ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

Master of Engineering in Computer Engineering

Master Thesis

Resource management of HPC infrastructures based on Kubernetes

Advisor
prof. Paolo Bellavista

Candidate
Michele Tagliani
0001140069

Company tutors
IBM Ireland
Michele Gazzetti
Christian Pinto

December 2025

To Serena

Studia che ti troverai contento
Nonna

Contents

Introduction

Within large research laboratories, such as the IBM Research Lab, infrastructure administrators are required to manage various types of infrastructure. With the rise in popularity of AI applications, users now request different type of computing resources depending on the type of workload that they want to execute. High Performance Computing (HPC) infrastructure, accessible via workload management platforms like Slurm, tends to be preferred for model training and classic computational applications (e.g., physics simulations). While cloud environments, with Kubernetes as the platform of choice, remain the preferred target to host data processing tasks, inference services or databases. Due to differences in refresh cycles and constraints set by the need to maintain legacy systems, cloud and HPC clusters tend to run on separated infrastructure. Each one with a dedicated cluster management services controlling the underlining computing resources. This approach increases the toil on administrators and hinders the reassignment of nodes between the different clusters. This is especially relevant for on-premises datacentre servers, where machines become part of an HPC or Kubernetes cluster and they seldom move to a different type of cluster. For example, an admin re-provisioning a node from Kubernetes to Slurm, and vice versa, would have to manually move ownership of the resource from one management service to the other. A lengthy procedure especially considering that many management tools inspect the machine before considering it available, increasing even further the toil on the administrator. An emerging trend is to adopt components from both cloud and HPC stacks to develop an hybrid solution capable of supporting both HPC and Kubernetes workloads on the same infrastructure.

An emerging approach is to converge at the workload management level running the HPC workload scheduler as an application inside the Kubernetes cluster [11], [45]. This approach provides high level of flexibility thanks to the easy to use Kubernetes APIs that simplify tasks like templating configurations and workload placement. However, one drawback of co-locating Kubernetes and HPC workloads on the same platform is that host computing resources are shared between workload managers, creating the potential for resource contention, a challenge that is generally mitigated with ad-hoc affinity policies ensuring that HPC and Kubernetes workloads do not get scheduled on the same host. Another approach is to converge at the node management level by defining a cluster life-cycle management layer capable of provisioning and dynamically scale separated Kubernetes and HPC clusters. This last approach ensures dedicated access to hosts and it simplifies the movement of nodes between clusters thanks to a global view of all the infrastructure resources and a unique cluster management API. This allows administrators to easily react to changes in user requirements.

In this thesis, we explore the adoption of Kubernetes management tools to achieve convergence of HPC and cloud at the node management level. We also present an extension to the Kubernetes cluster life-cycle management tool Cluster API [47] to automate the provisioning, bootstrapping,

and graceful de-provisioning of HPC batch scheduling platforms. For the HPC side, Slurm [46] was selected as target workload manager for this work. With this approach administrator can benefit from a single interface that simplifies the movement of nodes between Kubernetes and HPC clusters while keeping them isolated to avoid resource contention.

Chapter 1

Background on HPC and Cloud Computing infrastructure management

To better understand the problem and the proposed solution, it is important for me to present to the reader the HPC and the cloud computing history, motivations, and state of the art that are at the base of the work.

1.1 High Performance Computing

1.1.1 What is a supercomputer?

Computers have become fundamental in everyday life: even in science, for example, they allow us to model, study, and predict the events that surround us. Furthermore, they have enabled the creation of tools capable of generating volumes of data that were once impossible to manage or interpret. In the early days of the computing age, during World War II, computers were specialized machines, financed mainly by public bodies, designed to solve problems of strategic interest. These machines were mainly created to tackle numerical problems such as solving equations concerning the motion of projectiles and modeling nuclear fission processes. However, there were also non-numerical applications: it is therefore worth mentioning the work carried out at Bletchley Park by Alan Turing and his colleagues, who were able to decipher the codes used by the Nazis using the "Colossus" computer. This type of machine was capable of operating at speeds of between one and one hundred arithmetic operations per second. Once the war was over, the importance of computers for the study of complex scientific and mathematical problems was realized; thanks to them, it was finally possible to solve increasingly large systems of equations and develop simulations of phenomena for which, until then, only a few heuristics could be obtained.

As has always been the case in all fields of science, having tools capable of explaining phenomena whose solution was still unknown only served to generate new questions and new problems for which, very often, innovative techniques and physically larger machines were necessary. This was because, at that time, machines were made up of thermodynamic valves which, in addition to being extremely fragile, were also particularly bulky. This could lead to computers occupying entire

buildings. It was only in the late 1950s that some companies, including IBM, began to develop computers based on transistor technology, which would later become the standard technology and open the door to the era of microcontrollers.

In 1964, Control Data Corporation introduced the CDC600, the first *supercomputer*. The term supercomputer is actually a journalistic term that has been used since the early 20th century to refer to a machine capable of performing mathematical calculations faster than a human being. However, in this case, the term was borrowed by CDC to promote their machine, which at the time was the fastest computer in the world, being three times faster than the previous record holder, the IBM730 [18].

The world of automated computing is evolving at an enormous speed, and these records must always be contextualized in relation to their era: just consider that the number of operations that a smartphone can perform today represents an incredible performance when compared to that of supercomputers developed in the 1990s.

It is important to highlight that a machine's performance also depends on the characteristics of the problem to be solved: for this reason, standard problems, called *banchmark*, have been defined and are used to compare different computers.

In order to keep track of the 500 machines (not distributed) around the world, an official list was created in 1993. This list, called Top 500 [53], provides an overview of the characteristics of the various machines and allows us to understand which solutions are adopted to streamline and speed up computation. As mentioned above, a benchmark is used to compare these machines; in particular, since 1979, the "Linpack" benchmark has been used, a program that measures the speed of execution of an LU factorization. There are some criticisms of this problem, as it does not take into account some important aspects of modern applications, such as calculations performed on sparse matrices.

Given the high cost of building and operating these machines, it is reasonable to ask what they are used for today. For example, with the exponential growth of the artificial intelligence market, the computing power provided by these machines has been essential for training certain models. The improvement of these models has brought multiple benefits to the HPC world, attracting greater investment, which will be discussed in the next section. In addition to this, HPC systems are mainly used for research and development in many fields. For example, while working at IBM, I observed how the company's HPC systems were used in chemical research for the development of new materials and in physics for the analysis of atmospheric and space phenomena.

1.1.2 Systems architecture

The speed of supercomputers derives from two factors: high-performance hardware and parallel architectures. Since the early 1960s, hardware performance has followed an empirical law, formulated by Gordon Moore, which states that the number of transistors on a microchip, and therefore its computational power, would double every eighteen months. However, the current size of transistors is reaching a physical limit, i.e., the size of atoms [4]. Approaching this limit has therefore made the development of new transistors difficult and economically disadvantageous. It is therefore thanks to parallel architectures that the evolution of processing systems has been able to continue. In the 1980s, parallelism began to be exploited through the introduction of vector architectures and shared memory multiprocessing. In the first case, architectures are designed to run in *pipelines* in order to perform a single floating point operation on an entire series of data rather than on a single piece of data. In the second case, a small number of processors process

different data in parallel by accessing a common memory space. However, this memory sharing represented the biggest bottleneck: to increase speed, vector processors must be able to access memory quickly, but as the number of processors increases, so do conflicts. To address this problem, machines with distributed memory were developed, meaning that each processor began to have a dedicated portion of memory in which, by design, there could be no conflicts with other processors. The challenge in designing this type of machine lies in the communication protocols that must be implemented to allow the various processing units to communicate with each other: here, information is no longer shared via common memory, but through the exchange of messages. This trend has continued to the present day, where modern supercomputers are made up of independent processing units, known as nodes, which exchange messages using libraries that implement the Message Passing Interface (MPI). These libraries allow the use of standard functions that make HPC infrastructure programs portable and scalable, i.e., able to effectively exploit clusters independently of the number of nodes. Each node can be made up of either specific hardware or off-the-shelf components (i.e., components not designed for high-performance machines and which can normally be found on the shelves of any store): these components increasingly include accelerators capable of offloading certain types of operations from the CPUs, thereby speeding them up. These accelerators are very effective [22] and allow certain nodes, or even entire clusters, to be specialized to accelerate operations that would otherwise be very costly on general-purpose systems. Two examples of these accelerators are Graphics Processing Units (GPUs) and Tensor Processing Units (TPUs). The former were introduced to efficiently perform tasks related to graphics rendering, thus lightening the workload of the CPU. Thanks to the introduction of programmable shaders and their highly parallelizable architecture, GPUs began to be used to accelerate data-parallel computations not strictly related to graphics. This practice, called General-Purpose computing on Graphics Processing Units, began to grow in popularity after the release of CUDA, a toolkit produced by NVIDIA, which is used to program GPUs of the same brand [36]. With this hardware boost, it became possible to develop artificial intelligence models better and faster, which led to new investments in the sector, thus creating a sort of virtuous circle in which manufacturers of this type of accelerator began to optimize hardware for this type of computation as well.

In 2017, Google unveiled the first TPU, the first accelerator designed specifically to perform linear algebra operations (mainly matrix multiplications performed in batches), which represent the main workload in both training and inference phases in the field of artificial intelligence. Unlike GPUs, this type of accelerator is used exclusively in data centers and, due to its cost and limited availability, it is unlikely that all nodes in a data center will be equipped with this type of accelerator [22]. This has led to modern computing centers being made up of machines with different components and characteristics [3].

To illustrate how the technologies presented are installed on the most advanced systems, I have chosen to refer to the most powerful supercomputer currently ranked in the TOP500: El Capitan. This supercomputer consists of 11,136 nodes, each of them equipped with 4 MI300A cards. These cards are specifically designed for supercomputing and each includes, among other components, a combination of CPU and GPU, 128 GB of memory, and a communication interface called *Infinity Fabric*, which is necessary to enable communication between the components of the node [55].

Having illustrated how the machines that make up HPC systems are structured at the hardware level, it is important to analyze how they are organized at the logical level. The nodes not only have different hardware characteristics but also different roles. It is impossible to imagine that a system with thousands of nodes could use them all solely for computing; in fact, other operations

are also necessary within the cluster, such as storage, connection, and user management. Below are some types of nodes that can be found within an HPC system.

- Computational nodes: these are the main nodes within HPC systems, both in terms of number and importance; they are responsible for performing the *jobs* assigned to them by the scheduler and managing local resources such as memory and accelerators.
- Head nodes: these nodes are responsible for controlling the cluster or part of it; they communicate with other nodes to monitor the status of the cluster and run management programmes to coordinate the work of the worker nodes. These nodes are also used by administrators to respond to any failures or other issues. For these reasons, they almost never perform heavy workloads, so that they can always be responsive.
- Login nodes: these are the only nodes accessible to users of the HPC system. Users can connect to these nodes to compile their programmes, interact with the scheduler to submit their workloads, and download the results. As these nodes interface with the outside world, administrators must make them both easy for users to use and secure against attacks. It is important to note that, in some systems, the head nodes and login nodes are the same: this configuration is adopted to save resources that can be allocated to computational nodes.
- Storage nodes: these are nodes that only deal with managing the information exchanged between other nodes. Managing information flows is a fundamental aspect of HPC systems, as it very often constitutes a bottleneck. For this reason, some cluster nodes are reserved for storage management.

1.1.3 Cluster management

Managing a cluster is a complex process involving various activities aimed at supporting user needs and ensuring the proper functioning, efficiency, security, and reliability of the infrastructure. To access the computing power provided by these machines, users typically connect via the internet and request to submit a job, specifying the resources needed to execute it. In this context, the development of Internet networks has played and continues to play a key role, as it has allowed researchers and developers to exploit available computational resources regardless of their geographical location. These resources, available in large quantities but still limited, are divided among the various jobs and users, based on several factors, through the use of schedulers and load balancers. Requests typically made by HPC system users when submitting their jobs mainly concern the quantity or type of resources. Other cluster management activities carried out by infrastructure administrators include planning and installing software and hardware, configuring system resources, administering accounts, managing data security and privacy, monitoring system performance, and troubleshooting technical issues. In addition, it is necessary to collaborate with users and stakeholders to ensure that the system meets their needs and expectations.

The various infrastructures can be managed in very different ways and, as in the cloud environment discussed in the next section, administrators can also provide users with different levels of abstraction in the HPC environment. For the purposes of this thesis, only *bare metal* management will be explored in depth. The bare metal level represents an environment in which the user has complete control over the physical infrastructure, without filters imposed by virtualization systems or hypervisors. In fact, the end user of the solution presented in this work is a system administrator, who operates at the lowest level of abstraction available. The other users of the HPC system

are not technicians, but they must be able to exploit the infrastructure for their work. However, their view of the system must be limited, and they will have little, if any, configuration leeway. At this level, administrators must maintain updated numerous low-level HPC software components, such as drivers, operating systems, and parallel file systems, as well as tools used by users to develop programs, such as specific libraries or integrated development systems. To manage all this, administrators use both third-party tools, which can be either proprietary or open-source, and tools developed specifically for their infrastructure. In the latter case, administrators can leverage their specific knowledge of their infrastructure to create solutions that perfectly reflect their needs, but these tools must be maintained, thus contributing to the workload they face [2]. Therefore, the use of open-source solutions represents the right compromise between the adoption of flexible and maintainable tools: the most emblematic example of this phenomenon is the fact that since 2017, all supercomputers ranked in the Top500 use distributions based on the Linux kernel as their operating system [54].

1.2 Cloud Computing

According to ISO, the International Organization for Standardization, cloud computing is a paradigm for enabling network access to a scalable and elastic pool of shareable physical or virtual resources with self-service provisioning and administration on demand [1].

While the previous definition dates back to 2023, the idea of being able to access a computer system in a shared manner is much older, dating back to 1963, when DARPA (the Defense Advanced Research Projects Agency) began working with MIT (Massachusetts Institute of Technology) on building a system that would be general-purpose and time-sharing. These characteristics would allow this system to be used concurrently by multiple users, a situation described by the term "virtualization." Research in virtualization, operating systems, storage, and networking advanced in the next two decades with a speed never known before. For example In the early 80s, network operating systems were launched to allow computers to talk to each other, and by the end of the decade, around one hundred thousand computers were connected to the Internet [56].

Today the term virtualization indicates software that mimics the functions of physical hardware to run multiple instances (used by a multitude of users) on the same physical machine. This concept comes from the late 90s when the cloud became popular as companies started to use it to deliver software programs to the end users through the internet. The World Wide Web has helped in the spread of the foundational technologies of the cloud, leading to the *dot-com* revolution. The early architectures for distributed systems began to emerge; for example, the client-server model, which allowed, and still allows, connecting to a server to request resources or the remote execution of processes. It was precisely during these years that the term *Cloud* first appeared, defined as "the on-demand availability of computer system resources, especially data storage (cloud storage) and computing power, without direct active management by the user."

Since Amazon Web Services launched its services in 2002, more and more companies have started to provide a variety of cloud-based services. The resources were concentrated in large facilities to take advantage of large-scale economy, providing a cost-effective dream solution to the technical and management problems that many small and medium-scale businesses and organizations suffered. The cloud alleviated the burden of maintaining servers, upfront investment in compute resources, and scaling web services based on demand because now you can rent the resources that you need when you need them. This period is known as the *first generation cloud*

and represents the implementation of the original idea - one or more data centers where computational power and storage resources are concentrated. The primary use case for the client companies was to host a portion of an application, organized according to a two-tiered architecture, on the cloud, while end-users made requests through the web. Already at this stage, it is possible to see how the adoption of cloud solutions attempts to alleviate the burden of infrastructure lifecycle management from the shoulders of system administrators.

Using third-party resources for sensitive data or mission-critical operations was not convincing to the market; there was a need to process the most sensitive or urgent information on-premises using local resources while delegating aspects such as storage and application deployment to the cloud, which needed to be scalable and permanently available. To address these needs and accelerate development phases, cloud providers began offering services with an increasing level of abstraction. We transitioned from the Infrastructure-as-a-Service (IaaS) model, where customers rented physical or virtual machines and were responsible for their configuration, to the Software-as-a-Service (SaaS) model, where customers can access ready-to-use software hosted in the cloud at any time. This transition also included the Platform-as-a-Service (PaaS) model, where customers access a properly configured platform to develop or manage specific types of applications.

In this phase, known as the *second-generation cloud*, two main technologies have emerged: hybrid cloud and containerization systems. The hybrid cloud utilizes computational resources, storage, and services from a variety of public and private providers. This approach enables the management of workloads across different environments based on various needs, such as privacy, latency, complexity, and availability. Additionally, an increasing number of intermediate solutions are being developed to move data processing closer to the source, utilizing network structures like routers. This architectural paradigm is commonly known as *edge cloud* or *fog cloud*. The fundamental idea behind these approaches is to relocate computational processes as close as possible to the physical location where data is generated and where it will be consumed. By minimizing the distance between data sources and processing units, edge and fog cloud solutions are able to significantly reduce latency, thereby enabling faster response times for time-sensitive applications. Furthermore, this proximity enhances privacy and security, as sensitive information can be processed locally rather than being transmitted to distant data centers. However, it is important highlight that these benefits often come at the expense of computational power, since edge devices and intermediate network nodes typically possess less processing capability compared to centralized cloud infrastructures. As a result, there is a trade-off between achieving lower latency and improved privacy versus the potential limitations in available computational resources.

As previously mentioned, the second technology developed in this phase is related to containerization systems. Its history stems from the need to separate the execution of different users or applications that concurrently utilize the same machine; this necessity is not a modern issue, but has been present since the late 1970s. During that period, multiple users accessed the same mainframe simultaneously, often interfering with one another. To address this problem, the system call `chroot` was introduced within Unix, capable of relocating the root of a process and its children to a different point in the filesystem [40]. Over the years, a multitude of solutions has been developed to increasingly isolate the processes related to different users or applications. For instance, in 2000, the command `jail` was introduced in FreeBSD, one of the main distributions of the free Unix-like operating system. This command is very similar to the previously mentioned chroot but includes additional mechanisms to isolate the filesystem, users, and networking, thus allowing for a highly customizable configuration of each jail. A few years later, in 2004, `cgroups` were introduced within the Linux kernel, functionalities capable of managing and limiting access

to resources, which enabled the implementation of Linux Containers (commonly abbreviated as LXC). These containers allow for operating system-level virtualization, thereby creating isolated environments with their own processes and address spaces, while sharing the same Linux kernel, resulting in a significantly lighter resource footprint compared to virtual machines [29].

Docker is built upon the LXC technology, and it wants to address the challenges of application distribution in production systems. Docker allows for the inclusion of both the application and its dependencies within a container in a standardized manner. The standardization achieved through collaboration between the Open Container Initiative (OCI) and the Cloud Native Computing Foundation (CNCF) has facilitated the creation of a set of protocols that companies have trusted and upon which they have begun to build their products. However, Docker has a limitation: it can only manage containers on a single node, a situation that is no longer acceptable as applications have begun to consist of several dozen of them. To address this issue, Kubernetes was developed and, for over a decade, it has represented the standard tool for managing containers, contributing to the emergence and proliferation of cloud-native architectures such as microservices. Kubernetes has accelerated the development of applications and optimized their management through automation and observability. Its rapid adoption is a result of the momentum gained from major market players, primarily Google, which created and open-sourced the project, adapting it to be portable across most available cloud platforms. This adaptability has made Kubernetes a cornerstone of modern cloud infrastructure, enabling organizations to deploy, scale, and manage applications efficiently [50]. Furthermore, Kubernetes supports a wide range of deployment strategies, which allow for safer and more controlled application updates. These features are crucial in today's fast-paced development environments, where *continuous integration* and *continuous deployment* (CI/CD) practices are becoming the norm.

The evolution of containerization technologies, from the early days of Unix to the sophisticated orchestration capabilities of Kubernetes, reflects the ongoing need for efficient resource management and application deployment in increasingly complex computing environments. As organizations continue to embrace cloud-native architectures, the role of containerization and orchestration tools will only grow in significance, shaping the future of software development and deployment.

1.3 Motivations and objectives

Administrators of large-scale research centers manage different types of infrastructure deployments to respond to the diverse requirements coming from the scientists and developers that they support. With the rise in popularity of AI applications, users now request different types of computing resources depending on the characteristics of the workload they want to execute. High Performance Computing infrastructures, accessible via workload management platforms like Slurm (discused in the 2.2 section), tend to be preferred for model training and classic computational applications (e.g., physics simulations), while cloud environments, with Kubernetes as the most popular management framework, remain the preferred target to host big data processing tasks, inference services, or databases.

Because of differences in refresh cycles and due to constraints deriving by the need to maintain legacy systems, cloud and HPC clusters tend to run on separated infrastructures, each one with dedicated cluster services for resource management. This approach increases the burden on

administrators and hinders the reassignment of nodes between the different clusters. This is especially relevant for on-premises datacentre, where machines are often statically assigned to either a HPC or a Kubernetes cluster, and seldom move to a different cluster type. For example, an administrator re-provisioning a node from a Kubernetes to a Slurm cluster, or vice versa, has to manually move ownership of the resource from one management service to the other; this leads to a engthy, manual, and expensive procedure, in particular, by considering that many associated management tools inspect nodes before considering them available (increasing even further the toil on administrators).

For those motivations, an emerging trend is to adopt components from both cloud and HPC stacks to develop a hybrid solution capable of supporting both HPC and Kubernetes workloads on the same infrastructure presents three possible architectural guidelines to the deployment of Kubernetes and HPC batch scheduling platforms. The most common nowadays, as already stated, is keeping systems siloed with dedicated nodes and separated cluster management tools. An emerging approach is to converge at the workload management level, running the HPC workload scheduler as an application inside the Kubernetes cluster. This approach is more flexible thanks to the easy-to-use Kubernetes APIs that simplify tasks like templating configurations and workload placement. However, one drawback of co-locating Kubernetes and HPC workloads on the same platform is that computing resources are shared between workload managers, thus creating the potential for resource contention (the challenge is generally mitigated with ad-hoc affinity policies that ensure that HPC and Kubernetes workloads do not get scheduled on the same host.) The third, most novel, and most promising approach is to support computing convergence at the node management level, by defining a proper cluster life-cycle management layer, capable of provisioning and dynamically scaling separated Kubernetes and HPC clusters. This direction of solution ensures dedicated access to hosts by design and simplifies the migration of nodes between clusters of different types thanks to a global view of all the infrastructure resources, typically by offering a unique cluster management API. In addition, this approach allows administrators to more easily and more dynamically react to changes in user requirements.

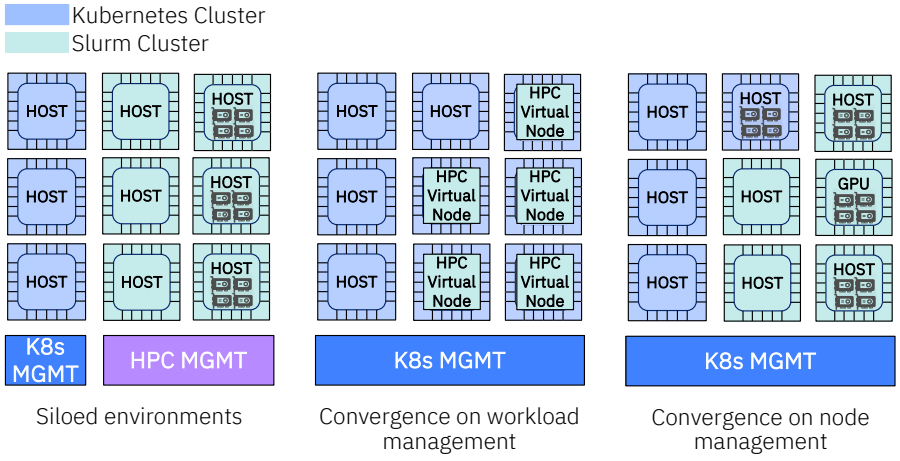

Figure 1.1: Approaches to Cloud/HPC convergence

This work explores how to efficiently extend Kubernetes management tools to achieve convergence of HPC and cloud at the most innovative node management level. I present 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 based on Slurm. As better detailed in the following sections, with this approach, administrators can benefit from a single interface that simplifies the migration of nodes between different types of clusters, while preventing resource contention issues.

Therefore, in short, the primary contributions of this thesis can be summarized as follows:

- A converged computing model that employs the Cluster API as the unified interface for the provisioning of on-prem clusters;
- An original bootstrap provider, called Cluster API Bootstrap Provider Virtual Kubelet (CABPV), which extends the standard Cluster API to manage HPC clusters;
- A Slurm Detach Handler, which enables the graceful cordoning and deletion of HPC nodes via an extended version of Cluster API.

1.3.1 Use case and requirements

In practical use cases, the lifecycle management of a computing infrastructure is a complex problem, in particular when users request different types of clusters and configurations to be supported, with possibly evolving requests and requirements. During my period at IBM, Michele Gazzetti and Christian Pinto have shared with me their experience of interaction with researchers and developers, in the domains of computational-based discovery of new materials and AI model development, just to mention a few, in which their users call for diverse sets of computing environments to carry out their research activities; this translates, for instance, to requests for Slurm clusters to host simulations or model training, and to requests for Kubernetes for model fine-tuning and inference.

The ability of on-demand hosting multiple system configurations in the same infrastructure opens new opportunities and enables global sharing of resources across clusters. One possible example relates to high-value GPU-based accelerators, essential for tasks such as AI model training, fine-tuning, and inference, but usually available in limited quantity in on-prem deployments. When cloud and HPC clusters are separated, unutilized accelerators in one cluster cannot be dynamically transferred to the other one due to separated control planes and networking. This leads to inefficient resource utilization and limits the possibility of scaling up applications with limited costs. A driving idea behind this approach is to be able, in a converged infrastructure, to dynamically and automatically migrate a node equipped with accelerators between different types of clusters, depending on workloads' demand.

1.4 Related Work

Before exploring the technologies that were used and the process that led to the development of the solution, I would like to briefly present the solutions already available on the market, highlighting the main differences we want to achieve.

1.4.1 Cluster management tools

xCat

xCat [9] is a comprehensive tool for infrastructure management that can control various types of clusters. It enables the provisioning of Operating Systems and configurations for both virtual and physical machines, which means it could potentially have helped us in achieving some of our objectives. While xCat is widely utilized in many high-performance computing environments, its adoption in cloud settings is less common due to its lack of native support for cloud technologies such as Kubernetes. By addressing this gap, our solution will enhance usability and streamline resource management across both environments, facilitating more effective operations for administrators.

Base Command Manager

Base Command Manager (commonly referred to as BCM) [35] includes tools and applications designed to facilitate the installation, administration, and monitoring of a cluster. Additionally, BCM aims to provide users with an optimal environment for developing and running applications that require substantial computational resources. Among its various features, it also supplies the necessary packages for setting up a Kubernetes cluster. Developed by NVIDIA, Base Command Manager is not an open-source solution and is compatible only with their infrastructure, which poses a significant risk of vendor lock-in. In the solution that will be presented, we want to ensure a complete agnosticism from an infrastructure perspective.

1.4.2 Workload convergence projects

Slinky

The Slinky project [45] aims to integrate Slurm with Kubernetes, leveraging its API and abstractions to manage both technologies effectively. While this high-level integration offers a streamlined approach, it also presents a potential drawback: different types of Pods may be scheduled on the same machine, leading to resource contention. Our approach emphasizes maintaining a clear separation at the machine level, thereby mitigating the risk of contention by ensuring that resources are allocated distinctly for both Slurm and Kubernetes environments. This strategy allows for more efficient workload management while preserving system integrity.

Supernetes

The Supernetes project [12] aims to make computing resources and nodes accessible to other Kubernetes tools by abstracting the specifics of HPC. Similar to Supernetes, our approach will expose the HPC nodes view in the Kubernetes cluster; however, we also want to empower system administrators with the ability to provision and de-provision resources as needed. While Supernetes primarily functions as a bridge enabling access to HPC resources from Kubernetes, our approach focuses on providing system administrators with more direct management capabilities.

