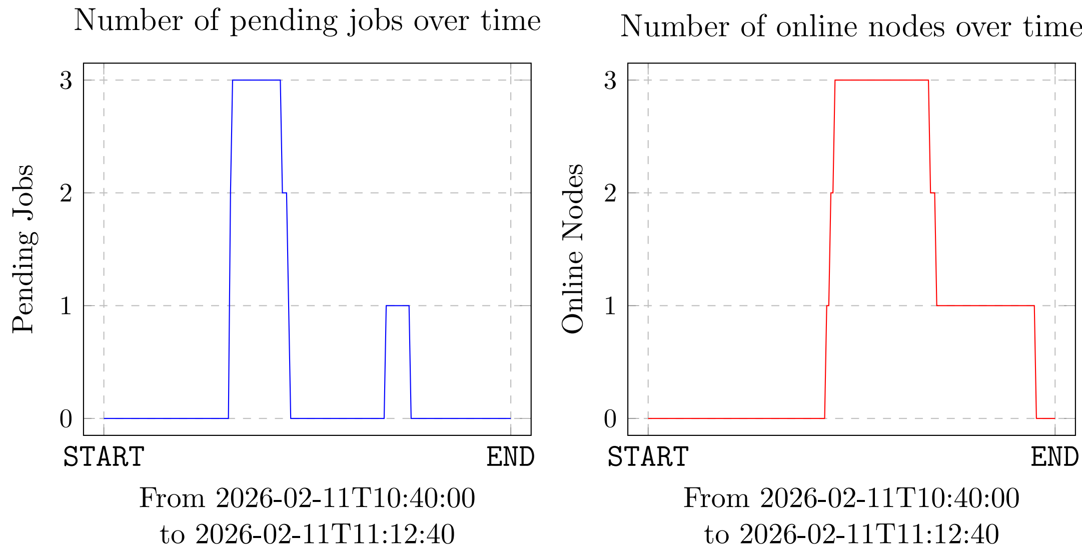


Figure 4.7: Effect of the refined prometheus query

It can be seen that there is a sort of “single bounce” after the scaling to 1 node, that happens while the query returns 1 (running) job, as expected since the threshold was set to 1. This problem, indeed is not inherently related to the choice of a specific metric, but instead it is to be attributed to the obliviousness of Kubernetes and KEDA/HPA to the concept of “busy worker node”. When the HPA decides to scale from 3 to 1, it commands it to the Kubernetes API server, but this latter entity does not have the awareness of which node is running a Slurm job, it just scales down the number of replicas to 1, thus deleting 2 nodes without any consideration on their execution status. Indeed, as it can be seen, the node that was executing the longer job got deprovisioned, thus the job got rescheduled (the shown bounce) and was picked up by a currently idle node, which should have been the correct target of the deprovisioning process. This is a fundamental problem that is not immediately solvable, since it would require to have a system that prevents the deprovisioning of a certain node if it is running a job and a way to map the concept of “busy node” from Slurm to Kubernetes. Possible ways to overcome this problem are discussed in Section “Future Work”.

4.3.2.2 Autoscaling integration between Slinky and Cluster API

In this section, a workaround solution to make the autoscaling system suggested by Slinky work together with the architecture based on Cluster API is presented. The key idea is enabled by the introduction of two components: an admission webhook, called `scaling-webhook` and a regular Kubernetes `Deployment`, called `mirror-deployment`.

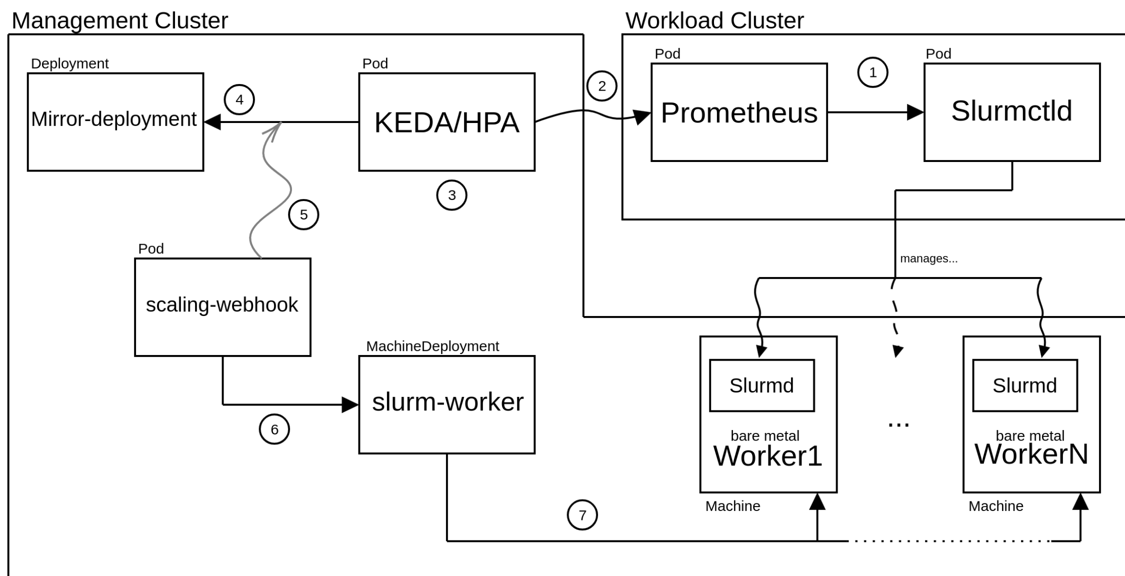


Figure 4.8: Simplified architecture for autoscaling integration with CAPI

Solution workflow

In particular, the workflow shown in Figure 4.8 is described as follows:

1. Prometheus scrapes the metrics exposed by Slurmctld on port 6817 from the endpoint /metrics/partitions
2. KEDA Operator periodically polls Prometheus for the number of pending jobs in the Slurm job queue for a specific partition, as described in the `ScaledObject` configuration.
3. KEDA Operator evaluates the retrieved metric against the defined threshold to decide whether to scale up or down the target
4. KEDA Operator scales the target `Deployment`, in this figure named `mirror-deployment` accordingly to the aforementioned metrics evaluation
5. The admission webhook `scaling-webhook` intercepts the scaling request and

reads from the updated `/scale` subresource what is the desired number of replicas for KEDA/HPA

6. The `scaling-webhook` replicates the scaling request to the target `MachineDeployment` by updating its `/scale` subresource accordingly

7. The target `MachineDeployment` then deploys the desired number of `Machines`, that will eventually become worker nodes in the Slurm cluster

Refer to Section 5.3 for the implementation details and the deployment steps for the *Scaling Webhook*.

4.3.3 Refined architecture overview

In this section, the definitive architecture of the system is described, it includes the components needed to make the autoscaling system proposed by Slinky work together with the Cluster API-based architecture of this thesis. Since Cluster API is designed to manage the lifecycle of Kubernetes clusters, the possibility of having multiple concurrent Slurm clusters was introduced. To ease the management of those multiple clusters from a single control plane, an even more flexible configuration that introduces the usage of a DNS-based service discovery system to retrieve the Prometheus metrics server address used for KEDA's `ScaledObject` was implemented. With this solution there is no more need to have a static IP address for the Prometheus server, in fact the addresses can be dynamically updated inside the DNS server by using DNS records provided and pushed by ExternalDNS. In this way it is possible to have multiple Slurm clusters with their own Prometheus server without the need to reconfigure anything in the KEDA `ScaledObject` definition, even if configuration changes occur. The object is created indicating only a logical name that will eventually be translated in different network addresses, as needed.

Firstly, the newly introduced technologies are described, then the overall architecture is described.

4.3.3.1 ExternalDNS

ExternalDNS [58] is an open-source project that aims to automate the lifecycle of DNS records for objects belonging to Kubernetes clusters. It works by watching a configurable set of Kubernetes resources, typically but not limited to resources of type `Service` and `Ingress`, and synchronizes their networking data with one or more external DNS providers, so that DNS entries always reflect the current state of the cluster without requiring manual intervention.

The architecture of ExternalDNS can be described by presenting three main abstrac-

tions, namely *Sources*, *Providers* and (a) *Registry.* A *Source* is responsible for discovering endpoints from Kubernetes API objects. For example, an ExternalDNS source of type *Service* typically extracts hostnames from the `external-dns.alpha.kubernetes.io/hostname` annotation of the Kubernetes `Service` and pairs them with the correspondent IP, that can be for example an external IP assigned by a load balancer, while an Ingress source reads the `spec.rules[*].host` fields. Each *Source* produces a list of one or more DNS records for the processed resources, in form of `Endpoint` objects. A *Provider* abstracts the interaction with a specific DNS backend, translating the data retrieved by *Sources* into the API calls required by the target DNS system. ExternalDNS ships with providers for major cloud platforms such as AWS Route 53, Google Cloud DNS and Azure DNS, but also supports on-premises solutions like CoreDNS (which is the default Kubernetes choice), PowerDNS and RFC 2136-compliant servers, such as BIND (Berkeley Internet Name Domain), which makes it also suitable for bare-metal, on premise environments like the one considered in this thesis. The *Registry* sits between Sources and Providers and is responsible for ownership tracking: it marks each DNS record with a unique identifier (the `txt-owner-id`) stored in companion TXT records that get persisted on the DNS provider. Thanks to this mechanism, multiple ExternalDNS instances can coexist on the same DNS zone without interfering with each other.

During each reconciliation loop, whose frequency is specified by setting the `--interval` flag, with the default value being one minute, ExternalDNS collects the desired state from all the available *Sources* and the current state from the *Provider.* Then a component called *Plan* bundles together the `Endpoint` objects created by the *Source* and sends them to the `Provider`, which translate them into the creation, updates and deletions needed to converge the two states, and applies it to the DNS service, labelling each endpoint with the result, which can be `Success`, `Failed`, or `Skip`. The reconciling nature of the process results in the fact that transient failures are "self-healing". Indeed, if a *Provider* application fails, the next loop will recompute the same set of `Endpoint` object and retry. The `--policy` flag controls the aggressiveness of the updating of the DNS records. The default value is `upsert-only`, which only creates and updates records and never deletes them. There is also the `sync` policy, that performs full synchronization including deletions, which is more appropriate for those environments where records needs to be cleaned up when the corresponding Kubernetes resource is removed.

Additionally, ExternalDNS can filter the resources it watches through several mechanisms: namespace selectors, label selectors, annotation filters and domain inclusion or exclusion lists. These filters allow multiple ExternalDNS deployments to coexist inside the same cluster, each responsible for a disjoint set of zones or resource types, which can be useful in multi-tenant environments or, for example, when different

teams manage different subsets of the DNS records.

A few annotations can be inserted inside the target Kubernetes resources to allow for a more fine-grained control over ExternalDNS behavior. One of the most important is `external-dns.alpha.kubernetes.io/hostname`, which sets one or more hostnames for a Service or Ingress (hence a *list* of `Endpoints`). The annotation `external-dns.alpha.kubernetes.io/ttl` controls the DNS record TTL to manage caching. Instead, `external-dns.alpha.kubernetes.io/target` lets you override the computed `Endpoint` with a custom values for the target, which is the RDATA field, hence this annotation can be used to set custom IP addresses. There is also the annotation `external-dns.alpha.kubernetes.io/controller` which allows to make the controller ignore the annotated resource if the value is not `dns-controller` Moreover, other provider-specific annotations are present, which are defined by the provider itself and have the name of the provider as a prefix.

In the context of this refined architecture, ExternalDNS is deployed in each workload cluster with BIND as the DNS provider backend. It uses the provider `rfc2136`, that can manage a pool of different RFC 2136 compliant DNS services [80]. Its role is to automatically register the addresses of the Prometheus metrics servers that are exposed by each workload cluster, so that KEDA's `ScaledObject` can reference a stable logical DNS name instead of a hard-coded IP address, which can be subject to changes. When a new workload cluster is provisioned and its Prometheus service becomes available through a `LoadBalancer` service (provided by MetalLB, as described in Section 4.3.3.2), ExternalDNS detects the new endpoint and creates the corresponding DNS record. If the service IP changes, for example because the workload cluster is redeployed, the record is updated transparently during the next reconciliation loop, requiring no manual reconfiguration in the KEDA `ScaledObject` definition.

4.3.3.2 MetalLB

MetalLB [51] is an open-source network load balancer for Kubernetes that is intended to provide this functionality with regard to bare-metal environments, since the orchestration platform does not implement a load balancer. Indeed, when creating a `Service` of type `LoadBalancer` inside clusters that are hosted in the cloud, the provider typically automatically assign a valid external IP through its infrastructure. In environments without this possibility, however, the service remains indefinitely in the `Pending` state because no component is available to allocate and advertise an external address by default. MetalLB solves this problem by running locally to the cluster and by managing a pool of IP addresses defined by the user that it can assign to `LoadBalancer` services when needed.

A working MetalLB installation is composed of two fundamental components. The *controller*, which is a `Deployment` that watches for `Service` objects of type `LoadBalancer` and is responsible for IP address allocation. When a new `Service` is created, the controller selects an available address from one of the configured `IPAddressPool` objects, and puts it inside the `status.loadBalancer.ingress` field of the `Service`, and reserves the the chosen address for the object to avoid conflicts. When the service is deleted, the address is returned to the pool as available. The second component is the *speaker*, deployed as a `DaemonSet` so that one instance runs on every node. *Speakers* are responsible for making the allocated addresses reachable from the network external to the cluster by advertising the IPs through the configured protocol, that can be ARP (for IPv4), NDP (for IPv6) or BGP. In addition, they also periodically monitor the health status of the endpoints, if the node currently responsible for an address becomes unavailable, another speaker takes over the announcement, reducing the unavailability window.

MetalLB supports two distinct address advertisement modes, namely L2 mode and BGP node, realized through `L2Advertisement` and `BGPAdvertisement` custom resources, each one referencing one or more `IPAddressPools`. In Layer 2 mode, the speaker on a single elected node responds to ARP (IPv4) or NDP (IPv6) requests for the service IP, making the rest of the network believe that the node is the owner of that address, apparently making it owner of multiple IPs. All traffic for the service therefore converges on that one node, which then forwards it to the backing pods through the normal `kube-proxy` operations. The speaker responsible for all the traffic handling for a specific `LoadBalancer` service is called the speaker leader. Leader election among speakers is indeed performed for each service, so different services can be announced by different nodes, de facto spreading the network load across the cluster nodes, the election is stateless, removing the need to keep memory of which speaker is the leader for a certain service. Layer 2 mode requires no particular router configuration and works on any Ethernet network, making it the simplest option to deploy. Its main limitation is that, for a given service, all external traffic enters the cluster through a single node, which can become a bandwidth bottleneck if the service handles large volumes of data.

In BGP mode, every node of the cluster establishes a BGP peering session with the network routers, which are configured through `BGPPeer` objects, and advertises the service addresses. This mode allows for true load balancing, differently from L2 mode. However, it requires a more complex network configuration and the presence of a BGP-capable router, which may not be available in all environments.

Address pools are defined through the `IPAddressPool` CRD, which accepts one or more CIDR (Classless Inter Domain Routing, e.g. 192.145.32.0/24) ranges or explicit address intervals. IP pools can have the `autoAssign` field set to `true` or

`false` (defaulting to `true`), which controls whether MetalLB can autonomously pick an address from the pool when a service does not request a specific IP. This exists to allow “expensive” IPs to be assigned only on specific request via annotation. When finer control is needed, a service can request a particular pool by adding the annotation `metallb.universe.tf/address-pool`, or even a specific address by setting the `spec.loadBalancerIP` (that is in process of becoming deprecated) field or the preferable `metallb.universe.tf/loadBalancerIPs` annotation. This makes it very simple to partition the available address space: for example, one IP pool can be dedicated to certain infrastructure services, while another can be reserved for other kind of services, each with its own advertisement mode if specific requirements e.g. on the bandwidth exist for the different kind of services.

In the context of this thesis, MetalLB is deployed inside each workload cluster operating in Layer 2 mode with a dedicated `IPAddressPool` drawn from the subnet that `metal3-dev-env` assigns to the provisioned virtual machines. Its primary role is to provide a stable external IP to the Prometheus `Service` of type `LoadBalancer` running in the workload cluster, so that the management-cluster components, namely KEDA and ExternalDNS, can reach the metrics endpoint from outside the workload cluster’s pod network. Because MetalLB pools are declarative Kubernetes objects, they can be reconfigured at any time without restarting any component: if the available address range changes, for example because additional subnets are allocated, it is sufficient to update the `IPAddressPool` resource and MetalLB will immediately reflect the change. Combined with ExternalDNS, which registers the allocated IP under a logical DNS name, this setup ensures that the only truly static IP address in the entire system is the one of the DNS server in the management cluster, which is an inherent requirement of DNS itself.

4.3.3.3 Architecture overview

As introduced at the beginning of this section, to further improve the flexibility of the architecture, a solution to ease the management of multiple Slurm cluster from a single control plane is introduced. The cluster-wise architecture stays almost the same, inside one workload cluster there is a single Slinky installation with its own Prometheus server that scrapes metric from `slurmctld`. But instead of having a static IP address for the Prometheus server, which would reduce flexibility and require manual reconfiguration if the Prometheus service changes its IP for any reason, a system to automatically update the network address of the metrics scraping server was introduced. To achieve this target, two main components have to be added to the workload cluster: MetalLB, a LoadBalancer service to expose the Prometheus server outside its own cluster and ExternalDNS, to automatically register the Prometheus server address under a logical DNS name that can be used

in the KEDA `ScaledObject` definition. Instead, regarding the management cluster, the only change is the addition of a BIND DNS server, that is used as the backend for ExternalDNS, and that is configured to be authoritative for a specific domain.

As can be seen in Figure 4.9, an high level overview of the architecture can be described as follows, after having deployed the Prometheus `Service`:

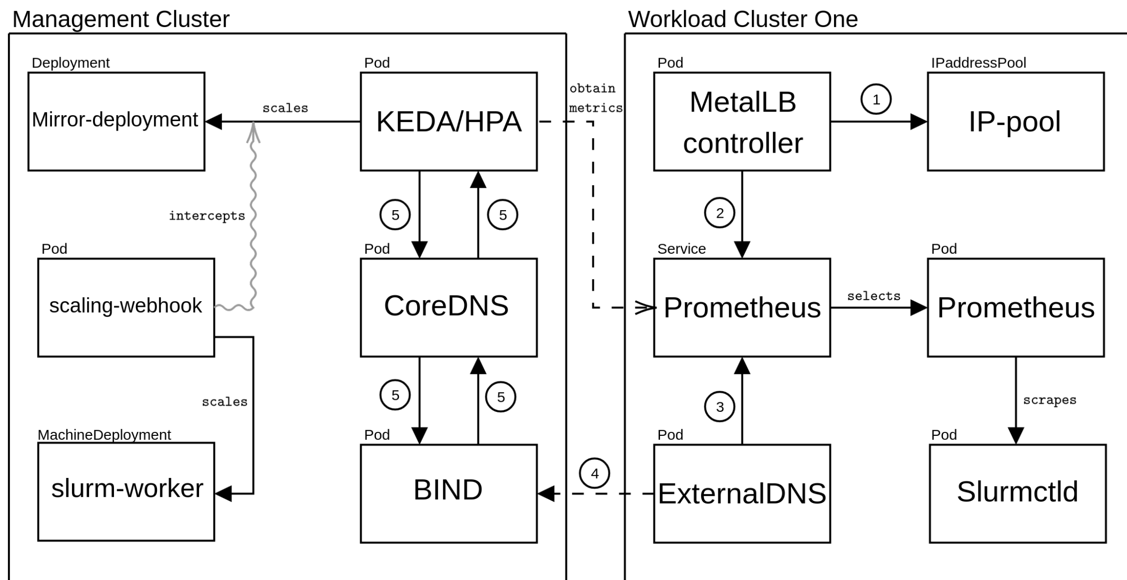


Figure 4.9: Refined architecture overview

1. The MetalLB controller picks a suitable IP address from the configured `IPAddressPool`
2. The MetalLB controller assigns the chosen IP to the Prometheus `Service` (and the non-represented speaker leader starts advertising it on the network)
3. ExternalDNS detects the new endpoint and prepares a DNS record
4. ExternalDNS updates of creates the new DNS record on the management cluster BIND server, that is authoritative for the domain used in the record
5. KEDA tries to retrieve the metrics from the Prometheus server using the logical DNS name written inside the `ScaledObject`, if it does not know the resolution, it queries CoreDNS, which forwards the query to the BIND server that handles the zone. The BIND server then returns the IP address and KEDA can reach the metrics Server

Obviously what depicted in Figure 4.9 is a simplified view of the architecture, in which only one workload cluster is shown, but an equivalent process happens for each

workload cluster, it is sufficient to replicate all the right-hand side of the figure. All of the remote ExernalDNS instances in the workload clusters will update the same BIND server, so that every KEDA `ScaledObject` can reference a logical DNS name that is discoverable via CoreDNS.

