UNIVERSITY OF TRENTO

Department of Information Engineering and Computer Science

Master's Degree in
Computer Science

Final Dissertation

reCluster

A resource-aware Kubernetes architecture for heterogeneous clusters

Supervisor
Maurizio Marchese
Co-Supervisor
Lorenzo Angeli

Student
Carlo Corradini
223811

Academic year 2021/2022

Acknowledgements

Firstly, I want to express my deepest gratitude to Dr. Lorenzo Angeli, who followed me during my internship and thesis writing, as well as Professor Maurizio Marchese, who oversaw it.

Thanks to all members of the Laboratory of Critical Technologies (CRIT).

A special thanks to my parents and my family for their constant support and encouragement to always pursue my dreams and passions.

Finally, I want to thank my friends who have shared my joys and concerns and have always been there for me.

Contents

Abstract

Many large-scale computing clusters employed to host online services are mainly concerned with maximizing overall performance and responsiveness, with the energy impact being considered only for cost savings, while overall sustainability is mostly marginal. Additionally, commercial data centers are largely homogeneous relying on modern and cutting-edge hardware components. If we want data centers to become more sustainable, we have to rethink their architecture and include awareness of the energy and resources involved. In this work, a possible solution for integrating and making sustainability as essential as performance and responsiveness in data centers is proposed. As a direct consequence, this new cluster architecture must preserve as much of the previous architecture's usability and overall requirements as possible, while also being more aware of the various resources and minimizing its energy consumption.

This thesis presents a case study of a possible heterogeneous data center architecture that is more resource-aware, and generally sustainable. We guide the reader through the design process, starting from guiding principles, progressing through theory and, finally, implementation and critical reflections.

We open the thesis by outlining the context, goals, and core principles around which the cluster is established. A dynamic and somewhat abstract context that is far from static and should adapt/transform to the needs and requirements of the organization managing the cluster. A set of goals aimed at making the cluster appear less heterogeneous and more like a single, homogeneous entity. A collection of core principles that enhances and provides a basis for improved sustainability, while minimizing waste as much as possible: Sustainability, Acknowledging Planetary Limits, Hardware Reusability, Energy Reduction, Insourcing, Interoperability, Free/Libre And Open Source Software, and Dependencies Reduction.

After the foundations, we depicts the theoretical architecture, which defines a high-level overview of the whole cluster. This is a collection of all the involved hardware and software components, along with their respective responsibilities outlining how they manage and maintain the cluster continuously and with an awareness of energy and resources. To communicate, the components must be connected, providing a network that enables workload orchestration and cluster management.

As the core point of the thesis, we detailed how we implemented a prototype heterogeneous cluster that integrates the core principles with the theorized networks and components. We named it reCluster, and the prefix "re" refers to the Right to Repair movement's "reduce, reuse, repair, recycle" slogan. Among the most notable reCluster components are the Server, which is a completely new implementation developed from the ground up, and the Cluster Autoscaler, which is a reimplementation that allows its core to communicate with the Server and vice versa. Combining Server and Cluster Autoscaler allows the cluster architecture to perform autoscaling procedures like upscaling and downscaling automatically. Moreover, we developed a script that automates all of the necessary installation, configuration, and deployment procedures. The prototype provides a totally autonomous, resource-aware, and sustainable cluster by employing only those resources that are strictly necessary, preferring those that are more sustainable than others.

The initial reCluster concept was theorized in [4], where I contributed to the Kubernetes analysis and the conceptualization of the various algorithms, which were eventually implemented in the prototype.

Adapting what is already available and specifically developed for large and energy-hungry data centers to a more sustainable and resource-aware architecture while preserving the same original computing industry requirements is not only possible, but also necessary for a more sustainable future.

1 Introduction

The purpose of this chapter is to provide an overview of the context, goals, and core principles that serve as foundations for the overall cluster architecture, both theoretical and implemented.
These so-called foundations are not restricted to this project and/or environment, but can be considered as the minimum requirements and principles to obtain a greener, energy and resource-aware software in general that is not only for the present, but also, and must be, for the future, because we, as the overall ICT community, have mostly ignored them in the past without giving them the appropriate weight.

reCluster is heterogeneous because it is composed of numerous and diverse components, ranging from computers to networking equipment, that have been reutilized after decommissioning. The fundamental goal of reCluster is to establish a cluster (hence the name) without reinventing the wheel, but rather to increase reusability by transforming and adapting heterogeneous hardware and software components designed for similar environments, enhancing the overall reduction of both energy consumption and resources involved without compromising overall performance, responsiveness, and ease of use.

1.1 Context

Most existing systems rely entirely on large data centers, which are excessively energy-hungry and do not provide any resource awareness owing to virtualization/abstraction layers. Furthermore, the servers are maintained permanently powered on to significantly minimize the time necessary for provisioning client instances while achieving virtually 100% uptime[1][2][3].
To achieve a greater degree of sustainability, the time necessary for provisioning must be rebalanced, allowing servers to be completely powered off while maintaining a similar level of uptime. It should be highlighted that this is ideal for many applications that may be classified as non-critical, while critical applications, such as emergency response and payment systems, require "five nines" or 99.999% or around five minutes and 15 seconds of downtime per year[4].