Chapter 2

Adopted Technologies

In this section I want to introduce the software stack that and technologies that I have adopted.

2.1 Kubernetess

In this thesis, the main structure and functioning of Kubernetes are presented as known to the reader, as an in-depth discussion of the topic would fall outside the set objective. Therefore, only the strictly necessary concepts will be discussed.

Kubernetes [27] is a powerful open-source system that is designed to manage and orchestrate cloud infrastructure and the containerized applications running on it. It reduces the complexity of coordinating many physical and virtual machines, networks, and environments so that developers can focus on building and deploying their applications. Kubernetes also allows system administrators to focus on application delivery, availability, and scalability instead of managing individual servers and networks. Many applications can run simultaneously on a cloud that is managed with Kubernetes, and each application will only use the memory, CPU, and network resources that Kubernetes allocates to them [21]. There are two types of machines in a Kubernetes cluster: masters and nodes. The former are the cluster's brains, providing an API that clients and users can use, and they are in charge of the majority of the logic that Kubernetes provides. These API server implements a RESTful interface, which means that many different tools and libraries can readily communicate with it. The way the user communicates with Kubernetes by default is through the command-line tool *kubectl*. The others machines receive work instructions from the master and react accordingly. To create or manage any resource, the user must provide a YAML or JSON file that contains a declarative set of instructions. One of the reasons that led us to use Kubernetes within the project is its great flexibility, which will be discussed in section 2.1.1 and will then be employed to enable Kubernetes to manage the lifecycle of HPC nodes in chapter 3. It is important to emphasize that Kubernetes is already widely known and used within the administrator community, eliminating the need to learn a new technology, which could facilitate the adoption of our solution.

2.1.1 Kubernetes Objects and Services

Kubelet

On each node exists a small service called Kubelet whose scope is to exchange information with the rest of the cluster. It usually communicates with the master node to authenticate to the cluster and receive commands and work. The Kubelet performs other essential tasks, including managing container lifecycle (starting, stopping, and restarting), check if they are running without errors and ensuring that they're in the desired state. One relevant aspect for the aim of this thesis is that a Kublet also checks the node's status reporting any errors.

Custom Resource Definitions

One of the biggest strengths of Kubernetes is its ability to remain agnostic to the underlying technology that it is managing. In fact, Kubernetes has been designed with an extensible API to be able to continuously change and grow. The main way to introduce a new type of resource is via the Custom Resource Definitions (CRD). A CRD is a Kubernetes resource itself that declaratively defines a new custom API for the cluster without the need to create or run a custom API server. Like any other resource in Kubernetes it is possible to define some controllers in order to properly react to CRD state changes. These custom resources behave like native Kubernetes resources but are specifically designed for your application's requirements; for this reason, they are used in almost every Kubernetes subject (like Cluster API, which will be discussed in a future section).

Replica Sets

Most of the time Kubernetes has to manage a pool of identical Pods that run some application. These Pods are managed by a controller called Replica Set that is able to horizontally scale them using a pod template in order to distribute load or maintain the desired state. A ReplicaSet is linked to its Pods via a field, which specifies what resource the current object is owned by. All Pods acquired by a ReplicaSet have their owning ReplicaSet's identifying information within their ownerReferences field. It's through this link that the ReplicaSet knows the state of the Pods it is maintaining and plans accordingly.

Deployments

Instead of working directly with Replica Sets, it is recommended to use Deployments. They are objects that use Replica Sets as building blocks, adding flexible life cycle management functionality. Kubernetes offers several deployment strategies to handle a broad range of application development and deployment needs. Once you define the desired state of the application, the deployment controller starts managing the correspondent Replica set. A deployment, operating at a higher level of abstraction, allows for update operations such as rolling updates and rollbacks, unlike a Replica Set, which works at a lower level, focusing solely on managing the health status of the pods and the number of available replicas.

ConfigMaps and Secrets

It is a good practice to externalize configurations from the various resources used by Kubernetes. Moving the retrieval of some configuration information to runtime allows for deployments in multiple contexts with greater flexibility. This approach not only enhances code organization but also

enables modifications to configurations without recompiling or redistributing the entire application. To manage parameters at runtime, Kubernetes provides two different resources: ConfigMaps and Secrets. ConfigMaps are files that contain data in the form of key-value pairs and can be presented to programs running in containers as environment variables or mounted as configuration files. If the program is capable of reading these configurations, they can be injected at runtime without the need to create a different build for each environment in which the container may be deployed. Secrets are similar to ConfigMaps; in fact, they can be read by applications in the same way, but the information contained within them is considered more sensitive and, therefore, their content is encoded in Base64. However, it remains the responsibility of the programmer or system administrator to handle encryption and access policies for the information contained in this type of file. Implementing proper security practices is essential to protect critical information and maintain a secure working environment.

2.2 Slurm

Slurm [46] is a highly scalable, open-source, fault-tolerant workload manager. Thanks to its features, it is widely used in the HPC community, in fact it is used in 65% of the supercomputers in the Top500. Slurm has three key features: the first allows shared or exclusive access to cluster resources to be allocated for a certain amount of time, the second is a framework for managing the work performed on the allocated resources, allowing the user to create, execute and monitor jobs, while the last feature allows the scheduler to manage, through the creation of queues and the possibility of defining access policies, situations in which some resources are contested between multiple jobs. Slurm currently boasts more than two decades of development, thanks to the work of a community of around 300 people who have contributed over time via the Github repository [49].

2.2.1 Slurm concepts

In this section, the goal is to define the concepts related to Slurm that will be used in the following sections.

Cluster

A cluster typically consists of a collection of nodes sharing a common network. A Slurm cluster can be on premises, in the cloud, or spread across both. In this last case the cluster is called hybrid.This thesis will only address the case of an on-premises bare metal cluster, because cloud providers that offer HPC resources do not usually allow bare metal management or a converged infrastructure approach. A set of Slurm clusters whose configuration depends on a common database is called a federation. Federations are usually designed to logically separate the resources that a category of users can use. In any case, all clusters belonging to a federation can be configured to work as if they were a single system.

Node

As already explained in section 1.1.2, a node is an independent processing unit that can take on different roles within a cluster. In Slurm, a machine takes on the role of a node when the *slurmd* daemon is run on it. This daemon performs various tasks (which will be discussed in

section 2.2.2): among these, it is important to highlight that it collects information about the machine on which it is running in order to share it with the rest of the cluster. This information includes: the number of CPUs, the amount of memory, and generic connected resources (e.g., accelerators, network interfaces). For each node, Slurm also maintains information unrelated to its physical characteristics, such as preference indexes (i.e., weights that favor the execution of certain workloads on certain nodes), node status (up, down, draining), and when and which user last changed the node status. It is important to note that generic resources are managed separately by Slurm, so that they can be allocated to various jobs regardless of the node on which they are located, billed separately, and have specific limits. Given the large number of nodes that can be present within an HPC system, Slurm allows nodes to be grouped by characteristics, and this division is called a *NodeSet*. NodeSets can be defined either explicitly by the administrator or managed automatically by Slurm. This type of division is particularly useful for applying identical configurations to multiple nodes simultaneously.

Job

A *job* in Slurm is a request to obtain a certain resource. It can take the form of a batch script, i.e., a simple text file containing commands interpreted by the command line interpreter, executed when the requested resources are allocated, or an interactive job for which the user waits to be granted access to the resources before using them in real time. A job includes multiple types of information, including: an ID, a name, time limits, the type and quantity of resources requested (number of nodes, processors, accelerators, etc.), and the names of specific nodes that the user wants to be included or excluded from the request. By setting these parameters, users can define both the quality and quantity of resources available to them, as well as the time they will have to wait for those resources to be reserved. It is up to users to evaluate this trade-off, while it is up to the system administrator to verify that users do not reserve, intentionally or by mistake, a disproportionate number of resources. An excessive request can lead to underutilization of resources and thus cause the work being done by other users to be blocked. Another parameter that is widely used when defining jobs is precedence constraints; in fact, in certain programs it is very important to ensure that the execution of a job is constrained to the end of the execution of a previous job. Finally, it is possible to define a series of behaviors based on the result of the execution of the predecessor job.

There are two special types of jobs: *array* jobs and *heterogeneous* jobs. The former allow the same job to be run multiple times, so they are usually used to run the same workload with different inputs; this allows the user to monitor and manage all jobs as if they were a single entity and also allows Slurm to optimize their management. The latter, on the other hand, allow the user to request certain resources only for a portion of the workload. If the user knows that only a portion of their job requires a greater number of resources, it is inefficient to reserve access to them for the entire execution time. With a heterogeneous job, it is possible to separate the job into several phases, specifying which resources are needed for each phase.

Job Step

A parallel task in Slurm is called a job step. In HPC, parallel tasks are usually part of a program that uses MPI to coordinate work between multiple nodes. A job can launch an arbitrary number of parallel tasks, but it is up to Slurm to organise these tasks so that they can be executed with the resources available to the job. If the required resources are not currently available within those

allocated for the job, the job step will be queued. As with jobs, job steps also have a series of parameters that can be set by the user; for example, Slurm allows certain nodes to be included or excluded from execution. It is important to note that it is not possible to define precedence constraints between job steps at runtime, but if these constraints are needed, it is up to the programmer to implement them through scripts to control the workflow.

At this point, it should be noted that Slurm allows detailed information on the performance of each job step to be collected; this collection can be continuous or periodic. Since this operation stores much more information than is normally stored in the Slurm database for each job, and since it adds overhead to the computation, the user must explicitly request that this data be collected.

Partition

Slurm often uses queues to manage resource allocation requests. These queues, which contain jobs, can be assigned names, making them partitions. These partitions are used to manage access control to resources by setting limits. Each partition has a set of nodes associated with it, and each node can be in multiple partitions at the same time. There are also 'float' partitions in which the administrator only defines the number of nodes contained in it, but does not specify which ones they are. These partitions are particularly useful as they allow unused nodes within a partition to be reassigned to other partitions that need them to support the required workloads. If the physical resources assigned to a partition are not sufficient to meet demand, resources borrowed from other partitions are distributed based on certain parameters (partition scheduling priority, job scheduling, etc.) also set by the administrator.

Partitions are also widely used for resource billing: costs can be assigned to the various partitions so that a price can be calculated for executing a type of workload. Some examples may involve partitions that contain nodes with a particular hardware accelerator, which will therefore cost more than generic nodes, or queues with higher priority that guarantee priority execution of a workload. Considering that a partition can contain very heterogeneous resources (a partition could even include the entire cluster), it is still essential that the user is able to correctly select the resources they need for a given job.

2.2.2 Node types and deamons

As described in section 1.1.2, there are different types of nodes in an HPC cluster. Slurm specialises each node by running the daemon related to the task it has to perform on it. Within a Slurm cluster, there must be at least one head node, which in Slurm is called a *controller node*, necessary for cluster management, and one computational node. Other types of nodes may be present in the cluster in order to enable certain features, but they are not strictly essential for functioning. Slurm developers recommend having physically separate nodes for separate roles, especially in production environments [32] [46].

Controller node

This is the machine on which the *slurmctld* daemon is running. This daemon runs as an unprivileged user called *SlurmUser* and orchestrates all activities performed by Slurm, manages partitions, monitors node status and allocates resources to jobs. It also handles almost all user requests, with the exception of some accounting-related requests that are handled by the *DBD*

node, a type of node that will be presented later. The controller node is unique within the cluster, but to ensure its presence in case of errors or failures, it is standard practice to configure backup nodes that can take its place. If the head node were to return to operation, it would immediately resume control of the cluster; for this reason, every time the status of the cluster changes, it is saved to disk by the daemon, thus allowing activities to resume even in the event of shutdowns.

Computing node

A *computing node*, often referred to simply as a *node*, performs the computational work within the cluster. It runs the slurmd daemon, which is the only one within Slurm that runs with root user permissions. The ability to run as root is a necessary condition for performing workloads and managing user jobs. In addition to the hardware monitoring tasks mentioned in section 2.2.1, the slurmd daemon monitors the processes running on the node, launches new jobs, and kills existing ones based on the instructions it receives. Furthermore, whenever a job step needs to be executed, slurmd launches a special handler, *slurmstepd*, which terminates together with the assigned job step. This handler is responsible for managing the input, output, and signals that the job step must launch.

To simulate a cluster larger than the physical one, you can launch multiple instances of slurmd on the same node, and thus use a single machine to simulate the behaviour of a cluster with a multitude of nodes. However, this property is only useful for testing purposes and for studying cluster configurations; it is not optimal for use in production environments.

DBD node

A DBD node is a machine on which the *Slurmdbd* daemon runs, which is dedicated to managing a database that can be either MariaDB [28] or MySQL [34]. The DBD node is optional within the cluster, and it is possible to have one shared between multiple clusters within the same organisation. It maintains accounting information and centrally manages certain configurations, including quotas and licences. When a system administrator modifies these centralised configurations, they are transmitted to the other nodes in the Slurm cluster. If the slurmdb daemon or the database it manages becomes unavailable, the Slurm nodes will save the billing information locally and use the latest version of the configurations received in order to continue operating. Since slurmdbd is not essential for the cluster to function, the Slurm documentation recommends enabling this feature only after the administrator has verified that the cluster consisting of only computational and control nodes is functioning.

Login node

A login node is a machine that is used by various users to request resources, prepare their workloads, and obtain results. Each node can be configured to be both a login node and a computational node: having these types of nodes separated on different machines can have advantages in terms of security and performance; in fact, using separate machines for login nodes can reduce the surface area available for attacks and errors (situations that can still occur since the infrastructure is shared by multiple users). Furthermore, keeping these two types of nodes separate avoids resource contention issues. It is the responsibility of system administrators to ensure that login nodes can perform all operations that are granted to users [46].

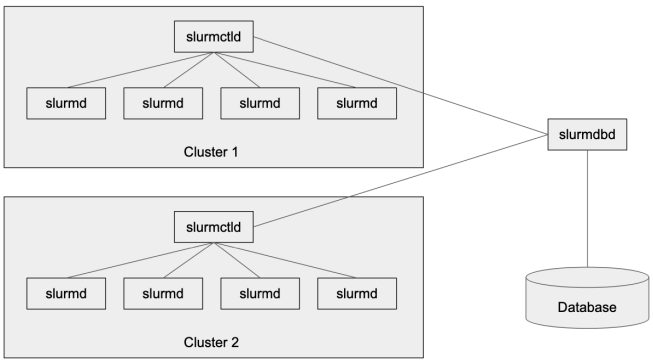

Figure 2.1: Classical Slurm cluster configuration

Restd node

The restd node is one of the latest additions to Slurm. Like the others, it is a type of node that runs a daemon, in this case called *restd*. This daemon manages a REST API for interacting with the Slurm cluster: it was introduced to enable the development of web services that would allow users to interact more easily with Slurm; to do this, the daemon translates requests, which are made in JSON or YAML format, into Slurm Remote Procedure Calls (RPCs) [46].

2.2.3 Configurations

For Slurm to function correctly, a configuration file called *slurm.conf* must be properly compiled or generated. This file must be distributed, identical, to every node in the cluster. The default location is the */ect/slurm* folder, but it is possible to change the location of this file, as long as it remains readable by all daemons. The minimum set of parameters that must be defined by the system administrator is shown in listing 2.1. Before installing Slurm, a user called *SlurmUser* must be created on each node. In addition, all directories containing log files, status save files, and process configuration files must be created. It is the administrator's responsibility to create these directories and make them writable and readable by the slurm user before starting the daemons.

Another directory that the administrator should pay attention to is the so-called *StateSave-Location*, as this is where the head node saves information about queues and jobs, whether they are running or have finished. It must be a directory that can be written to quickly, so that the status is always up to date in the event of errors or failures. If backup nodes are planned for the head node, the directory must be accessible to both; to achieve this, a shared NFS folder or a shared mount can be used. If a controller node were to activate without access to this folder, the cluster status would be lost and the jobs that were running would be deleted. The configuration file must also contain some information about the nodes that are part of the cluster; in order to avoid repeating information shared between nodes with the same characteristics, a particular syntax can be used to group them together. Each node can have up to three different names:

- NodeName: represents the name that will be used by Slurm tools to refer to that node.
- NodeAddr: is the name or IP address that Slurm uses to communicate with the node.

- NodeHostName: the name that is returned by running the command `/bin/hosrname -s` on the node

Other information that must be included in the configuration file concerns the hardware present on the nodes, such as: the number of processors, the available memory and how much disk space can be occupied. Any node that does not have this information in the configuration file is considered inactive (DOWN status) by Slurm and will not be used for job scheduling. There is also an alternative configuration for Slurm clusters that does not require the slurm.conf file to be shared on each node: this is called the *configless* configuration and requires the configuration file to be present only on the controller node, which, via the slurmctld daemon, distributes it to the other daemons that request it. Considering the large number of nodes that may be present within a cluster, Slurm allows nodes to be updated in a rolling manner: each daemon is able to manage up to three versions of Slurm simultaneously in order to ensure communication even while an update is taking place. Thanks to this feature, system administrators can gradually update the cluster [46] [32].

Other types of configuration, such as partition creation or permission management, must be performed by an administrator based on the nodes that make up the cluster. In the case of homogeneous clusters, where all nodes are the same, a single partition is created, limiting the number of nodes that a single user can reserve. If, on the other hand, the cluster is made up of heterogeneous nodes, it may be effective to create different partitions that group the various nodes according to common characteristics or recurring configurations. Other partitions are based on the speed with which the request can be fulfilled. Slurm allows you to submit a job to multiple partitions simultaneously so that it runs on the first available partition [46].

Slurm has introduced the possibility of offloading excess work to cloud resources (cloud bursting). Specifically, when the cluster experiences a spike in requests, it can leverage services such as Amazon Elastic Computing Cloud, Google Cloud Platform, or Microsoft Azure resources. Adopting these solutions increases the responsiveness and throughput of the cluster, but also significantly increases costs [32].

```
#
# Sample /etc/slurm.conf for mcr.llnl.gov
#
SlurmctldHost=mcri(12.34.56.78)
SlurmctldHost=mcrj(12.34.56.79)
#
AuthType=auth/munge
Epilog=/usr/local/slurm/etc/epilog
JobCompLoc=/var/tmp/jette/slurm.job.log
JobCompType=jobcomp/filetxt
PluginDir=/usr/local/slurm/lib/slurm
Prolog=/usr/local/slurm/etc/prolog
SchedulerType=sched/backfill
SelectType=select/linear
SlurmUser=slurm
SlurmctldPort=7002
SlurmctldTimeout=300
SlurmdPort=7003
SlurmdSpoolDir=/var/spool/slurmd.spool
SlurmdTimeout=300
StateSaveLocation=/var/spool/slurm.state
TreeWidth=16
#
```

```
# Node Configurations
#
NodeName=DEFAULT CPUs=2 RealMemory=2000 TmpDisk=64000 State=UNKNOWN
NodeName=mcr[0-1151] NodeAddr=emcr[0-1151]
#
# Partition Configurations
#
PartitionName=DEFAULT State=UP
PartitionName=pdebug Nodes=mcr[0-191] MaxTime=30 MaxNodes=32 Default=YES
PartitionName=pbatch Nodes=mcr[192-1151]
```

Listing 2.1: slurm.conf example

Plugins

To promote software extensibility and remain flexible in response to administrators' needs, Slurm allows plugins to be created and installed: there are more than a hundred of these, allowing support to be added for different types of hardware, software and scheduling algorithms. This system has allowed some features, which were initially provided as plugins, to be subsequently introduced into the Slurm implementation. When a job is submitted, the slurmctld daemon invokes the plugins that have been configured; each of them can either modify the arguments or modify the error messages that are returned to users. Plugins must be written in C and do not require access to the Slurm source code. To implement them, Slurm provides an interface called SPANK (Slurm Plug-in Architecture for Node and job (K)control). This flexibility is crucial for managing complex and constantly evolving infrastructures, where the need to support new technologies, such as advanced accelerators (e.g., GPUs and TPUs), requires a robust and standardised mechanism for integration.

Communications

Slurm uses a hierarchical, authenticated, and fault-tolerant mechanism for communication between computing nodes, with the added possibility of configuring fan-out. With this communication mechanism, it is possible to offload as much as possible from the slurmctld daemon which, as presented in the previous sections, manages numerous aspects of the cluster. Fan-out also allows to limit the time required to perform operations that require communication between a large number of nodes. The Slurm documentation recommends configuring fan-out so that the communication tree can reach every node in the cluster without exceeding five levels of depth. The communication hierarchy is not fixed and is created and destroyed as needed. Several hierarchies can also be created simultaneously, even overlapping ones.

To explain how fan-out works, let us assume that we have $\mathcal{N}$ nodes in the cluster and that a fan-out of $\mathcal{F}$ is set. If the control daemon needs to communicate with each node to execute a job, it takes a list containing the $\mathcal{N}$ nodes and divides it into $\mathcal{F}$ subgroups of equal size. The control node then launches $\mathcal{F}$ threads (one per group) which will communicate with a single slurmd, different for each group. These nodes are then informed of the action to be performed and to which nodes they must forward the request recursively. This process only ends when all nodes receive the communication. It is important to set the timeout parameter correctly so that any communication errors can be detected; in such cases, the daemon that originated the request selects another node in the group to send the command to. To avoid duplicate operations, each daemon maintains a list of communications it has already obtained and executed [32].

For security reasons, all communications between slurm daemons are authenticated. The administrator can select which slurm service or plugin to use by specifying the *AuthType* parameter in the configuration file. The default authentication method uses Munge, an authentication service designed to verify credentials in environments that require high speed and scalability [13]. These features make it ideal for use in HPC environments. Munge allows a process on any node in a group to authenticate the UID and GID of another process in the same group. In order to use Munge, a group of nodes must have users and groups in common and share an encryption key. Specifically, it is the system administrator's responsibility to create the key called *munge.key*, share it with all nodes in the cluster, and activate the *munded* daemon before starting the Slurm daemons [46].

2.3 Cluster API

Cluster API (CAPI) is a project created by the Kubernetes Cluster Lifecycle special interest group (SIG) to provide system administrators with a tool capable of managing Kubernetes clusters in a consistent, declarative and modular manner [47]. Cluster API uses CRDs and custom controllers to represent within Kubernetes a Kubernetes cluster and its infrastructure .

2.3.1 Goals

Although Kubernetes allows containers to be orchestrated independently of the environment in which they run, it does not provide any support for managing physical nodes independently of the infrastructure. This means that administrators have to manage the infrastructure in a customised manner based on the requirements that arise from time to time. The objectives that drove the Kubernetes Cluster Lifecycle SIG to create Cluster API are the following:

- Declarative cluster lifecycle management: Cluster API's declarative approach must be easy to integrate with GitOps, defined as a set of procedures that use Git repositories as the single source of truth for managing and provisioning infrastructure through code rather than manual operations [59].
- Infrastructure abstraction: Cluster API must allow infrastructure to be provisioned and maintained consistently across all environments, regardless of whether they are cloud, on-premises, or hybrid. Cluster management should not only cover computing and storage, but also other aspects of infrastructure such as networking and security.
- Integration with existing components: Cluster API must be created based on existing and widely used components, so as to avoid reinventing the wheel. in fact, the main components of CAPI are kubeadm [26] and Cloud Init [15]. Similarly, infrastructure management has been designed to provide the administrator with an approach similar to that used to manage Kubernetes workloads.
- Modular approach: Cluster API must be able to adapt to a wide variety of infrastructures and deployment environments, which is why its architecture must be modular and extensible. The aim is to implement a series of operations that can also be used on infrastructures that do not yet exist. By exploiting modularity and standardising a minimum set of operations that must be supported, administrators are able to use or develop alternative solutions as needed.

In summary, the goal of Cluster API is to replace the set of tools normally used by administrators to manage various aspects of infrastructure, providing an effective, centralised solution capable of functioning in a wide variety of environments [51].

2.3.2 Cluster API Concepts

To manage infrastructure and automate operations such as cluster creation, scaling, upgrading, and repair, Cluster API uses a set of components that create an abstraction layer capable of standardising the management of a large number of infrastructures. The project structure therefore consists of a core set of components, which remain unchanged regardless of the environment, and an outer layer of modular components that can adapt (or be adapted) to the context. The adaptable components include:

- CRDs, which model the physical components (or their virtual counterparts) of the cluster, such as servers and network infrastructure.
- Providers that specialise CAPI functionality for a specific technology or infrastructure.

Although these components are subject to many changes, Cluster API manages them declaratively, i.e. by asking the user for the desired state. By not having to specify how these components work, the code becomes a series of instructions and specifications that are, by their nature, more reusable.

Custum Resource Definitions and controllers

Cluster: represents a Kubernetes deployment and can be either Management or Workload type. In the first case, it is a cluster within which one or more Infrastructure providers run and where resources (e.g., Machines) are saved: it is used to provision and manage multiple Workload Clusters. On the other hand, the Workload Cluster is a Kubernetes cluster designed to be used by end users.

- Cluster: represents a Kubernetes deployment and can be either Management or Workload type. In the first case, it is a cluster within which one or more Infrastructure providers run and where resources (e.g. Machines) are saved: it is used to provision and manage multiple Workload Clusters. On the other hand, the Workload Cluster is a Kubernetes cluster designed to be used by end users.
- Control Plane: this is the set of services at the base of a cluster. In particular, it controls the configurations and life cycle of the nodes, reconciling the desired status via control loops. It can be managed entirely by Cluster API or by external services.
- Machine: declaratively specifies the configurations of a single Kubernetes node. Within Cluster API, a node can be considered as such if it is capable of running a Kubelet (illustrated in section 2.1.1). When a machine is created, the provider controller provisions and installs everything needed to register the new node within the cluster according to the configurations defined in the Machine object. With the exception of a few parameters (labels, annotations, and status), each machine is considered immutable by CAPI. Machines do not contain any information about the infrastructure and its provider; this information is stored in the object referenced by the *InfrastructureRef* field.

- MachineHealthCheck: this is a resource that allows the administrator to define the conditions under which a machine is considered *unhealthy*. MachineHealthChecks are defined within a Management Cluster and refer to a single Workload Cluster; in fact, if a node does not meet the conditions, the MachineHealthCheck starts the repair procedure, replacing the machine. The MachineHealthCheck controller is also able to check various aspects of a node's *health*, such as pod capacity, whether it is reachable via the network, and whether it is running out of disk space.

- MachineSet: represents the state of a set of Machines. It is not used directly by the administrator, but is used by MachineDeployment to reconcile machines with the desired state. For completeness, it is worth noting that a MachineSet behaves similarly to a ReplicaSet, one of the fundamental components of Kubernetes presented in section 2.1.1.

- MachineDeployment: maintains the state and allows the updating of Machines and MachineSets. A MachineDeployment reconciles changes to a Machine by destroying the old one and creating a new one. Since Machine objects are practically immutable, it is recommended to use MachineDeployment when it is necessary to modify them. A MachineDeployment represents the Cluster API counterpart of Deployments in Kubernetes. One of the fundamental attributes of these objects is `replicas`, which tells Kubernetes how many machines of that type must be present within the cluster.

Each CRD has a service dedicated to checking its status, known as a controller. Each controller ensures that objects of a given type reflect expectations and therefore comply with the status desired by the user. As in Kubernetes, the phase in which a controller modifies the status of an object to meet certain requirements is called reconciliation. To specify the desired state for any resource, a YAML file, called a manifest, is compiled, which must contain the attributes of the corresponding CRD. For each parameter defined in the CRD, the user must set a specific value that will be managed by Cluster API [47] [51].

Providers

In early versions of CAPI, the term referred only to infrastructure providers; however, it now has a more general meaning and refers to all elements that act as a bridge between the Cluster API and the environment.