Refer to Section 5.3.1 for the implementation details and the deployment steps for the components of this *refined* architecture.

Chapter 5

SliMe Implementation and Deployment

This chapter provides the implementation details of the components introduced in Chapter 4. For each component, the deployment procedure is described, together with the configuration choices that were made and their justification. Where applicable, the chapter also documents the issues that were encountered and resolved during the deployment process and the workarounds that were adopted. All the components described in this chapter were deployed and tested on the Morrigan cluster, using the `metal3-dev-env` environment or the OpenShift platform as needed. The described deployment details for each component are proven to be working on the test environment.

The chapter is organized as follows: Section 5.1.1 describes the steps required to deploy Slinky on OpenShift, with an emphasis on the modifications to the default installation procedure that were necessary to comply with the stricter security policies enforced by OpenShift compared to upstream Kubernetes. Section 5.1.2 covers the deployment of the preliminary architecture, mainly describing the provisioning of bare-metal Slurm worker nodes with Cluster API and Metal^3 and the network and authentication configuration required to connect them with the Slinky managed Slurm controller pod. Section 5.2 details the implementation of the *Injection Webhook*, including its TLS certificate management through cert-manager, an high-level description of the business logic and the required permissions to operate correctly. Subsequently, Section 5.3 describes the implementation of the *Scaling Webhook* that enable KEDA-driven autoscaling of Cluster API `MachineDeployments`. Finally, Section 5.3.1 presents the deployment of the necessary components to support the DNS service discovery (ExternalDNS, BIND, MetalLB) and the configuration required to make them operational.

5.1 Preliminary Evaluation

In this section, the deploying details for the preliminary evaluation architectures presented in Section 4.1 are described, along with the rationale behind the choices that were made during the deployment process. In this order, the first part of the section is focused on the deployment of Slinky on OpenShift (see left-side of Figure 4.1), then the deployment of the proposed architecture that joins bare metal nodes with a Slinky-managed Slurm controller pod (see right-side of Figure 4.1) is described and finally a possible deployment process for an hybrid architecture as mentioned in Section 1.1 is presented.

5.1.1 Deploying Slinky on OpenShift

Deploying Slinky on OpenShift required some adjustments to the standard installation procedure, since OpenShift has different and stricter security policies than standard Kubernetes. For this latter reason a specific nonstandard release of Slinky called `slinky-on-openshift` [65] was deemed as a possible starting point. It is a GitHub repository that contains a set of guidelines, manifests and container images that allows to deploy in a straightforward and adequate way an installation of Slinky on the OpenShift platform. The main differences with the standard Slinky installation is that the repository provides a Security Context Constraint (SCC) setting that allows the pods in the namespace to run as privileged pods, this would allow them to run essential services as `sshd`. Also, the repository provides a set of custom container images that are no longer based on the standard Slurm images provided by SchedMD, but instead on CentOS Stream 9 images on which the OpenHPC repository is enabled and the Slurm packages, along other tools like the aforementioned HPL benchmark, are installed from there.

It is necessary to note that the OpenShift installation on the Morrigan Cluster prevents any pod that is not allowed otherwise from running with a UID that is not contained inside an interval defined inside the namespace scoped SCC called `restricted`, which is by convention the strictest policy in the namespace. Hence it would not allow both the Slurm controller pod and the worker pods to run with the required slurm user, since it has UID 401 by default. Allowing a pod to be privileged do not bypass the check on the admissible UIDs, since it only means that the pod can run with extended capabilities and more relaxed security policies, not affecting whether or not the pod is allowed to run with any UID. For this reason, an hybrid installation of Slinky composed of the standard `slurm-operator` by SchedMD and an installation of the `slurm` helm chart with the custom `slinky-on-openshift` container images was chosen, without deploying the provided SCC settings, since they would not have allowed the pods to run with the required UID by themselves. The

core modifications were made at namespace level, with a temporary nature in mind, in fact the allowed set of UIDs was overridden to include the UID 401 by using the annotation `openshift.io/sa.scc.uid-range`, in this specific case it was set as `openshift.io/sa.scc.uid-range=399/100000` inside the `Project` object manifest. In this way, the allowed UID range would be from 399 to 100399 (since the two numbers define the starting UID and the pool size), thus allowing the pods to run with the required UID. Moreover, the pods would still need to be executed with the privileged security context, for the aforementioned reasons, and thus the annotation `security.openshift.io/MinimallySufficientPodSecurityStandard: privileged` was added to the `Project` manifest as well. This annotation sets the least restrictive Pod Security Standard that is sufficient to pass the checks on the security context of the pods, in this case it is the `privileged` standard, which allows to run privileged pods. Obviously, this combination of factors is not a secure solution and it is not adequate for production, but it was sufficient to deploy a testing environment on OpenShift that would allow to run the preliminary performance tests and to explore which limitations and issues a platform with stricter security policies would have caused on the Slinky installation process.

```yaml
- key: "slinky"
  operator: "Equal"
  value: "true"
  effect: "NoSchedule"
```

```yaml
requiredDuringSchedulingIgnoredDuringExecution:
  nodeSelectorTerms:
  - matchExpressions:
    - key: kubernetes.io/hostname
      operator: In
      values:
      - <nodename1>
      - <nodenameN>
```

Listing 1: Tolerations and node affinities for `NodeSet` pods

Once the Slinky installation was correctly setup and working, a set of 4 nodes were tainted with a specific taint that prevented any pod from being scheduled on them with the following command:

```
oc adm taint node <nodename> slinky=true:NoSchedule.
```

After having the taint applied, the nodes were drained from the preexisting workloads to maximize the amount of available resources for the performance tests. Subsequently a standard `NodeSet` object was created and the `spec.replicas` field was set to 4, which is the number of tainted nodes. Furthermore, the `spec.template.spec.tolerations` and `spec.affinity.nodeAffinity` fields were set as can be seen in the left side and in the right side of Listing 1, respectively.

This configuration allowed the Slurm worker pods managed by the `NodeSet` to be scheduled on the tainted nodes. Moreover, as also the Slinky documentation says,

it is reasonable to deploy only a single pod per physical OpenShift node, since said pod should mimic a real physical node and should access the whole of the available hardware resources. If this is not the case, i.e. if multiple `NodeSet`'s pods are scheduled on the same physical node, the performance tests would not be reliable since the pods would be competing for the same resources. To avoid this situation while maintaining generality, the resource requirements of the pods were set to the maximum value that would not prevent the scheduling of the pod in a node. In our specific situation, the maximum achievable values for `spec.template.spec.containers.resources.requests` were `cpu: 189` and `memory: 495Gi`. These values have the side effect of preventing the scheduling of more than one pod on the same node, since the total amount of resources available on a single node was 192 CPUs and 503 GiB of memory, hence the scheduling of a second pod on the same node would not be possible since it would require more resources than the ones available.

Once the worker pods were correctly scheduled on the tainted nodes, the performance tests were conducted and the metrics were collected from Prometheus as described in Section 6.1

5.1.2 Deploying the proposed architecture

To deploy the proposed architecture, there are not many relevant differences between the preliminary one which uses standard bare metal nodes and the one that leverages Cluster API provisioned bare metal nodes, besides the phase of preparing the environment for Cluster API, which is unnecessary in the first case but fundamental in the latter. Once provisioned, there is no difference between the two kinds of nodes, since they are both physical machines that run the `slurmd` daemon and that register with the `slurmctld` pod, thus the deployment steps are the same for both architectures once the Cluster API environment is ready. It is worth mentioning that in the specific case of this thesis, the bare-metal Cluster API nodes are not real, physical nodes, but instead VMs. There are no inherent issues in doing this, in fact, they behave in the same way as standard bare metal nodes since they are not running any Kubernetes/OpenShift related workload and they are completely not Kubernetes aware, thus they are still outside of the cluster network and, from the communication perspective, they interact with the Slurm controller pod over the ad-hoc shared network between the cluster and the bare metal nodes.

Therefore, for the Cluster API version, in addition to the aforementioned standard Slinky installation on OpenShift/Kubernetes, it is necessary to deploy a Cluster API environment that will support all the components needed to manage bare metal machines as Slurm worker nodes through the Kubernetes APIs. A fundamental tool that goes towards this direction is part of Metal^3 project (see Section 3.6), more

specifically metal3-dev-env (see Section 3.6.1). Once deployed using the provided scripts, the environment is ready to provide the infrastructure support to provision and manage bare metal machines as Kubernetes Machines, which is a CRD exposed by Cluster API. The environment also provides a set of tools to create a whole new cluster, referred as *Workload Cluster* inside CAPI, which is a Kubernetes cluster that is completely manageable via Kubernetes API calls and to which is possible to connect bare metal Machines with the role of standard Kubernetes worker nodes. The pre-existing cluster, running on kind, is instead referred as *Management Cluster* and it is the cluster on which Slinky is installed and where the Slurm controller pod is hosted.

metal3-dev-env offers a configurable environment: in the chosen configuration (file config_${USER}.sh), a number of 15 virtual machines were provisioned. In this manner 15 bare metal nodes, referred as BareMetalHost(s) in metal[3], are made available to be managed through the Kubernetes API. Subsequently a MachineDeployment, along with the corresponding Metal3MachineTemplate, containing the image to bootstrap on the node, the VirtualKubeletConfigTemplate (which is a CRD of cluster-api-bootsrap-virtual-kubelet), which defines the provisioning configuration for cloud-init and the Metal3DataTemplate, that contains required configuration data regarding the node, are deployed. By modulating the number of replicas for the MachineDeployment object, it is possible to control the number of bare metal nodes that are provisioned and that will be registered as Slurm worker nodes, obviously remaining bounded to the number of the physical machines. Once the MachineDeployment is deployed, the provisioning process starts and the required number of nodes is provisioned and registered with the Slurm controller pod, thus becoming potentially available for the execution of submitted jobs.

There is a fundamental problem that arises when deploying this architecture, which regards the communication between the Slurm controller pod and the worker nodes. The controller pod is hosted inside the Kubernetes cluster and the worker nodes are hosted on bare metal machines that are logically outside of the said cluster. All the entities are physically connected to the same network, which is deployed by metal3-dev-env, that creates a network interface called external that interconnects every VM with the host OS. That same OS incidentally also hosts the docker container(s) constituting the management cluster, hence there always is an available connection between the involved entities. But it is necessary to enable the communication between them, which is not available by default, since the controller pod is not directly reachable from the outside of the cluster (since it runs inside a docker container and it is inherently firewalled by how Docker handles its network). Instead, the worker nodes are always directly reachable from the inside of the cluster for the aforementioned reasons. To enable the communication between

the controller and the worker nodes, a `NodePort` service was created to expose the Slurm controller on a specific port, mapping the port 6817 of the controller pod to a port on the cluster nodes, in this specific case, port 32000 was arbitrarily chosen. Using a `NodePort` service may be seen as a fairly intrusive solution, but as can be seen in Section 4.2 this is not a problem since there should be a single instance of Slinky in each cluster and thus, the port mapping would not cause any conflict with other potential Slinky installations. Obviously, the chosen NodePort will be exposed on the docker container, which is still not an ideal solution if the goal is using the host IP as the one for the slurm controller. To solve this problem, it is not possible to use the `hostNetwork` setting in the controller pod manifest, which allows the pod to use the host network and thus to be directly reachable from the outside, since it would still refer to the docker container, and would be a non ideal deployment choice due to security concerns and network configuration flexibility issues. Instead, the solution that was introduced was the use of a set of iptables rules that allows to forward the traffic from the host IP to the control-plane Docker container NodePort in the following manner:

```
sudo iptables -t nat -A PREROUTING -p tcp --dport 32000 -j DNAT
↪  --to-destination $KIND_CONTAINER_IP:32000
sudo iptables -I FORWARD -p tcp -d $KIND_CONTAINER_IP --dport 32000
↪  -j ACCEPT
```

It is important that `sysctl net.ipv4.ip_forward` returns 1, meaning that the forwarding service is enabled in the kernel's IP stack. These rules allow to forward the traffic from the host IP on port 32000 to the control-plane Docker container IP on the same port, thus allowing the worker nodes to reach the Slurm controller pod by using the host IP and the NodePort. This solution is not ideal since it requires to have a set of iptables rules that are not managed by Kubernetes and that need to be manually applied, but it is sufficient for testing purposes, in the following iterations for the automatization of the solution, this constraint would be removed, see Section 4.2.

Another important caveat for this solution that will have its later relevance is the fact that port 32000 was chosen as a sort of substitution for port 6817, which is the default port for Slurm controller communications. The choice of port 32000 was arbitrary, but constrained by the allowed range for NodePorts, which is from 30000 to 32767 by default. Technically, there are two separate bands in that range: the static band, spanning from 30000 to 30085 and the dynamic band, spanning from 30086 to 32767. The static band is intended to be used for services that require a specific port, while the dynamic band is intended to be used for services that do not require a specific port and that can be assigned a random one by Kubernetes. Since the Slurm controller requires a specific port (due to the needed configuration

in the bare metal nodes), it would have been more appropriate to choose a port from the static band, to lower the possibilities of having a conflict when other nodePort services gets their ports reserved, but since this is a test environment and we have complete control over the deployed services, the choice of port 32000 was sufficient to avoid any conflict and to allow the communication between the controller pod and the worker nodes. In a more general and production-ready solution, it would be more reasonable to use the lower, static, band to reduce the chances of having a conflict.

The bare metal nodes cloud-init configuration contained inside the `VirtualKubeletConfigTemplate` was chosen to allow the nodes to register with the Slurm controller pod by using the host IP and the NodePort as *Dynamic Nodes* in the Slurm configuration, which means that they are not statically defined in the Slurm configuration file (`slurm.conf`, as in a more traditional Slurm deployment), but instead they are registered at runtime by the `slurmd` daemon when it starts. This is the predefined configuration for the Slinky installation, since it allows to have a more flexible and dynamic architecture, where the worker nodes can be added and removed at runtime without needing to modify the Slurm configuration file. This is particularly useful in the context of a Kubernetes-centric environment, where the number of worker nodes can be dynamically adjusted based on the workload and the available resources. Moreover, this configuration allows to have a more seamless integration with the Cluster API provisioning of bare metal nodes, since it does not require to have a predefined list of worker nodes in the Slurm configuration file, which would be difficult to maintain and update in such a dynamic environment. A Slurm dynamic node does not have a valid `slurm.conf` configuration until it registers with the controller, since it is not statically defined in the configuration file, it is necessary to specify where to find the controller in the daemon arguments, the daemon can be started with the following command

```
slurmd --conf-server=$SLURM_CONTROLLER_HOST_IP:32000 -Z
```

The `-Z` flag is used to specify that the node is a dynamic node, otherwise the daemon would try to find a valid configuration for the node and would fail.

One fundamental issue that arises is that the controller host IP cannot be known in advance without occurring in limitations of flexibility in the configuration. For what concerns this particular deployment flow, since it is still a testing environment, the controller host IP was hardcoded in the `VirtualKubeletConfigTemplate` manifest, but in a more general and production-ready solution, it would be more reasonable to have a more dynamic configuration, for example by using a mechanism that injects the controller host IP at runtime when the `Machine` is deployed. This would allow to have a more flexible and dynamic architecture, where entire sets

of worker nodes can be added and removed at runtime without needing to modify the `VirtualKubeletConfigTemplate` manifest. This will be the object of the description in Section 4.2, where a solution for the automatization of the integration between Slinky and Cluster API is proposed.

It is worth noting that there is another core missing piece of information that is needed for the worker nodes to be able to register with the controller, which is, the Slurm authentication key (`slurm.key`). Since Slinky does not use the default Munge authentication system, but instead leverages the `auth/slurm` plugin, a particular 1024 Bytes long key is necessary to communicate to the Slurm controller, this key is generated at the installation of Slinky and it is stored in a Kubernetes secret, more precisely at the first installation of Slinky in the cluster, since retention policies are active by design and the key is not deleted when Slinky is uninstalled. This key is needed to be injected in the provisioning process of a machine at some point and the methodology is described in Section 4.2.

For what concerns the preliminary tests, the required data were uploaded to an `httpd` container that would serve as an accessible cluster-ready http server. This data is statically put into the server directory and gets retrieved during the provisioning phase by the newly booted node. Obviously this is not a flexible solution, since it requires manual intervention for what concerns the controller host IP in the eventuality of changes in the deployment (e.g. the controller pod crashes and gets rescheduled) and it is not a secure solution, since storing an auth key in an http server that is not protected by some form of authentication is a security concern. But, again, in a preliminary testing environment it can be acceptable, the solution to this issue is addressed during the automatization of the integration with Slinky.

Since the Slurm configuration file gets downloaded from the controller when the node registers, and by design choice no breaking modification has been performed on the Slinky default config (to maintain the possibility to have an hybrid configuration without issues on Slinky's behalf, see Section 5.1.3), the effective port to reach `slurmctld` will be the default one, which is 6817, but the problem is that it is exposed on port 32000 of the host via the aforementioned NodePort plus IPtables configuration. This would render the communication impossible after having retrieved the `slurm.conf` file from the controller, since it overrides the parameters with which `slurmd` is called, with the bare metal node trying to communicate with the pod's host IP on port 6817. To overcome this issue, inside the config script that is used to bootstrap the nodes, another iptables rule is applied as follows:

```
sudo iptables -t nat -A OUTPUT -p tcp --dport 6817 -d
↪  $SLURM_CONTROLLER_HOST_IP -j DNAT --to-destination
↪  $SLURM_CONTROLLER_HOST_IP:32000
```

If the deployment steps were to stop there, the communication between the controller pod and the worker nodes would still not be possible. In Kubernetes there is the possibility of setting policies on how external traffic is routed to services, mainly for load-balancing purposes. There are two possible settings: `Cluster` and `Local`, when the former is set, the traffic is routed to all of the available and ready pods that are backing the service regardless of the node on which they are running, while when the latter is set, the traffic is routed only to the pods that are running on the same node as the one on which the traffic arrives. The default setting for NodePort services is `Cluster`, this is a problem since this setting causes the traffic to be subject to (Source Network Address Translation) SNAT by the kube-proxy running on the node that receives the traffic from the outside of the cluster, i.e. the source IP of the incoming traffic is replaced with the IP of the interface of the kube-proxy. The reason for the SNAT stays under the fact that the external packet gets forwarded to all the ready endpoints in the cluster, which means that it can be forwarded to a pod that is running on a different node, and thus the source IP needs to be replaced with the one of the kube-proxy to allow the return traffic to be correctly routed back to the external source, the recipient pod cannot autonomously reach back the external source since the initial sender wouldn't expect a response from a different IP than the one to which it sent the initial request. This is a problem with our setup since Slurm dynamic node registration relies on the source IP of the incoming traffic to identify the registering node, and if the source IP is replaced with the one of the kube-proxy, the controller pod would not be able to correctly reach back to the registering node and thus it would not be able to register it without problems, in fact the `NodeAddress` field of the node in Slurm would correspond to the kube-proxy, and it is not the correct host that is running `slurmd` trying to register. To solve this issue, it is necessary to set the `externalTrafficPolicy` field of the NodePort service to `Local`, this setting allows to preserve the source IP of the incoming traffic, since the traffic is only routed to the pods that are running on the same node as the one on which the traffic arrives, without needing to perform a SNAT to ensure the correct route back to the external source. It may be seen as a limitation, but in our configuration the daemon `slurmd` tries to register with the controller using the host IP, which means that the traffic will always arrive on the node on which the registering node is running. In this way, the communication between the controller pod and the worker nodes is enabled, since the source IP of the incoming traffic is preserved and thus the controller pod can correctly reach back to the registering node and register it without problems.

In addition to the issue of non-reachability of the controller pod from the worker nodes, there is another issue that arises when deploying this architecture, which is the fact that Slurm uses a certain amount of ports for the job-oriented communication between the controller and the worker nodes when using `srun`. They are chosen

among the range of the *ephemeral ports*, those ports, also called dynamic ports are short-lived TCP/UDP source ports that an operating system automatically assigns to client sockets when an application initiates a network connection (since typically the server side listens on a well-known port). The flow is this: one host calls `srun` to launch a certain, possibly parallel, workload, the controller daemon `slurmctld` receives the request and it requires the allocation to the known worker nodes communicating to them on the default `slurmd` port, which is 6818. Once the allocation is done, the `srun` host opens a certain number NP of ports on the *ephemeral* range defined by [69]:

$$NP = 4 + 2 \cdot \max\left(0, \left\lfloor \frac{NH - 48}{48} \right\rfloor\right) \tag{5.1}$$

with NH being the required number of hosts. In other words, `srun` opens 4 ports for each invocations, plus 2 additional ports every 48 hosts over the first 48, if `srun` is invoked with the argument `--pty`, another port is opened. After the ports are bound, the `srun` host contacts each involved `slurmd` to communicate the necessary info to run the tasks, among which there is the correspondent opened port to reach back to. This means that the worker nodes need to be able to reach the controller pod on the opened ports, which are not known in advance, since they are chosen by `srun` at runtime from a very extensive pool of ports, which in Linux typically spans from 32768 to 60999 by default. It is clear that it is not possible to have a `NodePort` service that exposes the whole range, since it only has a limited range, and even if it would be possible it could be a security concern, since it would expose a very large number of ports to the outside of the cluster. One possible solution that would work could be using `hostNetwork: true` in the pod configuration [16], which still exposes the whole range of ports, so it is not a good idea. For this reason it has been decided to use a more conservative approach, which is to choose a reasonable range of ports that can be used for the job-oriented communication and to expose only those ports on the cluster network. This can be done by setting the `SrunPortRange` field in the Slinky `slurm.conf` file (via the field `extraConf` available in the helm charts) to a reasonable range, for example from 30000 to 30010 as `SrunPortRange=30000-30010`, and then exposing those ports on the cluster network with a `NodePort` service, in the same way as for port 6817. This solution allows to have a more secure and controlled exposure of the necessary ports for the job-oriented communication, while still allowing the worker nodes to reach the controller pod on the required ports. Obviously, it is important to choose a range that is large enough to accommodate the expected number of involved nodes following Equation (5.1), but this should be manageable since the number of nodes involved in a job has an upper bound that corresponds to the total number of available physical nodes that can be provisioned with Cluster API.