Radovanović et al., in [24], describe Google's Carbon-Intelligent Compute Management system, which actively reduces electricity-based carbon footprint and power infrastructure costs by deferring temporally flexible workloads. It uses a collection of analytical pipelines to acquire the next day's carbon intensity projections, train day-ahead demand prediction models, and apply risk-aware optimization to generate the next day's carbon-aware capacity for all Google data centers. The system limits the resources available to temporally flexible workloads on an hourly basis while preserving overall daily capacity, allowing all such workloads to be completed within a day.
This type of computing solution only accounts for services and processes that can be delayed, not those that must be continually active with an acceptable response time; sending a critical email that is only delivered the following day is unacceptable. It is only compatible with Google's data centers and does not provide a more interoperable solution that is compatible with other data centers. Moreover, the code is closed source, which means it does not adhere to the FLOSS philosophy. Nevertheless, the notion of deferring non-real-time operations, employing analysis and projections, to be rescheduled when the system is underutilized, improving overall sustainability and lowering the costs, should be considered.

Enes et al., in [12], present a platform that manages a power budget to limit the amount of energy spent by users, applications, and individual container instances. The energy constraint is accomplished

[1]https://aws.amazon.com/it/compute/sla
[2]https://azure.microsoft.com/support/legal/sla
[3]https://cloud.google.com/compute/sla
[4]https://aws.amazon.com/blogs/publicsector/achieving-five-nines-cloud-justice-public-safety

by the platform's capability to monitor container energy consumption and dynamically modify its CPU and memory resources via Vertical Scaling as necessary (see section 3.4.1).
This approach is much more high-level and directly autoscale the resources allocated to containers based on the power allocation among users and applications, as opposed to a low-level approach that involves physically terminating/bootstrapping the server (used by reCluster). Moreover, the energy is viewed as a precious resource that can be split and shared, improving the container system's overall awareness of its resources. Nevertheless, it needs to be combined with additional approaches to achieve high and low levels of energy awareness and reduction.

Paya and Marinescu, in [21], presented an energy-aware operation model for load balancing and application scaling on a cloud, outlining an energy-optimal operation regime and aiming to increase the number of servers employed in this regime. Servers that are idle or only barely loaded are put into one of the sleep modes to conserve energy. Therefore, the following could be employed to reformulate the conventional idea of load balancing in a large-scale system: *"Distribute evenly the workload to the smallest set of servers operating at optimal or near-optimal energy levels, while observing the Service Level Agreement (SLA) between the Cloud Service Provider (CSP) and a cloud user. An optimal energy level is one when the performance per Watt of power is maximized"*.
To reduce energy waste, this approach can be applied to an external load balancer employed in the architecture. Even more, energy can be saved by completely turning off mostly inactive or lightly-loaded servers rather than switching them into sleep mode. This new type of load balancer can also be interfaced with the Cluster Autoscaler and Server to better understand the overall cluster state, both in terms of active/inactive nodes and also of overall power consumption, to obtain more fine-grained information and therefore increase the overall efficiency.

All of the above strategies attempt to minimize energy consumption in various contexts: non-real-time services, containerization, and load balancing. Yet, none of them address, or only partially address, the core cause of the highest energy usage in a cluster architecture, which is the server themselves. To establish a more sustainable and resource-aware cluster, the servers must be considered and therefore autoscaled by physically bootstrapping or terminating them; since a powered-off machine consumes (nearly) zero energy.
Nonetheless, the prior techniques should, and must, be extended into future versions of reCluster to achieve even higher levels of energy reduction while satisfying the SLA.
More than ever, a more sustainable cluster architecture is required.

1.2 Goals

A cluster is composed of several components, including hardware and software, that operate together as a single entity to accomplish one homogeneous goal, or multiple and different goals. The hardware components, which are typically computers of different kinds ranging from desktop computers to single board computers, are connected in a single network: the cluster network, by additional hardware components particularly designed for communication. Because multiple applications can achieve the same result, are constantly evolving with newer features and capabilities, and can be employed or not depending on the (current) needs and requirements, the software components can vary and are as heterogeneous as the hardware, if not more. Between hardware and software, two major distinctions complement one another. The first distinction is that hardware, particularly reconditioned hardware, is limited in its ability to be modified and transformed, whereas software is essentially limitless in this regard and may achieve the same goal using multiple technologies and/or languages. The second distinction is that there is no difference between the various hardware components in terms of whether they are used for cluster operations or user operations. Instead, software components can be divided into low-level ones that are used to keep the overall cluster operational and high-level ones that are used by the users for their specific services. In this context, software in all of its aspects must be used to compensate for the shortcomings of the hardware since it can be customized to fill the gaps where the hardware lacks.
Most used hardware components may be reconditioned by reusing decommissioned components from consumers and organizations due to upgrades. Because newer components are often more performant and energy-efficient than older ones, the hardware is directly and automatically managed by the var-

ious cluster-related software deployed.
The cluster must be as simple as possible, with human interaction minimized to nearly zero, and automatic solutions preferred. The most essential automated solution that must be implemented in a cluster is autoscaling, both upscaling and downscaling. In upscaling, the cluster automatically bootstraps an inactive hardware component. Whereas in downscaling, the cluster automatically terminates an active hardware component. Selecting which component(s) to autoscale can follow multiple criteria, such as choosing nodes that are more power efficient or performant, or even a combination of the two. The latter is an excellent illustration of the combination of hardware and software that needs to be present in the cluster.
The cluster should be used in a wide range of environments and use-case scenarios, from a modest and local deployment at home to a large and remote deployment comparable to that found in a data center. Given that the various software manages them automatically using the autoscaling technique, there should be no limit to the number of involved computers.
If there is no need to operate the cluster for specific periods and it is therefore considered worthless, it can be completely shut down, with the benefit of reducing its overall power consumption to zero. When it is necessary again, it is reactivated. Although the latter may create some performance and/or responsiveness issues, because no physical machines are managing any service, it is far preferable to an always-on technique. The latter impacts can be mitigated if the organization can calculate different models and/or employs a single board computer as the always-on hardware component thanks to its overall extremely low power consumption. As stated previously, the context is dynamic, and hence the latter is dependent on the organization in charge of the cluster.