There are three types:

- Infrastructure Provider: this is the provider that implements the provisioning of the infrastructure and computational resources required to form the Cluster. There are infrastructure providers capable of interacting with cloud platforms such as AWS (CAPA), Azure (CAPZ), Google Cloud Platform (CAPG) and other types of providers that are capable of directly managing bare metal infrastructure via $Metal^3$ (CAPM3) or VMware (CAPV). Multiple infrastructure providers capable of managing the same infrastructure can coexist, and in these cases they are referred to as *variants*. It is important not to confuse the term infrastructure provider with the provider of the infrastructure that will host the cluster.

- Control Plane Provider: this is the provider that instantiates the control plane for a Kubernetes cluster and ensures that all services provided by the control plane are functioning correctly.

- Core Provider: the provider that implements controllers for the fundamental components of CAPI.
- Bootstrap Provider: this is the provider that generates the configuration scripts needed to install the software that will turn the machine into a Kubernetes node. The default provider is called Cluster API bootstrap provider Kubeadm (CABPK) and creates a Cloud Init script to configure the machine from the first boot and then uses kubeadm to configure Kubernetes. A Kubernetes secret is used to pass the cloud-config generated by the provider to the node.

Each provider implements best practices to manage the environment resources for which it was designed. This architecture allows Cluster API to maintain a single declarative API while deploying Kubernetes on different types of physical and virtual resources. The ability to reuse the same configurations in very different environments, such as cloud and on-premises, is a major advantage for administrators, as they no longer have to worry about implementing configurations and best practices to be used in different environments [47] [51].

2.3.3 Working principles

Cluster API manages clusters using mechanisms very similar to those used by Kubernetes; however, the latter uses a small number of management nodes to administer a much larger number of worker nodes, on which the pods are running. Similarly, Cluster API uses a management cluster to administer one or more workload clusters used by users. The controllers and providers used by CAPI run in the management cluster and ensure that the status of the workload clusters complies with the status defined by the administrator. As explained above, the status of the workload cluster is declared via YAML manifest based on the CRDs of the resources being managed. CAPI CRDs are generic and infrastructure-independent, and providers implement specific CRDs and related behaviours to adapt them.

Tools

CAPI provides a set of tools for interacting with the cluster workload; the most common method involves interaction via the clusterctl command line tool. This simplifies recurring operations such as cluster configuration, creation, deployment and management through the creation of CRDs, without the need for the administrator to manually compile the manifest. Clusterctl also allows you to operate directly on Kubernetes clusters in order to make them management clusters and install CAPI components, create cluster workloads, and move their management from one management cluster to another.

Another widely used tool is the kubeadm control plane (KCP), a control plane used by default by Cluster API to control Kubernetes control planes; it allows to manage all components such as schedulers, etcd data stores and networking services in a punctual manner. KCP also allows control planes to be distributed across different failure domains, so that the failure of multiple control planes is a rather rare occurrence.

Deployment

To deploy Cluster API, you need to use two separate clusters: the first is a temporary cluster called *bootstrap cluster*, which is used to create the second, which will effectively be the management cluster. To achieve this, the bootstrap cluster must generate certificates, initialise one or

more control planes, and install the core components of Kubernetes. The steps for deploying the infrastructure are as follows:

1. Install kubectl, clusterctl, Kind, Docker and Helm.
2. Use Kind to create a small local cluster, which uses Docker containers instead of physical nodes, to be used as a bootstrap cluster to provision the management cluster on the selected infrastructure.
3. Transform the newly created cluster into a management cluster using the *clustrerctl init* command, which accepts, as a parameter, a list of providers to install in addition to the default ones. The default providers include the cluster api core provider, the kubeadm bootstrap provider, and the kubeadm control plane provider. Before initialisation, you can set some environment variables that tell clusterctl to enable certain experimental features.
4. Meet the prerequisites imposed by the chosen infrastructure provider; usually, these requirements consist of setting API keys, certain environment variables, or installing some additional software packages.
5. Initialise the infrastructure provider using the command *clusterctl init –infrastructure <infrastructureProviderName>*: at the end of this step, if there are no errors, the management cluster has been successfully created.
6. Create the first cluster workload by compiling the YAML template, which is returned by the *clusterctl generate cluster* command. If only one infrastructure provider is installed, the command partially compiles the template by applying best practices for that type of infrastructure. As with cluster management, some configurations may be necessary in this step based on the chosen infrastructure provider.
7. Apply the newly compiled template using the command *kubectl apply -f <templateName>.yaml*. This command triggers the creation of the cluster and the resources it will subsequently manage.
8. Once you have verified that the cluster control plane has been initialised correctly, you can proceed to download the kubeconfig file, which is required to access the workload cluster. This kubeconfig is the key that can be distributed to users who need to use that cluster.

Some steps have been deliberately omitted, but will be explained in later chapters. From the above list, it is clear that using cluster APIs can greatly simplify the process of provisioning a Kubernetes cluster [47].

Scaling

One of the most appreciated features of Cluster API is the ability to scale the cluster based on the amount of work that needs to be done. For worker nodes, CAPI must ensure that the right amount of hardware is provisioned to meet demand; for control nodes, however, the number of backup nodes must be adequate in case of failure. The Kubeadm control plane allows you to scale Kubernetes control planes by placing them in different failure domains, further reducing risk. The real strength of Cluster API lies in scaling the number of working nodes; in fact, it is possible to declaratively set the number of worker nodes that the administrator wishes to provision. The rest

of the process will be entirely managed by CAPI, which will increase or decrease the number of machines with the help of aggregator objects, such as MachineDeployment.

There is also a tool called Autoscaler, which automatically adjusts the number of machines in the Kubernetes cluster based on the number of pods and how they affect node utilisation [47]. Some of the metrics that the autoscaler can use to evaluate the status of the cluster are: average CPU utilisation per node and how long a node has been idle. The autoscaler can also take into account the hardware present on the various nodes, such as accelerators, particularly GPUs.

Self Healing

Cluster API gives Kubernetes the ability to self-heal by provisioning new infrastructure. When a node fails within a cluster, Kubernetes takes care of creating new instances of the pods that were running on that node on a new node; however, the operation can only be completed successfully if a node that is already configured and has sufficient resources is available. In extreme cases, where a significant number of nodes in the cluster fail, Kubernetes is unable to guarantee the proper functioning of the pods. Since CAPI is able to interact with both Kubernetes and the infrastructure, it can handle situations such as the one described above by provisioning new nodes. These nodes can be machines that are on-premises, unused, or already part of other clusters, which could be moved without causing problems. If necessary, new nodes can also be provisioned on cloud infrastructure.

To understand if a node is experiencing problems, the administrator can use the MachineHealthCheck resource and define the behavior to follow using a custom provider. The repair process is called remediation and is activated after the "unhealthy" condition defined by the administrator has occurred and remained unchanged for a certain period of time. By default, the behavior adopted to remedy a failure involves eliminating the malfunctioning machine and creating a new one, which, once correctly inserted into the cluster, will host the execution of the pods of the failed one [51].

2.4 Metal³

The open source project $Metal^3$ (pronounced: "Metal Kubed") [31] aims to manage hosts at the bare metal level using Kubernetes. With it, you can register your bare metal machines, provision the operating system, and, if desired, configure the machines as Kubernetes nodes. Once the machines are configured through this process, $Metal^3$ is able to operate and update them. $Metal^3$ is itself an application that runs on Kubernetes, uses its resources, and uses its API as an interface. In the context of this work, $Metal^3$ is fundamental in that it implements a provider for Cluster API, thus allowing CAPI to be used as an interface to control the infrastructure at the lowest possible level. In order to operate on the hardware, $Metal^3$ uses Ironic, an Open Stack component that will be introduced in the next section. Through Ironic, $Metal^3$ can then provision all machines that are composed of compatible hardware. What the administrator can do through $Metal^3$ is define a Kubernetes manifest in which the hardware and cluster layout are presented. Subsequently, $Metal^3$ will take care of:

- Reach the hardware inventory.
- Configure BIOS and RAID on hosts.

- Clean the disc if necessary.
- Install and start the image of the chosen operating system.
- Deploying Kubernetes.
- Update Kubernetes or the host operating system with a non-disruptive rolling strategy.
- Fix malfunctioning nodes by attempting to reboot them or, in extreme cases, removing them from the cluster.

2.4.1 Project structure

The *Metal*^{3} project consists of several sub-projects that interact both with each other and with external components. As in Cluster API, most of these components are Custom Resource Definitions with their respective controllers.

BareMetalHost

One of the main CRDs of *Metal*^{3} is the BareMetalHost (BMH), which represents the physical machine as a Kubernetes object; this object can be manipulated by the administrator to modify the configuration of each host. Some examples of configuration may include network settings and the operating system images to be used. Operations on BareMetalHost objects are applied using standard Kubernetes APIs.

Bare Metal Operator

The Bare Metal Operator (BMO) is the controller of BareMetalHost objects and manages their lifecycle within the Kubernetes environment.

The BMH represents the interface between the Kubernetes cluster and the physical infrastructure, while the operator handles fundamental processes such as discovery and provisioning. During the discovery phase, the BMO inspects the hardware installed on the various hosts and reports it within the corresponding BMH objects; the information reported within the BMH object includes CPUs, RAM, disks and network interfaces. During the provisioning phase, when the operating system is installed on the machine, the BMO configures the host's RAID and changes or updates the firmware settings. During both provisioning and deprovisioning, the controller can, if instructed by the administrator, delete the contents of the disks. Once the provisioning phase is complete, the node will be ready to join the Kubernetes cluster. To perform all these operations, the BMO uses the RESTful API provided by Ironic.

The automation of these tasks, which normally require considerable effort on the side of the administrator, greatly reduces both provisioning times and the risk of having inconsistent configurations within the cluster.

Cluster API Provider Metal³

Metal³ Cluster API Provider (CAPM3) is the CAPI provider implemented by Metal³, whose purpose is to act as a link between the concepts introduced in Cluster API and the underlying environment. In this case, the environment is not represented by the configuration of virtual machines made available by some cloud provider, but rather by a bare metal infrastructure.

The joint use of CAPI and Metal³, made possible by CAPM3, allows Cluster API to be used to declaratively manage the lifecycle of the Kubernetes cluster, while Metal³ manages, again declaratively, the lifecycle of the bare metal nodes on which the cluster resides. The image 2.2 shows how the components presented are organised. The functioning of Ironic will be presented in section 2.5.

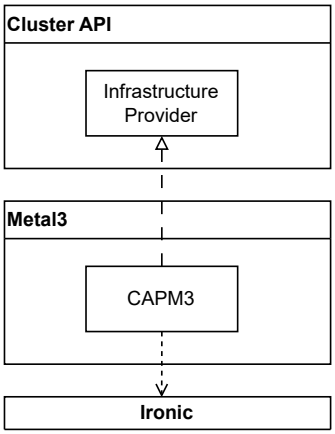

Figure 2.2: Architecture overview

IP Address Manager

The IP Address Manager (IPAM) project provides a controller for static IP address allocation via the Metal³ Cluster API Provider. IPAM allows you to reserve a pool of IP addresses to be provided to nodes at "provisioning time"; this mechanism is particularly useful when there is a need to reserve certain addresses for a specific purpose (or more than one), but the machines that will respond to those addresses have not yet been created. Usually, each group of machines that CAPI manages under the same MachineDeployment object has an associated pool of IP addresses, within which it is possible to declare certain addresses as reserved (e.g. the gateway address).

2.5 Ironic

Ironic was created as an OpenStack project with the aim of replacing the bare metal drivers that were present in Nova (the OpenStack compute instances project) at the time. It allows bare metal machines to be provisioned via a convenient API; in fact, the project was created precisely to compensate for the lack of standard APIs for machine management. The OpenStack group realised that there were projects and scripts available on the market that could configure every aspect of the machines, but none of them were able to manage their entire lifecycle. This forced infrastructure administrators to implement ad hoc solutions for the different machines they were operating on, with consequent portability issues. The complexity of creating scripts capable of

configuring machines at such a low level lies in the wide variety of components that can be present on a node, whose combinations are almost infinite [6].

Ironic is developed in Python and provides generic drivers, which act as interfaces, supporting standards such as IPMI and Redfish. These standards allow hardware to be abstracted, although in different ways, and can be used to control any type of machine independently of the brand of components. Ironic is API-driven and API-first; in fact, provisioning and infrastructure management operations are entirely managed through a RESTful API that is agnostic to the underlying hardware [6].

2.5.1 Ironic's role in Metal³

Within this work, Ironic is fundamental as it handles requests from Metal³, communicating directly with the machines; to do this, it communicates with the Baseboard Management Controllers (BMCs) using specific protocols. The BMC is a specialised component on a machine's motherboard, whose purpose is to interface with hardware management systems and the hardware itself. In order to function independently from the computer, the BMC has firmware and dedicated RAM. It is important to note that this component is capable of turning the machine on and off and changing the boot device. If more complex operations are required, Ironic can load the Ironic Python Agent (IPA) onto the machine so that it, rather than the BMC, can carry them out. The IPA is a Python service, contained in a particular disk image, which can run within the RAM and therefore does not require the installation of an operating system. Given the critical nature of the operations that can be performed via the IPA, such as writing images to the disk, communications between it and other components are encrypted using SSL/TLS [47].

In addition, Ironic maintains information about machines in a database, which is used by the Bare Metal Operator to determine the status of machines at the bare metal level, compare it with the desired status specified by the administrator and stored within the resources managed by Kubernetes. If discrepancies arise between these two sources, the BMO would use Ironic's APIs to reconcile the physical machine with the desired state. Ironic's database is based on SQLite and is part of the back end of Ironic's APIs. This database is ephemeral, so the data within it is deleted every time the Metal³ pod is restarted. When this happens, the operator applies different strategies to repopulate the Ironic database, taking care not to perform the reconciliation process for every node in the cluster. The information in the Ironic database is also entered by the BMO when a BareMetalHost resource is created; this hardware information is obtained through an inspection process that is launched on the node itself and executed by Ironic. To make this information available within Kubernetes, it is stored in a resource called HardwareData. It should be noted that the information contained in this resource is not updated unless an explicit request is made to re-run the inspection phase, as it is assumed that the machine components will not change suddenly. Ironic is then used in the provisioning and deprovisioning phases of the machines. Specifically, when the provisioning of a node is requested, the IPA downloads and writes the image, while in the deprovisioning phase, the node is completely cleaned before shutdown. The administrator can set certain parameters, such as the URL from which to download the disk image, using the *image* parameter of BareMetalHost [47].

2.6 Cloud Init

Managing the operating system configurations of numerous machines can be a complex and repetitive task. Cloud Init is an open source tool, mainly used by developers and system administrators, that allows you to configure servers or virtual machine instances in the cloud quickly, automatically and repeatably. When a machine is provisioned, Cloud Init applies the configuration provided by the user; the strength of this tool lies in the ability to reuse the same configuration on multiple machines, with the certainty that they will all be configured in the same way. In this way, even if the number of machines to be configured grows exponentially, the system administrator only has to provide a list of settings they want, leaving Cloud Init to figure out how to achieve them. This tool takes care of all the operations that are normally performed by the administrator, such as setting the hostname, configuring network cards, creating users, managing permissions and installing software packages. If an administrator needs a setting that Cloud Init cannot handle, it is always possible to request that a custom configuration script be run; this aspect proved to be of fundamental importance in the development phase of the solution to the problem addressed in this thesis. Using this tool ensures that all machines are configured in the same way, greatly limiting the possibility of human error [15].

2.6.1 Instance setup process

The operations performed by Cloud Init to configure a machine are carried out in two distinct phases of the bootstrap process. The first phase occurs during the initial moments of startup, when the machine is not yet connected to the network and therefore only local operations can be performed. The second stage takes place during the advanced boot phase, i.e. after the network configurations have been applied and it is therefore possible to communicate with the outside world. Many of the configurations provided by Cloud Init only need to be applied when the machine is first started up. For example, it is unnecessary to recreate all users every time the machine is rebooted. For this reason, Cloud Init has a mechanism to recognise whether it should perform all operations ("per-instance" configuration) or only some of them ("per-boot" configuration). This mechanism is based on a cache maintained internally by the program, in which the steps that have already been performed are recorded. If this cache is present when the machine starts up, it means that Cloud Init has already run on that node and that not all steps need to be performed [15].

```
#clod-config
groups:
  - admingroup: [root,sys]
  - cloud-users
users:
  - default
  - name: testuser
    gecos: Mr. Test
    homedir: /local/testdir
    sudo: ["ALL=(ALL) NOPASSWD:ALL"]
```

Listing 2.2: Simple cloud-config example

To provide Cloud Init with the configurations to be applied to the machines, the administrator must create a simple YAML file called *cloud-config*. It should be noted that Cloud Init uses the same markup language as Kubernetes.

First fase

This first phase is essential to enable all subsequent phases to function correctly. In particular, Cloud Init is responsible for:

- Identify the data source, i.e. the set of files containing the configurations that need to be applied.
- Fetch configurations from the data source. Configurations are usually separated into three files:
  - Meta-Data: these are essential configurations to be applied to the machine, such as hostname and network configurations.
  - Vendor data: these are configurations that must be applied based on the infrastructure; they are usually provided by cloud providers in order to implement certain best practices.
  - User data: these are the configurations that are normally requested by the user and will be applied in the second phase.
- Apply network settings.

Second fase

At this stage, Cloud Init applies the previously loaded configurations in order to configure the machine. The operations performed are:

- Creation and configuration of user accounts. Cloud Init can also inject SSH keys to ensure remote access to machines, a feature that is essential in cloud environments.
- Updating the operating system and installing additional software requested by the user.
- Execution of custom configuration scripts provided by administrators.

2.6.2 The relation with Cluster API

Cloud Init is a fundamental tool within the provisioning process carried out by the Cluster API bootstrap provider Kubeadm. In fact, the provider that is adopted by default by Cluster API generates a cloud-config file, which is able to configure the node so that it can join the Kubernetes cluster. To do this, CAPBK compiles a template containing generic information and inserts the specific information for the machine being provisioned. The cloud-config file generated in this way is placed in a Kubernetes secret and will be referenced by the Machine object, making it usable during the first boot. It should be noted that the configurations in Cloud Init are mainly used to launch the kubeadm tool, which will then configure the machine to make it a Kubernetes node correctly inserted into the Cluster [47].

2.7 Other technologies

In this last section of the chapter, I would like to briefly present the other technologies that have been adopted within the project, whose importance for the development of the solution is marginal compared to those presented above.

CentOS

CentOS is an open source operating system based on the Linux kernel. This feature, combined with a very active community, makes the CentOS project extremely flexible; in fact, there are many distributions of the operating system, each designed for specific areas (e.g. Cloud, Automotive, etc.). CentOS was chosen as the operating system mainly because of its close relationship with Red Hat Enterprise Linux (RHEL). The latter is one of the most widely used enterprise-level operating systems in the world and has been under IBM's control since 2018. CentOS, on the other hand, is a continuously delivered distribution that is "just one step ahead" of the RHEL development cycle. This strong link means that CentOS is used as a test bed for features that will soon be introduced into Red Hat's operating system distributions [7] [44]. As highlighted in this section, the choice to use this operating system was solely for the purpose of creating a solution that could be easily integrated into the IBM product range. Any other Linux distribution would have achieved the same results, but would have required some modification to be brought into the business ecosystems.

Helm

Helm is the Kubernetes package manager, which facilitates the installation and management of Kubernetes applications. To do this, Helm uses a collection of files that describe the Kubernetes resources that make up the application. This packaging format is called a chart. Each chart contains the application and all its dependencies, making it easier to deploy in different environments (e.g. development and production) and to share. When a chart is installed, Helm creates a "release" to track that specific installation; the same chart can thus be installed multiple times in the same cluster, simply by specifying different release names, allowing, for example, multiple independent instances of the same application. Charts can reside in repositories, which contain metadata about the charts available within an index file, facilitating sharing and discovery [16]. There are official Helm repositories (with 'Stable' and 'Incubator' charts) and unofficial repositories provided by companies such as IBM.

MPICH

The MPICH project is an open source implementation of the Message Passing Interface. Since this implementation can be found on many HPC systems, it has been implemented in a modular way, allowing infrastructure administrators to adapt the software to their needs by developing modifications to the way the library operates. In order to compile programs, MPICH provides wrapper compilers for certain programming languages (e.g. C, C++); the goal of these compilers is the same as that of classic compilers, so as to make the most of the features that the project offers. MPICH also provides commands to launch the developed programs correctly. It should be noted that if the execution environment is managed by a scheduler (such as Slurm), these commands are not necessary, as the scheduler takes care of the execution [37].

Chapter 3

Development process for bare metal HPC and Cloud convergence

In this chapter, which represents the heart of the project, I will illustrate the development process that was followed during the five months of collaboration with IBM to extend Kubernetes and make it a tool for managing bare metal HPC nodes. The presentation will follow the chronological order of the project's development and will also illustrate any alternative solutions that were explored. Based on the requirements that were defined, it was decided to use a bottom-up development process. This process involved the evolution of fundamental subcomponents that were integrated with each other to achieve the desired result.

3.1 Initial Provisioning and Automation Phase via Cloud Init

The first period was dedicated to understanding how Cloud Init works and how a Slurm node could be configured automatically. The decision to study the initialization of this type of node rather than a Kubernetes node was not casual; in fact, the configuration for Kubernetes nodes is already managed automatically by the Cluster API Bootstrap Provider Kubeadmin. In order to begin this study phase, a virtual environment was created in which to simulate the provisioning process of bare metal nodes, paying particular attention to the configuration of the operating system.

Cloud Init virtual enviroment setup

To virtualize a machine running Linux, we opted to use Qemu [43], an open source tool designed for emulation and virtualization. At this stage, there was no indication that CentOs would be required as the operating system, so I used a minimal Ubuntu image called jammy-server. This image was chosen because it is particularly lightweight and comes with Cloud Init already installed, thus simplifying part of the configuration process. Once the first virtual machine had been successfully

started, I proceeded with the setup to make Cloud Init operational. In order to function, it requires a server that simulates an Instance Metadata Service (IMDS). Specifically, an IMDS is a service that provides a virtual machine with metadata useful for configuration. In this case, these are the meta-data, user-data, and vendor-data files that are requested during boot by Cloud Init; therefore, I created a simple HTTP server in Python capable of providing the above-mentioned files.

To get the most out of Qemu, the virtual machine was launched by setting parameters for acceleration and bootstrapping. For example, the `accel=hvf` accelerator was used to take advantage of the capabilities of macOS (the host machine), making the VM start up particularly quickly. Other key parameters concerned instructions on how to reach the IMDS server. The parameter `-smbios type=1, serial=ds='nocloud; s=http://10.0.2.2:8000/'` specifies the data source to be used, instructing Cloud Init not to use one related to a specific cloud provider and to retrieve the configuration files from the specified URL. This URL pointed to the default gateway of the virtual machine and the default port of the Python web server, thus allowing Cloud Init to query the server running on the host from within the VM.

After the initial configuration, it became necessary to improve the cloud-config settings relating to users and security. The preliminary solution did not allow the creation of multiple users and had security issues relating to the use of passwords saved in plain text. As a result, changes were made to allow the creation of multiple users, and a first user with administrator privileges and a second standard user were configured. In addition, to improve security, password login was disabled in favor of enabling remote management via an RSA-encrypted SSH connection; an RSA key was therefore added to the administrator user to ensure remote access. To accept SSH requests, the virtual machine activation command was modified to include the value `hostfwd=tcp::2878-:22` in the `-net` parameter. Given the nature of the testing environment, which involves the frequent creation of new virtual machines and, consequently, changes to host keys, it was necessary to implement a standard procedure to delete previous keys from the `known_hosts` file. This prevented errors related to key changes from being interpreted as potential man-in-the-middle attacks.

Slurm Integration

The next goal was to integrate Slurm, the chosen workload manager. The first step was to manually initialize a single-node cluster capable of running a simple parallel hello-world program. Since the cluster would consist of a single node, it would be both management and computing at the same time. It is important to note that this type of configuration is not normally used in production environments; however, it is permitted for this type of experimentation. To install Slurm, first download the software packages using the Advanced Packaging Tool (APT) and then proceed with the configuration of the `/etc/slurm/slurm.conf` file. This file indicates the presence of a single node with 4 CPUs (each capable of utilizing 2 cores) and 256 MB of RAM. We chose to assign 4 CPUs to the virtual machine so that we could run the parallel hello-world program on a larger number of cores. To achieve this result, it was sufficient to launch the virtual machine with the parameter `-smp cores=4`; a small batch script was then written, visible in listing 3.1, capable of exploiting Slurm: this script uses the cores present in the node to execute two tasks in parallel. During the execution tests, it emerged that the order in which the tasks were printed (task 1 and task 2) was variable, confirming the parallel operation managed by Slurm.

```
#!/bin/bash
```

```
#SBATCH --job-name=singlecputasks
#SBATCH -N 1
#SBATCH --ntasks=2
#SBATCH --ntasks-per-node=2
#SBATCH --output=/home/mik/slurm.test.stdout

(srun --exclusive --ntasks=1 -c 1 echo  "Hello world, I'm task 1") &
srun --exclusive --ntasks=1 -c 1 echo "Hello world, I'm task 2"
```

Listing 3.1: First slurm script

During testing, various Slurm commands were used to monitor and manage the workload:

- squeue: to view the jobs in the queue.
- sinfo: to obtain information about the node.
- scancel <job_id>: to cancel a job from the queue.
- scontrol show job <job_id>: to check the execution status of a job.

Of course, since this is a test setup consisting of a single node, the complete configuration shown in section 2.2.3 was not performed, and the modules for communication via MPICH were not installed.

The next step focused on automation, with the goal of running the hello-world program directly at virtual machine startup through Slurm and Cloud Init. To achieve this, Slurm must be installed and configured automatically, ensuring that the configuration file `slurm.conf` is written correctly during the bootstrapping process. The cloud-config was then modified so that these two steps were performed and, subsequently, the script for executing the program was launched; to fulfill this objective, the Cloud Init directive `runcmd` was used. This configuration allowed the Slurm script to be launched correctly at system startup, verifying, only in part, the feasibility of using this tool for Slurm node provisioning.

3.2 Evaluation of alternative technologies

3.2.1 Bootc

During the development process, in collaboration with the IBM team, it was decided to evaluate various techniques and technologies. At this stage, once it was established that Cloud Init would be a valid tool for configuring Slurm nodes, it was agreed to explore the use of Bootc [5] (a project that will be described shortly) to change the configuration of the nodes. In fact, as explained above, the ultimate goal remains to be able to change the configuration type of a bare metal node from an HPC node to a Kubernetes node and vice versa.

Bootc is an open source project whose purpose is to create bootable containers, i.e., images that comply with OCI and Docker standards but contain the entire operating system inside them. Compared to classic containers, which, as already explained in paragraph 1.2, have been widely adopted, a bootable image also contains the Linux kernel. This element certainly makes the image heavier, but it opens up the possibility of replacing the entire operating system of a machine as if you were working on classic containers. This latter aspect has attracted particular attention,

leading to the formulation of two insights. The first involves the creation of two separate images, one designed for the HPC environment and one for the Kubernetes environment, which could be exchanged whenever it was necessary to change the intended use of a bare metal machine. The second involves the use of two minimal images that can be specialized via Cloud Init when the change of destination is made via Bootc.

Having already explored the potential of Cloud Init and established that it was a valid tool for configuring machines, we decided to experiment with the second idea. The primary objective was therefore to switch operating system images using Bootc, ensuring that the Cloud Init configuration was applied correctly once the request to change the bootable container was made.