Naturally, selecting the chosen range of ports for `srun` communications on the cluster

network is not sufficient to allow the communication between the controller pod and the worker nodes, since the same problem of non-reachability of the controller pod from the outside of the cluster arises for those ports as well. To solve this issue, it is possible to use the same solution that was used for port 6817, which is to create a service of type NodePort that exposes the required ports on the Kubernetes cluster, and then apply a set of iptables rules that allows to forward the traffic from the host IP to the control-plane Docker container network interface for each port in the chosen range. It is worth noting that, again, those iptables rules will not be needed for the subsequent iteration of the solution, as already said before.

Visually, the interactions between entities can be seen in Figure 5.1.

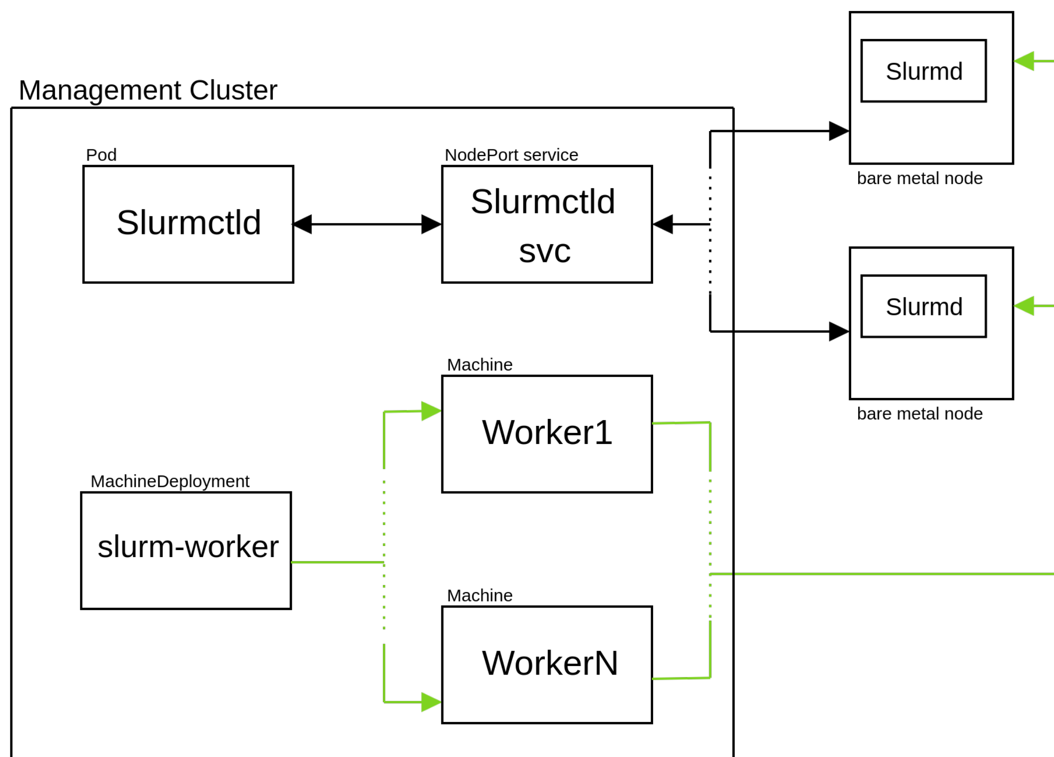


Figure 5.1: Relationships between entities in preliminary architecture

So in a concise summary, the deployment flow can be described in the following steps:

1. Deploy `metal3-dev-env` to prepare the environment for Cluster API to manage bare metal nodes.
2. Install Slinky on the management cluster, editing its config file by adding a reasonable range in field `SrunPortRange`

3. Deploy a `NodePort` service to expose both port 6817 and the just selected range on the Slurm controller pod
4. Set the needed iptables rules to allow the communication between the controller pod and the bare metal nodes
5. Deploy a `MachineDeployment` that:
   (a) Create a user called `slurm` with UID 401
   (b) Pull `slurm.key` from the httpd service and give the correct permissions (`600`) and ownership (`slurm:slurm`)
   (c) Pull and install the Slurm *rpm* packages (for CentOS) from the httpd service
   (d) Get the Controller Host IP from the httpd service and use it to set the aforementioned iptables to handle port 32000 and modify `/etc/hosts` adding the line `${SLURM_CONTROLLER_IP}<\t>slurm-controller.slurm`
6. Submit jobs to the Slurm cluster

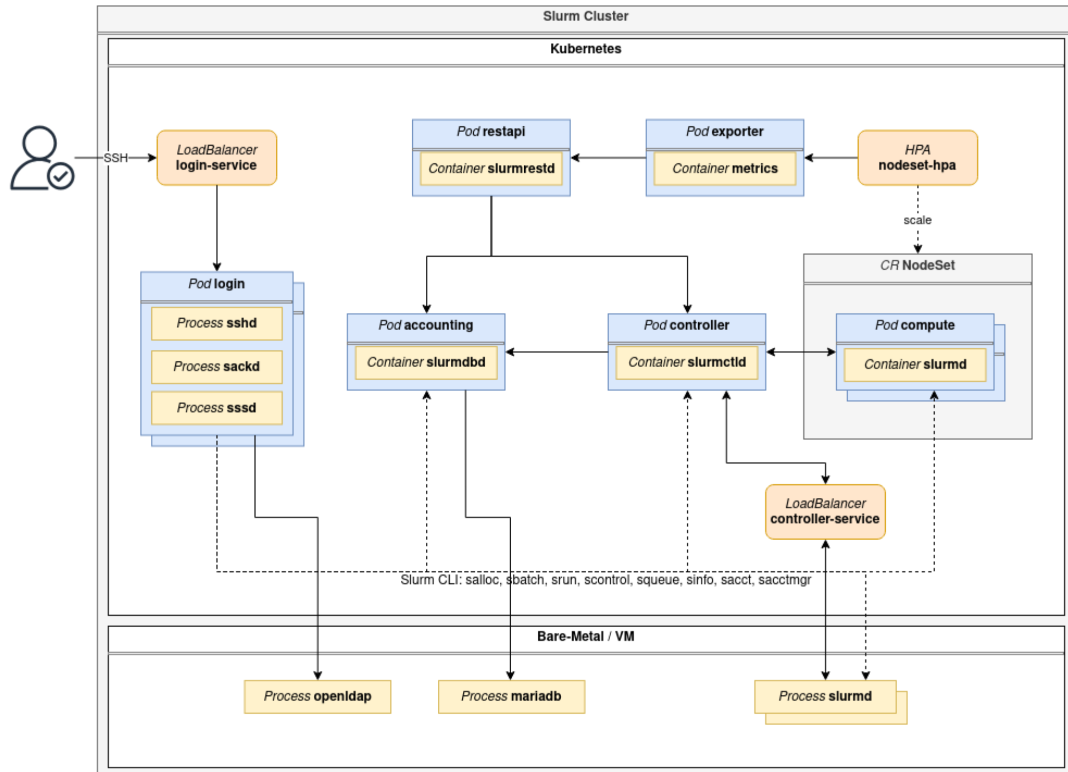
After having performed all these steps, it can be confidently said that the answer for **RQ2** is positive.

5.1.3 Enabling hybrid architecture

As can be seen in official Slinky documentation [68], it is possible to deploy a hybrid architecture where the Slurm controller is hosted inside a pod while the worker nodes are hosted on both bare metal machines and standard kubernetes pods, as can be seen in Figure 5.2.

This architecture is interesting since it allows to have a very flexible deployment, that can allow to have a certain amount of worker nodes hosted on pods and a certain amount of worker nodes hosted on bare metal machines, thus potentially allowing to have a more fine-grained control over the resources assigned to the worker nodes, if needed. Moreover, this kind of architecture could be useful when there is a preexisting Slurm cluster of bare metal machines that can be repurposed to be integrated with a Slinky-managed Slurm controller, without needing to dedicate a whole node to the controller itself.

Furthermore, this architecture could be useful in scenarios where there is a need to have a certain amount of worker nodes that are not suitable to be hosted on pods, for example because they require access to specific hardware resources that are not available on the Kubernetes/OpenShift cluster nodes, or because they need to run a specific workload that is not well suited to be hosted on pods [24]. In these cases,



source: `https://slinky.schedmd.com/projects/slurm-operator/en/release-0.3/_static/images/architecture-slurm-hybrid.svg`

Figure 5.2: Hybrid architecture in Slinky documentation

having the possibility to have an hybrid architecture could allow to have the best of both worlds, since it would allow choosing the desired destination for a particular workload among an hybrid cluster that has a certain amount of worker nodes that are hosted on pods for general workload, while having another amount of worker nodes that are powered by bare metal machines and that can provide the necessary resources for the specific workload.

To enable a configuration like the one in Figure 5.2, thus having standard Slinky `NodeSets` while still having bare metal nodes provisioned with Cluster API, a series of careful modifications to the deployment described in Section 5.1.2 are needed, since some choices that were made, such as defining a cluster-wide `NodePort` service, would lead to conflicts between the controller pod and the worker pods.

More precisely, the first issue that arises is that the `NodePort` service that exposes the ports in the `SrunPortRange` on the cluster network would reserve that range in a cluster-wide fashion. This means that the worker pods would not be able to use those ports for the job-oriented communication, since they would be already

reserved for the controller pod in every node of the cluster.

To solve this issue, it is necessary to stop exposing all the `SrunPortRange` ports on the cluster network with a single `NodePort` service, and instead to expose only the default port 6817 on the cluster network with a `NodePort` service, while leaving the ports in the `SrunPortRange` untouched by the service. It is obvious that exposing them is still needed, and this can be done through the use of `hostPorts` in `spec.<containerName>.ports` in a similar way:

```
for port_i ∈ SrunPortRange
- containerPort: $port_i
  hostPort: $port_i
  name: srun8
  protocol: TCP
```

whether `containerName` being `slurmd` or `slurmctld`, depending on the chosen pod assuming the role of a worker node or the controller, respectively. In this way, the ports in the `SrunPortRange` are exposed on the cluster network referring to the actual running pods. This solution is not ideal since it requires to have a set of `hostPorts` that need to be manually defined in the controller/`NodeSet` manifest, but it is sufficient for demonstration purposes.

Another core problem that surfaces when trying to enable the hybrid architecture and after having applied the workaround based on `hostPorts` is that the dynamic registration of the pod worker nodes with the controller pod would still complete successfully, but would not allow for bare metal worker nodes to connect on `SrunPortRange` ports on the pod worker nodes. This is due to the fact that the dynamic node registration in Slinky happens by default with the cluster IP of the pod, but since the ports are exposed on the host IP, if a bare metal worker nodes needs to communicate with a pod worker node on one of the ports in the `SrunPortRange`, it would try to reach the endpoint `<worker-cluster-ip>:<port>`, which is not reachable from the outside of the cluster, this is due to the fact that nodes retrieve the IP from the `NodeAddress` field of the node object in Slurm (visible with `scontrol show node <node_name>`), which is set to the cluster IP of the pod during the dynamic registration phase.

To overcome this issue, it is possible to manually set the correct `NodeAddress` field for the pod worker nodes to the host IP instead of the cluster IP, a bash script that does exactly that for each node belonging to a certain `NodeSet` has been developed. It substantially retrieve the host IP from the pod that is running the worker node, then it uses the `scontrol` command to set the `NodeAddress` field of the node to the retrieved host IP. Moreover, at the time of writing, if the Slinky cluster is run inside a workload cluster created with `metal3-dev-env`, it is possible to retrieve

the host IP from cluster secrets called `<node_name>-networkdata`, inside the field `ip_address`. To update the `NodeAddress` field it is sufficient to run:

```
scontrol update NodeName=<node_name> NodeAddr=$NODE_IP
```

Again, this solution is not ideal since it requires to have a manual step that needs to be performed after the dynamic registration phase of a pod worker node, but it is sufficient to show that enabling the hybrid architecture is possible, and that the communication between all involved parties, i.e. the controller pod and both types of worker nodes, can be ensured.

5.2 Injection Webhook - Integration with Slinky

In this section, the implementation details for the Injection Webhook, presented in Section 4.2 are described. The main goal of the Injection Webhook is to inject the Slurm controller pod host IP and the authentication key `slurm.key` into the provisioning process of the bare metal nodes, to allow them to register with the controller pod without human intervention.

5.2.1 Implementation details

The implementation of the Injection Webhook required particular care to be as least impactful on the already established configuration as possible. Only the addition of the label `slurm-controller-namespace` in the object of kind `Metal3DataTemplate` is required to allow the correct operations of the Mutating Admission Webhook. It is in fact needed to specify precisely to which controller the bare metal node has to connect to, because it is not possible to infer it from the sole context. The webhook is completely implemented in the Go language as it is the most common language for Kubernetes controllers and webhooks.

The main goal of the Injection Webhook is to inject the Slurm controller pod host IP and the authentication key `slurm.key` into the `metaData` field of the secret that is created during the provisioning of a bare metal node with Cluster API.

The webhook makes use of the Kubernetes `client-go` library [37] to interact with the Kubernetes API server and to retrieve from the cluster the necessary data that will be injected. As said before, it gets triggered by the creation of a `Secret` object that is owned by a `Metal3Data` object and that contains a `metaData` field. Once triggered, it examines the content of the *Admission Request* and, in particular of its `Object`field. This field contains the manifest of the `Secret` object that is being created, and it is in a raw format, so it needs to be unmarshalled to be used as a `Secret` object from the `core/v1` Kubernetes API. Once obtained the secret, the webhook

has, first of all, to retrieve the Slurm controller pod host IP, but it only has the metadata `Secret` available. To obtain the host IP, it is needed to know the namespace of the pod (which has to be the same as the workload cluster that it runs into), readable from the label `slurm-controller-namespace` that the user put inside the `Metal3DataTemplate` object, so the webhook climbs backward the `ownerReference` chain composed by `metadata-secret` → `Metal3Data` → `Metal3DataTemplate` that can be seen in Figure 5.3.

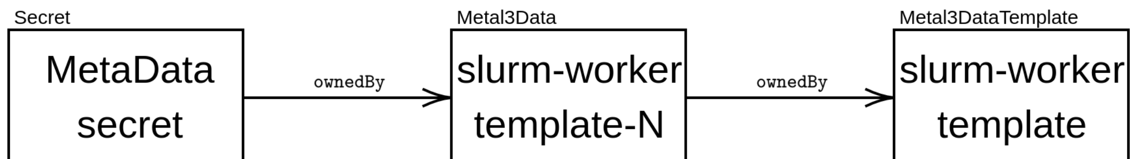


Figure 5.3: Ownership reference chain for Injection Webhook

To be able to handle those objects that are defined as CRDs by the Metal^3 project, the webhook needs to use a client that is able to interact with them. The standard Kubernetes client provided by `client-go` is a typed client that has specific methods for each standard Kubernetes resource (such as `Deployments`, `Secrets`, `StatefulSets`, `Pods`, ...) in its interface. But this standard, typed, client is not able by default to handle CRDs defined by external projects, so it is convenient to use a different kind of client, called dynamic client that can interact with any kind of Kubernetes object, including CRDs. It can retrieve resources by using an object of type `GroupVersionResource (GVR)`, which contains the group, version and resource of the object to retrieve, for example for a `Metal3DataTemplate` object, the used GVR is:

```
schema.GroupVersionResource{Group: "infrastructure.cluster.x-k8s.io",
↪  Version: "v1beta1", Resource: "metal3datas"}
```

Then, with the dynamic client, it is possible to retrieve the `Metal3Data` object by using the `Resource` method, passing to it the required GVR and the namespace and obtaining an unstructured object that can be then converted in a Metal^3 `Metal3Data` object. The same process is obviously repeated to retrieve also the `Metal3DataTemplate` object.

Once obtained the namespace identifier from the label, the `kubeconfig` for the corresponding namespace is retrieved from a secret called `<cluster_name>-kubeconfig` that can be found in the management cluster. Subsequently a client referring to that `kubeconfig` is created and used to retrieve the `StatefulSet` that manages the Slurm controller pod from the correct namespace. The reason why the `StatefulSet` is retrieved instead of the pod itself is because it can be used to obtain both the

Slurm controller pod host IP and the authentication key, in a process later described. The StatefulSet is obtainable by using a LabelSelector that is defined as app.kubernetes.io/component=controller, app.kubernetes.io/name=slurm-ctld, which gets statically assigned in the slurm-operator source code. Once the StatefulSet is obtained, it is possible to retrieve the controller pod by searching for pods that have the same LabelSelector of the StatefulSet and that are in the same namespace, and then extracting the value of the field hostIP from the pod Status.

After having obtained the host IP, the webhook needs to retrieve the authentication key slurm.key from the secret auth-slurm, which is actually called <controller_name>-auth-slurm and it is created upon Slinky installation, if not already present. <controller_name> is referred to an object of kind Controller that is created by Slinky and manages the StatefulSet which handles the controller pod, not the controller pod itself. Obtaining the Controller object is straightforward since it is the owner of the StatefulSet that was previously retrieved, so it is sufficient to refer to its ownerReference field to obtain its name. Then the name is used to retrieve a secret with the aforementioned name, which will contain the slurm.key in its data."slurm.key" field.

Finally, once the host IP and the authentication key are obtained, they are injected into the metaData field of the intercepted secret by emitting a patch that modifies the original field by appending the newly retrieved data. The patch is emitted in the form of a JSON Patch, which is a standard format for describing modifications to a Kubernetes object.

As good practices dictate, the operations of the webhook are idempotent, which means that if the same secret is intercepted multiple times, the webhook will inject data only once. In fact, a preliminary check on the content of the metaData field is performed to see if the data to inject are already present, and if they are, the webhook does not emit any patch and simply returns a positive admission response.

5.2.2 Deployment

To correctly deploy the solution based on the Injection Webhook, it is necessary to perform the Slinky installation on a workload cluster. But before being able to deploy Slinky, it is necessary to prepare the workload cluster. It is possible to use the script provided by the metal3-dev-env project to create it, there are three scripts, one for creating the Cluster, one for creating the Kubernetes control plane node(s) and another one for creating the Kubernetes worker node(s), obviously it is not strictly necessary to use those scripts, but they are a convenient way to create the workload cluster. After the creation of the workload cluster, it is necessary to

install a CNI plugin to enable the communication between the pods and to make the workload cluster's nodes state go to `Ready`, otherwise they would not have a functioning cluster network, which is required to have a working Kubernetes cluster. The CNI plugin that was chosen is Calico [63], but it is possible to use any other CNI plugin. After the preparation of the workload cluster, it is possible to deploy Slinky on it, following the standard installation procedure.

Deploying the Injection Webhook requires some steps, first of all, a secure connection between the webhook and the API server is required. This can be achieved by creating a TLS certificate for the webhook and configuring the API server to trust it. For this exact reason, an core component of the chosen deployment is `cert-manager` (see Section 4.2.2). A specific deployment configuration of `Issuers` was chosen for the infrastructure, as shown in Figure 5.4. A main, cluster scoped and self signed `ClusterIssuer` is created, which is a *bootstrap* signer that signs the `certificateRequest` generated by a `certificate` with the field `isCA: true`, obtaining a custom root certificate (CA certificate) for the internal cluster PKI (Public Key Infrastructure). Then a namespace scoped `Issuer` that refers to the root CA certificate is created, and it is the entity that signs the leaf certificate that is used for webhook server authentication, which indeed has the X.509 key usage set as `server auth`. The certificate then triggers the creation of a secret that is mounted as a volume in the webhook server pod and it is also used to retrieve the data of the CA certificate that are being put into the `caBundle` field of the webhook configuration object.