As a result, the cluster should appear less heterogeneous and more like a homogenous, and mostly automated entity.

1.3 Principles

This section is dedicated to illustrating some of the fundamental principles around which the overall design is based. These principles should be seen as the foundations for a more sustainable, resource-aware, free and open-source, and energy-efficient architecture that may be utilized not only in this specific context, but in all aspects of ICT.
It should be highlighted that these principles should not be regarded as negative to overall performance and responsiveness, but rather as equally important. Performance and responsiveness currently constitute the sole and most significant characteristics to consider when examining software, but in the future, it should be necessary, almost mandatory, to include, maybe gradually but starting to consider them as parameters for comparisons and essential aspects of a program.
The applicability of these principles is determined by the organization's requirements. Yet, some of these principles should be investigated and implemented in the future for greener and more resource-aware software.
The proposed list can undoubtedly be improved upon and/or expanded with additional, and possibly more strict, principles. Shifting the computing industry toward a more sustainable future should be possible, if not essential.

1.3.1 Sustainability

Source: `https://www.un.org/sustainabledevelopment/news/communications-material`

The United Nations (UN) has established 17 Sustainable Development Goals[5] (SDGs). The SDGs address climate change and strive to preserve the oceans and forests, while also addressing poverty and other deprivations via initiatives that improve health and education, reduce inequality, and stimulate economic growth.
Clearly, in an ICT context, several of these Goals may appear to be out-of-context, unreachable/unfeasible, or not directly related. However, the majority of them, including those that are not directly related to an ICT context, must be considered when developing new software applications and architectures; given that a minor improvement in a single SDG can benefit all other SDGs.

[5] `https://sdgs.un.org`

There is no need to change or entirely disrupt a project's primary goal, but rather to integrate the relevance of these long-term goals into the development process. If an ICT initiative improves or enhances one or more SDGs, it will undoubtedly benefit not only other direct and associated SDGs, but practically every other SDG as well.
A large end goal is nearly always comprised of a plethora of many smaller improvements/objectives that, when considered separately, may appear to be irrelevant, but when placed together form a sustainable chain of progress.

1.3.2 Acknowledging Planetary Limits

Icon made by Freepik from www.flaticon.com

When available resources are limited, it is essential to preserve them and reduce their overall consumption as much as possible. It is important to note that the term resources in this context is more generalized and does not designate any specific resource type, but they may be regarded as a distinct entity that groups them.
Rethink and change the availability of resources, and investigate a more realistic and conservative approach, referred to in [18] as computing within limits or simply LIMITS. LIMITS incorporates three major topics: present and near-future ecological, material, and energy constraints; new forms of computing that may assist promote well-being while living within these limits; and the influence of these limits on the area of computing. Not only is the concept of LIMITS confined within a computing environment, but it also aims to integrate other, and sometimes uncorrelated, heterogeneous areas. Moreover, because resources are not exclusively available to individuals, there is a need to think about and focus on collectives and larger contexts engaging in stronger connections to sustainable activities in disciplines other than computing. The term resource(s) should be considered not just to describe actual elements, but also to encompass socioeconomic, more intangible, factors that may be enhanced by diverse techniques pursued by the computing community, such as promoting development rather than economic growth.
As stated in [26], the incorrect underlying assumptions of endless, infinite, and replicable technological resources must be reconsidered. These assumptions are just unsuitable for a more constrained and sustainable future. To re-imagine how digital technologies may be developed and implemented in these new and limited computing architectures, new concepts of radically leaner and ecologically conscious techniques are necessary.
Another element of the architecture's involved resources is the relationship between the software and the hardware. Because of the current amount of processing power, some software has ridiculously high hardware requirements for even the most simple operations. Better and more performant programming languages and algorithms need to be employed to rethink, modify, and/or improve software resource requirements. A different technique is necessary with the adoption of a more permacultural approach, which is referred to as permacomputing in a computing environment. Permacomputing, as depicted in [23], is both a concept and a community of practice centered on issues of resilience and regenerative in digital technology, transforming problems into solutions, competition into cooperation, and waste into resources.
Transform what appears to be a disadvantage, a difficulty, or even a waste into a rich and valuable resource.

1.3.3 Hardware Reusability

Icon made by Freepik from www.flaticon.com

Use reusability and repairability approaches to reduce resource waste as much as possible.
The majority of unused hardware is perfectly capable of fulfilling the majority of tasks and operations in sustainable architecture, such as hosting a website or a cache server, without any particular difficulty. Unused hardware is frequently created by an upgrade to a newer and more recent version or by (major) scheduled decommissioning.
Typical electronic devices with even a single faulty hardware component are discarded and promptly replaced with newer hardware, even if the remaining components are healthy and completely functional. The expenses and work required for a repair vary