In order to operate with Bootc, in a manner similar to other containerization systems, manifests called *Dockerfile* must be created, containing the commands that the user wishes to execute to provision a machine. The initial idea was to create a base image that did not use Cloud Init and a second one that did use it to configure the operating system. The transition between one image and another and the restart of the machine with the new initialization should have used the `bootswitch` command. In order to experiment with Bootc, it was necessary to change the base operating system from the version of Ubuntu used in the previous section to Fedora. This change was made because Fedora is one of only two Linux distributions currently supported by Bootc; this distribution required modifying the cloud-config files that had been developed previously, as Fedora does not allow Slurm to be easily installed via APT. At this point, ad hoc scripts were created to install and configure Slurm on the new operating system. Subsequently, attention shifted to creating the operating system image using Dockerfile. To this end, a tool used for container management and image generation was installed. Podman [42] was chosen because it allows bootable containers to be created quickly and via a graphical interface. At this point, a bootable container capable of using Cloud Init to configure a Slurm node was configured. After several failures, it became clear that the configuration process was not successful because Fedora was unable to install the software packages necessary for Slurm to function. This error was due to a constraint imposed by Bootc, namely that the operating system images used by bootable containers must be read-only; as a result, once the image has been created and distributed, it is no longer possible to install additional packages. This requires that all software components essential for Slurm to function be included and configured directly via dockerfile before the build phase and that it not be possible to install them using Cloud Init during the initial startup phase. At this point, we proceed with testing the configuration changes, combining the use of Bootc and updating the user-data configuration file managed by Cloud Init. The following were created using dockerfile:

- A Fedora Bootc image with no software packages installed and a single user.
- A Fedora Bootc image with Cloud Init enabled and Slurm software packages installed.

After the build, both images were saved to the Red Hat Quay registry so that they could be downloaded from Bootc at any time. Next, Qemu was used to start the first virtual machine with the first image, setting the parameters that Cloud Init uses to locate the IMDS, even though we knew that the first image would not use them because Cloud Init was inactive. After verifying that the Fedora virtual machine was working, the Bootc image update process was launched using the `bootc switch <image_url>` command; this command started downloading the packages needed to update the operating system and, once completed, the virtual machine was restarted with the new image. During boot, the new image produced with Cloud Init configured successfully downloaded the configuration files from the server and configured the machine as a Slurm node.

Although the experiment was successful, it was necessary to investigate why the smbios parameter remained unchanged despite the change of operating system and the restart of the virtual machine. Consulting the Cloud Init documentation [15], it was discovered that the data relating to the configuration server search is written to the BIOS. Since Bootc only updates images but does not modify the BIOS, the two parameters transmitted via QEMU (i.e., 'ds' for the *datasource* 'nocloud' and 's' for the *source* of the Python web server) remain valid. These parameters are maintained and exploited by Cloud Init, if present on the virtual machine, to retrieve the configurations.

However, a potential problem has been identified in relation to the future goals declared by Bootc developers, who aim to allow changes to the operating system to be applied without requiring a reboot. The absence of a reboot would prevent Cloud Init from configuring the machine as desired. Although this is not the standard configuration procedure, a possible future solution could be to include an executable file capable of configuring the machine within the new image. This configuration file should be launched when the bootable container starts, using the `ENTRYPOINT` keyword typical of Dockerfiles. Using this solution would defeat the purpose of using Cloud Init, which is an easy-to-understand and highly repeatable system for configuring nodes. For these reasons, we decided not to continue further experimentation with Bootc.

3.2.2 Ignition

After experimenting with Bootc, attention was focused on analyzing Ignition [19], an alternative provisioning tool to Cloud Init. The documentation consulted highlighted how Ignition is launched only when the machine is first started, as it is intended as a tool for provisioning and not for managing system configurations. Despite this, Ignition proved to be a suitable tool for the objectives set. As in Cloud Init, Ignition files are also declarative in nature: they are not designed as a sequence of steps to be performed, but rather as a description of the state that the machine must reach at the end of the boot process. However, a critical constraint was encountered at this stage, namely that Ignition does not support the direct launch of configuration scripts. This greatly limits the configuration process of Slurm nodes which, as explained above, require the execution of small custom programs. It is still possible, albeit indirectly, to launch configuration scripts, provided that they are implemented through *systemd* services. This means that previously created configuration scripts must be rewritten to be compatible. To facilitate integration and discourage manual changes to the system, Ignition files should not be written manually: instead, *Butane*, a description language based on YAML files (very similar to Cloud Init configuration), must be used to generate them. The transformation process is performed using the command `butane -pretty -strict example.bu > example.ign`. The generated configuration file, or a URL that references it, is then passed to the machine according to the specifications given by the target platform. Although the reasons that led the Ignition developers to set such strict rules for the development of custom configurations were clear, it was decided not to continue experimenting with this technology.

3.3 Cluster API and Management Cluster Setup

The next steps were dedicated to configuring a development environment compatible with Cluster API, in order to study its features locally before operating on the systems provided by IBM. The strategy adopted involved creating a local Management Cluster, in accordance with the two-tier

architecture provided by CAPI. First, it was necessary to install the basic tools for interacting with the Kubernetes environment and the cluster lifecycle management infrastructure. The installations involved kind, kubectl, clusterctl, and Docker. The next step was to create a management cluster using kind. Kind [25], short for Kubernetes in Docker, is a tool that allows you to test Kubernetes by creating a local cluster whose nodes are represented by Docker containers. Kind's goal is to allow users to experiment with Kubernetes, local deployments, and Continuous Integration systems. To experiment with CAPI, we configured the cluster with a single node acting as the control plane, since, as it was not a production environment, there was no need for a redundant system capable of coping with any failures; a configuration with a larger number of control planes would only have slowed down the simulation. A crucial detail in the configuration of the Kind cluster was the use of the extramounts parameter within the configuration file: this parameter allows a portion of persistent space on the host to be assigned to the node; this space was used to enable communication with Docker through the socket used by the dockerd daemon. The daemon is at the heart of the Docker project as it manages the container lifecycle, which includes the creation, execution, and monitoring phases. Proper communication between dockerd and Kind is essential for Kind to simulate the nodes belonging to the cluster. Once the Kind cluster was created, it was initialized as a CAPI Management Cluster using the `clusterctl init -infrastructure docker` command.

Running clusterctl init is necessary to install the core components of CAPI, making the cluster a Management cluster capable of managing the lifecycle of other Kubernetes clusters. There are many providers that can be used via CAPI [47], and in order to perform the initialization phase correctly, clusterctl must be able to read the configurations and components to be installed. To know these configurations, CAPI maintains a list of providers where, for each of them, a URL to a repository containing the following is specified:

- metadata.yaml: a file that documents all the versions that have been released for that provider. For each release, the CAPI version that the provider implements must be associated.
- components.yaml: a file containing a list of all components (CRDs, controllers, etc.) that must be installed for the provider to function.

The presence of these files within the repository is mandatory; however, it may also contain templates to facilitate cluster configuration.

Once the initialization phase was complete, the Management Cluster could finally be used. At this point, the focus shifted to creating a Workload Cluster, the Kubernetes cluster, designed to host end-user workloads. This process required the generation of a YAML configuration file that indicated to CAPI the type of cluster and the type of machines to be provisioned. To facilitate this step, a parametric clusterctl command was used, as shown in the following listing.

```
clusterctl generate cluster capi-quickstart \
  --kubernetes-version v1.32.0 \
  --flavor development \
  --control-plane-machine-count=1 \
  --worker-machine-count=2 \
  > capi-quickstart.yaml
```

Listing 3.2: Command for generating the CAPI cluster configuration file

The parameters used are the following:

- The name given to the cluster, i.e. capi-quickstart;
- The version of Kubernetes to be used. We opted to use the version recommended by the CAPI documentation.
- The flavor, i.e. the type of template made available by the provider.
- The number of control plane machines you want to create.
- The number of worker machines that you want to make available to the user.

The parameters provided were used by clusterctl to generate the configuration file (capi-quickstart.yaml), which was then applied using the command `kubectl apply -f capi-quickstart.yaml`. This command triggered the Cluster API reconciliation mechanism, initiating the creation and provisioning of the resources required for the new Workload Cluster. Once the provisioning phase was complete, the `kubectl get cluster` command confirmed that the Workload cluster had been successfully created. Analysis of the machines belonging to the cluster showed that three Docker containers had been successfully started, one for each machine requested. Before destroying the two clusters and declaring the experiment a success, the configurations contained in the *kubeconfig* file were retrieved and studied.

Managing the kubeconfig file is essential for interacting with the Workload Cluster. A Kubeconfig is a YAML file that contains all the information needed to authenticate and connect to a Kubernetes cluster; it contains details about the cluster itself, including certificates and secret tokens. When the user uses the kubectl command-line tool, it reads the information in the kubeconfig file to establish communication with the cluster's API server. The default location for this file is `$HOME/.kube/config`. Obtaining this configuration file and distributing it to users is the final step in ensuring that they, or administrators, can access and use the Workload Cluster. The file was obtained using the command `kind get kubeconfig -name capi-quickstart > capi-quickstart.kubeconfig`, and the path was used to set the environment variable `KUBECONFIG`. This made it possible to use the commands provided by kubectl to access cluster objects such as secrets.

At this point, the Workload Cluster was deprovisioned using clusterct, eliminating the Management Cluster using Kind. The first experiment with Cluster API proved successful and opened the door to experimentation on the infrastructure provided by IBM.

3.4 Metal³ and Custom Provisioning Process

Applying the iterative development principles that were being followed, it was decided to simulate a lower level of abstraction than that used with Kind and Docker. At the same time, the objective of this phase of the project was to be able to provision a bare metal machine by providing a custom Cloud Init configuration file. The first task was to set up the *metal3-dev-environment*.

Metal3-dev-enviroment

The Metal³ Development Environment is an open source repository that provides the scripts needed to set up a development environment for Metal³. The primary goal of this environment is to allow developers to test Metal³ by simulating the bare metal infrastructure on virtual machines

(VMs). To use this environment, Metal³ developers provide specific requirements for the host machine; in particular, the use of a bare metal system is recommended, as the environment will create VMs to emulate the hosts managed by Metal³. For the purposes of this thesis, a machine within the IBM cluster was made available for the exclusive use of this and future activities. The machine in question had 192 cores and 503 GB of RAM, allowing clusters consisting of numerous nodes to be simulated. This proved to be of fundamental importance during the testing phases of the developed solution.

The development environment setup process is notoriously invasive, as it reconfigures part of the host operating system and installs numerous packages. At this stage, several choices must be made, such as which operating system to use. By default, metal3-dev-env supports virtual machines running Ubuntu or CentOS operating systems. For the reasons already described in section 2.7, it was decided to use the latter distribution. Deployment can be started by running a series of sequential scripts or, more simply, by using the `make` command. In addition, the `make clean` command is available to clean up the development environment. The scripts that are executed sequentially by `make` are the following:

1. `01_prepare_host.sh`: installs all the software packages needed for the environment.
2. `02_configure_host.sh`: creates the set of virtual machines that will act as emulated bare metal hosts and downloads the disk images needed for Ironic.
3. `03_launch_mgmt_cluster.sh`: launches the Management Cluster via Kind (following a process very similar to that described in the previous section), or via miniKube (recommended when using certain distributions of the Linux operating system), and launches the Bare Metal Operator (BMO) within it.
4. `04_verify.sh`: runs a set of tests to confirm that the deployment has been successfully completed.

These scripts can be modified in their behavior through numerous environment variables [31], thus allowing deployment with very different characteristics; for example, it is necessary to specify the desired version of Cluster API Provider Metal³ (CAPM3) and Cluster API (CAPI). To avoid having to export many variables each time the development environment is created, the documentation recommends creating files that collect them. A best practice is to name these files *<username>.sh* so that they are directly included by Metal³ in the deployment process. Before running the four scripts just presented, we decided to modify the environment variables to simulate six nodes, each with four GB of memory. Once these scripts were executed, the environment was correctly configured and the Management Cluster was ready to provision the Workload cluster. At this point, the *BareMetalHost* objects were visible and in *Ready* status; these objects represent the VMs that emulate the bare metal hardware that will be managed via CAPI and Metal³. Each of these VMs uses Ironic to communicate with Metal³ and execute requests from Cluster API. At this stage, it was possible to see how the BareMetalHost objects contained information about the hardware simulated by the virtual machine, collected by Ironic during the creation phase.

With both the infrastructure and the Management Cluster ready, it was possible to test the provisioning of a Workload cluster. To perform provisioning, the environment provides three scripts capable of creating the cluster, the control plane nodes, and finally the worker nodes. It should be noted that there are also reverse scripts capable of deleting these objects. Running these scripts triggers reconciliation via the control loop. an overview of the objects that are created

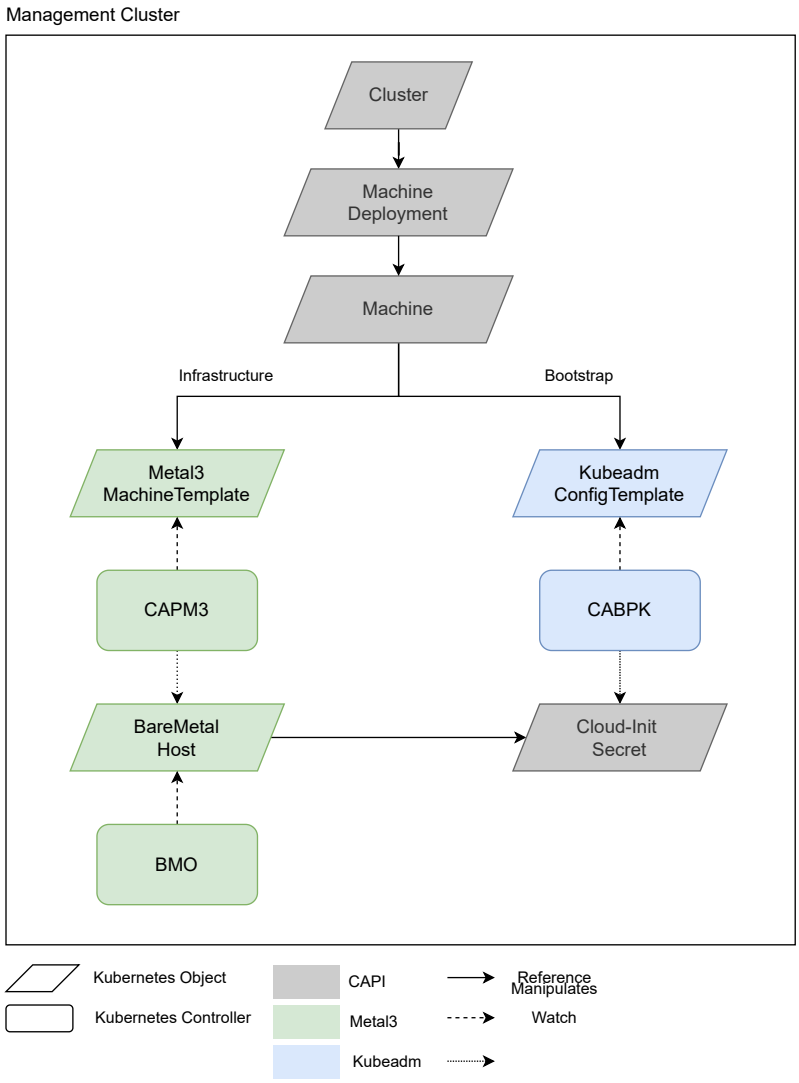

Figure 3.1: CAPI Architecture with Metal³ and Kubeadmin

during provisioning and the relationships between them are shown in Figure 3.1, which shows the default architecture: within it is the most abstract object, namely the Cluster, which references one or more MachineDeployments, whose role has already been described in section 2.3.2. Each MachineDeployment has two fundamental references: one for the infrastructure and one for the bootstrap phase, both managed by CAPI based on the infrastructure requested by the user and the type of configuration that the nodes must have at the end of the provisioning phase. In our case, the infrastructure is managed by Metal³, while the bootstrap is performed using the default technology, namely Kubeadm.

The bootstrap provider, in this case, is the Cluster API bootstrap provider Kubeadm (CABPK),

which reconciles the changes applied by the user to the machine configuration. The reconciliation procedure implemented by the controller requires it to generate a cloud-config containing both the configurations necessary to run Kubeadm and those related to custom configurations. The cloud-config produced is then encoded in base64 and written to a secret, which will be referenced by the machine chosen for provisioning; there will therefore be a different secret for each machine in the cluster.

The provider that handles the infrastructure is the Cluster API Provider Metal³ (CAPM3), which reads the configurations in the template and applies them to the BareMetalHost, i.e., the logical counterpart of the BareMetal machines. In this type of template, the most important information is contained in the *image* field, which must specify: the URL from which the image can be downloaded, its format, and the URL from which the checksum can be downloaded to verify its integrity. The presence of this information determines the start of the machine's provisioning and deprovisioning phases; if the image URL is added to a bare metal host object, the node provisioning phase will start immediately (and vice versa if it is removed). Any change to the Bare Metal Host is reconciled by the Bare Metal Operator (BMO) which, as illustrated in section 2.5.1, uses Ironic to provision the bare metal machine; it is important to emphasize once again that in all the tests carried out, the machine is not really a bare metal machine, but a virtual machine that simulates its behavior.

Provisioning with custom configurations

Once the development environment was successfully launched, the goal became to provision a bare metal host using custom configurations, specifically through the secret containing Cloud Init.

First, we verified that the worker nodes were functioning correctly using the `kubectl get machines` command. Unfortunately, the machines remained in a Pending state indefinitely, a condition that occurs when Kubernetes takes charge of the request but, due to some problem, remains waiting for certain resources. Reading the documentation, the cause of this problem was related to the failure of communication between the kubelet on the control plane node and the kubelet on the worker nodes. To resolve the issue, it was necessary to install a Container Network Interface (CNI), which is a framework that contains a set of libraries written in GO that can manage networking between containers [27]. In order to adapt to the needs of various Kubernetes users, CNIs have been defined as standard interfaces, and can therefore be implemented by different tools. These tools are called CNI plugins and are small programs that fully or partially implement the specifications defined by CNIs. There are several of these tools, but the one recommended by Cluster API is Calico [52]. These plugins are fundamental in the communication process between Pods, and when one of them is scheduled, the Kubelet invokes one of these plugins to communicate with the rest of the Kubernetes cluster. The plugin is responsible for assigning an IP address to the Pod, configuring the network interfaces, and applying specific configurations so that it can communicate. Similarly, when the Pod is deleted, the CNI plugin takes care of freeing up the allocated resources. Once the CNI plugin is installed, by applying a YAML configuration file provided directly by Calico, you can see the relevant Pods running and the machines finally in Ready status.

At this stage, the study of provisioning processes began, so Kubernetes objects were manually modified to attempt to trigger the creation of new machines to be configured as desired. To access the Metal³ cluster, it was necessary to retrieve the `kubeconfig` file from the Kubernetes secret using the command `kubectl get secret <cluster_name> -o json -n metal3`. Since all

secrets are encoded in base64, it was necessary to decode the secret's content in order to use it. Next, the environment variable `export KUBECONFIG` was set to simplify access.

At this point, the first attempt at modification involved cloning an existing machine by copying and pasting it, changing only its name. This attempt caused the new machine to start the provisioning process, but without completing it. The problem encountered was due to the fact that the Machine objects were created, but the provisioning process got stuck in the *WaitingForAvailableMachines* state. Upon further analysis of the error, it was immediately clear that the provisioning process failed both due to the lack of nodes available to accommodate the new machine and the lack of the secret containing the cloud-config configuration file. We then proceeded to study the secrets within the cluster using the `kubectl get secrets` command, which revealed that for each machine there were two secrets named *<machine_name>-networkdata* and *<machine_name>-userdata*. They contained network information and configuration information compiled by CABPK from the template, respectively. At this point, we were curious to understand whether re-provisioning a machine would cause new secrets to be created or existing ones to be reused. If the same secrets were reused, it would only be possible to modify the content of the current ones in order to customize the Kubernetes node configuration. Following this intuition, we manually deprovisioned a node. To do this, we decided to work at a lower level of abstraction than the previous modification attempt and therefore modified the BareMetalHost object with the command `kubectl edit bmh <node_name> -n metal3`. It should be noted that the parameter `-n metal3` is used to indicate to Kubernetes the namespace in which the object to be modified is located.

```
spec:
    image:
        url: http://172.22.0.1/images/centosraw.raw
        checksum: http://172.22.0.1/images/centosraw.raw.sha256sum
        checksumType: sha256
        format: raw
    userData:
        name: nodename-userdata
        namespace: metal3
    networkData:
        name: nodename-networkdata
        namespace: metal3
```

Listing 3.3: Section of a BareMetalHost object

There are numerous attributes within the object, but the only ones relevant at this stage are those listed in 3.3. Following what has already been explained in the previous section, the value of the URL parameter has been removed, leaving the values of the other fields unchanged. Applying the changes triggered the BMO control loop, which immediately began to command the deprovisioning of the node; in fact, after a few minutes, the BareMetalHost was in Available status. Despite the deprovisioning, the secrets related to that node had not been deleted, giving hope that they could be reused. The procedure was followed in reverse to start provisioning the node, but unfortunately, it was realized that the machine being created had a different name than the previous one and that the related secrets were being generated again.

At this point, the previous idea was abandoned in favor of manually creating a secret, which would be referenced by hand when the BareMetalHost object was modified to start provisioning. To create a secret that had the format required by CAPI, kubectl was used, via the command `kubectl create secret generic -n metal3 -type=cluster.x-k8s.io/secret my-secret`.

Subsequently, a very simple cloud-config was created, whose purpose was to check that all the Cloud Init features necessary to configure Slurm were working; among these, the following were verified: the creation of a user, the writing of a file, and the execution of a command. This configuration file was encoded in base64 and inserted into the secret created previously. Once the node was reprovisioned by referencing the new secret instead of the one created by kubeadm, it started correctly, and it was possible to access it via Virsh [57], a command line tool that allows interaction with virtual machines, but not via SSH. After a few changes related to the *networkdata* secret, this problem was solved and it was finally possible to access the machine, demonstrating the feasibility of creating custom Cloud Init scripts, which is essential for studying the BareMetalHost configuration method for creating a Slurm cluster.

It is important to note that manually configuring a node also raised some issues: for example, replacing the configuration file that launched Kubeadm configured the node in question, but it was not part of the Kubernetes cluster, thus making the node invisible and unmanageable to Cluster API. As proof of this, when the cluster was deprovisioned with the command `kubectl delete cluster -n metal3 <cluster-name>`, all nodes began the deprovisioning phase, except for the custom-configured node. Having achieved this result, it was decided to address the issue at a later stage, giving priority to the study of an automatic Slurm configuration which, unlike previous configurations, would have to include more nodes.

3.5 Automatic Slurm Cluster Configuration via Cloud Init

Having gained a better understanding of the mechanisms involved in provisioning BareMetalHosts, efforts were then focused on creating an automatic configuration capable of creating a Slurm cluster, initially static and subsequently able to add and remove nodes as needed.

The initial activities involved modifying the cloud-config file created during the early stages of the project. These changes were necessary because the Ubuntu operating system was used in the initial experiments, whereas CentOS is now used. The change in Linux distribution invalidated some of the previous configurations, as CentOS does not provide the installation packages distributed by APT; therefore, the configurations indicated in section 2.2.3 must be performed manually or using custom scripts. A virtual machine was configured manually in order to identify the configuration steps necessary to create a script that could repeat them automatically. The script in question was able to create the *slurm* and *munge* users and groups, install and configure the two software packages (e.g., creating the Munge key), and start the daemons, taking care to start Munge first. It was decided to create a script as it was more maintainable than the alternatives: it would have been possible to write all the commands to be executed within the `rumcmd` section of the cloud-config, but doing so would have made the configuration file difficult to interpret and modify. The script was therefore tested individually first and then inserted into a cloud-config. Once it was verified that the script configured the node correctly, the new cloud-config was tested within the metal3-dev-environment.

Since the clusters had to be recreated and the environment restarted, it was decided to try modifying the characteristics of the BareMetalHosts. The settings for RAM and CPU count were modified by editing the `config_mtagliani.sh` and `vm-setup/roles/common/defaults/main.yml` files, giving each node four CPUs and eight GB of memory; the number of Kubernetes worker machines was also increased from three to six. Although it may seem counterintuitive, these changes made it possible to study the configuration of a Slurm cluster consisting of two control

nodes and two worker nodes, with the aim of simulating for the first time a hybrid cluster that included both cloud and HPC resources. In fact, as it was not yet possible to correctly configure a MachineDeployment that did not use the default providers, it was decided to configure the entire cluster as Kubernetes and then modify it so that some of the nodes became Slurm. Following the creation of the development environment, the scripts for provisioning the Workload cluster were launched in sequence. Despite an initial delay in provisioning all the machines, probably due to changes in physical characteristics, the cluster, control plane, and workers were still configured as expected. At that point, all worker nodes were configured as Kubernetes nodes, and before proceeding with the configuration change necessary to make them Slurm nodes, SSH access was used to verify that all nodes had the required hardware characteristics.

Following what was done in previous experiments, a new secret containing the cloud-config was created that could configure a single Slurm node, which would interpret both the role of head node and worker node; A reprovisioning process was then started for one of the worker nodes of the Kubernetes Workload cluster, taking care that the `userData` field of the BareMetalHost object referenced the newly created secret. From the very first attempt, it became clear that a critical error had been made: the munge key, which should have been the same on every node, was created differently for each provisioning. The solution adopted was to manually create the key in advance and write it identically on each node using Cloud Init and the `write_files` parameter. It is important to note that this approach was also used to distribute the slurm.conf file. Once this problem was solved, it was possible to say that a BareMetalHost had been successfully configured via Cloud Init. At this stage, the features of the latter software were greatly appreciated, as most of the configurations used in the first phase of the project were reused; in fact, they remained almost unchanged despite the change in the context in which they were applied.

The goal then became to automatically configure not only one node, but an entire cluster. To this end, the four nodes that would be part of it were selected, two with the head role and two with the computing role. Next, the information needed to correctly modify the slurm.conf file was retrieved. The fields that were modified are shown in listing 3.4. From the listing, it is possible to appreciate, albeit in a limited way given the small number of nodes, the syntax that allows the declaration of nodes with the same characteristics to be grouped together. It can also be noted that the declaration of two head nodes does not create any kind of conflict; in these cases, Slurm will follow the order of definition to determine which node is the control node and which is the backup node to be used in case the first one fails.

```
ClusterName=my-first-cluster
SlurmctldHost=node-4
SlurmctldHost=node-5

# COMPUTE NODES
NodeName=node-[9-11] CPUs=4 RealMemory=7674 ThreadsPerCore=2 State=UNKNOWN
```

Listing 3.4: Modified sections of the *slurm.conf* file

The slurm.conf file was modified and distributed via secret and cloud-config to the selected nodes. In this case, two different secrets were created, one for the head nodes, which must launch the slurmctld daemon, and one for the worker nodes, which must instead run the slurmd daemon. However, no provisioning of other types of nodes (e.g., DBD, login, etc.) was planned, as this would only have made the configuration more difficult and would not have been strictly necessary to make the cluster operational. Subsequently, the provisioning of the nodes involved was performed again, and the Slurm cluster was created correctly. Once the provisioning of all machines was

complete, the Slurm control commands were working: `sinfo` allowed the nodes belonging to the cluster to be displayed, while `scontrol show node <node-name>` allowed the `idle` status of the worker nodes to be displayed correctly. To verify that the configuration was working correctly, the `srun -N 2 hostname` command was launched, which in turn launched the command specified as a parameter on the two persistent worker nodes in the cluster. Having correctly returned the name assigned to the two nodes, the cluster proved to be suitable for creating jobs and executing workloads.