In this way, leaf certificate rotations are completely transparent to the webhook configuration, since the CA certificate is injected into the `caBundle` field, which is certainly a desirable feature, considering the trend in shortening certificate durability [77]. Moreover, the key rotation policy for the leaf certificate is set to `always`, which means that every time a new certificate is issued, a new private key is generated, this allows to enhance the security of the system by reducing the risk of a compromised private key being used for a long time. Instead, for what instead concerns the root CA certificate, the rotation policy is, more conservative from the perspective of frequency [44], but still would affect the freshness of the `caBundle` field in the webhook configuration, to automatically update it, the aforementioned CA Injector can be used. This allows to ensure that the API server can always verify the authenticity of the TLS certificates used by the webhook servers, even in cases where the certificates are renewed or rotated, without requiring manual intervention to update the `caBundle` field.

Since the webhook is being deployed alongside a service to make it reachable from the API server, the `certificate` is configured to have matching DNS names:

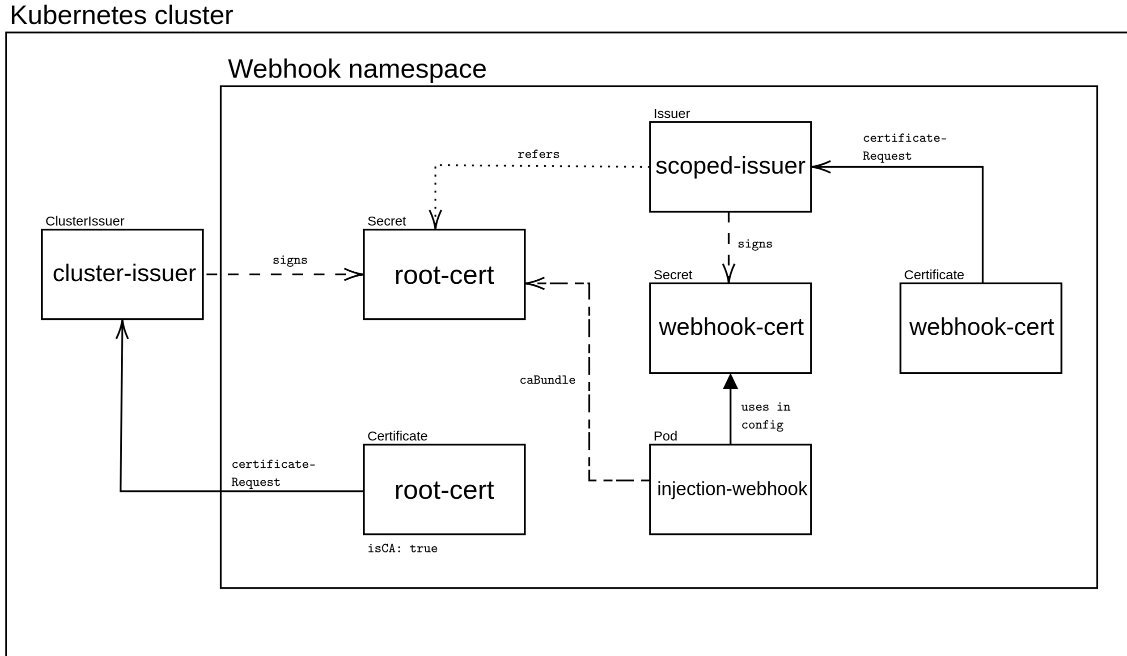


Figure 5.4: In-cluster PKI infrastructure

```
dnsNames:
  - <service-name>.<service-namespace>.svc
  - <service-name>.<service-namespace>.svc.cluster.local
```

as the standard Kubernetes convention for service DNS names dictates.

Another core piece of configuration are the `Roles` and `ClusterRoles` and the relative bindings that refer to the webhook pod's `ServiceAccount`, which allow the webhook to have the necessary permissions to retrieve the data from the cluster that it needs to perform its injection operations. In particular, the webhook needs to have permissions to retrieve the `StatefulSet` that manages the Slurm controller pod, the `Pods` that gets created by that `StatefulSet`, the `Secret` that contains the authentication key (and the one to inject the data into) and the `Metal3Data` and `Metal3DataTemplate` objects that are part of the ownership reference chain. Moreover it has to access `Controller` objects as Slinky's CRD, to retrieve the name for the `slurm-auth` secret. Since the namespace of the Slurm controller pod is not known a priori, the webhook needs to have permissions to retrieve those resources in all namespaces, so it is necessary to create `ClusterRoles` and `ClusterRoleBindings` for those permissions. The only exception could be made for the `Metal3DataTemplate` and `Metal3Data` objects, since they are indirectly created by the user with the `MachineDeployment` for the bare metal Slurm worker nodes,

and they are expected to be in a specific namespace, so it would be sufficient to create a `Role` and a `RoleBinding` for that particular resources.

For what concerns the Injection Webhook deployment, it is important to expose port 443 of the webhook server pod to allow the API server to reach it and to mount the secret containing the TLS certificate. The webhook is deployed as a Deployment with a single replica, since it is not expected to have a high load in this testing environment, but it is possible to scale it up if needed. Moreover, as already mentioned, it is important to create a `Service` to expose the webhook server pod that matches the `dnsNames` field of the `certificate`.

The last fundamental piece of configuration is the `MutatingWebhookConfiguration` object, that, among other properties, defines the rules for which the webhook gets triggered, in this case it is configured to intercept the creation of `Secret` objects that are owned by a `Metal3Data` object and that contains a `metaData` field, this can be done in this way:

```yaml
matchConditions:
  - name: owned-by-Metal3Data
    expression: 'has(object.metadata.ownerReferences) &&
    ↪  object.metadata.ownerReferences.exists(o, o.kind ==
    ↪  "Metal3Data")'
  - name: has-MetaData-key
    expression: 'has(object.data) && object.data.exists(k, k ==
    ↪  "metaData")'
```

After having set the `matchConditions`, it is important to add a reference to the created `Service` to specify the path on which the API server should contact the Injection Webhook server and then finally populate the `caBundle` field with the CA certificate to allow the API server to verify the authenticity of the TLS certificate, or alternatively to set the annotation `cert-manager.io/inject-ca-from-secret` to let cert-manager handle the injection of the CA certificate into configuration object.

Once having deployed the webhook, the same deployment steps described in Section 5.1.2 can be followed to obtain a working architecture, with the difference that there is no more need to deploy the iptables rules to forward the packets directed to the controller pod to the docker container running inside the Kind cluster, since the controller pod is hosted inside the workload cluster and it is directly connected with the worker nodes.

5.3 Scaling Webhook - Autoscaling integration

In this section, the implementation details for the *Scaling Webhook*, presented in Section 4.3 are described. The main goal of the Scaling Webhook is to allow the integration of the autoscaling system proposed by Slinky with the Cluster API-based architecture presented in this thesis, by intercepting the scaling requests from KEDA and forwarding them to the target `MachineDeployment` that manages the bare metal Slurm worker nodes.

Implementation details

The newly added `Deployment` acts as a sort of proxy, or mirror, between KEDA and the desired `MachineDeployment` that has to be scaled to have bare metal Slurm worker nodes dynamically deployed. In fact, the `mirror-deployment` is the actual target of the `ScaledObject` defined for KEDA operator, since it is perfectly compatible with HPA scaling operations and has the sole purpose of receiving scaling requests from KEDA. At the same time, the `scaling-webhook` intercepts those scaling requests, more precisely the *AdmissionReview* request from the Kubernetes API server, with the goal of extracting the number of desired replicas that KEDA or HPA decided to set based on the defined metrics evaluations. This happens by examining the `object` field of the *AdmissionReview* request, that contains the updated `/scale` subresource for the target `Deployment`, then, once the object is obtained, the desired number of replicas is read from the `Spec.Replicas` field. Finally, the webhook performs an update operation on the `/scale` subresource of the target `MachineDeployment`, whose name and namespace metadata can be retrieved by reading from a configmap called `scaling-webhook-configmap` that is user-defined and can be found in the same namespace as the webhook itself.

The `scaling-webhook` is implemented in Go language using the (dynamic) client-go library to allow the interaction with custom CRDs defined by Cluster API as shown in Listing 2.

```
gvrMachineDeployment := schema.GroupVersionResource{Group:
↪  "cluster.x-k8s.io", Version: "v1beta2", Resource:
↪  "machinedeployments"}
unstrScale, err := dynCl.Resource(gvrMachineDeployment).Namespace
↪  (machineDeploymentNameSpace).Get(context.TODO(),
↪  machineDeploymentName, metav1.GetOptions{}, "scale")
```

Listing 2: Get the `/scale` subresource of a `MachineDeployment` with dynamic client

The webhook is deployed as a `Deployment` with a single replica and exposed through a `Service` of type `ClusterIP` on port 443 with TCP traffic. The webhook uses TLS

encryption for secure communication with the API server, thus a self-signed certificate is generated and referred inside the webhook configuration. A `ValidatingWebhookConfiguration` object is created to register the webhook with the Kubernetes API server, specifying, in addition to certificate configuration, that the Admission Webhook has to intercept `UPDATE` operations on the `/scale` subresource of `Deployments` in the cluster. To narrow the scope of the intercepted scaling requests, those were the choices for the `matchConditions` field:

```
matchConditions:
  - name: is-scale
    expression: '(request.kind.kind == "Scale")'
  - name: is-deployment
    expression: '(request.resource.resource == "deployments")'
```

Listing 3: Webhook *matchConditions*

in this way, only the *AdmissionRequests* whose have as an object the `/scale` subresource of a `Deployment` object gets forwarded to the webhook for processing, saving unnecessary computation and most importantly avoiding loops, since the webhook itself triggers an update on another `/scale` subresource, but this time of a `MachineDeployment`. In fact, if the condition `is-deployment` in Listing 3 were not present, the processing would be endlessly retriggered.

Another interesting aspect to notice is that to access the `AdmissionRequests` that involve Scale subresources, it is not sufficient to specify `resources: ['*']` inside the `ValidatingWebhookConfiguration`, but instead it is necessary to specify it in an explicit way such as:

```
rules:
  - operations: ...
    resources: ['*/scale']
```

In conclusion for this deployment section, it is important to notice that the response for **RQ4** is affirmative.

5.3.1 Refined architecture implementation

In this section, the implementation details for the refined architecture presented in Section 4.3.3 are described. The main goal of this architecture is to allow for having multiple Slurm clusters hosted inside the workload clusters, while still having a simple and flexible configuration for the KEDA `ScaledObject` definitions in the management cluster.

The first thing to do to employ this architecture is to prepare the management cluster by deploying the BIND DNS server and configuring it. It is important to modify the configuration file for CoreDNS, which is the default DNS service for Kubernetes, to tell the server to forward queries for certain names to the BIND server itself, it is sufficient to add this to the Corefile inside the coredns `configmap`:

```
slurm.metrics:53 {
   forward . <static_BIND_server_IP>
}
```

In this way, all the queries for names ending with `slurm.metrics` will be forwarded to the BIND server, that is listening at the specified IP address, which will be the only static IP address in the entire system, that also all the ExternalDNS instances in each workload cluster need to know. Moreover, it is clearly necessary to deploy a service that exposes the BIND server on the network, for example a `NodePort` service, specifying the same IP address as the one in the CoreDNS configuration and mapping pod's port 53 (on both TCP and UDP) with a port that does not conflict with other services. The last additional piece of configuration needed for the management cluster is the one relative to the BIND server pod itself, which can be configured through a Configmap as can be seen in Appendix A. If the management cluster is generated with `metal3-dev-env`, or, in general, runs inside Kind, it is necessary to add some iptables rules to allow the communication between the workload clusters and the BIND server, since the NodePort service would bind to the docker container instead of the physical host. Those iptables rules are similar to the other aforementioned and can be also found in Appendix A.

For what instead concerns the workload clusters, the first step is to deploy MetalLB, which is used to provide an external IP address to the Prometheus server LoadBalancer service. The configuration of MetalLB, once installed, is not particularly complex, it is sufficient to create an `IPAddressPool` object with the desired address range inside `.spec.addresses` and an `L2Advertisement` object that references the created pool. In this manner MetalLB is ready to assign IP addresses to any `LoadBalancer` service that is created inside the cluster. The second step is to deploy ExternalDNS, that is responsible for automatically registering the Prometheus server address under a logical DNS name. The configuration requires to set the provider to `rfc2136` and to set a list of arguments that identify the target DNS server. In particular, the `--rfc2136-host` argument is set to the IP address of the BIND server in the management cluster, while `--rfc2136-zone` is set to the domain for which the BIND server is authoritative, e.g. `slurm.metrics`. Moreover, `--rfc2136-port` is set to the NodePort of the service that exposes the BIND server. To allow for dynamic updates of the DNS records, it is necessary to pass the TSIG key used for authentication with the BIND server, that can be passed with the

`--rfc2136-tsig-secret` argument, that contains the secret used to create a TSIG signature and `--rfc2136-tsig-secret-alg` that specifies the algorithm used for the signature. Alternatively it is possible to use `--rfc2136-insecure` to disable the authentication, note that this is not recommended for production environments, but for testing purposes it can be a valid option to avoid the complexity of setting up and distributing the TSIG key.

The last step is to create a `Service` of type `LoadBalancer` to expose the Prometheus server outside the cluster, so that it can be reached by KEDA in the management cluster. It is important to set the annotation `external-dns.alpha.kubernetes.io/hostname: <desired_name>.slurm.metrics`, which purpose is described in Section 4.3.3.1. Once the service is created, MetalLB will automatically assign an external IP address to it, and ExternalDNS will create the corresponding DNS record pointing to that IP address, making the Prometheus server reachable through a stable logical name that can be used in the KEDA `ScaledObject` definition.

Thanks to this set of tools, the only static IP address in the entire system is the one of the BIND server in the management cluster, while all the other addresses, including the one of the Prometheus server, can be dynamically updated without requiring any manual reconfiguration, since ExternalDNS will automatically update the DNS records when changes occur. This means that if the IPAddressPools of MetalLB are reconfigured, for example because certain conflict arises, or if the workload cluster is redeployed and the Prometheus service gets a different IP address, there is no need to update anything in the KEDA `ScaledObject` definition. This setup allows for a flexible configuration that can easily accommodate changes and scaling of the workload clusters without requiring manual intervention for network configuration.

Chapter 6

Performance Evaluation

6.1 Comparing bare-metal nodes and Slinky pods

In this section the results of the performance evaluation of the Slinky-managed pod worker nodes are presented and compared with the performance of the bare-metal worker nodes. The evaluation is based on a set of benchmarks that are described in Section 3.9 and that target different aspects of the system: the performance in terms of FLOPS in solving linear equation systems, the network performance over MPI operations and the bandwidth available between the nodes.

All tests were performed on the same cluster, the aforementioned Morrigan cluster, by isolating 4 identical nodes from it. Each single node is equipped with 2 AMD EPYC 7643, having 48 cores and 96 hardware threads each, for a total of 192 per node, 512 GB of RAM, 4 TB of local NVMe storage and a Supermicro SIOM network card with 4 SFP+ ports driven by an Intel XL710 controller, that provides Ethernet network connectivity.

In the following sections, each performed benchmark results are visually presented and analyzed to discuss the different aspects of the system performance.

6.1.1 HPL

The first performed benchmark is the HPL (High-Performance Linpack) benchmark, specifically a portable version of the software[1]. It was chosen since it can be a reasonable indicator for the performance of a system in terms of FLOPS when subject to strongly CPU-bound workloads. Since the HPL benchmark needs to be tuned to give its best results, the first iteration of the test, which was conducted on

[1] `https://www.netlib.org/benchmark/hpl/`

the OpenShift Slinky installation (see Section 4.1.1.1), with the final goal to obtain a satisfying problem size N that followed the tuning guidelines provided in the documentation, i.e. to fit at least 80% of the available memory. The obtained value of N was the one used for the subsequent comparisons. The chosen process grid was 24×32, which is the "squarest" grid that can be obtained with 768 processes, which is the total number of logical cores available on 4 nodes with 192 hardware threads each. The results obtained for the different problem sizes are shown in Figure 6.1.

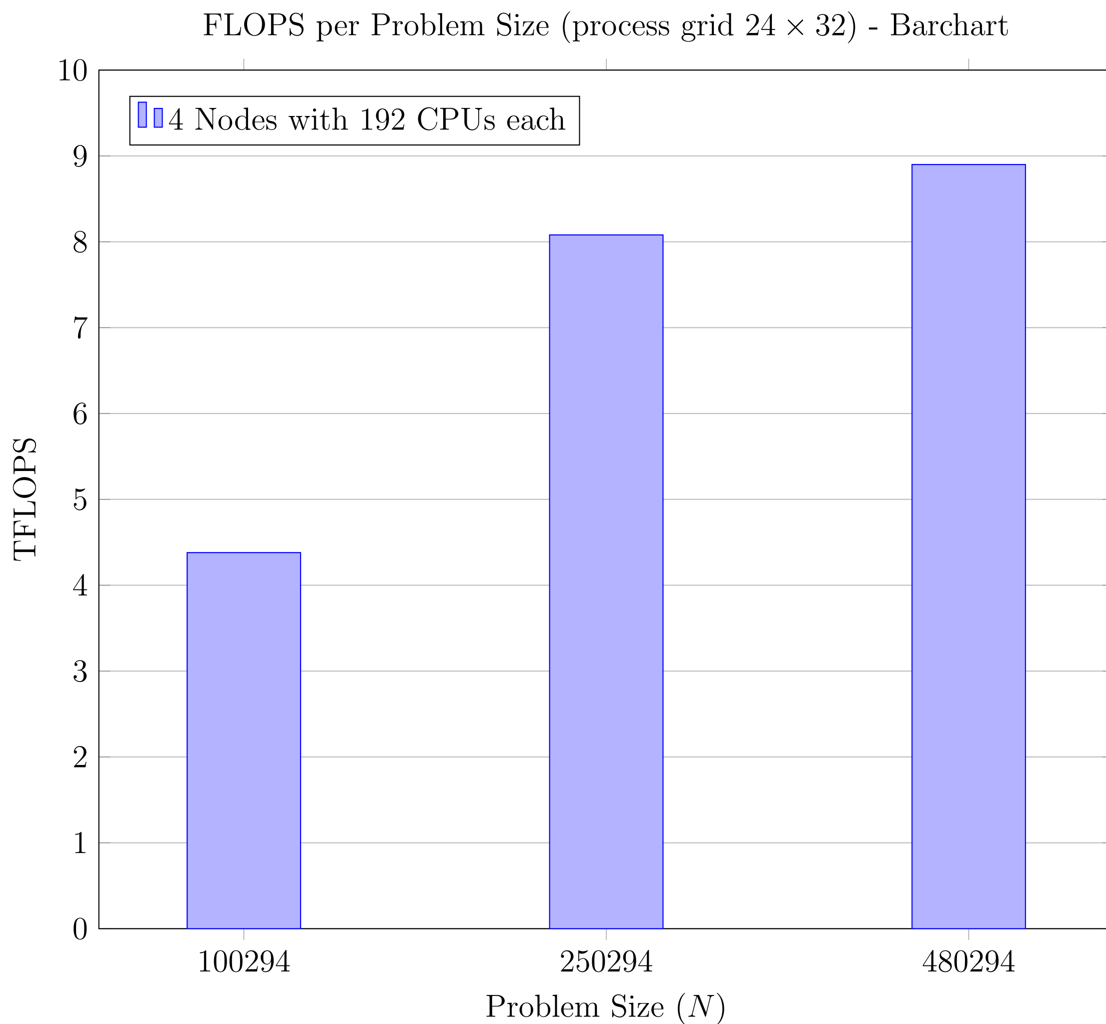


Figure 6.1: HPL performance for different problem sizes on pod workers

Due to benchmark execution time, 3 runs with different N were performed. By observing the results, it can be seen that the performance increases with the problem size, with the improvement diminishing as the problem size increases going towards the maximum N that can be used with the available memory without swapping.

The best performance is obtained with $N = 480294$, giving around 8.9 TFLOPS by using approximately 90% of the available memory. So the following experiments are based on runs of the benchmark with $N = 480294$.

A potentially insightful investigation is examining the effective usage of the available CPU resources, to see how much CPU is effectively used by the benchmarking operations compared to the total used CPU, if there is a difference. For this reason, the CPU usage during the execution of the HPL benchmark was measured from two different perspectives: the total CPU usage of the system, which includes all the processes running on the node, and the CPU usage by the submitted HPL Slurm job. The results of this procedure are shown in Figure 6.2 and Figure 6.3.