depending on the device, but most of the time the repair may be performed quickly and cheaply by replacing the faulty component with a new one. Furthermore, and this is especially true for desktop computers, if two devices are compatible and one has some broken components that the other does not, the healthy components can be combined to re-create a single but healthy and usable device with the benefit of drastically reducing the overall amount of resource waste.
The latter two approaches not only recycle abandoned technology but also reduce the overall quantity of electronic waste (e-waste) that they can generate. Consequently, the energy gap between overall utilization energy and production energy, formerly expressed as embodied energy, can be narrowed or even attained and exceeded.
Yet, the ease and feasibility of a device's repairability is entirely dependent on its manufacturer. Moreover, there are cases where the devices have a planned obsolescence after which some of the components will probably break and the cost or effort required for a repair is unfeasible, and it is far easier to buy a new one while discarding the old and broken one, causing additional (e-)waste. It is difficult to avoid the latter, and almost certainly additional issues, induced by the manufacturer. Yet, in recent years, there have been an increasing number of consumer activities aimed at reducing or eliminating the behaviors perpetuated by these manufacturers. The Right To Repair[6][7] movement is the most well-known, to advocate for repair-friendly policies, regulations, statutes, and standards. Another approach is to provide a repairability index, such as the French repair index[8], for the products so that potential customers may know and understand how easy it is to repair the corresponding device before purchase. It should be noted that not all manufacturers are opposed to repairability, and there are certain, albeit small, manufacturers, such as Framework[9], that have repairability as one of their core missions, building longevity products and improving the overall availability of spare parts and ease of upgrade, customization, and repairability. The objective/hope is that more manufacturers will begin to design, produce, and distribute more sustainable products.
Instead of planned obsolescence, consider planned longevity.

1.3.4 Energy Reduction

Icon made by Freepik from www.flaticon.com

The architectures and software should strive to reduce their total energy footprint as much as possible while providing similar, if not identical, outcomes. Concerning the preceding point, this is only possible if the organization has complete control over all aspects of the architecture, since the union of software and hardware is the only conceivable and feasible strategy to establishing a more sustainable and energy-aware architecture.
Energy is one of the most significant factors in achieving sustainable architecture. Because energy is closely related to carbon emissions, there is a need for low-carbon and sustainable computing with a path towards zero-carbon computing, as stated in [30], because current computing emissions account for about 2% of global emissions and are expected to rise dramatically in the upcoming years. This rate of expansion is unsustainable and as a community, we must begin to regard computational resources as finite and valuable, to be used only as essential and as effectively as possible. The latter is known as frugal computing, and it strives to get the same outcomes while using less energy by being more frugal with the available computing resources. In the current context, for example, if a system/computer in a cluster is not needed or is underutilized, it is a waste of both energy and resources. It must be terminated and, if any services are deployed on it, migrated to another active system to avoid service interruptions, increasing its utilization and lowering the architecture's total power consumption while maintaining the same level of Quality of Service (QoS). Humans are unable to perform these operations since there is the need to continuously monitor and check the overall status of the entire architecture. The core principles of energy reduction must be developed into automatic systems that could be easily extended to existing applications and tools which are focused solely on overall system performance and responsiveness. The system should be as efficient as possible, employing only those components

[6] `https://www.repair.org`
[7] `https://repair.eu`
[8] `https://www.ecologie.gouv.fr/indice-reparabilite`
[9] `https://frame.work`

that are strictly necessary, avoiding bootstrapping unneeded machines and terminating those that are unnecessary, and avoiding the paradigm of an always-on architecture that wastes energy and resources. Another factor to consider is the embodied energy of each hardware system. Embodied energy is the energy necessary to build all of the electronic components (both network and especially consumer appliances). It emerges that the energy used to manufacture electronic components is significantly higher, and in many cases dominant, than the energy needed throughout their whole operation[9]. This means that practically every hardware component is underutilized in terms of the energy required for manufacturing it. As a result, there is a need to correct this imbalance by attempting to reach at least the same amount of energy. It should be noted that the latter does not want to employ the hardware unconsciously, but rather with better and more conscious principles and logic that not only tries to reduce the overall "live" energy consumption, but also tries to equal, if not even surpass, the embodied energy of each electronic device that is employed in the architecture.
Previously, the term carbon emission/footprint was used concerning energy consumption. Yet, it is necessary to accurately define how the overall energy of the architecture is estimated, as well as the fundamental principles on which the calculation is based. This is not as simple as it appears. [20] reveal that quantifying the carbon footprint has produced contradictory results, implying the need for different and alternative approaches that delineate the specific relationships between elements - geographic, technical, and social - within the broader information and communication technologies infrastructure. The common method of assessing the energy impacts is the kilowatt hours of electricity and gigabytes of data through a network, expressed as the functional unit `KWh/GB`. The `KWh/subscriber` metric, which divides the larger energy draw inside a study's system boundary by the number of subscribers within the network, can better represent worldwide patterns in network access expansion over time. Rather than calculating `KWh/GB` or `KWh/subscriber`, a relational approach would assess the carbon, water, and land footprints of powering the whole architecture in specific regions of the world, highlighting the significant disparities between the various locations. Network shapes, rather than scales, provide new possibilities for assessing the existing and future Internet[20].