MPICH Configuration

Despite the success of the experiment, the Slurm node configuration phase was not complete. As a secondary requirement, the cluster configurations also needed to include the installation of MPICH [33]. MPICH is a high-performance, open-source implementation of the MPI standard, designed for portability and efficiency, and is widely used in HPC clusters, including those used within IBM. The same procedure used previously was followed, studying the installation on a machine already configured and then automating the process using scripts and Cloud Init. The installation process was fairly quick thanks to the presence of software packages that can be easily installed via Yum (the package manager for RHEL and CentOS), while some problems were encountered with the configuration. These problems occurred following a test in which a simple *helloworld* program was created that produced a different greeting message for each machine participating in the computation. Once the program was compiled and launched via `srun`, errors occurred; these were caused by the failure to distribute a copy of the compiled program to all nodes in the cluster, which is a fundamental requirement for MPICH to work correctly. To temporarily overcome the problem, it was decided to share the file manually: however, this immediately highlighted the need for a sharing system more suited to HPC systems, which can comprise thousands of nodes. For this reason, a Network File System was set up to provide the various nodes with a common location in which to share the compiled files and the related results. The shared folder was exported from the head node and then mounted in the home directory of each node. By moving the compiled files into this folder, it was possible to run the program correctly. In this case, again, the configuration process was automated by creating two scripts, both intended for the installation of MPICH, with the difference that the first is executed by the control node and exports the shared NFS folder, while the second, executed by the computing nodes, mounts the folder in question. These scripts, as well as those for configuring Slurm, are written and executed on the respective nodes by Cloud Init during the first boot phase of the machine.

At the end of these activities, it was possible to automatically configure BareMetal machines so that they formed a functioning Slurm cluster; at this stage, however, the configurations are static, i.e., developed with prior knowledge of which machines would be part of the cluster and what roles they would assume. Important questions have been raised about the incompatibility between the rigidity of Slurm configurations and the dynamic nature of Kubernetes and Cluster API:

- The slurm.conf file requires node names to be specified statically, which conflicts with the dynamic allocation of machines in a Kubernetes-managed environment.
- The secret containing the configurations, including the slurm.conf file (which assigns roles to nodes statically), is created before the actual creation of the cluster, raising a potential issue with the dynamic assignment of physical nodes to Slurm's logical resources.

- The MPICH and NFS setup scripts for the compute nodes depend on the network address of the machine exporting the shared NFS folder.

These questions add to the problem presented in the previous section, namely the lack of visibility of Slurm nodes within the Kubernetes cluster.

3.6 Virtual Kubelet convergence exploration

At the current stage of the project, it is therefore possible to configure, albeit manually, bare metal nodes as Kubernetes nodes or as Slurm nodes. However, the ultimate goal is to use the Kubernetes API to control both types of nodes, and to do so, it is clearly necessary for the nodes of both clusters to be visible within the Kubernetes cluster. At this stage, as shown in the image 3.2, the Slurm nodes are not visible because they are not aware that they are in a cloud environment and therefore do not register with the Kubernetes cluster, as the other nodes do.

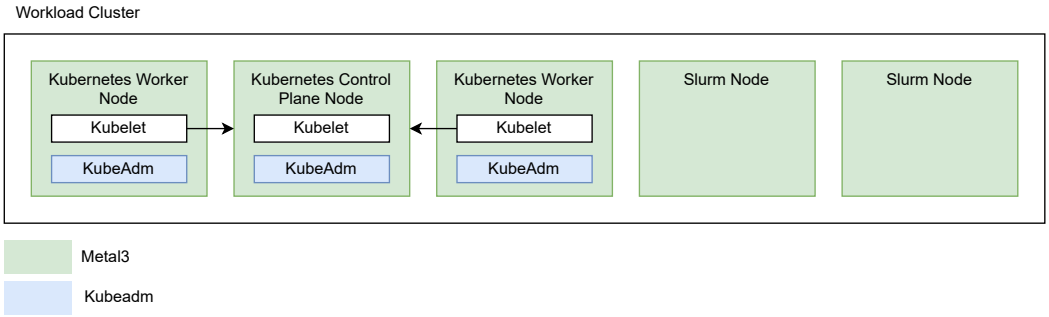

Figure 3.2: Bare Metal Machine roles and communication

The first idea, designed to solve this problem, involved reusing the mechanism used on Kubernetes nodes on Slurm nodes as well. By installing Kubeadm on the HPC cluster nodes, it would have been possible to start a Kubelet, whose main function is to report the presence and characteristics of the nodes. If this idea had been implemented, it would certainly have been possible to correctly register the Slurm nodes within Kubernetes. However, the presence of a Kubelet on this type of node would have irreversibly compromised the separation that we wanted to maintain at the node level. in fact, a Kubelet is responsible for managing the entire lifecycle of Pods, and having one running on a Slurm node would allow a distracted user to schedule Pods on it, thus creating a conflict between the bare metal resources assigned to the cloud environment and those assigned to the HPC environment.

A second intuition, which represents an evolution of the first, concerned the use of special annotations provided by Kubernetes, whose purpose is to prevent certain Pods from being scheduled on particular nodes. In this scenario, the Slurm nodes would have used Kubeadm to join the cluster, as in the previous case, but at that point, a controller would have taken action to promptly modify the node configuration so that it would reject Pod scheduling. In particular, this controller would have used the *Taints* [27] mechanism, which is a property that a Kubernetes node can have, allowing it to reject certain Pods either totally or partially. Usually, this mechanism is used to filter only certain types of Pods, while in this case it would have been used to block scheduling completely. However, the addition of this type of controller, responsible for managing

Taints, represents an anomaly within the CAPI architecture, and for these last two reasons, this path was not pursued either.

To address this issue, an investigation was launched into the functioning of Virtual Kubelet [58], an open-source project that is part of the Cloud Native Computing Foundation and represents an abstraction of Kubelet, specifically designed to mask APIs and other objects from the eyes of a Kubernetes cluster. Unlike a conventional Kubelet, which performs operations on Pods and containers directly on the physical or virtual node on which it resides, a Virtual Kubelet allows the programmer to implement, through the creation of a specific provider, the interface and have custom behavior when asked to perform some operation on the Pods. The goal of the Virtual Kubelet is to achieve a convergence solution for node management, i.e., an approach that aims to unify resource management regardless of their characteristics. In fact, the main use of this special type of Kubelet is to mask serverless environments from Kubernetes nodes, where containers can be deployed without knowing or managing the infrastructure.

The idea was therefore to use the features provided by the Virtual Kubelet project, while foregoing the implementation of the methods responsible for managing requests related to the Pod lifecycle. By doing so, the Slurm nodes would be visible within the Kubernetes cluster, while scheduling requests would not interfere with the HPC workload. To verify the feasibility of the proposal, it was decided to try to register the machine being worked on within the Kubernetes cluster; in principle, any machine could have become part of the cluster. To do this, the project repository was downloaded and its structure was studied. The simplest configuration of Virtual Kubelet requires three configuration files and a provider that implements its behavior. Fortunately, the project provides a test provider, called *mock provider*, which reports the message received to a log file for each scheduling or Pod management request.

In Kubernetes, both communications between the various components and those with the user are protected by TLS; therefore, it is not surprising that the first two configuration files are a certificate and a key, which are necessary for authentication and communication within the cluster. The secret containing these last two files was downloaded via kubectl, immediately decoded, and finally, two environment variables were set: `APISERVER_KEY_LOCATION` and `APISERVER_CERT_LOCATION`. The third configuration file is a JSON file containing the characteristics of the node that will become part of the cluster. The parameters that were set can be found in listing 3.5; among them, it is important to highlight the fact that it is possible to specify a maximum number of Pods that the node will be able to manage. Setting that value to zero would allow Slurm nodes to be isolated from the Pod scheduling phase, maintaining a good level of isolation.

```
{
  "adcpu28":
  {
    "cpu": "2",
    "memory": "32Gi",
    "pods": "128",
    "labels":
    {
        "metal3.io/uuid":"1234"
    }
  }
}
```

Listing 3.5: Virtual Kublet configuration file

After compiling the project using the `make build` command, the `./virtual-kubelet` executable was launched, specifying, via a specific parameter, to use the mock provider. By checking the nodes with the `kubectl get nodes` command, it was verified that `adcpu28` had successfully registered in the cluster with the role of *agent*. This role is nothing more than a label that is assigned to each node that joins the cluster; since it has no operational purpose, it was decided not to investigate how to change this label. In the future, a specific role could be created for Slurm nodes that register via Virtual Kubelet.

The experiment proved successful, validating the effectiveness of the intuition. Once again, an automatic procedure was studied that would allow Cloud Init to bring a node into the Kubernetes cluster using Virtual Kubelet. A script was implemented that could download the source files for the Virtual Kubelet project from the public repository on GitHub and compile them. Once the executable was produced, the script allowed the environment variables to be set and Kubelet to be run using the mock provider. After a series of tests and refinements, the configuration was correctly applied to the node during the provisioning phase, thus creating a cluster like the one shown in 3.3. It is important to note that the script is based on a rather strong assumption, namely that the three configuration files presented above were present on the node. At this point, there was a need for an entity within the CAPI ecosystem that had access to this type of information during the cluster provisioning phase. Nevertheless, the adoption of Virtual Kubelet brought us closer to managing nodes that are part of a non-Kubernetes cluster through CAPI.

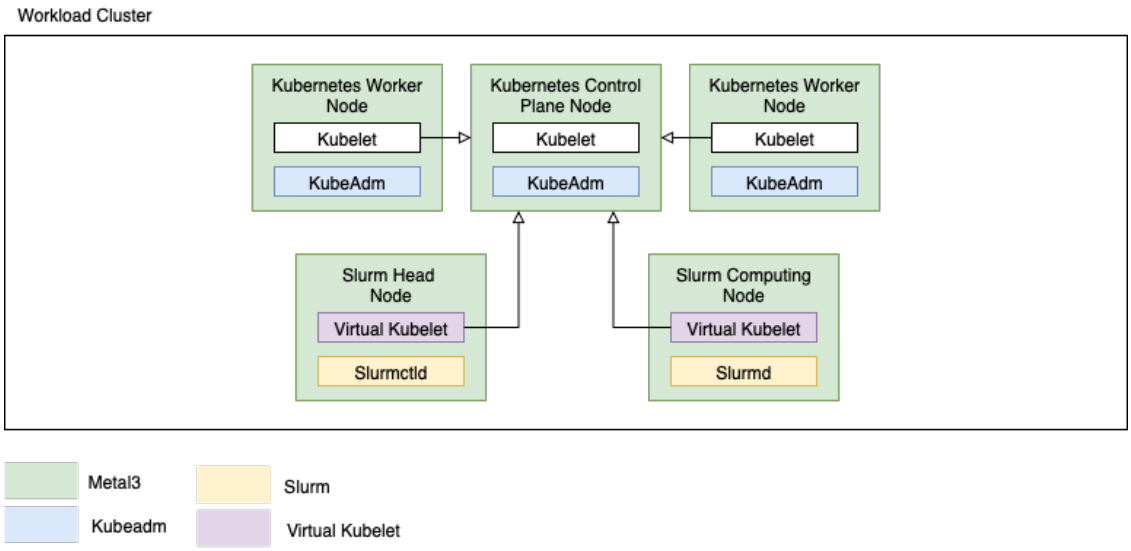

Figure 3.3: Bare Metal Machine roles and communication with Virtual Kubelet

3.7 Slurm Cluster automatic upscaling

Once we verified that it was possible to automatically configure a Slurm node via Cloud Init and that it was possible to make that node visible within a Kubernetes cluster, we still needed to study how to expand a Slurm cluster by adding new infrastructure. Once this question had been resolved, in theory, it would be possible to integrate all these features into CAPI to obtain a tool

capable of configuring and expanding both a Kubernetes cluster and an HPC cluster as needed.

First, we studied the process that an administrator must follow when adding new nodes to a Slurm cluster. In this section, we assume that the node being added to the cluster is a computing node. After installing the operating system and applying the configurations, already discussed in section 2.2.3 and necessary to run Slurm, the administrator must update the slurm.conf file with the characteristics of the node that has been added; this configuration file must then be redistributed to all nodes in the cluster. Updating the file is quite invasive, as the nodes refer to it in order to communicate with the other elements of the cluster. A sudden and non-simultaneous update of the file causes the various nodes to have a different view of the infrastructure and leads to the immediate failure of the workloads they are executing. The Slurm documentation recommends scheduling the addition of new nodes only when it is possible to drain the cluster, i.e., when it is possible to terminate the execution of existing workloads without scheduling new ones. Once the slurm.conf file has been updated, the slurmd and slurmctld daemons must be restarted on their respective nodes, which will reload the new file containing the updated list of available nodes [46].

The main challenge in expanding the cluster as desired was dynamically updating the slurm.conf configuration file, which had to be done every time a machine was added to the HPC cluster. Having successfully added the Slurm nodes to the Kubernetes cluster, we decided to use some of the resources and mechanisms provided by the latter to modify the configuration file on all nodes. For example, we studied Kubelet to see if there was a standard method for providing certain configurations to the node. Unfortunately, this research was unsuccessful, so other solutions were explored based on:

- DaemonSets: objects that allow the same Pod to be launched on each node.
- ConfigMaps: objects that, as presented in section 2.1.1, allow configurations to be provided to programs running on nodes; in this case, Kubelet is responsible for transforming these configurations into files accessible to the node.
- Operator: Kubernetes software extension capable of reacting to what happens within the cluster; it also allows administrators to create custom behaviors without having to directly modify the way Kubernetes operates.

The idea was to create a daemon to be launched, via a special DaemonSet, on each Slurm node in the cluster. The task of this service would be to perform, on behalf of the administrator, the steps necessary to correctly update the Slurm configuration, obtaining the updated configuration file via ConfigMaps. However, the problem of informing the node of the need to change the configuration remained; therefore, it was decided to implement an operator that would be able to react to the provisioning of the machines by creating an updated configuration file and then triggering the distribution mechanism. Go was chosen as the programming language to be used for the development of the operator, in order to remain consistent with the Kubernetes environment, since the latter is also developed with the same language [14].

A preliminary attempt to implement the above idea involved the scheduling of a simple Pod on one of the nodes already configured as a Slurm computing node. The goal of this experiment was to mount a folder in the path required for the slurm.conf file. Although it was possible to schedule the Pod by specifying the name of the node on which it should reside, it was stuck in a *pending* state. This type of status occurs when Kubernetes is unable to satisfy the user's request, generally due to a lack of resources on the selected node. After a little debugging, it became clear

that the Pod was not being scheduled because the Virtual Kubelet of the Slurm node had been configured not to accept it. As discussed in the previous section, the Virtual Kubelet does not accept Pod scheduling in order to maintain isolation between the Cloud and HPC environments; changing this setting would have caused a breach in the separation that we wanted to maintain.

This unsuccessful experiment led to two new insights: the first was that the dynamic update script, instead of being executed within a Pod scheduled on each node, could be launched directly from cloud-config during provisioning and executed in parallel with Virtual Kubelet, while the second was a custom implementation of Virtual Kubelet. This custom implementation would launch the configuration update program in response to the operator's request to schedule a new Pod.

The procedure recommended by Slurm documentation for adding new nodes to the cluster is in stark contrast to the requirements that the solution should have met. It is not possible for a highly dynamic architecture, composed of Slurm and Kubernetes nodes, to wait for the HPC cluster to drain every time in order to modify the configuration. Furthermore, the implementation of methods made available by Virtual Kubelet for purposes unrelated to Pod scheduling was more of a hack than a solution to the problem. Therefore, it was decided to abandon this approach and study alternative Slurm configurations instead of the classic one.

To experiment with alternative solutions, a naive approach was explored in which all nodes belonging to the cluster were included in the slurm.conf file, without distinguishing between those configured as Slurm and those configured as Kubernetes. As expected, some of the nodes actually became part of the Slurm cluster, while others were not considered. Subsequently, the provisioning of a new Slurm node was started to verify whether it was able to join the cluster without having to update the configuration file. Unfortunately, the experiment failed because the node was unable to join the cluster. Then, the *configless* configuration mode was studied, which, as illustrated in section 2.2.3, allows the Slurm controller to take care of the distribution of the configuration file. This centralization allows only the head node to be informed of the changes, delegating the creation and distribution of configurations to the latter. Despite this, in order to retrieve the updated configurations, the computational node must, even in this case, undergo a restart of the slurmd daemon, requiring the cluster to be drained in order to make the changes.

Slinky Architecture and Dynamic Nodes Configuration

Unable to find a method that would allow Slurm and the nodes it managed to be configured dynamically, the open source project Slinky [45] was studied. The decision to study this software was due to the fact that, through it, it is possible to add or remove Pods that simulate HPC nodes. The idea was to reuse the same mechanism to manage bare metal nodes.

Analyzing the documentation, we found a diagram illustrating the architecture created by Slinky. As can be seen in image 3.4, Slinky also uses an operator to observe both Kubernetes resources and custom resources created specifically to run Slurm within Kubernetes. In addition, Slinky defines several CRDs: among these, the main one is *NodeSet*, which manages the Pods that simulate the worker nodes on which the slurmd daemon runs. Further analysis shows that all communications (which travel from within the Slinky cluster to the manager, which runs within the Kubernetes environment) pass through the Slurm API service. Since the manager is the connection point between Kubernetes and Slinky, it was assumed that requests for cluster scaling and the related configuration file update would also originate from it.

Within the documentation, additional alternative methods were found to instruct the Slurm

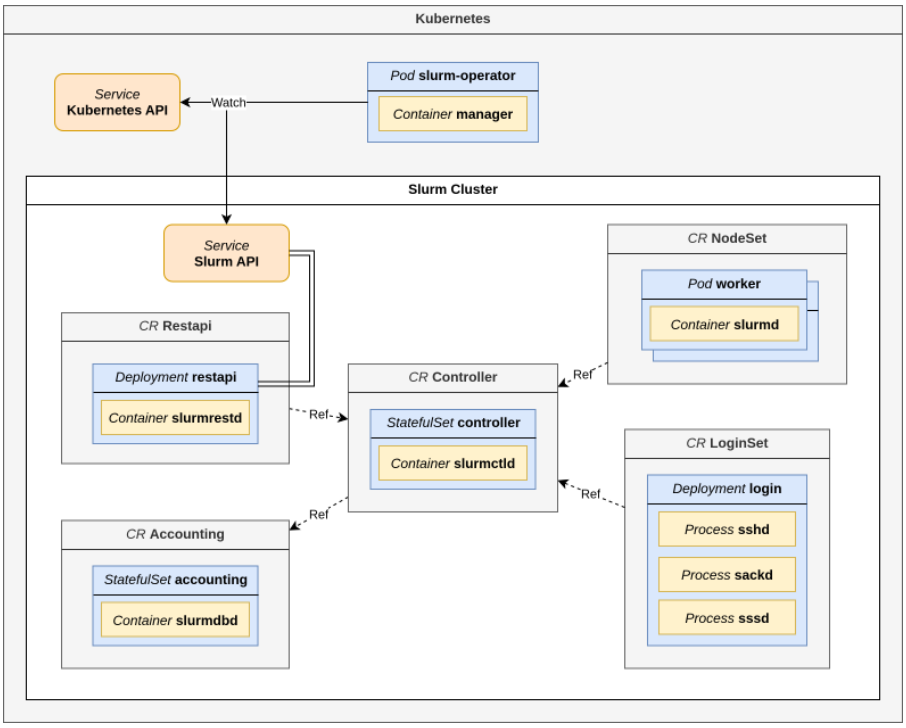

Figure 3.4: Slinky Architecture

node to reload the configuration file. These methods involve sending the `SIGHUP` signal or executing the `scontrol reconfigure` command on the node; the latter command can also be launched in response to an API call from a restd node. Unfortunately, however, none of these methods guaranteed that the workload running on the node would be maintained, so they were unsuitable for dynamically updating the configuration file.

Subsequently, an attempt was made to deploy Slinky in order to compare it with the implementation available on GitHub [48]. Examining the Slurm configuration file used by Slinky, it was noted that nodes can be added to the cluster without first being declared. This can happen as long as they communicate their presence to the controller node and have a particular *feature*; this feature is nothing more than a label that can be assigned to each node, and is used by Slinky to recognize which nodes belong to a given NodeSet. The idea that emerged was that Slinky would allow each worker node to register within the Slurm cluster in a similar way to what happens within Kubernetes, receiving an updated version of the cluster structure in response. Analyzing the node code revealed a registration process, but no process for downloading or updating the slurm.conf file; therefore, it was assumed that there was a particular configuration in which Slurm nodes did not receive this file.

Finally, after further study, we learned about the Slurm configuration with *Dynamic Nodes*. These particular nodes can be added and removed from the Slurm cluster without the administrator having to modify the slurm.conf file and without the other nodes in the cluster having to restart the daemon. The discovery of this configuration represents a turning point in the development of the project, as it allows the Slurm cluster to be made scalable without the need for custom Kubernetes operators or special management programs running on the HPC nodes. The

use cases recommended by the Slurm documentation for the use of this type of node are:

- Multiple dynamic clusters, where nodes are added or removed frequently
- Temporary addition of a new node(s)
- Cloud services adding and removing nodes

All three use cases align perfectly with the context under consideration. Slurm developers implemented this type of node so that they could be added to a cluster with a classic configuration. However, in the case being modeled, the entire cluster would have to be composed of this type of node in order to allow for the reconfiguration of each machine. No longer having the slurm.conf file available, Slurm nodes rely on a mechanism based on a dynamic list to understand the cluster structure and communicate reliably; this list is called *alias_list* and is queried by each node every time it needs to communicate with the rest of the cluster. This additional step, which is not necessary with a classic Slurm configuration, results in overhead that Slurm developers consider negligible. However, it is important to note that no studies have been conducted to estimate how much this overhead impacts cluster performance. At this point, it has been determined that this overhead would be acceptable in any case [20].

As in previous experiments, a cluster was configured manually so that the fundamental steps could be understood and the process could subsequently be automated. The documentation was followed and the required changes were made to the configuration file in order to use Dynamic Nodes. The parameter `TreeWidth=65533` was also introduced, a value that disables the fan-out mechanism, which is normally used to optimize communications between nodes. The `MaxNodeCount` parameter was also set, which estimates the maximum number of nodes that can be present in the Slurm cluster; the value of this parameter was set to the number of nodes that make up the bare metal cluster, whether they are configured as Slurm nodes or as Kubernetes nodes. Finally, similar to what happened in Slinky, the `NodeSet` and `Feature` parameters were specified to select which nodes would be part of the cluster. After modifying the slurm.conf file, the configuration was activated by stopping the daemons, copying the new configuration file to all nodes, and restarting the slurmd and slurmctld daemons. Checking the status of the cluster using the `scontrol snow nodes` command showed that it had no worker nodes; therefore, the process for adding a dynamic node to the cluster was followed. Slurm provides two different methods for performing the addition operation:

- through dynamic registration: using the command sudo `sudo slurmd -Z` asks the computing node to register within the cluster. The command also includes additional parameters such as `-conf`, which is useful for declaring certain hardware characteristics of the node; if this parameter is not specified, Slurm uses the information at its disposal to understand the available hardware. This information is obtained using the command `slurmd -C`.
- using the *scontrol* tool: using the command `scontrol create NodeName=<node_name>`. In this case, however, the node is not registered when the command is executed, but only its presence is declared. By doing so, it is possible to declare the presence of multiple nodes even if they are turned off or unreachable. The declared nodes will only be effectively added to the cluster when the Slurm daemon is run.

Since there is no need to register worker nodes before they start, we chose to use the first of the two methods. Furthermore, imagining that we would have to automate this task, it seemed

more consistent, compared to what already happens in the Kubernetes cluster, for the worker node to be added to the cluster once provisioning is complete. The command `sudo slurmd -Z -conf "Feature=my-worker"` was then launched on a node configured as a Slurm worker but not part of the cluster. It should be noted that the `Feature` parameter was set consistently with what was declared in the slurm.conf file. Indeed, after launching the command, the worker node appeared within the cluster structure. Furthermore, the hardware that was indicated as present on the node was correct, demonstrating Slurm's ability to accurately retrieve such information.

At this point, the process of deregistering a node dynamically added to the cluster was studied. To perform this operation, it is essential that the node is in `IDLE` state and not involved in any reservations. To be in that state, it must not be performing any type of workload at that moment; therefore, it is often necessary to perform a drain using the command `scontrol update state=drain nodename=<node_to_change> reason=<reason>`. In this case, unlike the registration phase, draining is not a problem, as only one node is being stopped and not the entire cluster. Once the node has been freed, it can be removed from the cluster using the `scontrol delete nodename=<node_name>` command. By executing this command on the previously dynamically added node, it has been successfully removed from the cluster.

The integration of these changes with Cloud Init proved to be successful. In fact, a dynamic registration command was added to the worker node configuration scripts. This allowed them to configure themselves correctly and register independently with the Slurm cluster during the provisioning phase. This eliminated the need to manually update the slurm.conf file, solving the problem of dynamic worker node management. Of course, it is important to note that at this stage the provisioning was still performed manually.

3.8 Cluster API and Virtual Kubelet integration

Having verified the feasibility of dynamically configuring a Slurm cluster and made the nodes visible within the Kubernetes cluster using Virtual Kubelets, it became necessary to formalize and automate the configuration process within the Cluster API ecosystem, eliminating the need for manual configuration. It was therefore decided to implement a bootstrap provider capable of replacing the Cluster API Bootstrap Provider Kubeadm (CABPK), used by default by CAPI, with one capable of providing the configurations as shown in the previous sections. The new provider would be based on Virtual Kubelet, so the name Cluster API Bootstrap Provider Virtual Kubelet (CABPV) was chosen. The goal was to extend the CAPI architecture shown in Figure 3.1 to that shown in Figure 3.5.

As can be seen from the diagram of the architecture to be implemented, in order to extend Cluster API, we needed to create a new Kubernetes object and a controller capable of generating cloud conf files. The combination of these two elements in Kubernetes is called an operator, and in order to implement it in a simplified manner, we opted to use the Operator SDK [39]: It consists of a series of Go tools and libraries that allow us to automate part of the development process by automatically generating part of the code. Adopting this tool has incredibly sped up the development process, as it has avoided having to develop code that uses low-level APIs and having to manually create repetitive parts of the code, the so-called boilerplate code.