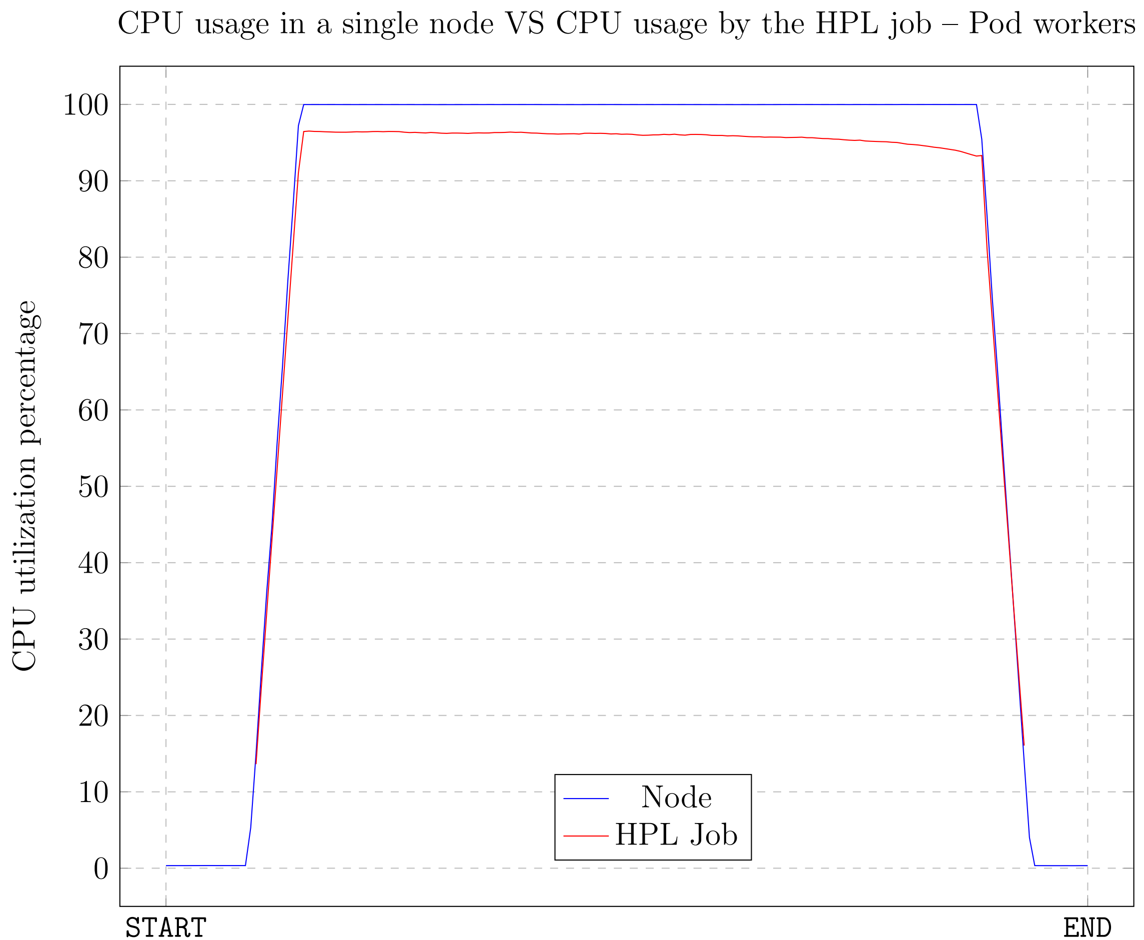


Figure 6.2: Pod workers: total CPU usage on a node and CPU usage by HPL job

The performance results are retrieved via Prometheus, by using the following queries:

```
# Total CPU usage on the node in percentage
100 * avg(1 - rate(node_cpu_seconds_total{mode="idle",
↪  instance="<node_name>"}[5m]))

# CPU usage by the HPL job in percentage
rate(container_cpu_usage_seconds_total{id=~"<hpl_job_cgroup_name>",
↪  node="<node_name>",}[5m]) / 192 * 100
```

It is worth noting that the CPU usage by the HPL job is obtained by looking at the cgroup that gets generated by Slurm, since the default configuration of Slinky foresees the use of `task/cgroup` plugin, which enforces the resource constraints for Slurm jobs by creating ad-hoc cgroups for each submitted job. Moreover, cAdvisor considers cgroups as containers, so it is possible to retrieve the CPU usage of the HPL job by looking at the corresponding cgroup, which is identified by the ID of the Slurm job. This is also the reason why the red curve starts and stops respectively after and before the blue curve, since the cgroup is created after the job submission and deleted after the job completion.

By looking at the plotted data, the first thing that can be observed is that the CPU usage by the HPL job is less than the total CPU usage of the node, which is expected since there are other processes running on the node that are not part of the HPL job, such as OpenShift core components that need to run on the node to keep the cluster functional. By looking at a more in-detail view of the same data, shown in Figure 6.3:

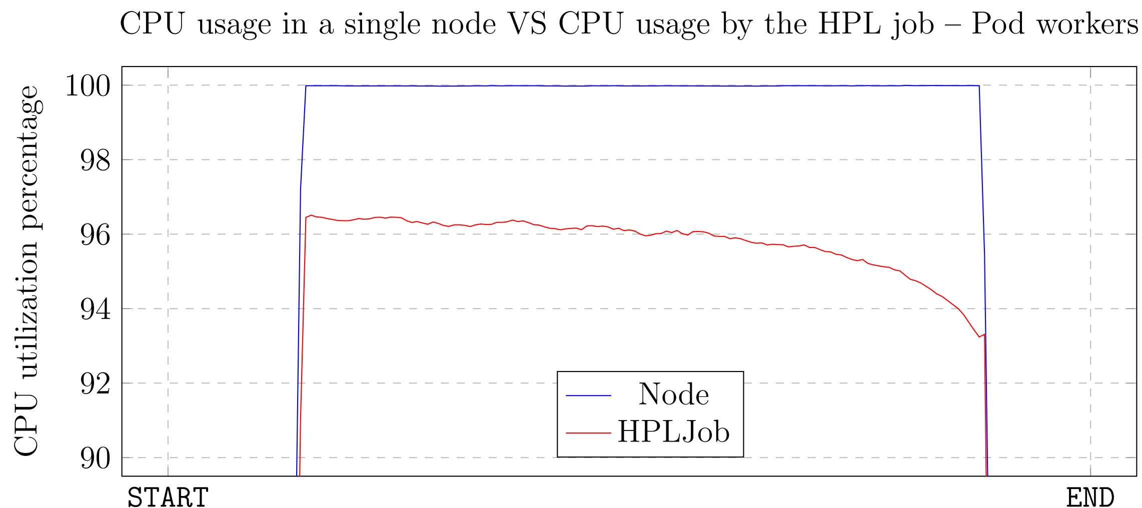


Figure 6.3: Detailed view of the results in Figure 6.2

It can be seen that the overall CPU usage of the node is around 100% during the

whole execution of the HPL job, while the CPU usage by the HPL job is around 96%, with a steady drop in the latter phases. This means that there is a chunk of around 4% of available resources that cannot be fully used by the HPL job. This resource amount is used by other processes running on the node, and in any case it is not possible to reduce it to a great extent, since due to resource requirements, the maximum amount allocable to the worker pods is limited to 189 CPUs (see Section 5.1.1), which means that at least 3 CPUs are not accessible to the HPL job at cgroup level. This is due to the fact that `systemd` creates the job cgroup inside the `system.slice` cgroup that lies inside the pod's cgroup, this makes it subject to at least the same limitations as the pod.

Subsequently, the same workload was executed on the bare metal nodes setup (described in Section 5.1.2), the obtained results are visually quite similar to the pod workers case and the plot in Figure 6.4 shows the outcome of the experiments

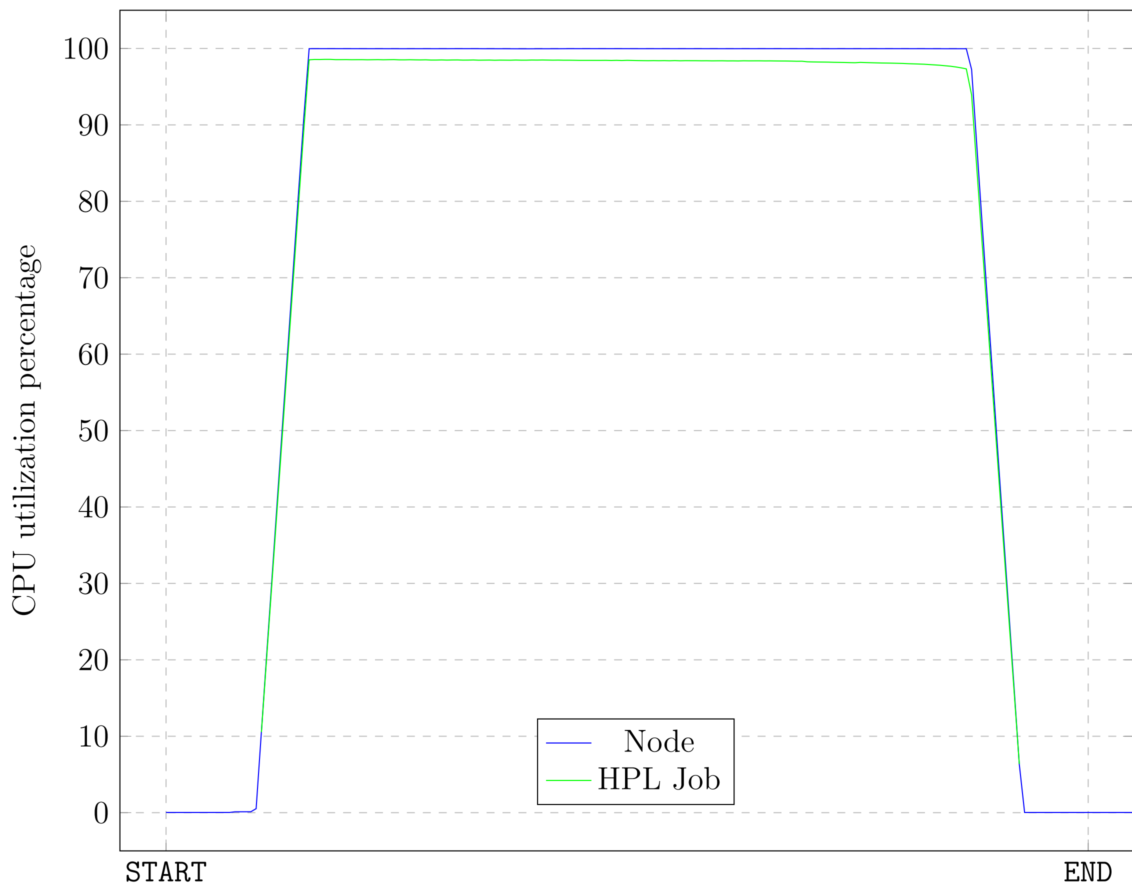


Figure 6.4: Bare-metal: total CPU usage on a node and CPU usage by HPL job

The performance results were again retrieved via Prometheus, by using the same queries presented for the pod workers case. Clearly, the default OpenShift metrics retrieval system is not running on the selected bare metal nodes, for this reason, an instance of cAdvisor was deployed on each of the 4 nodes together with an installation of Prometheus, which was configured to scrape the metrics from cAdvisor and expose them to be queried. At a first glance, the results are quite similar to the pod workers case, in Figure 6.5 a more detailed view is provided and a comparison between the two cases is more straightforward.

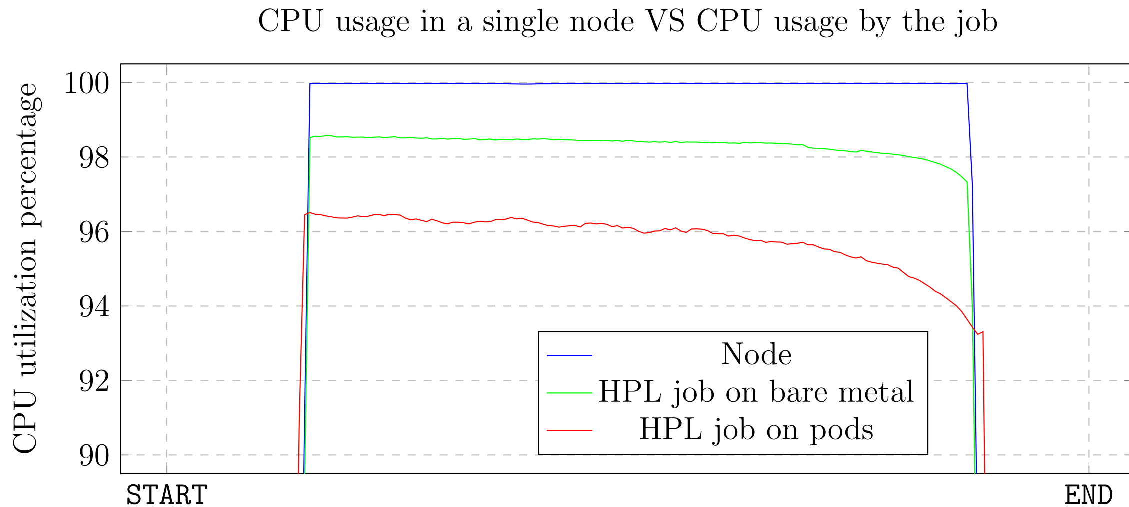


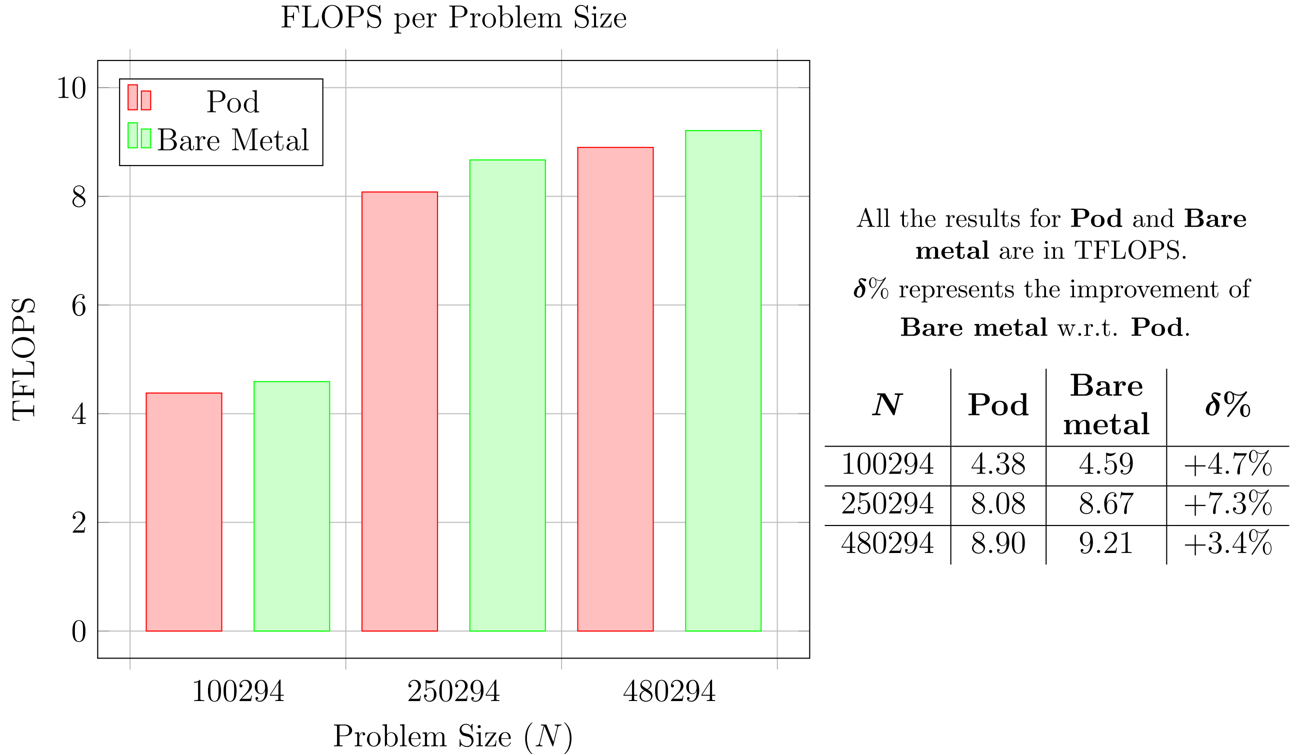
Figure 6.5: Detailed view of the results in Figure 6.2 and Figure 6.4

By looking at the detailed view, it can be seen that the CPU usage by the HPL job is around 98.5% if the benchmark is run on bare metal nodes, which is approximately 2.5% more CPU time than the equivalent case with pods. This difference in CPU usage by the HPL job between the two cases can be explained by the fact that there is no underlying orchestrator infrastructure that needs to be run on the node and that the HPL job can directly use all the available resources without any system-imposed limitation, in contrast with the aforementioned pod's cgroup constraint.

This kind of difference is partially noticeable in the testbed architecture, and reasonably even less on an infrastructure that provides a lot of computational power, like an extensive and powerful HPC cluster used for scientific computing. It can instead be more significant in a smaller cluster, as the ones in [39], where the prevention of usage of a certain amount of CPU on each node can become impactful on the overall performance of the cluster, since the resources are definitely scarcer.

To finally provide a comparative overview on the results of the HPL benchmarks, a comparison of the obtained TFLOPS for the different problem sizes between the

pod workers and the bare-metal workers is shown in Figure 6.6. It can be seen that the performance of the HPL benchmark is quite similar between the two cases, with a slight improvement in the bare-metal case, which is consistent with the higher CPU usage by the HPL job observed in Figure 6.5.



All the results for **Pod** and **Bare metal** are in TFLOPS.
δ% represents the improvement of **Bare metal** w.r.t. **Pod**.

N	Pod	Bare metal	δ%
100294	4.38	4.59	+4.7%
250294	8.08	8.67	+7.3%
480294	8.90	9.21	+3.4%

Figure 6.6: HPL result comparison between pod workers and bare-metal workers

Moreover, Figure 6.6 shows that the improvement of the bare-metal case w.r.t. the pod workers case in the analyzed experiments lies between 3% and 7%. This quantifies how much the performance of the HPL benchmark is improved on bare-metal nodes compared to Slinky-managed pod nodes, with a difference that is not negligible but also could not be considered very significant. It is up to the user to decide if this performance difference is important for their specific use case, keeping in mind the advantages and the disadvantages that either party have.

6.1.2 iPerf

The second performed benchmark is iPerf, which is a tool that allows to measure the available network bandwidth between two endpoints. This benchmark was chosen to see if there are differences in the available bandwidth between the pod workers and the bare-metal workers, which could affect the overall performance of the system in

presence of network-bound workloads. iPerf allows the user to conduct tests using a variable number of parallel processes that communicate simultaneously on the shared network between the endpoints. The collected results of the iPerf benchmark that represents bandwidth for different number of parallel processes are shown in Figure 6.7.

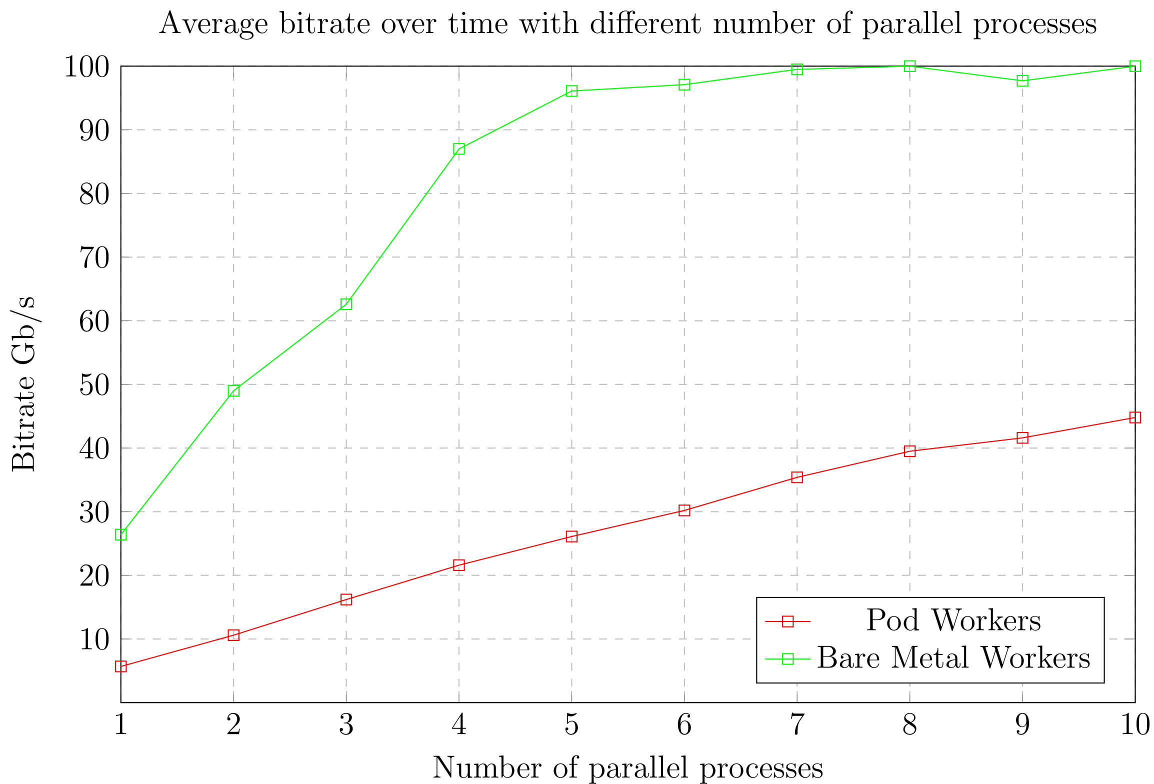


Figure 6.7: iPerf results for different number of parallel processes

By looking at this figure, it is reasonably clear that there is a significant difference in the available bandwidth between the two cases, with the bare-metal workers that reach the maximum available bandwidth of 100 Gb/s in a stable way starting from 5 parallel processes, with each process adding approximately 20 to 25 Gb/s in an almost linear fashion, until the bandwidth gets saturated. The main reason behind the per-process bandwidth limitation is that each single process runs on a single CPU, which on the test environment manages to handle only that specific amount of data per second, so the limitation is not determined by the network stack, but instead by the CPU performance while executing iPerf code. On the other hand, the network traffic between two pod workers can only reach a maximum of around 45 Gb/s with 10 parallel processes, with the parallel process contribution dropping to around 5 Gb/s for unit. This is a significant difference that can have a noticeable

impact on the performance of the system in presence of network-bound workloads, since the available bandwidth is significantly lower in the pod workers compared to the bare-metal workers.