1.3.5 Insourcing

Icon made by kerismaker from www.flaticon.com

Invert the trend of constantly outsourcing applications to third-party cloud providers; we might express this with a neologism: "Insourcing".
While there are undeniable benefits to outsourcing in terms of convenience and better pricing, it erodes institutional and human independence by centralizing a single point of failure and delegating relevant choices regarding privacy, data ownership, and environmental impact on these external actors[4].
There is a need to obtain, direct control over the whole software stack, beginning with the hardware infrastructure and progressing through the application's development and final release. At first look, reversing the ICT outsourcing trend may appear to be impracticable, unneeded, and comparable to an old and mostly outdated approach. Nonetheless, overall technology, tools, and usability have significantly improved in recent years. Most importantly, there has been a shift in the software community toward standardization and interoperability across heterogeneous systems, which has not only made deploying software locally and with an in-house infrastructure trivial, but has also aided in abstracting away from the final user common and not-so-trivial problems, such as handling and coordinating multiple connections across different hardware and software components/layers.
As discussed in [28] current infrastructures and approaches to climate change are mitigation-oriented, aiming only to support or investigate social changes that reduce the material causes of the more pernicious facets of global change, with the implicit assumption that these infrastructures will persist as the risks of current unsustainable practices become urgent threats to well-being and survival. Instead, new architectures, and possibly old ones through refactoring and rethinking the overall design, must take and address new approaches, denoted as post-apocalyptic computing, to address the many, dramatic, and complex phenomena associated with global change, allowing to prepare for nonlinear changes. If the organization has direct control over almost every aspect of the architecture, the latter can be accomplished with almost negligible effort, whereas if everything is outsourced to an external entity, the organization has only a limited amount of control over the possible future outcomes that

can arise from unpredictable changes that will occur sooner or later. As a result, avoid remote and generally unknown architectures of third-party organizations that only raise the fragile state on which the application is deployed, establishing an inherent single point of failure that not only can be an issue for the present but, more significantly, for the future.
Another significant aspect of current architectures and approaches is that most operations are performed remotely and outsourced to external organizations, even though the participating entities are mostly close together, resulting in extra workload and networking equipment, which translates to an increased amount of overall power consumption. In this regard, the usage of external videoconferencing applications that enable remote entities to connect is a significant example. When two close entities, that can be even in the same building, seek to establish a teleconference, their data is routed to remote videoconferencing servers. Instead of depending on an external and remote service, it is possible to set up a local videoconferencing service in the local (insourced) architecture, employing open-source software and improving overall security and privacy. The distance between the entities and the local server instance may be drastically reduced, decreasing latency and lowering total power consumption for the videoconference, as theorized in [19], resulting in better outcomes and more sustainable service.
Lastly, insourcing enables better fine-grained and conscious control over the whole application's data and everything that comes with it, such as privacy and security, because it is known where and how it is stored, distributed, and protected. Even though these later elements are not the primary focus of this document, they should not be ignored or underestimated because they are essential for almost every software application. Furthermore, various initiatives in recent years have attempted to improve data protection for individuals and organizations, including the well-known General Data Protection Regulation[10] (GDPR) drafted and passed by the European Union (EU).
As a result, returning to an insourcing approach and having direct control over almost every aspect of the architecture and software stack is not only feasible and almost trivial nowadays, but also necessary for establishing a better and more sustainable future for everybody, not just the individual.

1.3.6 Interoperability

Icon made by Freepik from www.flaticon.com

Interoperability and standardization of hardware and software must be at the core of a more sustainable architecture. Most, if not all, of the components, employed in the design must agree on common interfaces to prevent the need for hardware adapters and software translation layers, which not only increase resource waste and energy consumption but also decrease overall system performance.
When all of the applications employed by the architecture agree on or adapt to a common set of interfaces, achieving software interoperability is virtually straightforward. Upgrading previously incompatible software is feasible, albeit depending on the complexity of the program, refactoring may be required. Fortunately, most software programs and, more broadly, the computing environment depends on standards that are equivalent in practically every part of the world. Consider the Internet itself, which is built on standards and interoperability between all sorts of heterogeneous components.
On the other hand, implementing hardware interoperability and standardization is considerably more challenging than software since it is heavily dependent on design decisions made by the manufacturer throughout the development phases. As a result, there is a need to persuade, mostly through legislation, the many manufacturers that are reluctant to adapt and do not meet or do not want to establish a set of standards that improve interoperability on their devices. A perfect example of the latter is the European Parliament's approval of a law, following an impact assessment study[11], that mandates by the end of 2024 a common, standardized, and interoperable charger for all mobile devices sold in the EU internal market[12]. The latter has been highly awaited, and it may be regarded as a first step toward a much broader trend of standardization and interoperability that can establish the foundation

[10]https://gdpr.eu
[11]https://data.europa.eu/doi/10.2873/528465
[12]https://www.europarl.europa.eu/news/en/press-room/20220930IPR41928/long-awaited-common-charger-for-mobile-devices-will-be-a-reality-in-2024

for a more sustainable future.
Furthermore, they can significantly reduce the effort required while also increasing the ease of establishing an insourcing architecture. Various organizations can exploit a uniform and de-facto standard that enables high-level interoperability between heterogeneous architectures, accelerating the process of migrating various services and deployments from remote to local architectures.
Lastly, having an interoperable architecture, combining software and hardware, eliminates the so-called technology vendor lock-in effect, in which a customer is dependent on a single manufacturer. Because the entire architecture might be composed of reconditioned/refurbished hardware and thus is intrinsically heterogeneous, the vendor lock-in effect cannot be tolerated.

1.3.7 Free/Libre And Open Source Software

Source: `https://opensource.org/logo-usage-guidelines`

The software developed and employed in the architecture must be Free/Libre and Open Source in all aspects. If a software application is incompatible with the latter, an alternative must be identified or developed.
Following the GNU Project's philosophy[13]: *Free Software means that users have the four essential freedoms: (0) to run the program, (1) to study and change the program in source code form, (2) to redistribute exact copies, and (3) to distribute modified versions.*
This principle's fundamental is presented as Free/Libre and Open Source Software[14] (FLOSS).
Being FLOSS-compatible has also the significant advantage of allowing external programmers to contribute to the software, consequently improving overall quality and providing for more performant and efficient logic and algorithms.