In developing this part of the project, a portion of the operator implementation produced by the IBM team was used; however, some of the code was rewritten in order to better study its behavior and extend it so that it could support all the configurations developed previously. At this

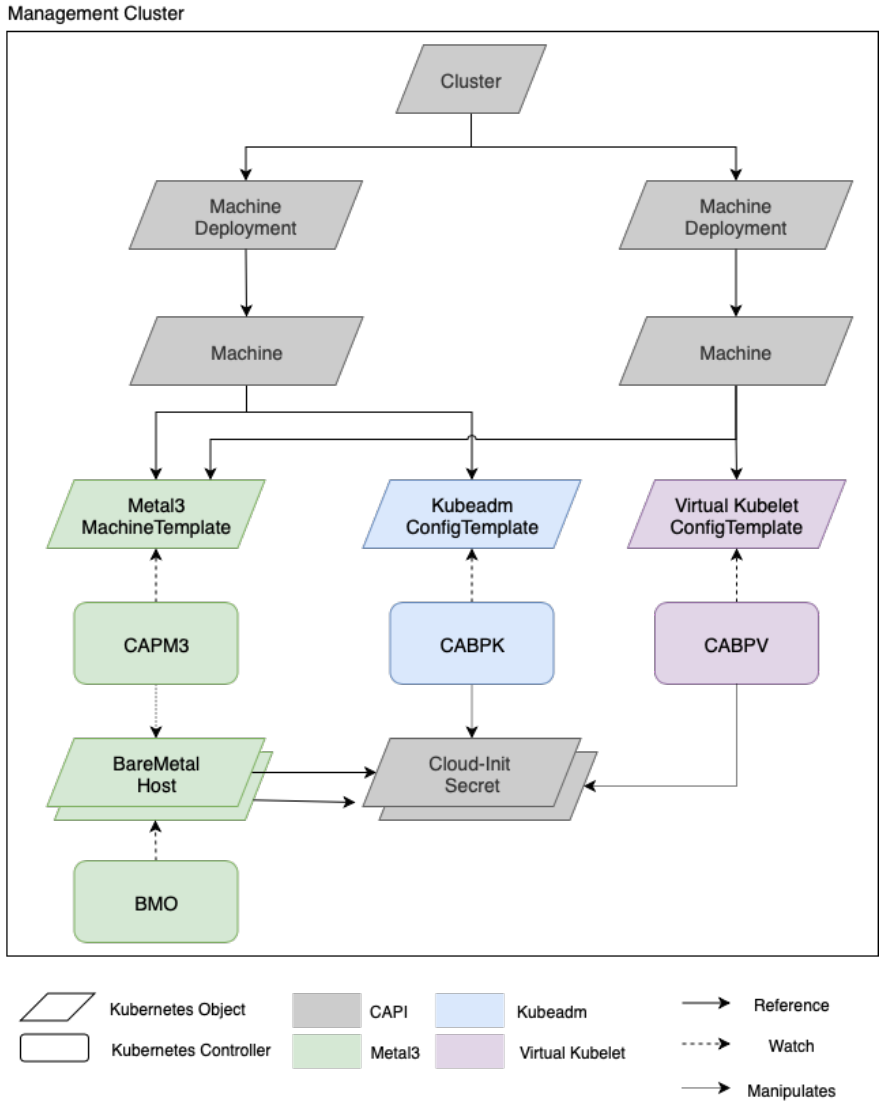

Figure 3.5: CAPI Architecture with CABPV

point, the already implemented project was downloaded and the steps necessary to obtain the same result were studied. First, the project was initialized using the `operator-sdk init` command, which creates all the script, manifest, and configuration files needed to create the operator. Next, the `operator-sdk create api` command was used, which generates a scaffold, i.e., a preliminary structure in which the programmer can define the properties of the CRD being developed. The resource created was named *VirtualKubeletConfig* and consisted of the following fields: the users to be created on the node, the files to be written during configuration, and the type of file to be produced by the controller. Once the fields that will make up the resource status have been defined, the commands `make generate` and `make manifest` must be invoked: the first generates the part of the code common to all operators, which must be integrated by the programmer so that the custom behavior is implemented; the second command produces template files, in yaml

format, which describe the new resource to Kubernetes. Among the files generated, a controller was created, but it lacked the implementation of the `Reconcile` method, a fundamental method as it is invoked by the control loop every time the state requested by the user differs from the current one. After checking that all the steps up to this point had been carried out correctly, an experiment was conducted by implementing the Reconcile method, which was required to write a log every time it was invoked by the control loop.

To perform this experiment, Kubernetes was instructed of the presence of this new resource; luckily, even in this case, Operator SDK provides a script that can be invoked using the `make install` command, which applies the manifests created in the previous steps. It is important to note that, in order to proceed with this operation, certain permissions must be granted. Using the `kubectl get crd` command, it was possible to see that Kubernetes had indeed become aware of the new resource; the controller was then started using the `make run` command and a *MachineDeployment* was applied. The latter specified Metal³ as the InfrastructureProvider and the newly created one as the bootstrapProvider. The provisioning phase started successfully, and the custom controller began writing to the log, demonstrating that the previously defined reconciliation cycle was working. Unfortunately, however, the provisioning process was unable to complete because the Bare Metal Operator (BMO) was waiting for the bootstrap provider to generate the secret containing the instructions for configuring the node. As you can imagine, the newly developed operator was not currently able to generate this secret. Once this experiment was complete, it was clear how to use the Operator SDK; at this point, the implementation provided by the IBM team was explored in greater depth. The implementation of the Reconcile method was studied first, as it had several distinctive features. Before performing any operation, this method waits for the infrastructure to be ready so that it can correctly configure access permissions to the secret. This prevented some BearMetalHosts from referencing a secret containing configurations designed for other nodes. Once the permissions were correctly set, the controller read the configurations provided by the user via the CRD and extended them to include the instructions necessary to run the Virtual Kubelet. As already presented in section 3.6, these configurations include a certificate, a key, and a JSON file: these three files are added to the `files` section of the cloud-config produced in the output. Among these files, the JSON file was studied with particular attention, as it lacked the explicit indication of the hostname, a parameter necessary for the node to run the Virtual Kubelet and join the Kubernetes cluster. In fact, unlike the configuration file presented in listing 3.5, here the hostname had been replaced by what appeared to be a reference to another source of information, as can be seen in listing 3.6.

```
{
  "{{ ds.meta_data.local_hostname }}":
  {
    "cpu": "2",
    "memory": "32Gi",
    "pods": "0",
    "labels":
    {
        "metal3.io/uuid":"{{ ds.meta_data.uuid }}"
    }
  }
}
```

Listing 3.6: Virtual Kublet configuration with template variables

The creation of this type of file was an open point in the experiments carried out in section

3.6. By manually checking the secret generated by the provider at the end of the provisioning phase, it was possible to see that the file in question no longer contained the template variables, but the correct value for the node that had been provisioned. To investigate further, the Go debugger was used to understand, step by step, how this type of information was resolved within the reconciliation cycle. The analyses carried out revealed that it was not the reconciliation cycle that resolved the variables, but that it was CAPI itself, in a subsequent phase, that set the correct value. In particular, it was discovered that the information used to compile the template comes from a secret called `<node_name-metadata`, which in turn is created from an object called *Metal3Data*, containing certain characteristics of the machine that was chosen to host the node.

In the next phase, some provisioning tests were performed using the new provider. First, a new MachineDeployment was created, similar to those used to provision Kubernetes nodes, but replacing the *Kubeadm ConfigTemplate* with the *Virtual Kubelet ConfigTemplate*. Some simple configurations were added to the latter, such as writing files containing some template variables. Following the same steps outlined in the previous paragraphs, Kubernetes was notified of the presence of the new operator, and provisioning was started by applying the MachineDeployment. In doing so, it was discovered that the template variables are kept intact by the provider and can therefore also be set by the user via the Virtual Kubelet Config. This last aspect, combined with the possibility of obtaining information about the physical machine through the Metal3Data object, led to some insights into possible solutions that could resolve the dependency issues that still persisted between the configuration file and the physical machines that needed to be configured. At the end of provisioning, the node selected by CAPI had been successfully provisioned and added to the Kubernetes cluster. For the first time, it was possible to apply one of the previously developed configurations automatically and with the help of Cluster API.

3.9 CABPV for Slurm configuration and upscaling

This phase proved crucial as it addressed the remaining challenges that had emerged during the automatic configuration of the Slurm cluster, including the need to manage node IP addresses dynamically and to make custom setup commands executable through the Cluster API provisioning mechanism. At this point in the project, the feasibility of configuring bare metal nodes as Kubernetes or Slurm machines had been demonstrated. In addition, the bootstrap provider had been implemented, making non-Kubernetes nodes visible in the Kubernetes cluster using the Virtual Kubelet project. The main obstacle therefore remained the automation of Slurm configuration in a dynamic environment, given its inherent rigidity in requiring static network names and addresses. The configuration files that were still dependent on the machine to be provisioned were:

- Configuration of the distributed file system of the Slurm head node: the script in question needs to know the IP address of the node and the mask to be used.
- Configuration of the Slurm nodes: the IP address of the control node must be declared in the slurm.conf file. This setting must be present even if a configuration with dynamic nodes is used.
- Configuration of the distributed file system of the Slurm worker node: in order to access the shared folder, the script requires knowledge of the IP address of the head node.

Before proceeding with the search for a solution to these problems, it was decided to test that all configuration files were written correctly within the cloud-config produced by the new bootstrap provider. The Virtual Kubelet Config created in the previous section was then integrated, adding both the configuration files necessary to create a Slurm controller node and the commands that must be launched during provisioning. Checking the secret that was created following the application of the manifest, a significant error was found: although the Custom Resource Definition provided a section for entering custom commands, this was systematically overwritten by the controller with the commands necessary to start the Virtual Kubelet; this problem was preventing the scripts needed to set up Slurm and the distributed file system from running. So, we debugged and modified the Virtual Kubelet Config controller. The solution adopted was to alter the behavior of the controller so that it added the user-defined commands to the standard commands required to add the node to the cluster: this modification allowed the successful execution of a first, partial deployment of a Slurm node with the help of the new bootstrap provider.

During this phase, the use of template variables was also tested to resolve some of the aforementioned issues. Specifically, the script for configuring the distributed file system was modified so that it could retrieve information about the network and the corresponding mask through the Metal3DataTemplate object. The information was successfully retrieved, which solved the first problem; however, the problem of how worker nodes could reliably discover the IP address of the head node remained. Template variables did not solve this problem, as the information that could be obtained through Metal³ was limited to that concerning the single node.

One of the ideas explored during this period to solve the problems presented was the creation of a new controller, which would be responsible for managing the IP addresses of the Slurm nodes. Specifically, the controller would monitor the cluster and, whenever it was necessary to provision a Slurm controller node, it would store the IP address of the machine on which it was to be scheduled. It would then share this information with the other nodes by modifying the Metal³ objects. The main problem with this idea was that, if a head node failed, the computational nodes would have no way of receiving the new address of the node assigned to replace the deleted one. Furthermore, the creation of a new controller was not justified by the task it had to perform. Usually, a controller is created to manage fundamental operations within the cluster lifecycle, whereas in this case, the controller would only have managed IP addresses for Slurm nodes.

At this point, after a research phase, tools were studied that allow one or more IP addresses to be reserved within a network. In particular, we began to analyze the Keepalived software [24], which provides load balancing and high-availability services to Linux-based systems and infrastructures. Among the many features offered by the software, one of the most important is the implementation of the Virtual Router Redundancy Protocol (VRR). This protocol ensures a routing path even if part of the network infrastructure, such as the default gateway, fails. To achieve this, Keepalived allows one or more IP addresses, called *Virtual IPs*, to be reserved and assigned to one or more machines. This way, if one node fails, another one is immediately ready to take its place to ensure the network continues to function. The Virtual IP is not linked to any physical network card, so the idea was to use this tool to reserve a special IP address, which would be chosen as the IP address of the Slurm cluster control node. By convention, the last available address within the network was chosen, and with this solution it was possible to set the address of the head node within all configuration files before the actual provisioning of the cluster.

At this point, manual tests were performed to correctly configure the software and verify its functioning. Once the installation was complete, a configuration file was created, shown in 3.7; this is used by Keepalived to reserve the address and set certain variables. Once the program was

launched with the command `sudo systemctl start keepalived`, it was possible to verify that the node was actually responding to the specified address. The software configuration process was then automated and, once its operation had been verified with Cloud Init, it was added to the MachineDeployment for the Slurm control nodes. Provisioning was then carried out via Cluster API, verifying that the automatic configuration of Keepalived had correctly reserved the specified IP address.

```
{
  vrrp_instance VI_2 {
    state MASTER
    interface eth1
    virtual_router_id 2
    priority 255
    advert_int 1
    virtual_ipaddress {
      192.168.111.248
    }
  }
}
```

Listing 3.7: Keepalived configuration file

Compared to the insights explored before using Keepalived, adopting this type of solution solved the problem of IP address management without introducing new elements into the CAPI architecture that were only necessary to manage the network addresses of the Slurm cluster.

Subsequently, the slurm.conf configuration files were updated by setting the controller node address, and then a new MachineDeployment was created, designed specifically to represent the entire Slurm cluster; in fact, it included Metal³ as the infrastructure provider and two different VirtualKubeletConfigs as bootstrap providers. It is important to note that the need for two separate VirtualKubeletConfigs arises from the fact that there are different configuration files based on the role that the nodes will play. This new MachineDeployment involved the creation of a Slurm cluster consisting of a head node and two computing nodes; once produced, it was applied and the bare metal machines were correctly configured. By accessing the head node via ssh, it was possible to verify the actual creation of a cluster capable of managing dynamic nodes and that the two worker nodes had successfully communicated their presence. In addition, all three nodes had activated their respective Virtual Kubelets and were successfully recognized as part of the Kubernetes cluster. The configuration files relating to NFS were also updated immediately so that a complete configuration could be carried out. Once provisioning was complete, the Slurm cluster was reconfigured correctly and, with the distributed file system available, it was possible to launch the program described in section 3.1; it worked, demonstrating that the distributed file system had been successfully configured.

At this stage, the implemented solution was able to use Cluster API to provision bare metal machines in order to create a Slurm cluster. This result represents a first milestone within the project, because it demonstrated the possibility of adapting CAPI to manage non-Kubernetes nodes, while maintaining a high level of isolation. Through all the integrations developed up to this point, it would have been possible, in theory, to already scale the cluster by provisioning new infrastructure; in fact, adding new nodes to the cluster would not have interfered with its operation in any way. We therefore proceeded to scale the number of computational nodes within the MachineDeployment using the command `kubectl edit machinedeployment <machinedeployment_name>`.

At the end of provisioning, the new worker node was successfully added to the Slurm cluster, confirming the scalability of the mechanism.

3.10 CAPI and Slurm cluster downscaling

Having created a solution capable of provisioning and upscaling a Slurm cluster via Cluster API, it was necessary at this stage to study additional configurations that would allow cluster nodes to be removed without affecting the operation of the other nodes.

There was a need for a program to be run within the Slurm cluster that was more complex than the one developed in section 3.1. This program had to be capable of potentially involving all the nodes of the HPC cluster for a long time, in order to verify that the deprovisioning operations did not cause errors. To do this, it was decided to implement a classic distributed computing algorithm, namely matrix multiplication. This type of problem was chosen because: it is common in various fields of science and engineering, it is easy to implement, and the computational complexity of the solution is $O(n^3)$. The latter aspect allows the execution time to be extended by slightly increasing the input provided to the program.

Once the matrix calculation program had been implemented and tested, we moved on to studying Slurm's behavior during the deprovisioning phase. We started with a Slurm configuration with one head node and four worker nodes. First, we attempted to deprovision a worker node by simply changing the number of `replicas` within the MachineDeployment. Once the node had been completely removed, within the Slurm cluster it was still recognized as part of the cluster, but an asterisk appeared next to its name. By analyzing the node status more closely using the `scontrol show node <node_name>` command, it could be seen that it was in the `State=DOWN+DYNAMIC_NORM+NOT_RESPONDING` state. This asterisk therefore indicated that the node was considered available but currently not reachable by Slurm. The next experiment followed the same procedure, but this time a workload was put into execution on the cluster. Again, the removed node was reported as inactive, but it wasn't removed from the cluster; moreover, the workload was aborted due to the sudden lack of part of the infrastructure. The results of these experiments highlight the need for nodes to follow a de-registration procedure before being removed from the cluster. This is because, if the node were to be re-provisioned for other purposes, it could still be part of the cluster when, in reality, it would be performing other tasks.

The use case we wanted to implement required the administrator, once they had decided to deprovision part of the Slurm cluster, to modify the replicas field of the corresponding MachineDeployment. At this point, to ensure the correct removal of the node from the cluster, the classic deprovisioning process must be paused until the node draining phase is complete. First, the Slurm documentation was studied to understand how draining should be done in environments with dynamic nodes.

The draining phase can be performed in two ways:

- Graceful: using the command `sclontrol power down asap <node_name>` or `sudo scontrol update NodeName=<node_name> State=DRAIN Reason=<reason>`. The node stops accepting new Job Steps and reservations, but completes the tasks assigned before becoming unavailable.
- Forceed: using the command `sudo scontrol power down force <node_name>` or `sudo scontrol update NodeName=<node_name> State=DOWN Reason=<reason>`. The selected node immediately becomes inaccessible and causes any workload it was executing to be aborted.

Once the draining phase is complete, a node added dynamically in Slurm can be deleted using `sudo scontrol delete nodename <node_name>`.

3.10.1 Cluster API scale down mechanism

Once we understood the steps necessary to correctly remove a dynamic node from the Slurm cluster, we needed to introduce them into the scale-down mechanism adopted by Cluster API and Metal³. As explained above, the process starts with modifying the replicas field of a MachineDeployment object, followed by a chain of reconciliations based on controllers and CRDs.

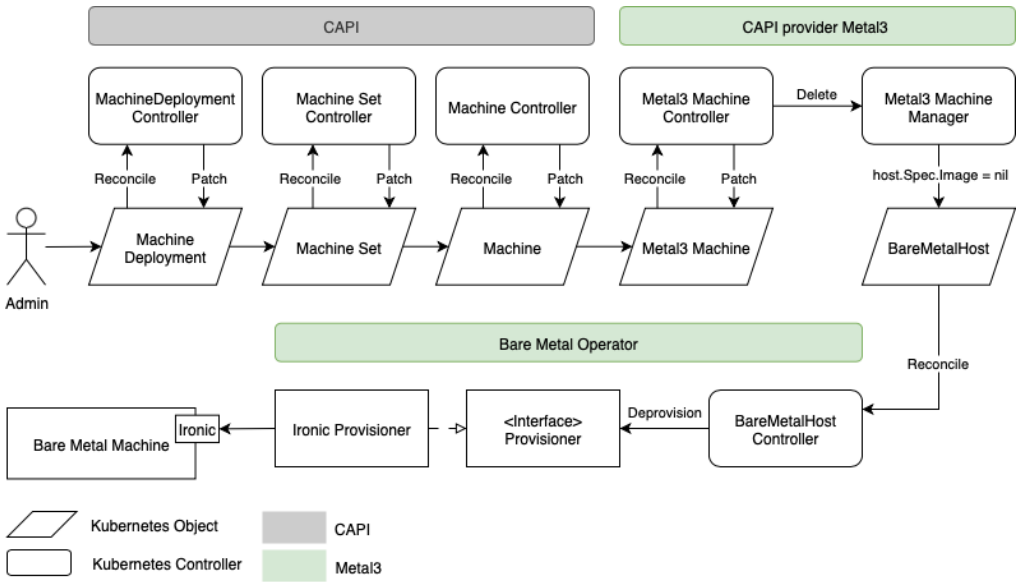

Figure 3.6: Deprovisioning process

This chain, shown in image 3.6, has been analyzed in detail; its stages are as follows:

- The administrator modifies the MachineDeployment object by decreasing the value of the replicas field.
- The reconciliation cycle of the *MachineDeployment controller* is invoked, which checks the status of each machine that is part of the deployment. Once the need to change the number of machines has been established, this controller replaces the old MachineDeployment with an updated one; this operation is called *Patch*.
- The MachineDeployment patch operation triggers the reconciliation cycle of the connected MachineSet. The relevant controller will be responsible for checking whether it is necessary to increase or decrease the number of nodes and then modifying the MachineSet by indicating a `MachineSetScalingDownCondition` Condition.
- As before, the *Machine controller* reacts to the MachineSet Patch and modifies the Machine object, setting the `DeletionTimestamp` attribute.

- The *Metal3machine controller* intercepts this change and understands that the machine must be deprovisioned; therefore, it invokes the `Delete` method of the *Metal3 Machine Manager*, indicating which machine is selected.
- The Metal3MachineManager removes the values of the following attributes from the BareMetalHost object associated with the selected machine: Image, UserData, networkData, metaData, and customDeploy. During the testing of files for the automatic configuration of Slurm nodes, this operation was performed manually to trigger the deprovisioning.
- At this point, the actual deprovisioning of the bare metal infrastructure begins, carried out by the Bare Metal Operator. In particular, the *BareMetalHost controller* uses the `Deprovision` method defined within the *Provisioner* interface.
- The concrete Deprovision method, implemented by the *Ironic Provisioner* class, uses the restful API, made available to Ironic, to communicate with the ironic python agent, in order to perform deprovisioning and node cleanup.

The idea that came out of this analysis was to re-implement the Provisioner interface. With a new provisioner, it would be possible to stop the deprovisioning phase so that the necessary operations could be performed to correctly remove the node from the Slurm cluster. However, the provisioner would not have been able to directly control the node since it was not running on it; therefore, there was a need for an entity capable of actually removing the node from the cluster. The idea was to design a daemon capable of removing the node on which it was active; this operation would only be performed in response to a deprovisioning request made by the new provisioner.

Adopting this solution would have allowed a custom feature to be added to the deprovisioning process without substantially modifying it. Furthermore, it would have been possible to extend the existing provisioner, thus avoiding code duplication. It was therefore decided to try this type of solution, outlined in the image 3.7.

3.10.2 Slurm Detach Handler Implementation

Before implementing the custom Provisioner, an application called *Slurm Detach Handler* was implemented, which is necessary for draining and removing nodes from the Slurm cluster. It was structured as a client-server application implemented in Go, a language chosen for its compatibility with the other elements of the architecture.

The first component of the system, i.e., the server, was designed to run as a daemon on each Slurm node. It acts as an HTTP server listening on a specific port, waiting for requests; when contacted, it uses the endpoint that the client used to invoke the service in order to understand which draining mode it should execute. Once draining has been performed using the requested mode, the server proceeds to remove the node from the Slurm cluster.

The second component of this architecture is the client, which is a simple Go program that takes two parameters: an IP address and a draining mode. The task of this program is to make an HTTP GET call to the IP address of the machine that needs to be removed from the cluster, using the correct endpoint for the draining mode requested by the user.

Once both components had been implemented, a test was carried out: the server was manually installed on the Slurm computational nodes and, subsequently, a deletion request was made via the client. This verified that the nodes were effectively drained and removed from the cluster

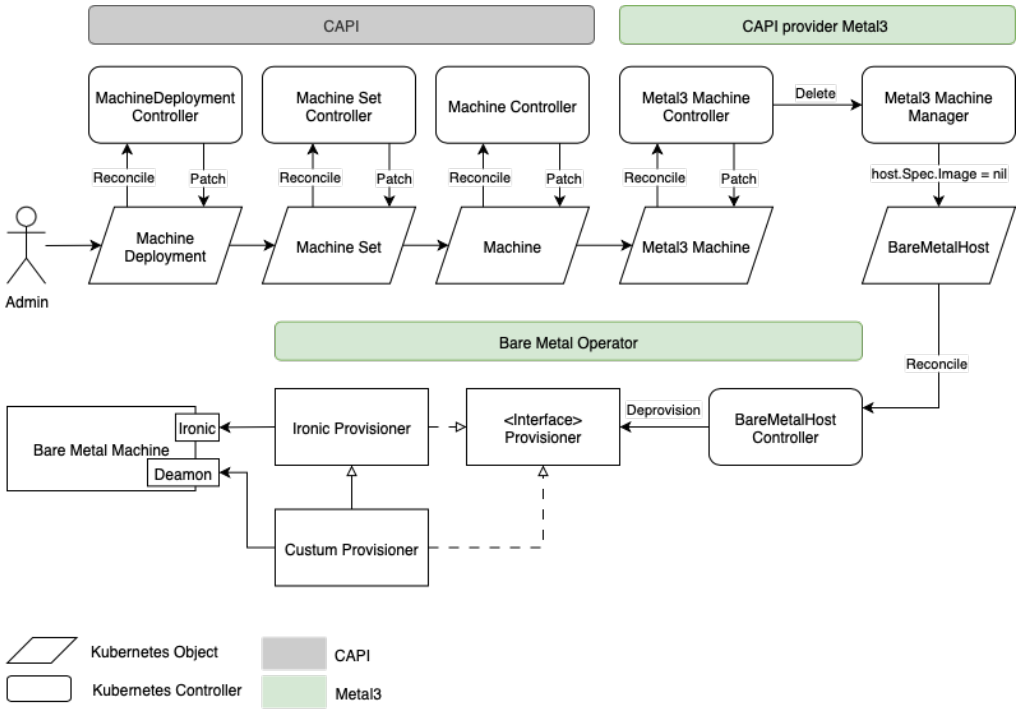

Figure 3.7: Custum deprovisioning process proposal

according to the available methods. Once this experiment was complete, the server installation was integrated into the cloud-config used to configure the computational nodes, so that the program would launch when the node was first started. It is important to note that, for Slurm control nodes, it would be necessary to implement a server that performs a *takeover* operation, i.e., instructing the head node to transfer control of the cluster to another node. After consultation with the IBM team, it was decided not to implement this type of functionality at this time, focusing solely on the management of computational nodes.

3.10.3 Custum Provisioner Implementation

Once this stage was reached, a mechanism had been created that could effectively remove Slurm nodes from the cluster; however, it was still necessary to understand the exact point at which the Bare Metal Operator could be modified in order to use a custom provisioner. In order to effectively extend the Ironic provisioner, it was essential to acquire the ability to run the Bare Metal Operator in debug mode. The `metal3-dev-env` repository, used to create the development environment, provides the `BMO_RUN_LOCAL` environment variable, which, when set to `true`, allows the BMO to be run directly on the host, rather than as a Pod within the Management Cluster. The environment was further modified by setting the `FORCE_REPO_UPDATE` variable to `false` to prevent the system from repeatedly overwriting local changes to the BMO source code. With the environment reconfigured, the virtual machines simulating the bare metal environment were destroyed and recreated, taking care to launch the operator manually. At this stage, some minor bugs arose within the metal3-dev-env automatic configuration scripts; these bugs were probably caused by the community's lack of use of this type of feature. Once corrected, it was possible to

recreate the virtual machine cluster. Finally, the debugging process was handled using *Delve*, Go's native debugger, connecting it directly to the operator's process.

We therefore began developing the new provisioner as shown in the diagram 3.7. The Go language does not provide classic inheritance mechanisms (such as those found in Java), but requires the use of mechanisms that exploit interfaces and embedding. In particular, embedding is a technique that allows data structures and interfaces to express the composition between types. In this case, the custom provisioner, called *myCustomProvisioner*, exploits embedding by incorporating an instance of the *ironicProvisioner* class. By doing so, it is possible to exploit the methods already implemented by redefining only some of them.

As a first experiment, within the myCustomProvisioner class, the `Provision` and `Deprovision` methods were redefined, adding the writing of a trivial log before invoking the same method of the ironicProvisioner class. This verified that the chain of calls used for provisioning and deprovisioning continued to function despite the presence of a small portion of custom code. Subsequently, it was necessary to modify the provisioner used by the Bare Metal Operator, instructing it on the presence of myCustomProvisioner; the BMO uses the factory pattern to obtain a provisioner whenever the BareMetalHost controller needs to perform a reconciliation. The `NewProvisioner` method was then modified to return an instance of myCustomProvisioner.

To make the changes effective, it was necessary to recreate the VM cluster managed by metal3-dev-env, as simply restarting the Bare Metal Operator did not trigger the recompilation of the project, but simply created a new instance without the changes made. It was therefore necessary to speed up this process to facilitate debugging and relaunching the operator after each change; therefore, a new command was created in the BMO's `Makefile`, called `run-debug`. This command handled both the compilation of the code with the flags necessary for debugging and the launch of the operator. Subsequently, this command was inserted into the `/hack/run-bmo-loop.sh` script, whose task is to ensure the presence of the BMO. This allowed the BMO to be updated quickly by simply killing the program when new changes were made; the absence of the Bare Metal Operator immediately triggered a new compilation and execution of the most up-to-date version possible without having to recreate the cluster of virtual machines each time.

Once the correct functioning of the call chain, which now passed through the custom provisioner, had been tested, the client logic developed in the previous section was inserted into the deprovisioning process. However, errors were found caused by the lack of information essential for the correct functioning of the provisioner. In fact, the information available to the provisioner did not include the IP address of the node that was to be removed from the Slurm cluster. To quickly resolve the issue and arrive at a first working version of the provisioner, an attribute containing all the information about the host that was about to be removed from the cluster was introduced into myCustomProvisioner. This attribute was set during the creation of the provisioner by performing a `DeepCopy` operation: in this way, it was possible to obtain the missing information and test the solution.

Having a complete copy of the machine's characteristics was certainly effective, but clearly inefficient, because the only information needed was the IP address. Therefore, it was necessary to create a second version of the NewProvisioner method, within which the network address of the machine's only network card capable of communicating with the outside world is selected. By doing so, the overhead of the new deprovisioning function was reduced, albeit modestly.