This difference cannot be completely attributed to the CPU computational capabilities, as in the case of bare metal workers, since pod workers have the exact same hardware and their cgroups do not limit their resource access to this extent (as can instead be seen in the HPL case). This phenomenon, instead, can be explained by the fact that pod workers are subject to the limitations imposed by the underlying orchestrator infrastructure that virtualizes and abstract the network stack. These operations introduce substantial overhead and thus limit the available bandwidth, while the bare-metal workers can directly use all the available network resources without any limitation or virtualization by default. To test the limits of the available setup, another experiment was conducted with pod workers, shown in Figure 6.8, where the number of parallel processes was increased up to 100, to see if the available bandwidth could be improved by increasing the number of parallel processes, but the results show that there is an approximately linear growth of around 5 Gb/s for every parallel process up until around 15 of them, then the growth decreases and the curve tends to form a plateau around 85 Gb/s, which is still lower than the maximum available bandwidth of 100 Gb/s. The maximum bandwidth gets almost reached with at least 85 parallel processes, which is a significant difference if compared with the 5 required for the bare metal workers.

As previously introduced, this behavior can be explained by the abstractions of the networking stack that are introduced by OpenShift, with the main component being OVN (Open Virtual Network) Kubernetes, which uses OVS (Open vSwitch) Kubernetes as the per-node underlying infrastructure. This abstraction, while certainly needed to enable pod communication over network, introduces additional layers of further computation and also performs tunneling if the communicating pods reside in different nodes. The main responsibility of OVN Kubernetes is orchestrating the networking between pods and to maintain a "logical scheme" of the network in form of a set of *Logical Flows*, those objects represent the logical path between two cluster objects, for example two `Pods` or a couple (`Pod`,`Ingress`). Those *Logical Flows* are translated by the OVN controller in objects called `Openflow Rules`, that represent the rules that an underlying virtual switch (OVS) should follow to construct a communication path. After being emitted, packets get passed through a virtual interface to the OVS layer, which represent a virtual software switch to which all the pods are logically connected, that is responsible for forwarding the packets to the correct destination. To find the correct path, OVS needs to perform a lookup on its internal tables, which contains the rules for forwarding the packets in form of the aforementioned `Openflow Rules`, those tables are called *Flow Tables*

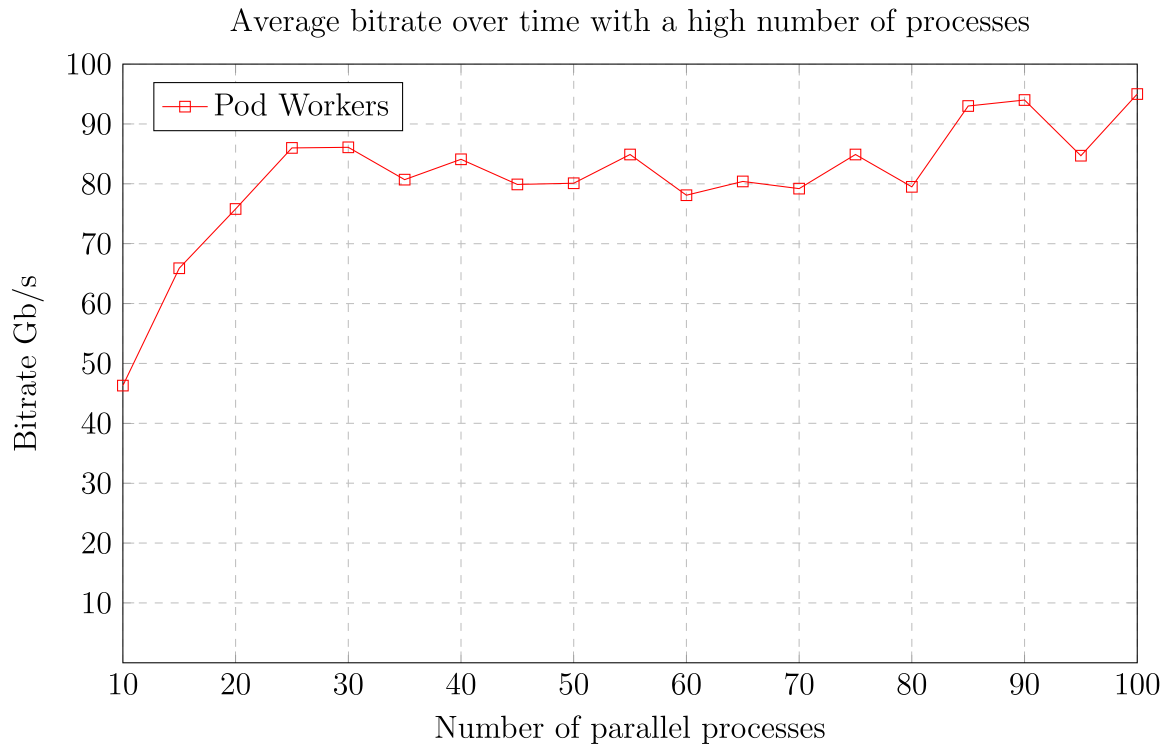


Figure 6.8: iPerf results for pod workers with high number of parallel processes

and comprise the OVS *pipeline* [11]. The OVS pipeline is then traversed until a matching rule is found, at this point the packet is forwarded to the next hop, which can be the destination pod if it resides on the same node, or a tunnel interface if the destination pod resides on a different node. Indeed, a tunneling is performed, in the case of the testing, a GENEVE (Generic Network Virtualization Encapsulation) tunneling is applied, and this requires the MTU (Maximum Transfer Unit) to be reduced to at least 100 bytes less than the MTU of the underlying network, which is 1500 bytes for the testing setup, to make the encapsulation header fit inside the packet. This means that the maximum size of the packets that can be sent between two pods is 1400 bytes, which can also contribute to the performance degradation. After the encapsulation, the packet is sent to the physical network stack and then gets emitted on the communication mean, to be received and decapsulated by the recipient. Concisely, the network abstraction introduced by OpenShift causes additional CPU cycles (and potentially more packets, due to lower MTU), meaning that the same CPU used in bare metal workers is capable to process less data per second. It is straightforward to note that bare metal nodes are not subject to any network virtualization and thus they can directly access the physical network stack, without any additional overhead.

Another additional experiment was conducted to see if, by completely bypassing the networking stack of the OpenShift cluster, it was possible to reach the maximum available bandwidth of 100 Gb/s. This was done by using the `hostNetwork: true` option in the pod specification for the Slinky workers, which allows the pods to use directly the host's physical network stack, thus bypassing the OVN Kubernetes and OVS layers. The plot shown in Figure 6.9 shows the results of this experiment.

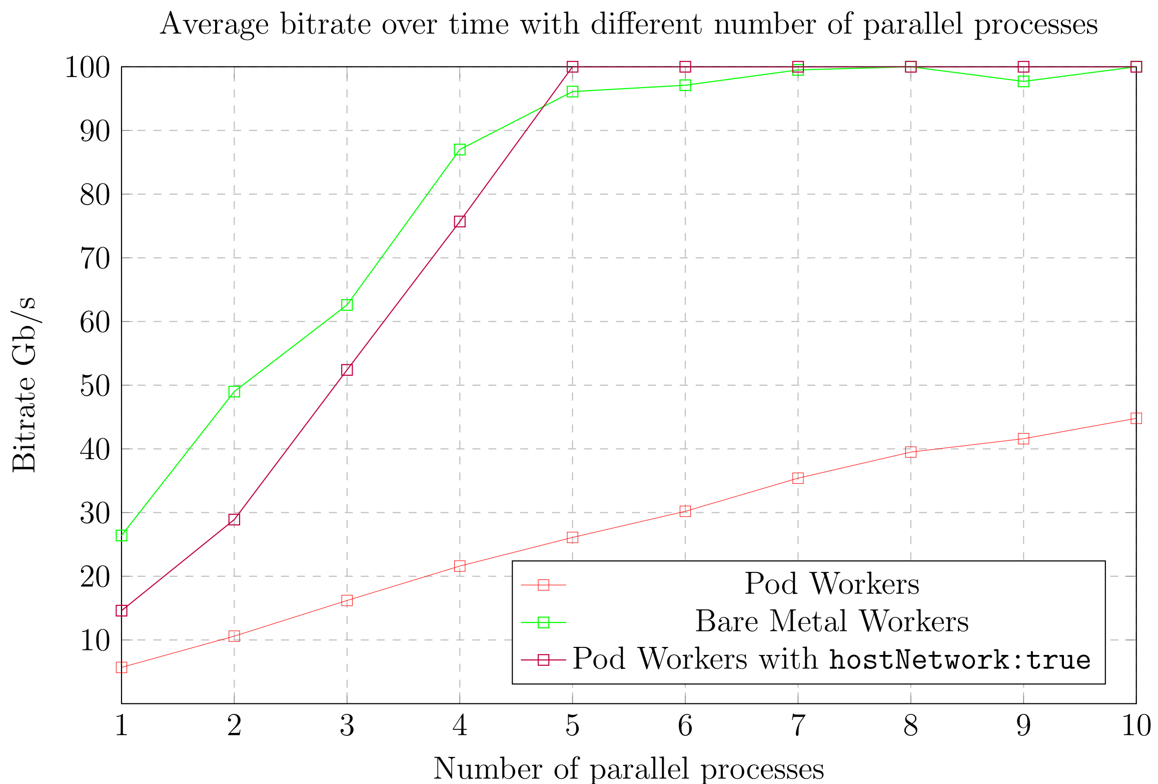


Figure 6.9: iPerf results when using `hostNetwork: true`

By looking at the graph, it can be seen that by using the `hostNetwork: true` option, the performance of pod workers is significantly improved and almost similar to the one of the bare-metal workers, with the maximum available bandwidth of 100 Gb/s that is reached with 5 parallel processes. But it is worth noting that by using this option, the isolation between pods is lost, since they are sharing the same network namespace as the host, which can lead to security issues and potential conflicts between different pods. Therefore, while this option can be useful for specific workloads that require high network bandwidth, it should be avoided to not lose the benefits of containerization in terms of isolation.

6.1.3 OSU Micro Benchmarks

The final chosen benchmark is the OSU Micro Benchmarks suite, which is a set of benchmarks that contains, among other, a number of tests that target the performance of MPI operations in terms of latency. For the case of this thesis the *collective* benchmarks `alltoall` and `allreduce` were selected, see Section 3.9.3 for more details. The choice of the benchmarks was performed to have a quantitative measure on how the system would behave in presence of a typical parallel HPC workload that leverages MPI for the network communications. This comes in contraposition with iPerf that measures the maximum available bandwidth, eventually saturating it, which is not what a typical communication between two endpoints represents, especially at higher scales such as 100 Gb Ethernet. In the following sections, the results of the two benchmarks are presented and analyzed, with a comparison between the pod workers and the bare-metal workers cases.

6.1.3.1 All-to-all

The first benchmark of the OSU Micro Benchmarks suite that was executed is the `alltoall` benchmark, which measures the latency of the all-to-all MPI operation for different message sizes. The results of the benchmark for the two cases are shown in Figure 6.10.

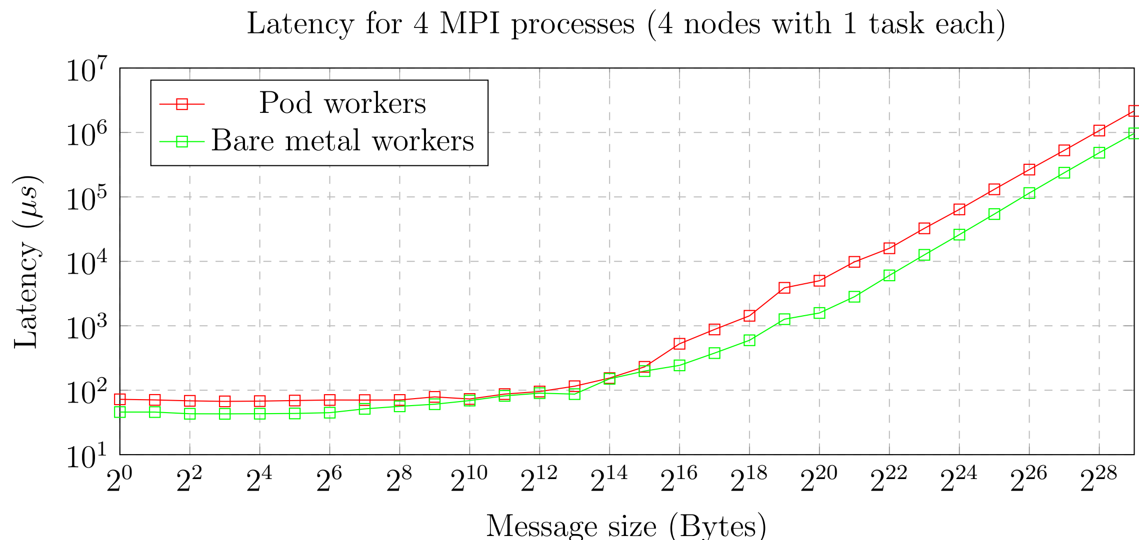


Figure 6.10: Latency measured with the `alltoall` OSU benchmark

The results shown in Figure 6.10 show that the latency of the `alltoall` operation is higher in the case of the pod workers compared to the bare-metal workers. The absolute difference between the latencies is substantially steady for messages

increases together with the message size from a certain value of it (approximately 2^{15} Bytes). The precise values for the latency in the two different cases is shown in Figure 6.11 with a table that presents the data along with the relative improvement of the bare-metal case with respect to the pod workers for each message size, in form of a multiplier (`pod_value` divided by `baremetal_value`).

Msg size	Pod	Bare metal	Improvement	Msg size	Pod	Bare metal	Improvement
1B	72.20	45.83	**1.57**×	32kB	230.62	198.14	**1.16**×
2B	70.71	45.81	**1.54**×	64kB	525.46	242.92	**2.16**×
4B	68.41	43.10	**1.58**×	128kB	876.82	376.74	**2.32**×
8B	67.12	42.98	**1.56**×	256kB	1428.66	595.85	**2.39**×
16B	67.76	43.18	**1.56**×	512kB	3883.43	1265.49	**3.06**×
32B	69.13	43.58	**1.58**×	1MB	4998.49	1580.39	**3.16**×
64B	70.54	44.82	**1.57**×	2MB	9812.97	2822.93	**3.47**×
128B	70.31	51.37	**1.36**×	4MB	15943.63	6051.88	**2.63**×
256B	70.57	56.23	**1.25**×	8MB	32512.41	12614.84	**2.57**×
512B	78.79	60.65	**1.29**×	16MB	63950.85	25977.12	**2.46**×
1kB	73.07	69.06	**1.05**×	32MB	131116.41	54143.04	**2.42**×
2kB	87.06	81.74	**1.06**×	64MB	265282.02	114832.70	**2.31**×
4kB	95.40	90.08	**1.05**×	128MB	527813.83	237152.75	**2.22**×
8kB	115.39	86.98	**1.32**×	256MB	1070327.40	484858.46	**2.20**×
16kB	154.35	150.34	**1.02**×	512MB	2161097.57	967785.47	**2.23**×

Figure 6.11: Latencies in μs for the `alltoall` OSU benchmark

From Figure 6.11, it can be seen that the performance of bare-metal workers is always better than the one of pod workers, with an improvement that spans from 1.02× to 3.47× depending on the message size, with a general trend of bigger benefits for the higher message sizes. This degradation of the performance of the `alltoall` operation in the case of pod workers can be again be explained by the fact that the communication between the nodes is handled by the underlying network infrastructure, the aforementioned OVN Kubernetes stack, which adds an overhead to the communication between the nodes that becomes more significant as the application-level message size increases. When instead the message size is low (range 1B - 64B), the latency for both cases is quite stable and the improvement of the bare-metal case with respect to the pod workers is around 1.5×. This seems to be compatible with the fact that for small message sizes, the communication performance is not particularly influenced by the available bandwidth (if the communication mean is not already congested). The latency difference is influenced by the overhead of the OpenShift network stack (passing through virtual interfaces to be processed by

OVS, the package getting directed to the right path with *flow rules*, ...), which contributes to this delta being around 20-25 μs. On the other hand, with a bigger message size, the communication performance is more influenced by the available bandwidth, which is apparently lower in the case of pod workers (as shown in Figure 6.7) and thus the latency value can be negatively affected by this, effectively opening the performance gap.

6.1.3.2 All-reduce

The second benchmark of the OSU Micro Benchmarks suite that was executed is the `allreduce` benchmark, which measures the latency of the all-reduce MPI collective operation for different message sizes.

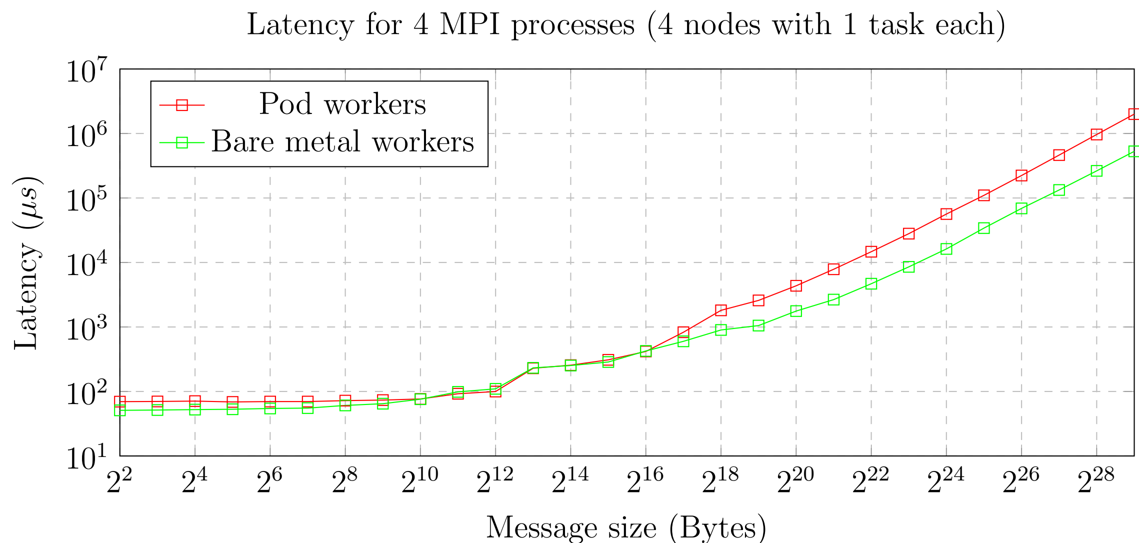


Figure 6.12: Latencies in μs for the `allreduce` OSU benchmark

The performance difference between the pod workers and the bare-metal workers is similar to the one observed in the `alltoall` benchmark and the same explanation provided for the previous benchmark can be applied to this case as well. Indeed both benchmarks measure communication latency and thus they are influenced by the same factors: available bandwidth and overhead introduced by the OpenShift network stack. Moreover, there are inherent algorithmic differences between the two benchmarks that also influence the performance.

To summarize the content of this section, it is possible to answer to **RQ1** in a positive way: there is in fact an advantage, even if not too marked from the CPU perspective, but for sure noticeable for those applications with high bandwidth requirements.

6.2 Injection Webhook

The first developed webhook, the Injection Webhook (see Section 4.2), is responsible for injecting specific data inside the Kubernetes objects involved in the provisioning of the Slurm bare metal nodes. It is designed to be sequential, so it handles one AdmissionRequest at a time, in a classical blocking request-response fashion. The main reason behind this design choice is that the webhook is not expected to be a bottleneck for the system. Since the webhook is thought to work on typically smaller, on premise clusters and it is only called once for each node provisioning, it is unlikely that the webhook will ever need to handle a very high number of requests in a very short amount of time. Moreover, the operations performed by the webhook are not computationally intensive, since it explores only a few Kubernetes objects and performs a limited number of API calls to the Kubernetes API server.