1.3.8 Dependencies Reduction

Icon made by Freepik from www.flaticon.com

For decades, as stated in [8], discussion about software reuse was more widespread than real software reuse. Nowadays, the situation is reversed, with developers reusing software produced by others regularly in the form of software dependencies, and the situation is largely unexamined. Software dependencies pose substantial risks that are all too often ignored. The transition to easy, fine-grained software reuse has occurred so quickly that there is little knowledge of the best practices for identifying and using dependencies effectively, or even determining when they are appropriate and when they are not. Installing a package as a dependency outsources the effort of developing that code — designing, writing, testing, debugging, and maintaining — to someone else on the Internet that is mostly unknown. Using that code exposes the program to all of the problems and weaknesses in the dependency. Therefore, if a dependency appears to be too problematic and there is no easy or effective procedure to isolate it, the best solution may be to avoid it totally, or at least the parts that have been recognized as the most dangerous.
The latter is essential to acquiring a critical perspective on the dependencies used in the overall architecture. There is a need to evaluate the real and unreplaceable dependencies and avoid those that are unnecessary and too critical to be adopted by the architecture. Thinking about future design with a more LIMITS, post-apocalyptic, and mostly resource-aware perspective should begin from the foundation. Lowering and simplifying the software stack's core and external dependencies significantly minimizes the time required to maintain and deploy it.

[13]`https://www.gnu.org/philosophy`
[14]`https://www.gnu.org/philosophy/floss-and-foss.html`

2 Architecture

This chapter is dedicated to displaying and describing the numerous components, their relationships, and the general needs for the cluster architecture's composition.
Understanding how the cluster works lays a good foundation for the following chapters, in which almost everything seen here is extensively explained in terms of how it was developed and why certain decisions were taken. Furthermore, the design is dynamic and may be modified to include more or fewer components and/or requirements to better meet the requirements of the end user.
To prevent resource waste at the cost of a little decreased service interruption, the overall design is structured around a high availability model rather than a fault tolerance approach. Fault tolerance is based on specialized hardware that detects a hardware fault and switches to a redundant hardware component immediately. Although the transition seems to be seamless and provides continuous service, a significant price is paid in terms of both power consumption and performance since the redundant components do no processing but are constantly operational. More crucially, the fault-tolerant paradigm ignores software errors, which are by far the most prevalent cause of downtime. High availability, on the other hand, considers availability to be a collection of system-wide, shared resources that cooperate to ensure essential services, rather than a series of replicated physical components. When a system, component, or application fails, high availability combines software and hardware to minimize downtime by quickly restoring essential services. While not instantaneous, services are generally restored in less than a minute[14].
Section 2.3 depicts a real-world functioning example of the cluster design given. It has been extensively tested and is continuously operational 24 hours a day, hosting a variety of services. Furthermore, it upscales or downscales automatically dependent on demand or requirements.
The diagram below illustrates the architecture, encompassing its components and how they are interconnected, as well as a potential connection to an external network.