The code for the deprovisioning function, extended by the new provisioner, is shown in listing 3.8.

```
func (p *myCustomProvisioner) Deprovision(restartOnFailure bool) (result
    provisioner.Result, err error) {
        nodeIP := p.hostIP
        mode := "detach"
        client := &http.Client{}
        msg := fmt.Sprintf("detaching node %s from slurm with mode %s",
            nodeIP, mode)
        p.log.Info(msg)

        url := net.JoinHostPort(nodeIP, "8090")
        url = fmt.Sprintf("http://%s/%s", url, mode)
        ctx, cancel := context.WithTimeout(context.Background(),
            1000*time.Second)
        req, _ := http.NewRequestWithContext(ctx, http.MethodGet, url,
            http.NoBody)
        resp, err := client.Do(req)
        defer cancel()

        if err != nil || resp.StatusCode != http.StatusOK {
                p.log.Error(err, "Cannot detach")
                return p.ironicProvisioner.Deprovision(restartOnFailure)
        }

        defer resp.Body.Close()

        p.log.Info("Detached")
        return p.ironicProvisioner.Deprovision(restartOnFailure)
}
```

Listing 3.8: Custum Provisioner Deprovision Method

At this point, a test was carried out by provisioning a Slurm cluster consisting of a head node and six computational nodes. Provisioning was much faster than manual provisioning, as it was possible to take advantage of the cluster creation mechanism developed previously. To obtain a functioning Slurm cluster, it was sufficient to apply the manifests developed in section 3.9, taking care to modify the number of replicas. Once the cluster was correctly formed, the number of worker nodes was reduced to start the deprovisioning process. By monitoring the Slurm cluster, it was possible to see how the nodes were being correctly eliminated. Although the implemented solution was working, it was immediately apparent that the hybrid context in which the Slurm nodes would be located had not been managed in any way; the deprovisioning process that had been implemented within the Bare Metal Operator would be performed for both Slurm and Kubernetes nodes. This situation would certainly have caused problems if Kubernetes nodes had been deprovisioned, as they do not perform node draining. To try to solve this problem, an idea was formulated that involved using labels associated with each machine to distinguish which nodes needed to be notified before deprovisioning. However, this idea was not implemented as it was decided to explore alternative solutions.

3.10.4 Alternative Approaches and Custom Controller Implementation

Following an analysis of the implementation, several critical issues were highlighted regarding the proposed solution. Although the solution was clean from an implementation point of view, part of the architecture was modified, which would have resulted in a different Bare Metal Operator for each type of cluster within the infrastructure. This aspect was not a problem in the use case covered by this thesis, which assumes the presence of only one Slurm cluster and one Kubernetes cluster, but it would have made the solution difficult to scale in environments where more than two types of clusters are present at the same time. Therefore, it was decided to abandon this solution and evaluate alternatives.

The first idea was to use Virtual Kubelet. Specifically, it could receive requests to remove the node from the cluster by implementing one or more methods provided for Pod scheduling. This idea was quickly discarded because it would have exploited interfaces with a well-defined purpose for custom behavior that did not concern Pods. Furthermore, even in this case, it was unclear who could communicate with Virtual Kubelet within the deprovisioning process.

Therefore, it was decided to investigate backwards from what had been done previously, and thus to search for the point in the deprovisioning chain at which the node to be deleted was selected. If the object that first becomes aware of the machine selected for deprovisioning had been found, it would have been possible to design a mechanism to discriminate between the various types of nodes and choose the most suitable deprovisioning method. The object in question was then searched for: it is the Machine object, which is managed by CAPI and determines which node is sacrificed in the scaling down process.

Based on this discovery, the second idea was to stop the deprovisioning process when the machine was selected. In this way, while the deprovisioning process was blocked, a new operator would have to determine whether or not the machine needed to be informed of its imminent deletion. Although this solution worked in a similar way to the one presented in section 3.10.3, it did not require any changes to the Bare Metal Operator, but was added as an extension to Cluster API. Working at a higher level of abstraction, an operator would have been easier to develop, less invasive, and easier to install. As can be seen in the image 3.8, the project includes the *Slurm Operator* within the Management Cluster; this operator was designed to be able to communicate with the nodes that form the Workload Cluster. The Slurm Detach Handler service, presented in the previous sections, would be launched on the nodes concerned. It should be noted that, for simplicity, the MachineDeployment relating to the Kubernetes nodes has not been included in the diagram 3.8.

At this point, it became necessary to find a mechanism that would block the chain of deprovisioning calls, allowing the operator to communicate with the nodes. Luckily, Cluster API provides users with a mechanism to introduce custom behavior into the cluster lifecycle; this mechanism is called *Runtime Hook*. Due to their characteristics, hooks can interfere with CAPI's work; therefore, it is important that developers design them carefully and that administrators monitor those that are installed, as they could easily be used for malicious purposes [47]. In particular, the documentation dictates a series of practices that the hook manager, i.e., the entity responsible for custom behavior, must comply with so that it does not interfere with the work performed by Cluster API. The most interesting rules, which could affect the development of the operator, are as follows:

- Timeout: the hook manager must execute the custom behavior as quickly as possible; if it takes more than thirty seconds, CAPI will consider the operation failed. However, it is still

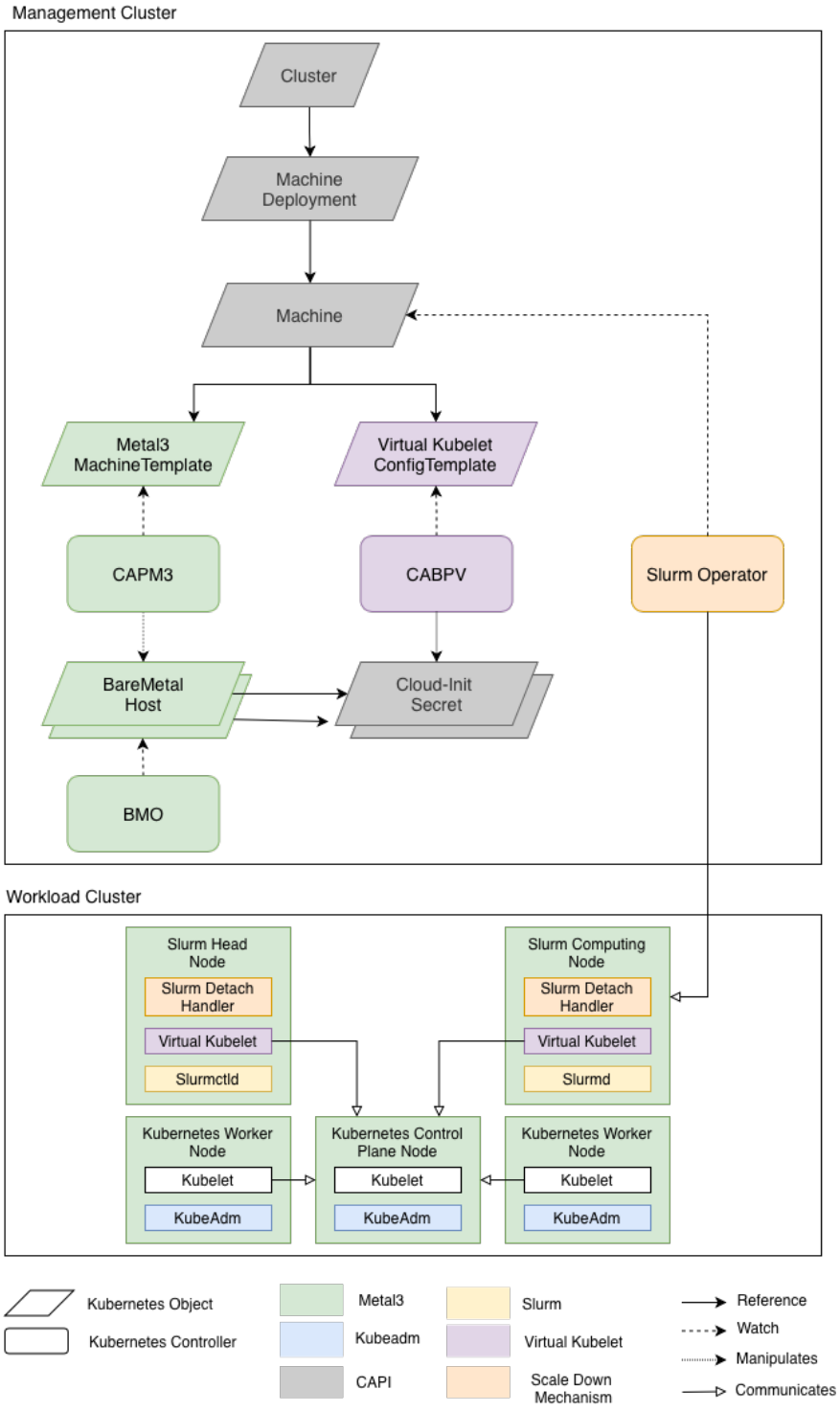

Figure 3.8: CAPI Architecture with Slurm Detach Handler

possible to perform operations that take more than thirty seconds, taking care to execute them in the background so as not to block the Cluster API Runtime.

- Availability: the administrator must keep the hook management systems always active, especially in the case of hooks that block the cluster lifecycle.
- Idempotence: the operations performed by the hook handler must be idempotent because they may be invoked multiple times within the reconciliation cycle.

Among the various hooks available, the one that has been adopted is `pre-drain.delete.hook.machine.cluster.x-k8s.io`, which blocks the Machine Controller immediately after a Machine is selected for deletion by setting the `metadata.deletionTimestamp` attribute. For hooks to be effective, they must be added to the annotations of the Machine object. Specifically, the name of the hook must be specified as the annotation key, while the value can be chosen by the administrator. At this stage, there was a need for an entity capable of setting this value when the Machine object instances were created. The idea was to use the operator that had to be implemented to add the hook to the machines that needed it; in this case, it was decided to do so only for nodes configured as Slurm computational nodes.

The operator therefore had two tasks: during the machine provisioning phase, they had to add the annotation, while during the deprovisioning phase, once they had verified that the machine had to be removed from the Slurm cluster, they had to invoke the Slurm Detach Handler and remove the annotation. Once the annotation was deleted, the deprovisioning process managed by Cluster API would continue following the path illustrated in section 3.10.1. The Operator SDK [39] framework was used to develop this operator, but unlike the process illustrated in section 3.8, the controller did not manage a custom resource, but rather the Machine objects defined by Cluster API. Once the change to the controller necessary to manage Machine objects had been made, the correct invocation of the reconciliation cycle was tested. To do this, a provisioning process was started, verifying that the controller was activated by displaying the information of the machine involved in the operation. Once this verification was successfully completed, the annotation addition mechanism was implemented as shown in listing 3.9.

```
if machine.Status.Phase == "Provisioning" && machine.Annotations == nil
    && strings.Contains(machine.ObjectMeta.Name, "slurm-worker") {
  logger.Info("Provision a machine")
  // Add annotation to wait the draining before deprovisioning
  machine.Annotations = make(map[string]string)
  machine.Annotations["pre-drain.delete.hook.machine.cluster.x-k8s.io"] =
      "drain-slurm"

  err := r.Client.Update(cctx, machine)
  if err != nil {
    return ctrl.Result{}, err
  }
  return ctrl.Result{}, nil
}
```

Listing 3.9: Slurm Operator Reconcile Loop portion

As can be seen, the hook setting is based on three conditions that the machine must meet. These conditions have been selected because they allow a Slurm worker machine in provisioning

status to be identified and, furthermore, by checking for the absence of the annotation, they ensure that the operation is performed only once. This last point is important because, even if the reconciliation cycle is invoked several times on the same machine, the hook setting would only be done once. At this time, their names are used to distinguish Slurm machines from Kubernetes machines. This is certainly not the best method from an implementation point of view, but it has allowed us to develop the controller more quickly. In the future, when more types of nodes need this hook, custom metadata could be used to indicate which machines need to undergo a particular deprovisioning process that is different from that of Kubernetes nodes.

Subsequently, in a manner very similar to that described above, the method for identifying machines undergoing the deprovisioning process was implemented. In this case, the behavior followed by the controller was that of the client implemented in section 3.10.2, which had already been used by the Provisioner implemented in the previous section. The only substantial difference was that, once the request to remove the Slurm node from the cluster had been made, the controller removed the hook from the list of annotations of the Machine object using the method `delete(machine.Annotations, "pre-drain. delete.hook.machine.cluster.x-k8s.io")`. This allowed the deprovisioning process managed by Cluster API to resume.

At this point, a method had finally been created to reduce the number of nodes within a Slurm cluster using Cluster API. This, combined with the solutions adopted to dynamically configure Slurm, allowed both types of clusters to be scaled up and down using the same API. Specifically, using the command `kubectl scale -replicas=<number_of_nodes> machinedeployment <machine_deployment_name> -n metal3`, it was possible to change the cluster structure at the bare metal level independently of the role that the machines played.

3.11 Simplifying Slurm Operator installation using Helm

The final phase of the project involved automating the Slurm Operator installation process. The goal of this phase was to ensure easy installation of some of the components developed in order to facilitate adoption of the solution. For the reasons already explained in section 2.7, it was decided to use Helm [16].

First, we decided to create a Docker image containing everything needed to run the operator developed in the previous section. We used the Makefile provided by the Operator SDK, which contains the `make docker-build` directive. The first attempt failed due to some limits that Docker has introduced on the number of pull operations that can be performed daily. By replacing the source of the base image, we were able to circumvent the limit and generate the Docker image. It was then uploaded to the local registry to simulate the distribution of the project. At this point, we tried to pull and run the container with `docker run`. Unfortunately, even though the container was running, the Machine objects were not being handled correctly. This type of error was caused by the lack of hooks within the Machine object annotations.

To understand the causes of the error, another directive provided by Operator SDK was used, namely `make deploy`, which deploys the operator within the Management Cluster. The Pod logs immediately highlighted the need to provide certain permissions to the operator so that it could access CAPI objects. Once the correct permissions were provided by creating a Kubernetes object of type *ClusterRoleBinding*, the Pod related to the Slurm Operator was successfully started and was finally able to access the Machine objects.

Once the error was resolved and the necessary changes made, we moved on to automating

the deployment process using Helm. Unlike in the previous steps, Operator SDK does not provide a make directive for creating a Helm chart. We therefore tried to create the package using the command `helm create slurm-operator-chart`, instructing Helm to use the image created previously. The need to perform two steps and the fact that Operator SDK does not natively support Helm led us to search for a solution that could further simplify deployment. Therefore, we adopted the *Helmify* [17] tool, which is able to convert an operator created with Operator SDK directly into a Helm chart without having to create the Docker image. To simplify use and future deployments, the directive shown in listing 3.10 was added to the MakeFile of the slurm operator.

```
helm: manifests kustomize helmify
	$(KUSTOMIZE) build config/default | $(HELMIFY)
```

Listing 3.10: Operator's Makefile with Helmify integration

In this way, using the `make helm` command, it was possible to generate the correctly configured chart. Once installed with `helm install` and given permissions, the Pod started successfully and was operational.

Chapter 4

Performance Evaluation, Optimization and Future Prospects

In this chapter, which concludes the thesis, we will illustrate some of the experimental results used to evaluate the performance of the proposed solution. In addition, we will present some of the ideas that the IBM team intends to explore further based on the results obtained from the development of the solution described above.

4.1 Data collection

Once a solution capable of meeting all the requirements specified at the start of the project had been obtained, a series of data on provisioning and deprovisioning times was collected, with the aim of quantifying the efficiency of the solution developed so as to identify any critical issues and suggest possible improvements. In particular, the time required to make the different types of clusters operational and the time required to safely remove nodes were analyzed.

In order to obtain a sufficient number of samples in the shortest possible time, a much larger development environment was created than those created in previous development phases. In particular, the number of virtual machines simulating the bare metal hosts that need to be configured was increased so that thirty could be active simultaneously. At this juncture, the ease with which the Metal³ development environment can be expanded was appreciated; in fact, to activate the expanded environment, the same steps used in section 3.4 were followed, modifying only the environment variable `NUM_NODES`. It was possible to start a considerable number of virtual machines only thanks to the computing power made available by IBM.

As was often the case in the experiments described in this thesis, data collection also involved an initial phase of manual information research, followed by automation of the process, which was necessary to speed up collection. The first phase involved searching for information that would be useful for evaluating machine provisioning times. First, a small Slurm cluster and a small Kubernetes cluster were provisioned in order to analyze the information contained in the log files and objects managed by Cluster API. The first useful information, namely the start and end times of the provisioning phase, was identified using the Bare Metal Host object. It is very

important to note that these last two pieces of information, although called `provisioning.start` and `provisioning.end`, refer to the time taken by Metal[3] to install and start the operating system on the node, excluding the time taken by Cloud Init to configure the machine. This last piece of information is considered essential in order to best estimate the time required to obtain a functioning cluster.

Subsequently, we began searching for information regarding the execution times required by Cloud Init to apply the configurations contained in the various manifests. Consulting the documentation [15], we discovered that Cloud Init saved information regarding the last execution in the file `cloud-init-output.log`. In particular, this file made it possible to trace the moment when the configuration was completed. Another set of information was retrieved using the `cloud-init analyze boot` command; this included time references for the moments when:

- The kernel is started.
- The kernel completes the boot phase.
- Cloud Init is launched.
- Cloud Init begins to apply the configurations specified via cloud config.

Once we understood where to find the relevant information, we automated the collection process. To do this, we implemented a Python script; we chose this programming language because there are numerous libraries available for communicating with Kubernetes and via SSH. With the help of Python and the libraries, we were able to implement the data collection script in a short time. The script consisted of three phases: in the first phase, the script connected to the management cluster and requested the available information relating to all Bare Metal Host objects present; in the second phase, it obtained information relating to the start and end of the provisioning of the operating system only, and also read the machine's IP address; in the last phase, the script used the address retrieved in the previous phase to connect via SSH to the machine to retrieve the log files and the output of the command presented in the previous paragraph. Once all three phases were completed, the script wrote the information to a CSV file. This file format was chosen for its simple structure and ease of integration into data analysis programs, such as the one that was implemented and will be presented later.

Once the data collection process for the node provisioning phase had been automated, it was necessary to collect the same type of information for the deprovisioning phase. A script was created, parallel to the previous one, capable of extending the CSV file produced by the first script with data relating to the removal of nodes from the two clusters. Unlike what has already been described, the deprovisioning data could be obtained directly from the Bare Metal Host object, as the attributes `deprovisioning.start` and `deprovisioning.end` took into account the entire process, from the administrator's request to the moment the node was cleaned up. This greatly simplified the operations that had to be performed by the program; in fact, there was no longer any need to communicate with the node via SSH. In addition, the two BMH attributes also allowed the execution time of the hook created specifically for Slurm nodes to be counted, thus ensuring a direct comparison with the Kubernetes node deprovisioning process.

At this point, all the tools needed to collect data automatically had been implemented. The only process that still had to be performed manually was the start of the provisioning and deprovisioning phases. However, these processes were simplified by CAPI and our solution, which allowed us to modify the structure of dozens of bare metal machines using a single Kubernetes

command. There were three types of machines analyzed: Kubernetes machines, Slurm control nodes, and Slurm compute nodes. To obtain an adequate number of samples, two hundred bare metal machines were provisioned for each node type.

4.2 Data Analysis

Once the data had been collected, it was then necessary to interpret it in order to identify any inefficiencies. Python was again chosen for this task, this time accompanied by specific libraries dedicated to data analysis: Pandas [41], MatPlotLib [30], and Jupyter Notebook [23].

The first graph that was created, shown in 4.1, shows the average time taken by Cloud init to configure each node type.

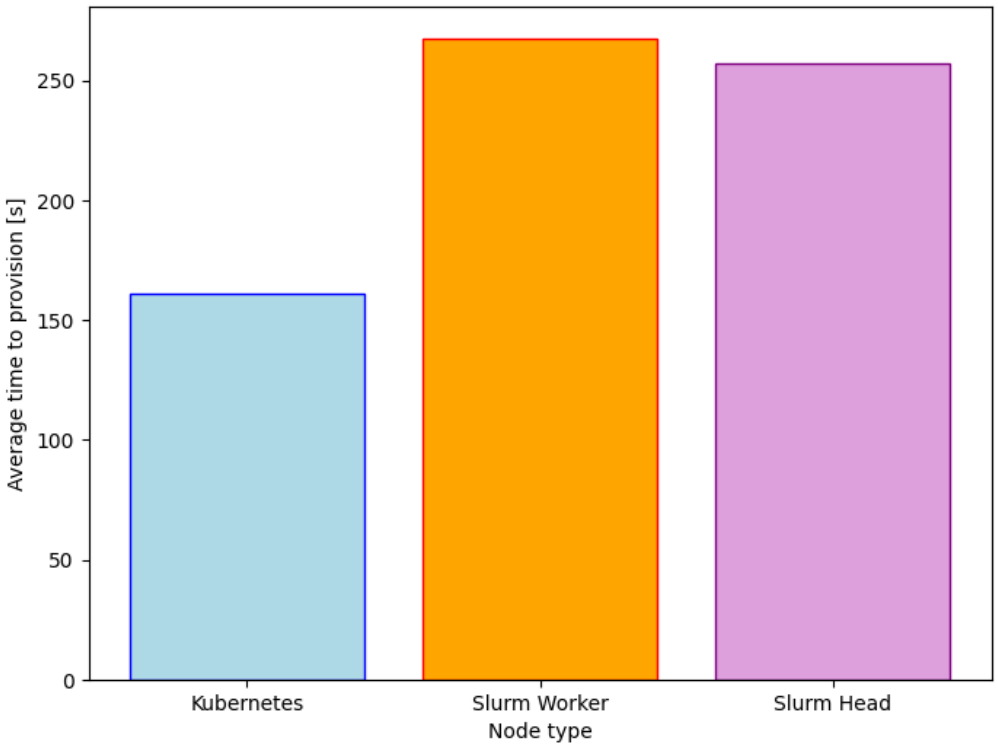

Figure 4.1: Cloud Init execution time during provisioning

The graph shows that the time required to configure nodes as Slurm Head has an average overhead of 59% when compared to the time required to configure Kubernetes nodes. Furthermore, the overhead rises to 66% when considering the configuration of Slurm Worker nodes. It is important to note that the data shown in 4.1 only concerns node configuration and not the provisioning of the operating system carried out by Ironic: this is due to the fact that all machines used the same CentOS operating system image and; therefore, the time taken for provisioning, as shown in the graph 4.2, was the same.

Subsequent analyses focused on the distribution of the samples collected. Box plots were produced, shown in image 4.3, relating to the samples collected for each type of node.

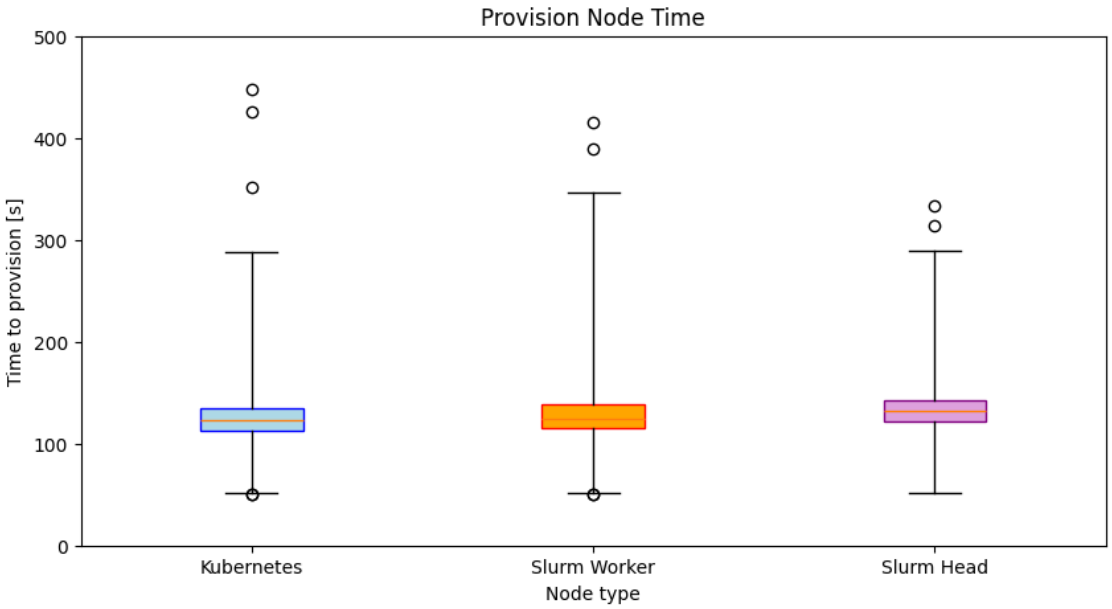

Figure 4.2: Cluster API and Ironic infrastructure provisioning distribution

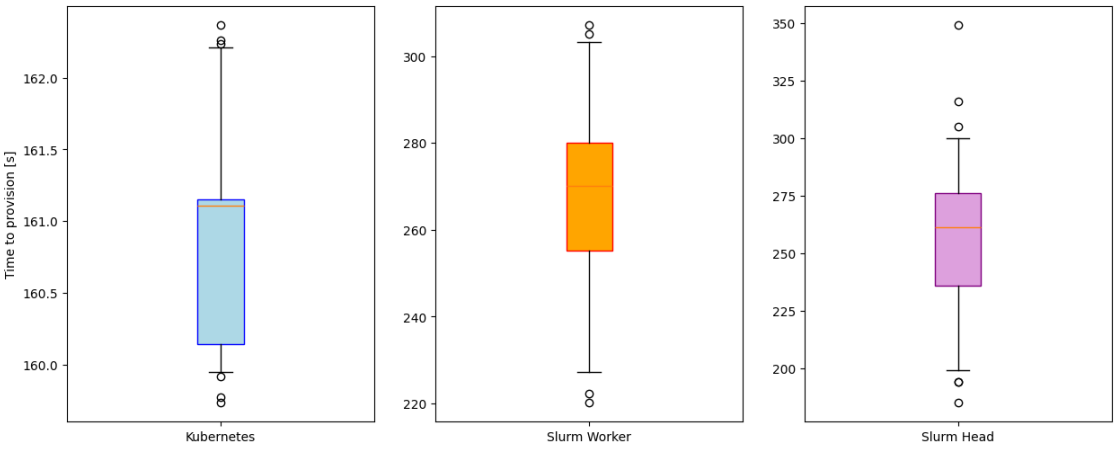

Figure 4.3: Cloud Init execution time distribution during provisioning

As might be expected, the time required to configure Kubernetes nodes was fairly deterministic: 99% of samples fell within a time range of just over two seconds. However, the same could not be said for Slurm nodes, which showed variations that could exceed one minute for Worker nodes and two minutes for Head nodes. This high variance arises from the installation of software packages, which, when using the network, is influenced by external factors (e.g., traffic, distribution system, etc.). Although these times were excellent when compared to the time it takes a system administrator to manually change the configuration of a node, we wanted to investigate whether it was possible to improve them in any way. The investigation will be described in the next section (4.3).

Once the analysis of the provisioning phase was complete, we moved on to analyzing the data relating to the deprovisioning phase. As before, we produced a graph of average times, shown in 4.1. In this case, unlike the previous one, the Slurm nodes did not show any particular overhead compared to the Kubernetes nodes.