6.2.1 Performance evaluation

An experiment was conducted to measure the introduced latency by the Injection Webhook. By measuring the time taken to process an AdmissionRequest and to send back an eventual JSON patch and the response to the API server. The results of the experiment are shown in Figure 6.13, and it was conducted by monitoring the system for an extended period of time, during which multiple node provisioning were triggered and the latencies retrieved to have a statistically significant sample of around 200 entries.

The boxplot shown in Figure 6.13 shows that the latencies for the Injection Webhook calls are quite low, with a median value of 50.575 μs. The interquartile range is approximately 12.8 μs and the Coefficient of Variation $CV = \frac{\sigma}{\bar{x}} = 0.168$, without including the two outliers (i.e. the values $V | V < Q_1 - 1.5 * IQR \lor V > Q_3 + 1.5 * IQR$). So the webhook is not expected to introduce significant delays in the node provisioning process (especially considering that it is a time consuming process, as shown in Section 6.3), which is one of the main goal of the tool. This low latency can be explained by the fact that the operations performed by the webhook are not computationally intensive, and thus they can be executed quickly, and even if the webhook is designed to be sequential, all the requests can be handled in an acceptable amount of time.

6.3 Scaling Webhook

The other, fundamental, component of this thesis is the Scaling Webhook (see Section 4.2). This webhook is responsible for intercepting the scaling operations per-

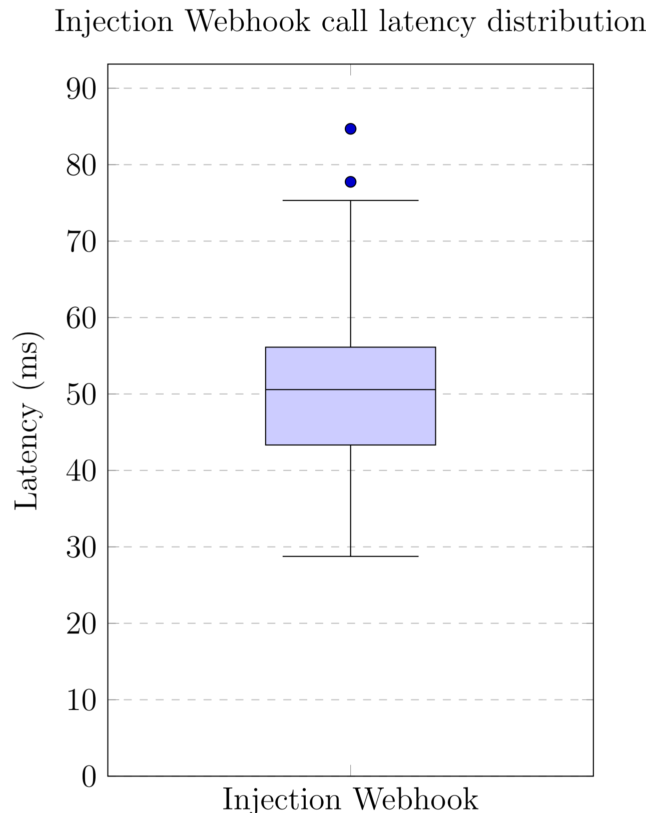


Figure 6.13: Distribution of individual call latencies for the Injection Webhook

formed by KEDA and replicating them on a `MachineDeployment` that manages the bare metal Slurm nodes. As the Injection Webhook, the Scaling Webhook is designed to be sequential, since it is not expected to handle a high number of requests in a short amount of time, due to automatic scaling operations being typically triggered with low frequency, considering the time it takes to provision a node. Moreover, akin to the Injection Webhook, the operations performed by the Scaling Webhook are not computationally intensive, since it reads into the request and replicates the operation before admitting the request.

6.3.1 Performance evaluation

The same experiment conducted for the Injection Webhook was also performed for the Scaling Webhook, by monitoring the system for an extended period of time and retrieving the latencies of the single calls to the webhook. The results of the experiment are shown in Figure 6.14.

As expected, the boxplot in Figure 6.14 shows that the latencies for the Scaling

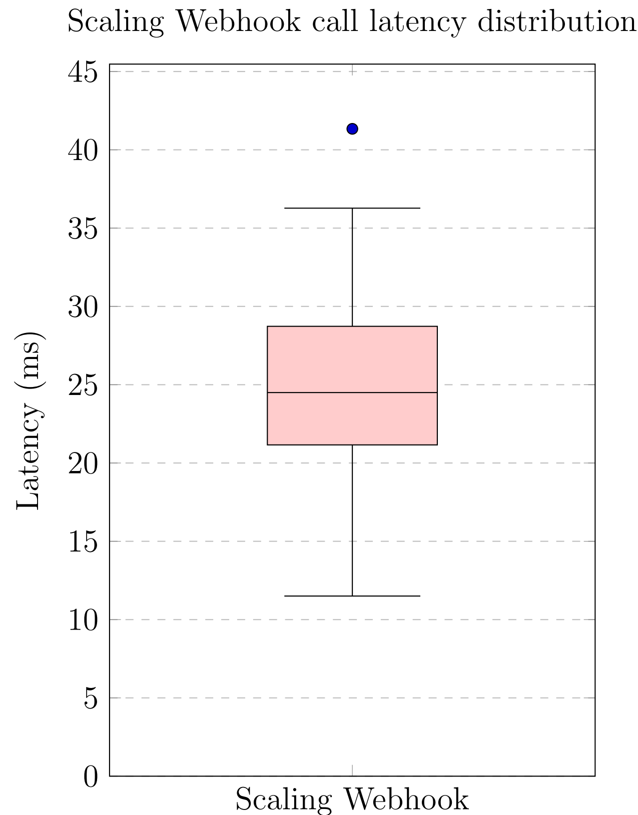


Figure 6.14: Distribution of individual call latencies for the Scaling Webhook

Webhook are even smaller than the ones of the Injection Webhook, with a median value of $24.4980\,\mu s$, an interquartile range of approximately $7.57\,\mu s$ and a Coefficient of Variation $CV = 0.179$, without including the outlier. It can be said that both of the webhooks have a similar "stability" in terms of latency, since the Coefficient of Variation is quite similar for both of them, but the Scaling Webhook has a lower latency overall. This can be explained by the fact that the operations performed by the Scaling Webhook are even less computationally intensive than the ones performed by the Injection Webhook, since it only needs to read the request and replicate it, without performing any additional API calls to the Kubernetes API server or exploring other Kubernetes objects.

All the results for both the webhooks were retrieved by using an installation of Prometheus in the management cluster, since the Kubernetes API server exposes some metrics related to the webhooks, such as a global counter called `apiserver_admission_webhook_admission_duration_seconds_count` that sums the total amount of latency introduced by the webhooks. By default, it does not provide a breakdown

of the latency of each call to the webhooks, but in the test environment, a scaling up and down of a `MachineDeployment` was triggered approximately every 5 minutes, which means that there were one call to the Injection Webhook every 10 minutes (since it is only triggered by the provisioning of a node) and one call to the Scaling Webhook every 5 minutes. This regular pattern allowed to extract the latencies of the single calls by checking the relative increase of the total counter with a certain resolution, lower than 5 minutes. This returned a series of values that represented the latencies of the single calls, the used PromQL query is:

```
increase(apiserver_admission_webhook_admission_duration_seconds_count
↪  {name="<webhook_name>",type="<webhook_type>"}[1m])
```

6.3.2 A scaling up example

In this section, an example of a scaling up process for a bare metal node is shown, the metrics chosen for the autoscaling is the number of pending Slurm jobs, as per Section 4.3.2.1. With the objective to provide the reader with a more extensive understanding of the durations of the different phases of the scaling up process, a plot is shown in Figure 6.15 that shows the elapsed time between the submission of a new Slurm job (`JOB_SUBMIT`) and the effective start of the execution of the job on a node (`JOB_START`), this process, assuming that there are not enough worker nodes to which assign the job, contains three different phases: the KEDA reaction, the action of the Scaling Webhook and the node provisioning, which itself contains the latency introduced by the Injection Webhook.

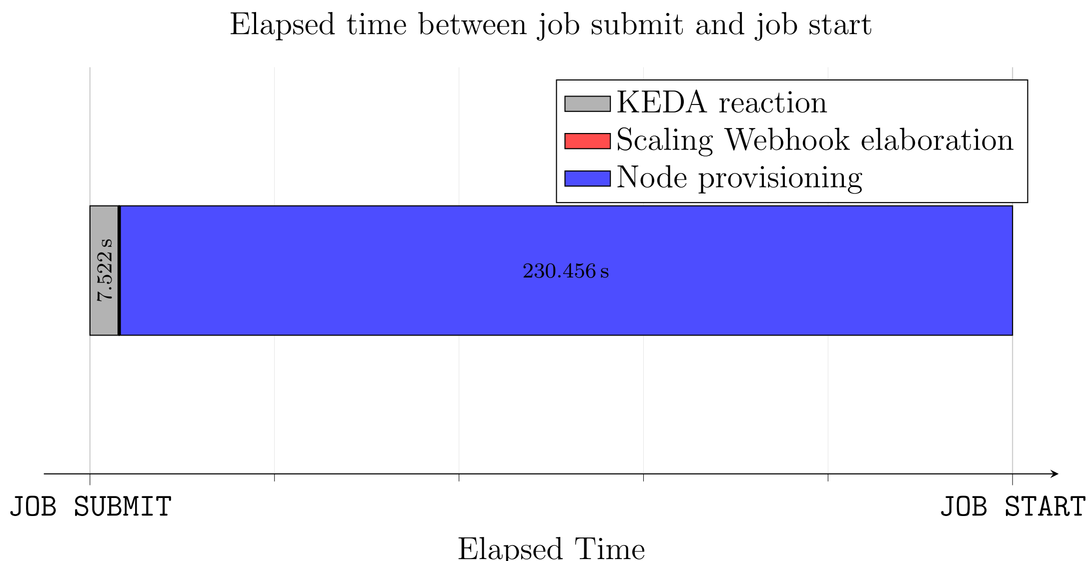


Figure 6.15: Total time required for a scaling up instance

As it can be seen, the orders of magnitude of the different phases are quite different, with the KEDA reaction being in the order of seconds, the Scaling Webhook elaboration being in the order of tens of milliseconds and the node provisioning being in the order of minutes. The reaction times of KEDA can be influenced by the configuration of the polling interval, in this particular case was set to 15 seconds, so the job submission arrived approximately in between of two checks. The time taken by the Scaling Webhook is quite low, as would be expected from the results in Section 6.3.1, and it is not expected to be a bottleneck for the system. The node provisioning time is the most significant part of the scaling up process, that includes the boot time of the node and the provisioning phase performed by cloud-init, which includes the installation of Slurm and the connection to the controller pod. Another, more detailed view of the scaling up process is shown in Figure 6.16, where the time axis is zoomed in to show more clearly the time taken by the Scaling Webhook and to give a better understanding of the different time scales of the phases involved.

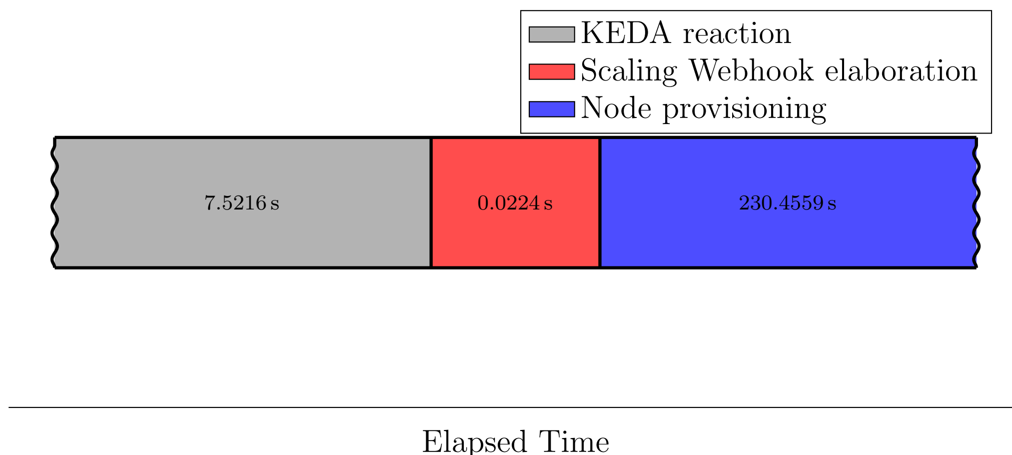


Figure 6.16: Detailed view of the total time required for a scaling up instance

Conclusions

This thesis investigated aspects on the integration of cloud-native orchestration technologies with traditional High Performance Computing infrastructures. The main focus was on the possibility of deploying a Slurm HPC cluster whose control plane is hosted inside a Kubernetes environment while worker nodes are in form of dynamically provisioned bare metal machines, managed via Cluster API from the Kubernetes control plane. The study addressed both architectural feasibility and performance comparisons between an unmodified Slinky cluster and the proposed architecture, following the research questions formulated in Section 1.1.

To investigate these aspects, an experimental environment was designed in which the Slurm control plane runs inside Kubernetes using the Slinky's `slurm-operator`, while compute nodes were either containerized workers or bare-metal machines provisioned through a modified version of the approach developed in [76]. The preliminary phase of the research focused on building a working integration between the containerized Slurm controller and external physical worker nodes, solving the necessary network implications that arose. Once this architecture was made operational, it provided a baseline that positively addressed the solution of **RQ2** and enabled the possibility to evaluate **RQ1**.

The performance evaluation was realized using a set of benchmarks to analyze different aspects of typical HPC workloads, as described in Section 3.9. The experiments measured floating point computational performance, available network bandwidth and communication latency for certain common MPI operations. These benchmarks were executed on clusters with worker nodes deployed according to the modalities described in **RQ1**, to determine whether there was any difference. The obtained results showed that bare-metal worker nodes generally provide higher per-process network bandwidth and lower communication latency compared to workers running inside Kubernetes pods. The difference is mainly attributable to the network virtualization layers used by container orchestration platforms, which introduce additional processing steps. When MPI communication benchmarks were executed, the measurements also showed generally higher latency for operations executed on

pod workers compared to bare-metal nodes, reflecting the same additional overhead. Lastly, the benchmark showed slight advantages even in floating-point performance. This evaluation confirmed positively the interrogative of **RQ1**, showing that the performance of HPC workloads can be affected by the choice of execution environment, and justified the possibility of using bare-metal nodes to achieve better performance, allowing to continue the investigation for **RQ3**.

Addressing **RQ3**, the conducted research introduced a mechanism based on Kubernetes admission webhooks. A component referred to as the *Injection Webhook* was implemented to intercept the provisioning process of new machines and automatically inject the configuration required for them to correctly join the Slurm cluster in an automated way. Through this mechanism, newly provisioned nodes automatically receive authentication credentials and register with the controller pod without manual intervention. The successful operation of this mechanism demonstrates that the deployment of this particular configuration can be fully automated within a Kubernetes infrastructure, positively answering **RQ3**.

Given the inherent dynamicity of Kubernetes and the potential benefits of autoscaling solutions [8], the investigation for **RQ4** led to the development of a component referred to as the `Scaling Webhook`. It was introduced to address the structural limitations of the Cluster API based approach for deploying bare metal nodes with respect to the default solution used by Slinky (see Section 4.3.2.1). The webhooks intercept scaling operations triggered by the autoscaling system and translate them into modifications of the bare metal node number. Through this mechanism, scaling requests originating from workload necessity can automatically increase or decrease the number of worker nodes available in the Slurm cluster, finally confirming the question posed in **RQ4**.

The architectural solutions developed in this thesis for addressing **RQ3** and **RQ4** integrate into a system referred to as SliMe, which ultimately combines workload-level convergence (Slinky) with node-level convergence ([76]). In this architecture, the Slurm control plane exists as a containerized entity within Kubernetes, while worker nodes are dynamically managed through infrastructure automation tools. This *Integrated* and *Dynamic* solution developed in this thesis allows for another step towards the convergence between traditional HPC systems and cloud.

All the produced code for SliMe is openly available on GitHub at [29].

Future Work

The research conducted in this thesis opens up several possibilities for future work and further investigations. Some of the most interesting directions include investiga-

tions and extensions of different autoscaling logics, potentially introducing improved PromQL queries to better capture the workload needs and trigger scaling operations in an efficient way. Another possibility would be to integrate the concept of Slurm job in Kubernetes, in a similar way to what is done in [47], to have autoscaling decisions that are more aware on what is the current state of the workload, this could help to address the "single bounce" problem presented in Paragraph "Issues" in Section 4.3.2.1. Moreover, as introduced in Section 1.1, it can be interesting to conduct research on specific workloads that can benefit from the hybrid architecture proposed in this thesis. Finally, it can be interesting to explore the possibility of introducing a position-aware provisioning for bare metal nodes, in which the creation of new nodes is not only based on the workload volume but also on the physical position of the nodes in the cluster. This can help provide precise and targeted support to specific topology domains (zones, areas, ...) that need more computation power.

Bibliography

[1] Subil Abraham et al. "On the Use of Containers in High Performance Computing Environments". In: *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*. 2020, pp. 284–293. DOI: 10.1109/CLOUD49709.2020.00048.

[2] Gojko Adzic and Robert Chatley. "Serverless computing: economic and architectural impact". In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2017. Paderborn, Germany: Association for Computing Machinery, 2017, pp. 884–889. ISBN: 9781450351058. DOI: 10.1145/3106237.3117767. URL: https://doi.org/10.1145/3106237.3117767.

[3] Dong H. Ahn et al. "Flux: Overcoming Scheduling Challenges for Exascale Workflows". In: *2018 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS)*. 2018, pp. 10–19. DOI: 10.1109/WORKS.2018.00007.

[4] Guillaume Ambal et al. "Semantics of Remote Direct Memory Access: Operational and Declarative Models of RDMA on TSO Architectures". In: *Proc. ACM Program. Lang.* 8.OOPSLA2 (Oct. 2024). DOI: 10.1145/3689781. URL: https://doi.org/10.1145/3689781.

[5] Kubernetes SIG Autoscaling. *Cluster Autoscaler*. [Online]. URL: https://github.com/kubernetes/autoscaler/tree/master/cluster-autoscaler.

[6] Mark Baker and Rajkumar Buyya. "Cluster computing: the commodity supercomputer". In: *Software: Practice and Experience* 29.6 (1999), pp. 551–576. DOI: https://doi.org/10.1002/(SICI)1097-024X(199905)29:6<551::AID-SPE248>3.0.CO;2-C. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/%28SICI%291097-024X%28199905%2929%3A6%3C551%3A%3AAID-SPE248%3E3.0.CO%3B2-C.

[7] Lucas Benedicic et al. "Portable, high-performance containers for HPC". In: *CoRR* abs/1704.03383 (2017). arXiv: 1704.03383. URL: http://arxiv.org/abs/1704.03383.

[8] Rajat Bhattarai, Howard Pritchard, and Sheikh Ghafoor. "Enabling Elasticity in Scientific Workflows for High-Performance Computing Systems". In: *Euro-*

Par 2025: Parallel Processing. Ed. by Wolfgang E. Nagel, Diana Goehringer, and Pedro C. Diniz. Cham: Springer Nature Switzerland, 2026, pp. 307–321. ISBN: 978-3-031-99854-6.

[9] Sharon Boeyen et al. *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. Tech. rep. 5280. May 2008. 151 pp. DOI: 10.17487/RFC5280. URL: https://www.rfc-editor.org/info/rfc5280.

[10] Reuben D. Budiardja et al. "Ready for the Frontier: Preparing Applications for the World's First Exascale System". In: *High Performance Computing*. Ed. by Abhinav Bhatele et al. Cham: Springer Nature Switzerland, 2023, pp. 182–201. ISBN: 978-3-031-32041-5.

[11] Axel Busch and Martin Kammerer. "Network Performance Influences of Software-defined Networks on Micro-service Architectures". In: *Proceedings of the ACM/SPEC International Conference on Performance Engineering*. ICPE '21. Virtual Event, France: Association for Computing Machinery, 2021, pp. 153–163. ISBN: 9781450381949. DOI: 10.1145/3427921.3450236. URL: https://doi.org/10.1145/3427921.3450236.

[12] Christophe Cérin, Nicolas Greneche, and Tarek Menouer. "Towards Pervasive Containerization of HPC Job Schedulers". In: *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. 2020, pp. 281–288. DOI: 10.1109/SBAC-PAD49847.2020.00046.

[13] cert-manager. *Cert-manager*. [Online]. URL: https://cert-manager.io/.

[14] Antony Chazapis et al. "Running Kubernetes Workloads on HPC". In: *High Performance Computing*. Ed. by Amanda Bienz et al. Cham: Springer Nature Switzerland, 2023, pp. 181–192. ISBN: 978-3-031-40843-4.

[15] ODU Research Cloud Computing. en. [Online]. URL: https://wiki.hpc.odu.edu/en/slurm.