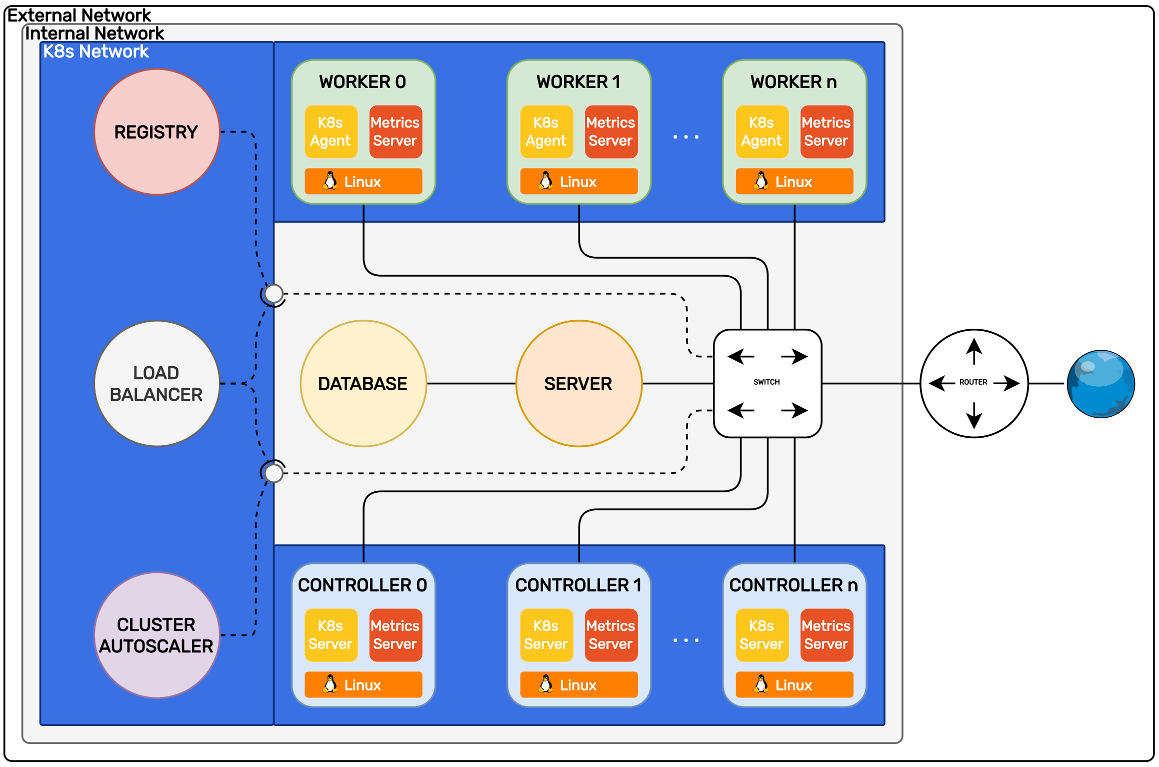


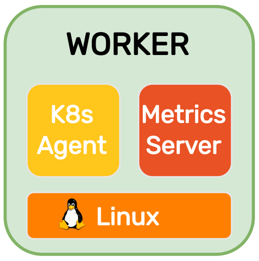
Figure 2.1: Architecture overview

2.1 Components

2.1.1 Node

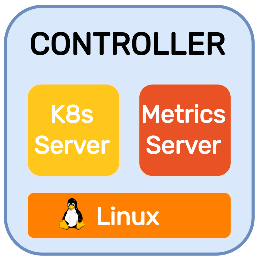
A node is a physical computer that runs the `GNU/Linux`[1][2] and constantly executes software that is specific to the cluster's composition.
Each node is physically connected to the other nodes via Ethernet and to the many operating services/-components through a virtual network. Section 2.2 goes into further detail about cluster networking.
A node can be in one of two states. The `active` state indicates that a node is turned on and is actively contributing to the cluster. The `inactive` state, on the other hand, shows that a node has been turned off and is no longer actively contributing to the cluster. This does not imply that the node is worthless and will never be utilized again, but simply that it is no longer required for the current cluster demand. A node state can be changed manually by switching the power button on or off, or automatically through the Cluster Autoscaler component, which monitors the current cluster state. More details about Cluster Autoscaler may be found in section 2.1.5.
Two core services are continuously operating on each node. The first service is a Kubernetes-compliant distribution. Kubernetes[3], also known as K8s, is an open-source solution for automating containerized application deployment, scaling, and administration. The second service, Metrics Server, is a server that constantly monitors the node, exposing hardware and operating system metrics.
Finally, each node is conceptually divided into two types depending on its role in the cluster: Worker nodes and Controller nodes. These are discussed in the sections that follow.

2.1.1.1 Worker



A worker node is designed to handle only deployable units of computation and services that are not critical components of the cluster. It is not in charge of scheduling the work over several nodes; rather, it only accepts it from a cluster-available authenticated and authorized controller node.
Even though a worker node executes the effectively scheduled workload in the cluster, it is not considered a critical component of it. At any given time, the total number of active workers might be zero. That is, there is no scheduled workload, and previously worker nodes have been shut down automatically to prevent precious resource waste that is no longer required.
The majority of the cluster's accessible machines are worker nodes. This raises the overall amount of schedulable workload as well as heterogeneity. Heterogeneity is helpful because it may help schedule workloads to nodes with the bare minimum of requested resources, preventing waste. Assume that the total number of `active` nodes in the cluster are zero and there are two `inactive` worker nodes. The first node has 4 GiB of memory and consumes 100W of power, whereas the second node has 8 GiB of memory and consumes 150W of power. A workload using around 3GiB of memory is then planned for the cluster. Because it decreases resource waste, notably memory waste, the first worker node will be chosen. It is important to note that if both nodes have an equal amount of memory, the conclusion remains the same since it has the lowest power consumption.

2.1.1.2 Controller



A controller node is an essential component of the cluster, acting as a coordinator between the active worker nodes and the overall workload in the cluster. It continually monitors the cluster's state in terms of available nodes, how and where the workload should be scheduled, and much more. A consistent and secure API must be provided for administration and end-users who wish to deploy custom services in the cluster. The API can be made available to an external network if it is available and appropriately configured, allowing remote control and improving overall usage.
To ensure the integrity and management of a cluster, at least one controller node must be constantly available. It is strongly suggested to have multiple controller nodes that meet the high availability

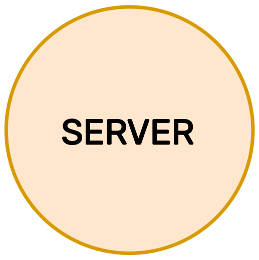
[1] `https://www.gnu.org`
[2] `https://kernel.org`
[3] `https://kubernetes.io`

model to withstand potential system, component, or application failures. This is possible considering an odd number of controller nodes (i.e. three) that are always active. A quorum of controller nodes is required for a cluster to agree on cluster state updates. Quorum[4] in a cluster with `n` controllers is `(n / 2) + 1`. Adding one node to any odd-sized controller group will always increase the number of nodes required for a quorum. Although adding a node to an odd-sized controller group appears to improve fault tolerance since there are more machines, it worsens it because the same number of nodes can crash without losing quorum but there are more nodes that can fail. If the cluster cannot withstand any more failures, adding a node before removing nodes is dangerous because if the new controller node fails to register, the cluster quorum would be permanently lost. The latter is not a strict necessity, but rather a preferable practice, even though it may slightly increase total resource waste. Consider this: if the only available controller node encounters a software or hardware failure and becomes unavailable, the entire cluster becomes unreachable and unusable.
To further decrease overall resource waste, a controller can also become a worker at the same time. If the overall workload in the cluster is very low and non-zero, having one worker and one controller active with minimal utilization at the same time is a waste. A single machine can perform the same task, saving precious resources. If the entire demand grows later and the sole active node becomes overloaded, the cluster reverts to its previous state. This is a configuration that may be enabled or disabled based on the management needs of the cluster.
It should be noted that the total number of active or inactive nodes in the cluster is not limited. However, a large number of nodes increases the workload on controller nodes, which must maintain the cluster state updated and synchronized. As a result, as shown in table 2.1 (extracted from the official K3s documentation[5]), their number and hardware requirements must be carefully balanced.