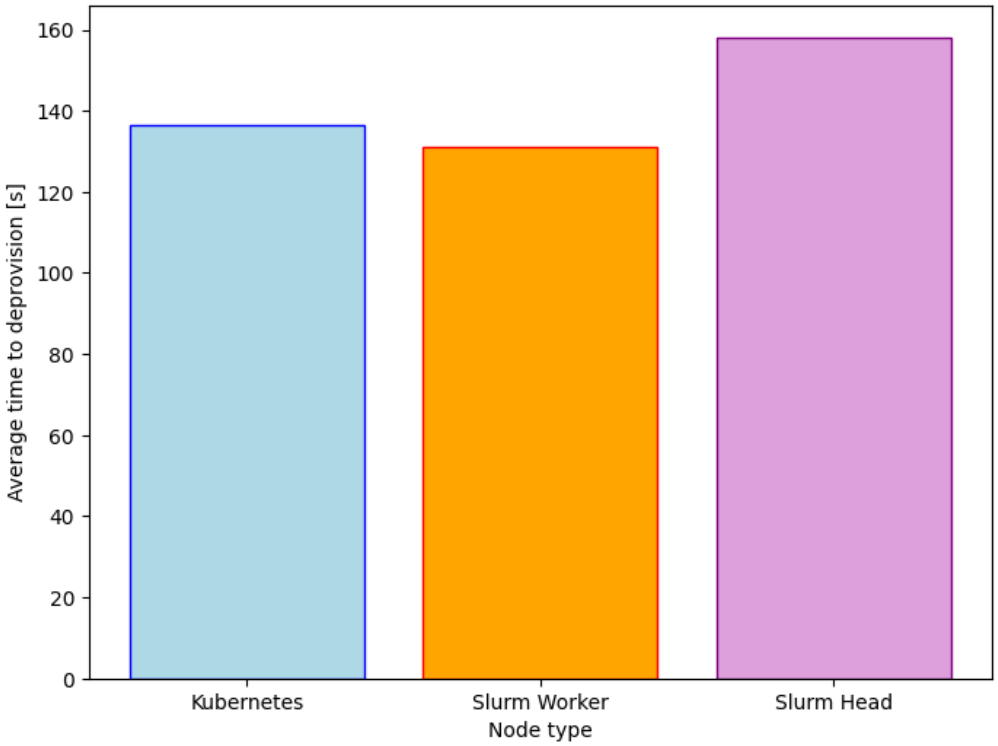

Figure 4.4: Cluster API and Ironic deprovisioning time

In this case, too, the distribution of samples was analyzed, creating the graph 4.5. As can be seen, the distributions are almost identical, demonstrating how the overhead introduced by the deprovisioning system consisting of the Slurm Operator and Slurm Detach Handler is essentially imperceptible to the end user of the system.

4.3 Optimization

As explained in the previous section, after the data analysis phase, we explored ways to reduce and make the provisioning time of Slurm nodes more consistent. The idea was to create a custom image with all the software packages already installed. This way, since we would no longer need to use the network to configure the node, the times should have been shorter and more consistent. Of course, achieving faster provisioning is a compromise, as the use of ad hoc images reduces the flexibility of the solution. This is because custom images need to be recreated to apply updates, which was not the case in the solution presented above, as the software packages downloaded to the machines during configuration were always the most up-to-date.

To mitigate, although only partially, the problem of custom image flexibility, an attempt was

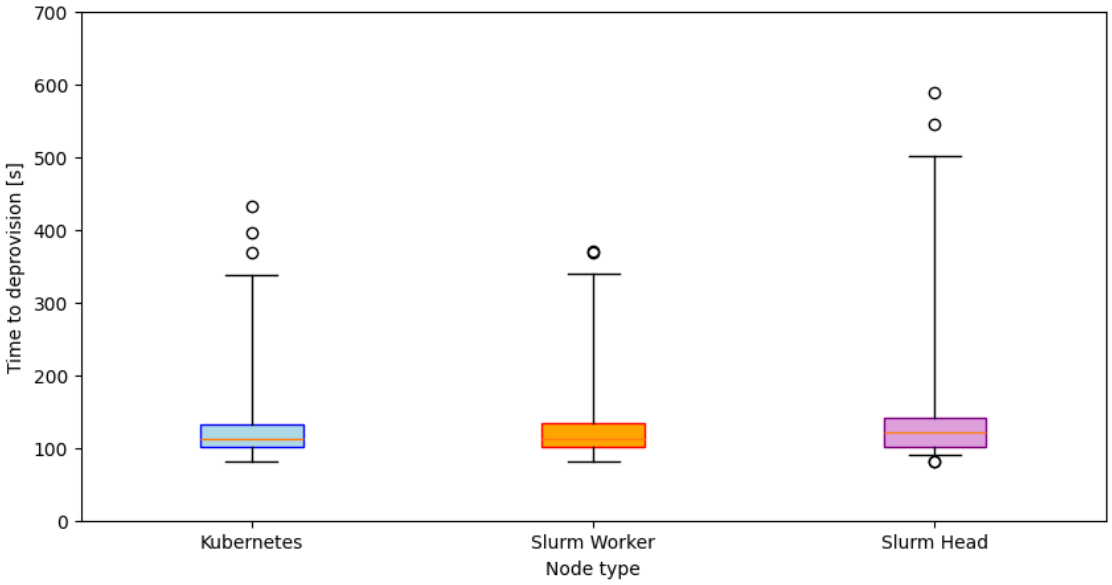

Figure 4.5: Deprovisioning time distribution

made to create an image using Bootc. By reusing some of the skills acquired in the study of bootable containers, it would have been possible to create Dockerfiles that declared the operating system and the packages to be installed. In this way, it would have been feasible, using Podman, to generate an image from the file created in the previous step, thus allowing the system administrator to update the image and related packages in an easy and maintainable manner. However, a blockage was encountered due to the hardware architecture: the machine provided by IBM used ARM architecture, while the node within the IBM cluster used x86 architecture. This meant that it would have been necessary to build a cross-platform image, an option unfortunately not supported by Podman. Therefore, although this solution seemed the most promising from a maintainability point of view, we chose to adopt a manual solution for creating the image in order to obtain data on provisioning times and thus be able to assess the impact of downloading the packages.

The manual solution in question involved, as in the experiments described in the first sections of chapter 3, the use of Qemu and Cloud Init. A cloud-config was created to create a user with administrator permissions, and once the virtual machine was started via Qemu, access was granted. Subsequently, all the packages necessary to make that machine a Slurm worker node were manually installed. The Qemu image thus created was converted into a disk image using the `qemu-img convert` command. The same procedure was followed for the Slurm control node image, and then the checksum of both images was produced so that Metal³ could verify their integrity.

At this point, two new MachineDeployments were created, which used Metal³ to manage the infrastructure, but unlike what had been done so far, the operating system image was the custom one created previously and no longer the CentOs image without any packages installed. Furthermore, these MachineDeployments continued to use the Cluster API bootstrap provider Kubeadm, but the cloud-config used to configure the nodes was modified by removing the portions that specified the software packages to be installed. An initial test provisioning consisting of five machines was performed to verify that the configuration had been done correctly. Already at this stage, it was possible to see a significant reduction in provisioning times. Once it was verified

that the configuration was working, data on the provisioning times of both types of machines was collected, following the same procedure described in the previous sections.

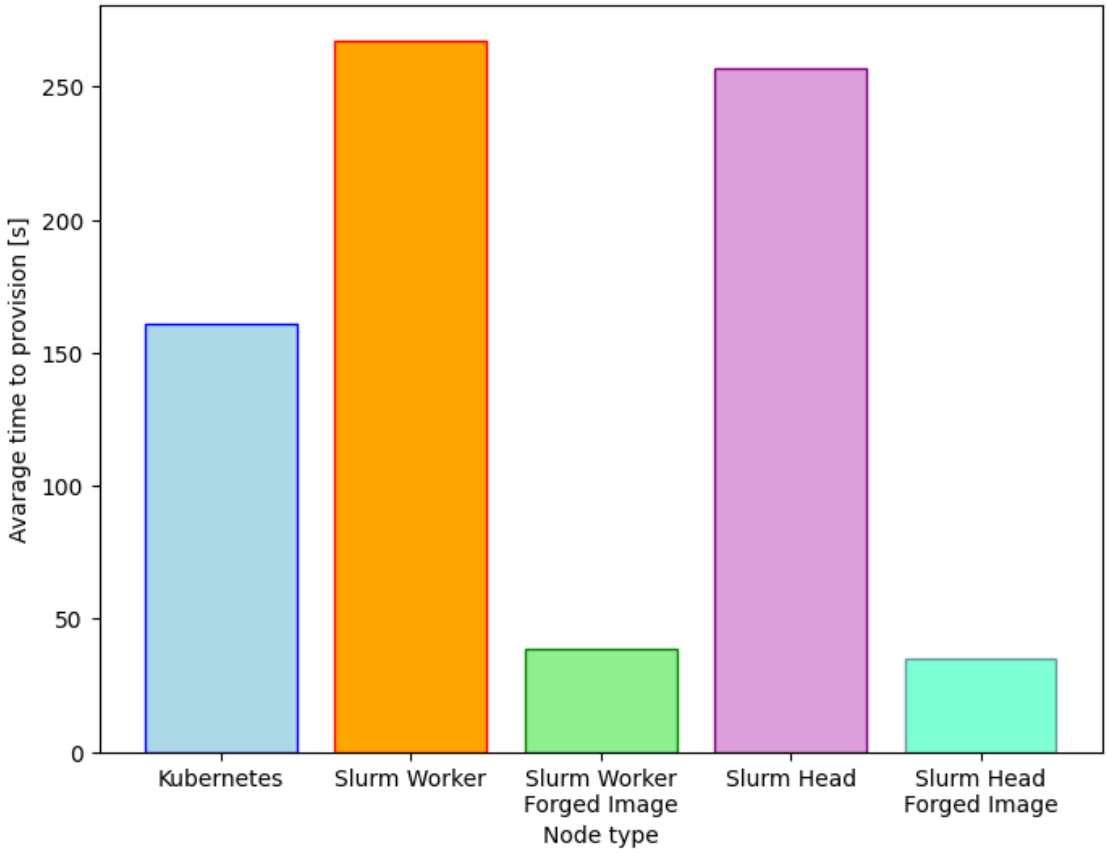

Figure 4.6: Cloud Init execution time during provisioning of custom images

As can be seen from the graph 4.6, provisioning times have been significantly reduced. In particular, the image created specifically for worker nodes allows cloud-config to run 6.9 times faster, while in the case of control nodes, execution is 7.37 times faster. In this case too, the variability of provisioning times was analyzed.

The graph 4.7 shows that provisioning times are much less variable than in experiments using the CentOS image as distributed by the developers. The average time to configure a bare metal node as a Slurm worker node went from 267.59 to 38.74 seconds, while for control nodes, the average time dropped from 257.15 to 34.87 seconds. If we combine these latest figures with the average time required to deprovision a bare metal node (which, as mentioned above, is independent of the node type), we can estimate the time required to reconfigure a Kubernetes cluster into an HPC cluster. Specifically, considering 145.17 seconds to deprovision and 38.74 seconds for subsequent provisioning, the total time is 183.91 seconds. In just over three minutes, an administrator using this solution could change the configuration of an entire cluster of bare metal machines.

As highlighted in the previous pages, this speed is the result of a compromise, as the image created must be kept up to date manually by the administrator. It will be up to the latter, based on user needs, to decide whether to prioritize speed of reconfiguration or maintainability.

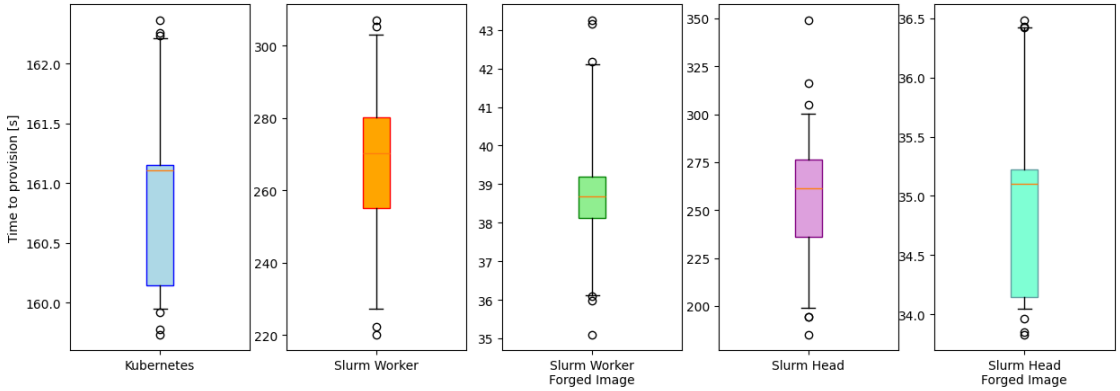

Figure 4.7: Cloud Init execution time distribution during provisioning with custom images

4.4 Future Prospects

The solution for unified management of hybrid environments presented in this thesis should be viewed as a proof of concept rather than a product ready for use in production. The project served to demonstrate the feasibility of adapting Cluster API to environments for which it was not designed; therefore, although the solution presented is functional, there are some open issues that will need to be addressed if it is to be adopted on a large scale. This section will illustrate these issues and some possibilities for integration with existing tools.

4.4.1 Possible Implementation Improvements

The first improvement to the existing solution is the implementation of the deprovisioning system extension so that it can also be used for Slurm control nodes. Currently, only Slurm computational nodes are removed correctly. The advantage of the adopted solution is that the operator will not need to be adapted in any way; instead, a new daemon will be implemented that can launch the takeover operation using the `scontrol takeover` command. Improvements should also be made to the system to recognize which nodes need to be managed via the Slurm Operator. Currently, the operator recognizes Slurm nodes using only the node name, whereas it would be more robust and extensible to introduce specific annotations, independent of the node's role.

Subsequently, in order to speed up the creation and increase the maintainability of custom images, the use of tools such as Bootc should be explored. The use of declarative tools would allow administrators to more easily balance configuration speed and the ability to precisely control everything that is installed on a specific type of machine; moreover, the image creation process would certainly be faster when compared to the manual process used in the previous section.

Although the development process and performance evaluation were conducted using virtual machine simulations that closely replicate the management processes adopted by Metal[3] and Ironic, the primary objective of the solution is the converged management of bare metal infrastructures. It is essential that the proposed solution is also tested in environments other than the virtual one produced by the metal3-dev-environment, particularly on clusters of physical machines. This is because, at the bare metal level, the administrator interacts with numerous low-level software components that are not filtered by virtualization systems (e.g., drivers). This is a crucial aspect of infrastructure management, especially in HPC environments where maximum performance is

sought. Unfortunately, it was not possible to carry out this type of experimentation using IBM's infrastructure during the collaboration period. It is possible that this experimentation will be carried out by the team in order to discover how similar the data shown in section 4.2 is to that collected in an environment closer to the production one.

It would also be advisable to conduct a further performance study in order to assess the impact of the Slurm configuration with Dynamic Nodes. The decision to use this type of node was fundamental to the success of the project, as it made it possible to overcome the need to create an ad hoc system for modifying and distributing the `slurm.conf` configuration file. However, it has emerged that the use of this type of configuration causes overhead compared to the classic Slurm configuration, and unfortunately there are no studies available that quantify how much this impacts performance. In particular, a study of this kind should include:

- Communication latency measurements: dynamic nodes use a communication system that is also dynamic, which affects the speed of information exchange.
- Benchmarking: to verify the absence of overhead introduced by our solution when compared to a Slurm cluster with dynamic nodes but configured manually.
- Hybrid environment stress testing: the overhead of a hybrid environment should be evaluated by simulating continuous switching between Kubernetes and Slurm clusters to understand the impact on stability, cluster performance, and how running workloads are affected.

4.4.2 Integrations with existing workload convergence projects

As already explained in section 1.4.2, there are already open source tools on the market that seek to create workload convergence solutions. Given that the solution presented in this thesis works at a lower level, it would be interesting to study how it could be integrated with the functionalities already developed by other projects.

Notably, the Slinky project [45] presents an opportunity for integration. By leveraging Slinky's capabilities, we can run Slurm management services, generally hosted on head nodes, directly within Kubernetes and utilize the VirtualKubelet Bootstrap Provider to provision bare metal Slurm workers, thereby increasing the space of potential deployment configurations and removing the need for dedicated Slurm head nodes.

Additionally, the Supernetes project [12], which exposes HPC cluster information in Kubernetes by mapping one Virtual Kubelet to each HPC node, offers another promising avenue for integration. This would not only increase the visibility of HPC workloads for administrators but also provide a more comprehensive set of tools for workload management. For example, with greater visibility of running workloads, an auto-scaling system could be developed, similar to the one used by Kubernetes to adjust the number of Pods, but capable of providing new infrastructure. In this way, the system could automatically choose cluster sizes based on user requests, completely removing the need for the administrator to intervene to modify the infrastructure.

Among all the tools and products developed by IBM, the project implemented for the purpose of this thesis could be perfectly integrated within Morrigan, a test bed created within IBM that aims to use OpenShift [38] as a unified control plane for bare metal infrastructure. Openshift, created by RedHat but owned by IBM, is a Kubernetes-based containerization system designed to manage, deploy, and scale enterprise applications. Morrigan was created to address the need to dynamically manage from a single OpenShift instance the heterogeneous environments (such as

Kubernetes and Slurm) used by IBM researchers. Thanks to the multiple points of contact between the two projects, the solution presented in this thesis could be adapted to extend Morrigan and enable this tool to provision Slurm clusters. Morrigan also uses CAPI to manage bare metal nodes and VirtualKubelet to give visibility to non-OpenShift nodes; however, it is still necessary to adapt the project's peculiarities, such as the Slurm Detach Handler, to ensure compatibility with OpenShift.

4.4.3 Publications and feedbacks

As already mentioned several times, this thesis is the result of collaboration between the University of Bologna and IBM Research Lab. Since the latter is a research laboratory, the goal is not to develop products that are suitable for market launch, but rather to search for new solutions that, once validated, can be integrated to develop products. In order to validate the effectiveness of solutions in the field of research and development, it is essential to spread the work by publishing the results obtained. This allows the communities that are expected to adopt the technologies and products to provide valuable feedback. This feedback allows development to continue, taking into account requirements that have not been addressed until then. The development of the project presented in this thesis would not have been possible without the publications of the developers who first worked on tools such as Cluster API and VirtualKubelet. In order to disseminate the implemented solution, it was decided to try to participate in conferences in both the Cloud and HPC sectors: in the first case, research focused on conferences that dealt directly with Kubernetes, while in the second case, conferences that dealt specifically with high-performance infrastructure administration were sought.

In order to gather feedback from the Kubernetes community, events called Kubernetes Community Days were analyzed. These days are organized worldwide by community members to discuss how Kubernetes is being adopted and modified. In this case, a request was submitted to give a presentation entitled *Cluster API Beyond Kubernetes: Unified On-Prem Management with Metal³ and Virtual Kubelet*, the purpose of which was to show the original use of Cluster API to enable bare metal management of non-Kubernetes nodes. The proposed solution, being a sort of bridge between two environments that rarely come into contact, must be presented differently to the two communities so that both can appreciate its benefits and provide useful insights. The proposed presentation has been accepted; therefore, on December 5, this project will be presented at CERN in Geneva, with the participation of the IBM Team. The event will be a great opportunity to exchange opinions and may be an opportunity to identify further directions for development based on the solution produced during the course of this thesis.

Conclusions

This thesis addressed the challenge of unified management of on-premises infrastructures, exploring the adoption of orchestration tools typical of the cloud environment, such as Kubernetes, to achieve management convergence with High Performance Computing (HPC) systems. The main objective was to overcome the siloed approach, in which HPC and cloud clusters are separated, requiring dedicated management services. The lack of a unified solution hinders the dynamic reassignment of bare metal nodes between different clusters, leading to underutilization of hardware resources within data centers.

Unlike solutions already explored by other tools, the proposed solution focused on convergence at the node level, outlining a common layer for cluster lifecycle management capable of provisioning and scaling both Kubernetes and Slurm nodes. This approach provides a global view of resources and a single management API, enabling more efficient use of resources and a high level of isolation to prevent contention.

The core of the project lies in the extension of Cluster API (CAPI), the Kubernetes cluster lifecycle management tool, to automate the lifecycle of non-Kubernetes machines, specifically HPC nodes organized through Slurm. To achieve this, a bootstrap provider called Cluster API Bootstrap Provider Virtual Kubelet (CABPV) has been developed, which is capable of provisioning non-Kubernetes nodes and uses the Virtual Kubelet project to make them visible to CAPI. The provider developed allows for a very high level of isolation between different types of clusters.

A mechanism for de-provisioning Slurm nodes was also developed through the implementation of Slurm Detach Handler, a client-server service introduced within the Cluster API architecture that manages the draining and removal of cluster nodes. Performance evaluations of the proposed solution revealed some inefficiencies; therefore, some optimizations were implemented that significantly reduced configuration time. Thanks to these optimizations, it has been estimated that an administrator could reconfigure an entire cluster of bare metal machines in just over three minutes (i.e. 184 seconds).

The results demonstrate that Kubernetes cluster management tools can be extended to support different cluster types, simplifying administration. Looking ahead, the IBM team plans to integrate the solution developed in this thesis with existing open-source tools to increase visibility of Slurm workloads within Kubernetes and further automate infrastructure management. Integration with Slinky [45] would allow Slurm management services to run inside Kubernetes, using the VirtualKubelet Bootstrap Provider to provision Slurm workers on bare metal, removing the need for dedicated head nodes and increasing configuration flexibility. Supernetes [12] could be used to share information about HPC nodes with Kubernetes, improving workload visibility and enabling systems that can autoscale this class of infrastructure. The IBM team's ultimate aim is to integrate the proposed solution into Morrigan, the OpenShift and CAPI based testbed

used by IBM researchers, to enable the provision of Slurm nodes and offer unified control of bare metal infrastructure. Supported by these results and future plans, the thesis work will be presented alongside the IBM team at Kubernetes Community Days in Geneva (CERN), offering an excellent opportunity to gather community feedback and identify new development directions.

Bibliography

[1] *Information technology - Cloud computing - Part 1: Vocabulary*. 2023. URL: https://www.iso.org/obp/ui/en/#iso:std:iso-iec:22123:-1:ed-2:v1:en.

[2] McLay et. al. "Best practices for the deployment and management of production HPC clusters". In: *Supercomputing* (2011). URL: https://dlnext.acm.org/doi/10.1145/2063348.2063360.

[3] Silvano et. al. *A Survey on Deep Learning Hardware Accelerators for Heterogeneous HPC Platforms*. Tech. rep. ACM Computing Surveys, 2025. URL: https://dl.acm.org/doi/full/10.1145/3729215.

[4] Fulai Zhu et. ali. *Moore's Law: The potential, limits, and breakthroughs*. 2023. URL: https://www.researchgate.net/publication/374208634_Moore's_Law_The_potential_limits_and_breakthroughs.

[5] *Bootc*. URL: https://bootc-dev.github.io/bootc/.

[6] *Building the Future on Bare Metal*. Tech. rep. Ironic.

[7] *CentOS*. URL: https://www.centos.org.

[8] Shaohao Chen. *Introduction to High Performance Computing*. Tech. rep. Boston University, 2017. URL: https://www.bu.edu/tech/files/2017/09/Intro_to_HPC.pdf.

[9] IBM Corporation. *xCat*. URL: https://xcat.org.

[10] Megan Crouse. *Businesses Work on Adapting to Generative AI, Hybrid Cloud*. Tech. rep. IBM Study, 2023. URL: https://www.techrepublic.com/article/ibm-hybrid-cloud-user-survey/.

[11] et. al. DanielJ.Milroy. *One Step Closer to Converged Computing: Achieving Scalability with Cloud-Native HPC*. 2022.

[12] Dennis Marttinen. *Supernetes*. URL: https://github.com/supernetes/supernetes.

[13] Chris Dunlap. *Munge*. URL: https://dun.github.io/munge/.

[14] *Go*. URL: https://go.dev/doc/.

[15] Canonical Group. *Cloud-Init*. URL: https://cloudinit.readthedocs.io/en/latest/index.html.

[16] *Helm*. URL: https://helm.sh.

[17] *Helmify*. URL: https://github.com/arttor/helmify.

[18] *History and overview of high performance computing*. 2020. URL: https://www.math-cs.gordon.edu/courses/cps343/presentations/History_and_Overview_of_HPC.pdf.

[19] *Ignition*. URL: https://coreos.github.io/ignition.

[20] Nick Ihli. *Cloudy, With a Chance of Dynamic Nodes*. 2022. URL: https://slurm.schedmd.com/SLUG22/Dynamic_Nodes.pdf.

[21] Kathleen Juell Jamon Camisso Hanif Jetha. *Kubernetes for Full-Stack Developers*. DigitalOcean, 2020.

[22] Ian Smalley Josh Schneider. *What's the difference between AI accelerators and GPUs?* 2024. URL: https://www.ibm.com/think/topics/ai-accelerator-vs-gpu.

[23] *Jupyter Notebook*. URL: https://jupyter.org.

[24] *Keepalived*. URL: https://keepalived.org.

[25] *Kind*. URL: https://kind.sigs.k8s.io.

[26] *Kubeadm*. URL: https://kubernetes.io/docs/reference/setup-tools/kubeadm.

[27] *Kubernetes*. URL: https://kubernetes.io/docs/home/.

[28] *MariaDB*. URL: https://mariadb.org/.

[29] Ell Marquez. *The History of Container Technology*. 2023. URL: https://www.pluralsight.com/resources/blog/cloud/history-of-container-technology.

[30] *MatPlotLib*. URL: https://matplotlib.org/stable/.

[31] *Metal3*. URL: https://book.metal3.io/.

[32] Tim Wickber Morris A. Jette. *Architecture of the Slurm Workload Manager*. SchedMD, 2023. URL: https://jsspp.org/papers23/JSSPP_2023_keynote_SLURM.pdf.

[33] *MPICH*. URL: https://www.mpich.org/.

[34] *MySQL*. URL: https://www.mysql.com/it/.

[35] Nvidia. *Base Command Manager*. URL: https://docs.nvidia.com/base-command-manager/index.html.

[36] Nvidia. *Cuda Documentation*. URL: https://developer.nvidia.com/cuda-toolkit.

[37] *OpenMPI*. URL: https://docs.open-mpi.org/.

[38] *OpenShift*. URL: https://www.redhat.com/en/technologies/cloud-computing/openshift/.

[39] *Operator SDK*. URL: https://sdk.operatorframework.io.

[40] Rani Osnat. *A Brief History of Containers: From the 1970s Till Now*. 2020. URL: https://www.aquasec.com/blog/a-brief-history-of-containers-from-1970s-chroot-to-docker-2016/.

[41] *Pandas*. URL: https://pandas.pydata.org.

[42] *Podman*. URL: https://podman.io/.

[43] *Quemu*. URL: https://www.qemu.org.

[44] *Red Hat Enterprise Linux*. URL: https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/10/.

[45] SchedMD. *Slinky*. URL: https://www.schedmd.com/slinky/why-slinky/.

[46] SchedMD. *Slurm*. URL: https://slurm.schedmd.com.

[47] Kubernetes SIG. *The Cluster API Book*. URL: https://main.cluster-api.sigs.k8s.io/introduction.

[48] *Slinky Repository*. URL: https://github.com/SlinkyProject.

[49] *Slurm Repository*. URL: https://github.com/SchedMD/slurm.

[50] Stephanie Susnjara. *Kubernetes History*. 2023. URL: https://www.ibm.com/think/topics/kubernetes-history.

[51] Saad Malik TenryFu. *ClusterAPI and Declarative Kubernetes Management*. Tech. rep. O'Reilly, 2022. URL: https://www.oreilly.com/library/view/cluster-api-and/9781098126865/.

[52] Tigera. *Calico*. URL: https://docs.tigera.io/calico/latest/about/.

[53] *Top500*. URL: https://www.top500.org.

[54] *Top500 Statistics*. URL: https://www.top500.org/statistics.

[55] *Using El Capitan Systems: Hardware Overview*. URL: https://hpc.llnl.gov/documentation/user-guides/using-el-capitan-systems/hardware-overview.

[56] Blesson Varghese. *History of the cloud*. 2019. URL: https://www.bcs.org/articles-opinion-and-research/history-of-the-cloud/.

[57] *Virsh*. URL: https://www.libvirt.org/manpages/virsh.html.

[58] *Virtual Kubelet*. URL: https://virtual-kubelet.io/.

[59] *What is GitOps?* URL: https://www.redhat.com/en/topics/devops/what-is-gitops.