[16] Brandon Cook et al. "Seamless end-to-end containerized HPC environments". In: *Proceedings of the SC '25 Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC Workshops '25. New York, NY, USA: Association for Computing Machinery, 2025, pp. 126–134. ISBN: 9798400718717. DOI: 10.1145/3731599.3767355. URL: https://doi.org/10.1145/3731599.3767355.

[17] Marcin Copik et al. "XaaS Containers: Performance-Portable Representation With Source and IR Containers". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '25. Association for Computing Machinery, 2025, pp. 533–555. ISBN: 9798400714665. DOI: 10.1145/3712285.3759868. URL: https://doi.org/10.1145/3712285.3759868.

[18] Jonathar Decker, Sören Metje, and Julian Kunkel. "Running Kubernetes Workloads on Rootless HPC Systems using Slurm". In: *CLOUD COMPUTING 2025, The Sixteenth International Conference on Cloud Computing, GRIDs, and Virtualization*. 2025, pp. 100–107. ISBN: 978-1-68558-258-6. URL: https://www.thinkmind.org/articles/cloud_computing_2025_2_60_20036.pdf.

[19] Jack J. Dongarra, Piotr Luszczek, and Antoine Petitet. "The LINPACK Benchmark: past, present and future". In: *Concurrency and Computation: Practice and Experience* 15.9 (2003), pp. 803–820. DOI: https://doi.org/10.1002/cpe.728. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.728.

[20] ESnet. *iPerf3*. en. URL: https://iperf.fr/.

[21] Álvaro Fernández-González et al. "Historical review and future challenges in Supercomputing and Networks of Scientific Communication". In: *The Journal of Supercomputing* 71.12 (Dec. 2015), pp. 4476–4503. ISSN: 1573-0484. DOI: 10.1007/s11227-015-1544-3. URL: https://doi.org/10.1007/s11227-015-1544-3.

[22] flannel-io. *Flannel: a network fabric for containers*. en. [Online]. URL: https://github.com/flannel-io/flannel.

[23] CA/Browser Forum. *Ballot SC063v4: Make OCSP Optional, Require CRLs, and Incentivize Automation*. en. [Online]. July 2023. URL: https://cabforum.org/2023/07/14/ballot-sc063v4-make-ocsp-optional-require-crls-and-incentivize-automation/.

[24] Surya Kant Garg and J. Lakshmi. "Workload performance and interference on containers". In: *2017 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computed, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*. 2017, pp. 1–6. DOI: 10.1109/UIC-ATC.2017.8397647.

[25] Lisa Gerhardt et al. "Shifter: Containers for HPC". In: *Journal of Physics: Conference Series* 898.8 (Oct. 2017), p. 082021. DOI: 10.1088/1742-6596/898/8/082021. URL: https://doi.org/10.1088/1742-6596/898/8/082021.

[26] Deepa Gurung et al. "Cloud Revolution: Tracing the Origins and Rise of Cloud Computing". In: *2026 IEEE 16th Annual Computing and Communication Workshop and Conference (CCWC)*. 2026, pp. 1100–1106. DOI: 10.1109/CCWC67433.2026.11393790.

[27] *History and overview of high performance computing*. Presentation slides / PDF file. [Online]. 2020. URL: https://www.math-cs.gordon.edu/courses/cps343/presentations/History_and_Overview_of_HPC.pdf.

[28] Md Rajib Hossen, Mohammad A. Islam, and Kishwar Ahmed. “Practical Efficient Microservice Autoscaling with QoS Assurance”. In: *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*. HPDC ’22. Minneapolis, MN, USA: Association for Computing Machinery, 2022, pp. 240–252. ISBN: 9781450391993. DOI: 10.1145/3502181.3531460. URL: https://doi.org/10.1145/3502181.3531460.

[29] IBM. *cluster-api-bootstrap-virtual-kubelet: Custom Cluster API (CAPI) bootstrap provider designed to enable the provisioning and lifecycle management of non-Kubernetes nodes*. en. [Online]. URL: https://github.com/IBM/cluster-api-bootstrap-virtual-kubelet.

[30] Red Hat Inc. *OpenShift*. en. [Online]. URL: https://www.redhat.com/en/technologies/cloud-computing/openshift.

[31] Nikhil Jain et al. “Evaluating HPC Networks via Simulation of Parallel Workloads”. In: *SC ’16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2016, pp. 154–165. DOI: 10.1109/SC.2016.13.

[32] Hamza Javed, Adel N. Toosi, and Mohammad S. Aslanpour. “Serverless Platforms on the Edge: A Performance Analysis”. In: *New Frontiers in Cloud Computing and Internet of Things*. Ed. by Rajkumar Buyya et al. Cham: Springer International Publishing, 2022, pp. 165–184. ISBN: 978-3-031-05528-7. DOI: 10.1007/978-3-031-05528-7_6. URL: https://doi.org/10.1007/978-3-031-05528-7_6.

[33] Kedaorg. *Keda*. [Online]. URL: https://keda.sh/.

[34] Laurence J. Kedward et al. “The State of Fortran”. In: *Computing in Science & Engineering* 24.2 (2022), pp. 63–72. DOI: 10.1109/MCSE.2022.3159862.

[35] Rafael Keller Tesser and Edson Borin. “Containers in HPC: a survey”. In: *The Journal of Supercomputing* 79.5 (Mar. 2023), pp. 5759–5827. ISSN: 1573-0484. DOI: 10.1007/s11227-022-04848-y. URL: https://doi.org/10.1007/s11227-022-04848-y.

[36] *Kubernetes*. en. [Online]. URL: https://www.kubernetes.io/.

[37] Kubernetes. *client-go: Go client for Kubernetes*. en. [Online]. URL: https://github.com/kubernetes/client-go.

[38] Kubernetes. *Horizontal Pod Autoscaling*. [Online]. URL: https://kubernetes.io/docs/concepts/workloads/autoscaling/horizontal-pod-autoscale/.

[39] Dileep Kumar, Sheeraz Memon, and Liaquat Ali Thebo. “Design, Implementation & Performance Analysis of Low Cost High Performance Computing (HPC) Clusters”. In: *2018 12th International Conference on Signal Processing and Communication Systems (ICSPCS)*. 2018, pp. 1–6. DOI: 10.1109/ICSPCS.2018.8631769.

[40] Gregory M. Kurtzer, Vanessa Sochat, and Michael W. Bauer. “Singularity: Scientific containers for mobility of compute”. In: *PLOS ONE* 12.5 (May 2017), pp. 1–20. DOI: 10.1371/journal.pone.0177459. URL: https://doi.org/10.1371/journal.pone.0177459.

[41] Kubernetes SIG Cluster Lifecycle. *Cluster API*. en. [Online]. URL: https://cluster-api.sigs.k8s.io/.

[42] Boris Lublinsky, Elise Jennings, and Viktória Spišaková. “A Kubernetes ‘Bridge’ Operator between Cloud and External Resources”. In: *2023 8th International Conference on Cloud Computing and Big Data Analytics (ICCCBDA)*. 2023, pp. 263–269. DOI: 10.1109/ICCCBDA56900.2023.10154770.

[43] Theo Lynn et al. “Understanding the Determinants and Future Challenges of Cloud Computing Adoption for High Performance Computing”. In: *Future Internet* 12.8 (2020). ISSN: 1999-5903. DOI: 10.3390/fi12080135. URL: https://www.mdpi.com/1999-5903/12/8/135.

[44] Zane Ma et al. “What’s in a Name? Exploring CA Certificate Control”. In: *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 4383–4400. ISBN: 978-1-939133-24-3. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/ma.

[45] Zaigham Mahmood. “Cloud Computing: Characteristics and Deployment Approaches”. In: *2011 IEEE 11th International Conference on Computer and Information Technology*. 2011, pp. 121–126. DOI: 10.1109/CIT.2011.75.

[46] Johannes Manner et al. “Cold Start Influencing Factors in Function as a Service”. In: *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. 2018, pp. 181–188. DOI: 10.1109/UCC-Companion.2018.00054.

[47] Dennis Marttinen. *Supernetes: Kubernetes bridge for Supercomputers*. en. [Online]. URL: https://github.com/supernetes/supernetes.

[48] Peter Mell and Timothy Grance. *The NIST Definition of Cloud Computing*. en. Sept. 2011. DOI: https://doi.org/10.6028/NIST.SP.800-145.

[49] Metal^3-io. *Metal^3 - Metal Kubed*. en. [Online]. URL: https://metal3.io/.

[50] Metal3-Io. *metal3-dev-env: Metal3 Development Environment*. [Online]. URL: https://github.com/metal3-io/metal3-dev-env.

[51] MetalLB. *MetalLB*. [Online]. URL: https://metallb.io/.

[52] Daniel J. Milroy et al. “One Step Closer to Converged Computing: Achieving Scalability with Cloud-Native HPC”. In: *2022 IEEE/ACM 4th International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*. 2022, pp. 57–70. DOI: 10.1109/CANOPIE-HPC56864.2022.00011.

[53] Claudia Misale et al. “Towards Standard Kubernetes Scheduling Interfaces for Converged Computing”. In: *Driving Scientific and Engineering Discoveries*

Through the Integration of Experiment, Big Data, and Modeling and Simulation. Ed. by Jeffrey Nichols et al. Cham: Springer International Publishing, 2022, pp. 310–326. ISBN: 978-3-030-96498-6.

[54] Paul Mockapetris. *Domain names - implementation and specification.* Tech. rep. 1035. Nov. 1987. 55 pp. DOI: 10.17487/RFC1035. URL: https://www.rfc-editor.org/info/rfc1035.

[55] Subrota Kumar Mondal et al. "Kubernetes in IT administration and serverless computing: An empirical study and research challenges". In: *The Journal of Supercomputing* 78.2 (July 2021), pp. 2937–2987. DOI: 10.1007/s11227-021-03982-3. URL: https://doi.org/10.1007/s11227-021-03982-3.

[56] Nina Mujkanovic et al. "Survey of adaptive containerization architectures for HPC". In: SC-W '23. Denver, CO, USA: Association for Computing Machinery, 2023, pp. 165–176. ISBN: 9798400707858. DOI: 10.1145/3624062.3624588. URL: https://doi.org/10.1145/3624062.3624588.

[57] *Munge.* en. [Online]. URL: https://dun.github.io/munge/.

[58] Kubernetes SIG Network. *ExternalDNS.* [Online]. URL: https://github.com/kubernetes-sigs/external-dns.

[59] John Nickolls et al. "Scalable parallel programming with CUDA". In: *ACM SIGGRAPH 2008 Classes.* SIGGRAPH '08. Los Angeles, California: Association for Computing Machinery, 2008. ISBN: 9781450378451. DOI: 10.1145/1401132.1401152. URL: https://doi.org/10.1145/1401132.1401152.

[60] Diego Ongaro and John Ousterhout. "In Search of an Understandable Consensus Algorithm". In: *2014 USENIX Annual Technical Conference (USENIX ATC 14).* Philadelphia, PA: USENIX Association, June 2014, pp. 305–319. ISBN: 978-1-931971-10-2. URL: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro.

[61] Tapasya Patki et al. "Fluxion: A Scalable Graph-Based Resource Model for HPC Scheduling Challenges". In: *Proceedings of the SC '23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis.* SC-W '23. Denver, CO, USA: Association for Computing Machinery, 2023, pp. 2077–2088. ISBN: 9798400707858. DOI: 10.1145/3624062.3624286. URL: https://doi.org/10.1145/3624062.3624286.

[62] Reid Priedhorsky and Tim Randles. "Charliecloud: Unprivileged Containers for User-Defined Software Stacks in HPC". eng. In: Assoc Computing Machinery, 2017, p. 10. ISBN: 9781450351140. DOI: 10.1145/3126908.3126925.

[63] ProjectCalico. *Calico: Cloud native networking and network security.* en. [Online]. URL: https://github.com/projectcalico/calico.

[64] Rolf Rabenseifner, Georg Hager, and Gabriele Jost. "Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes". In: *2009 17th*

Euromicro International Conference on Parallel, Distributed and Network-based Processing. 2009, pp. 427–436. DOI: 10.1109/PDP.2009.43.

[65] Redhat-Hpc. *slinky-on-openshift: Pattern for running the Slurm operator on OpenShift*. en. [Online]. URL: https://github.com/redhat-hpc/slinky-on-openshift.

[66] Nathan Rini. *Containers in Slurm*. Presentation slides / PDF file. [Online]. 2024. URL: https://slurm.schedmd.com/SC24/Containers.pdf.

[67] Stefan Santesson et al. *X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP*. Tech. rep. 6960. June 2013. 41 pp. DOI: 10.17487/RFC6960. URL: https://www.rfc-editor.org/info/rfc6960.

[68] SchedMD. *Slinky*. [Online]. URL: https://github.com/slinkyproject.

[69] SchedMD. *Slurm Workload Manager - slurm.conf*. [Online]. URL: https://slurm.schedmd.com/slurm.conf.html.

[70] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. "Architectural Implications of Function-as-a-Service Computing". In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-52. Columbus, OH, USA: Association for Computing Machinery, 2019, pp. 1063–1075. ISBN: 9781450369381. DOI: 10.1145/3352460.3358296. URL: https://doi.org/10.1145/3352460.3358296.

[71] Marcin Skwarek et al. "Characterizing Vulnerability of DNS AXFR Transfers with Global-Scale Scanning". In: *2019 IEEE Security and Privacy Workshops (SPW)*. 2019, pp. 193–198. DOI: 10.1109/SPW.2019.00044.

[72] Vanessa Sochat et al. "Converged Computing: A Best of Both Worlds of High-Performance Computing and Cloud". In: *Computing in Science & Engineering* 26.3 (2024), pp. 4–7. DOI: 10.1109/MCSE.2024.3489732.

[73] Markus sosnowski et al. "An Internet-Wide View on HTTPS Certificate Revocations: Observing the Revival of CRLs via Active TLS Scans". In: *2024 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. 2024, pp. 297–306. DOI: 10.1109/EuroSPW61312.2024.00038.

[74] Thomas Sterling, Matthew Anderson, and Maciej Brodowicz. "Chapter 1 - Introduction". In: *High Performance Computing*. Ed. by Thomas Sterling, Matthew Anderson, and Maciej Brodowicz. Boston: Morgan Kaufmann, 2018, pp. 1–42. ISBN: 978-0-12-420158-3. DOI: https://doi.org/10.1016/B978-0-12-420158-3.00001-0. URL: https://www.sciencedirect.com/science/article/pii/B9780124201583000010.

[75] Jayachander Surbiryala and Chunming Rong. "Cloud Computing: History and Overview". In: *2019 IEEE Cloud Summit*. 2019, pp. 1–7. DOI: 10.1109/CloudSummit47114.2019.00007.

[76] Michele Tagliani. “Resource Management of HPC Infrastructures based on Kubernetes”. MA thesis. URL: https://amslaurea.unibo.it/id/eprint/37321/.

[77] Emin Topalovic et al. “Towards Short-Lived Certificates”. In: *IEEE Oakland Web 2.0 Security and Privacy (W2SP 2012)*. 2012. URL: https://www.ieee-security.org/TC/W2SP/2012/papers/w2sp12-final9.pdf.

[78] Ohio State University. *OSU Micro-Benchmarks (OMB)*. en. URL: https://mvapich.cse.ohio-state.edu/benchmarks/.

[79] *Virtual Kubelet*. en. URL: https://github.com/virtual-kubelet/virtual-kubelet.

[80] Paul A. Vixie et al. *Dynamic Updates in the Domain Name System (DNS UPDATE)*. RFC 2136. Apr. 1997. DOI: 10.17487/RFC2136. URL: https://www.rfc-editor.org/info/rfc2136.

[81] Andy B. Yoo, Morris A. Jette, and Mark Grondona. “SLURM: Simple Linux Utility for Resource Management”. In: *Job Scheduling Strategies for Parallel Processing*. Ed. by Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 44–60. ISBN: 978-3-540-39727-4.

[82] Rehmana Younis et al. “A Comprehensive Analysis of Cloud Service Models: IaaS, PaaS, and SaaS in the Context of Emerging Technologies and Trend”. In: *2024 International Conference on Electrical, Communication and Computer Engineering (ICECCE)*. 2024, pp. 1–6. DOI: 10.1109/ICECCE63537.2024.10823401.

[83] Naweiluo Zhou, Huan Zhou, and Dennis Hoppe. “Containerization for High Performance Computing Systems: Survey and Prospects”. In: *IEEE Transactions on Software Engineering* 49.4 (Apr. 2023), pp. 2722–2740. ISSN: 0098-5589. DOI: 10.1109/TSE.2022.3229221. URL: https://doi.org/10.1109/TSE.2022.3229221.

[84] Naweiluo Zhou et al. “Container orchestration on HPC systems through Kubernetes”. In: *Journal of Cloud Computing* 10.1 (Feb. 2021), p. 16. ISSN: 2192-113X. DOI: 10.1186/s13677-021-00231-z. URL: https://doi.org/10.1186/s13677-021-00231-z.

Appendix A

BIND configuration

In this appendix, a sample configuration for the BIND server is provided, a similar one is used in the deployment of the solution in Section 4.3.3.3.

A.1 Configuration files

As stated in Section 4.3.3.3, the BIND server can be configured through a configmap mounted as a volume inside the pod.

An example for configuration files can be the following:

```
data:
  named.conf: |
    include "/etc/bind/named.conf.options";
    include "/etc/bind/named.conf.local";

  named.conf.options: |
    options {
      directory "/var/cache/bind";
      recursion no;

      listen-on port 53 { any; };
      listen-on-v6 { any; };

      allow-transfer { none; };
      pid-file "/var/cache/bind/named.pid";
      session-keyfile "/var/cache/bind/session.key";
    };
```

```
  named.conf.local: |
    zone "slurm.metrics" {
      type master;
      file "/var/cache/bind/db.slurm.metrics";

      allow-update { any; };

    };

  db.slurm.metrics: |
    $TTL 3600
    @   IN  SOA ns1.slurm.metrics. admin.slurm.metrics. (
            2026010100 ; serial
            7200       ; refresh
            1200       ; retry
            2419200    ; expire
            3600 )     ; minimum
        IN  NS  ns1.slurm.metrics.

    ns1 IN  A  <static_DNS_server_IP>
```

`named.conf` is the main config file for the `named` daemon, which is one of the main components of BIND, which includes a global option file and a zone configuration file.

Inside the file `named.conf.options` there is the `directory` specificating the working dir for the server, it will be the base for relative path and the directory in which the output files will be put, so it has to be writable. The option `recursion` is self-explicative, and in this case is disabled, this server only honors requests relative to its zone. Moreover, The option `allow-transfer` set to `none` is a security policy intended to mitigate some types of attack [71] related to unrestricted DNS zone tranfers.

For what instead concerns `named.conf.local`, it defines the server as authoritative on the zone `slurm.metrics` with `type master`. The option `allow-update` set to `any` is a deliberately insecure choice due to lab environment, it should not be used in production environments and allows for any entity to send dynamic DNS updates, which is clearly not desirable for a publicly reachable service.

The last file (`db.slurm.metrics`) is a Master file [54] that contains data for the zone `slurm.metrics`. It begins with a `$TTL` directive that sets the default Time To Live for all records in the zone to 3600 seconds, which is arbitrary for the test environment. The SOA (Start of Authority) record declares `ns1.slurm.metrics.` as

the primary nameserver and `admin.slurm.metrics.` as the administrative contact, followed by the required timing parameters that sets zone transfers and caching. The NS record specifies `ns1.slurm.metrics.` as the authoritative nameserver for the zone, and the final A record maps `ns1` to the static IP address of the server.

With its operations, ExternalDNS will then dynamically add further A and TXT records to this zone file through RFC 2136 updates.

A.2 Iptables

As referred in Section 4.3.3.3, it is possible to deploy the management cluster inside a kind cluster, which runs inside one or more Docker containers. For this reason it is necessary to configure the iptables of the host machine in order to allow the traffic to reach the Docker container on which the BIND server exposes its NodePort. The set of rules can be used to achieve this:

```
sudo iptables -t nat -A PREROUTING -p tcp -m tcp --dport $NODEPORT -j
↪  DNAT --to-destination ${CONTAINER_IP}:${NODEPORT}
sudo iptables -I FORWARD -d $CONTAINER_IP -p tcp -m tcp --dport
↪  $NODEPORT -j ACCEPT

sudo iptables -t nat -A PREROUTING -p udp -m udp --dport $NODEPORT -j
↪  DNAT --to-destination ${CONTAINER_IP}:${NODEPORT}
sudo iptables -I FORWARD -d $CONTAINER_IP -p udp -m udp --dport
↪  $NODEPORT -j ACCEPT
```

It is necessary to define rules for both TCP and UDP traffic, since BIND uses both protocols to handle DNS queries, as by protocol requirements.

And if the BIND server runs on the control plane node as default in `metal3-dev-env`, to obtain `${CONTAINER_IP}` it is sufficient to issue this command:

```
docker inspect -f '{{range .NetworkSettings.Networks}}
↪  {{.IPAddress}}{{end}}' kind-control-plane
```