Deployment Size	Nodes	CPU Cores	RAM Memory
Small	~10	2	4 GiB
Medium	~100	4	8 GiB
Large	~250	8	16 GiB
X-Large	~500	16	32 GiB
XX-Large	500+	32	64 GiB

Table 2.1: Controller node requirements based on cluster size

2.1.2 Server

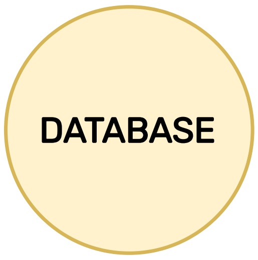


A server handles all cluster nodes, user authentication and authorization, and much more. It does not directly monitor the workload in the same way as a controller node does, but it does serve as a low-level middleware controller. Every action is made because of human intervention (e.g., administrators) or another component of the cluster that has much higher-level knowledge of the current state and reacts appropriately.
It is both an essential and a non-essential component of the cluster. It is essential in the sense that it is aware of all registered nodes, both active and inactive, and understands how to switch them on and off automatically. It reduces resource waste by automatically increasing or decreasing the number of nodes in the cluster when used in conjunction with the Cluster Autoscaler component (see section 2.1.5). Without prior information, no component in the cluster can operate as an oracle about the nodes, leaving the Cluster Autoscaler worthless and increasing total resource waste. It is also deemed non-essential in the sense that, while decreasing resource waste is the overall architecture's goal, there is no requirement for a high availability model as in controller nodes. If a server instance crashes and does not restart, leaving no more servers in the cluster, the entire cluster continues to function normally. However, unless failures, human interaction, or server restarts, the number of active nodes remains constant, potentially increasing resource waste.
A server instance does not have to run on a dedicated node. A node can hold numerous components

[4]`https://etcd.io/docs/latest/faq/#why-an-odd-number-of-cluster-members`
[5]`https://docs.k3s.io/installation/requirements#large-clusters`

at the same time, as previously mentioned. A node may be a server, a controller, and a worker all at the same time, drastically decreasing resource waste. It is crucial to note, however, that a server should not be executed on a worker node since, as previously explained, the total number of active workers might be zero, terminating the server instance and the capability of automatically adjusting the cluster size.
The server provides an API to facilitate cluster administration. Furthermore, as previously said, it is in charge of user authentication and authorization. Unharmful queries (providing information about active nodes or listing non-sensible information about a user) should not require any authentication or only a minimal one. Queries that mutate the state of the cluster or display sensitive information (turning on or off a node or displaying sensitive user information) must be protected by a high-security mechanism.
A heartbeat daemon must be implemented on the server to continuously check the condition of a node and detect any problems. A heartbeat[6] is a periodic signal or message created by hardware or software that is sent between devices at regular intervals. If the endpoint does not receive a heartbeat for an extended period, often many heartbeat intervals, the machine that should have sent the heartbeat is assumed to have failed. The latter is commonly implemented by controller nodes since they must constantly monitor and acquire information about active nodes for workload scheduling. As a result, a server can use the controller's API to eliminate duplication, increase consistency and availability, and simplify implementation.
For more information, section 3.3 covers in detail how the Server component is implemented.

2.1.3 Database



The database component is strictly related to the server component. Its primary function is to store all cluster-related data in a safe, persistent and fault-tolerant system. The data in question is generally static and does not change frequently. Static data includes all node-related information, such as CPU type and RAM quantity, that is hardly modified after the node is added to the cluster. However, some data, like information regarding the current node state and its last heartbeat, are intrinsically dynamic in the sense that the system regularly changes them to ensure integrity. Like the server component, is seen as both necessary and non-essential. A server is worthless without the database, resulting in the same outcome as previously explained.
This component is seen as the union of numerous modules that form it rather than as a single entity. A database[7], in general, is a structured collection of information or data that is persistently stored in a computer system. A database management system (DBMS) is a software program that acts as an interface between the database and its end users or applications, allowing for the retrieval, updating, and administration of how information is structured and optimized. A DBMS also facilitates database supervision and control by providing several administrative operations such as performance monitoring, tuning, and backup and recovery. The latter indicates that each module and feature is dependent on the implementation. Only for the database itself, there are several distinct types, and for each type, there are numerous distributions with varying features and capabilities. To better follow the overall design criteria, the ultimate decision on which one to take must be carefully examined.
Persistency and fault tolerance are strongly related, most importantly, in the data layer. Even if an error occurs, the only vital and critical component that must be preserved and not lost is all the stored data. Data persistency[8] is the preservation of data after the program that produced it has been terminated. To do this, the data must be written to non-volatile storage, a type of memory that can maintain the information indefinitely even if the program is no longer functioning. Fault tolerance, which differs slightly from the previous definition, refers to a non-volatile storage system's capacity to recover from an error or faulty condition, most typically an irreversible hardware failure of some type, without losing any previously stored data. Disk fault tolerance is often achieved by disk management

[6] `https://wikipedia.org/wiki/Heartbeat_(computing)`
[7] `https://www.oracle.com/database/what-is-database`
[8] `https://www.mongodb.com/databases/data-persistence`

technologies such as mirroring[9], data striping[10], duplexing[11], and Redundant Array of Independent (or Inexpensive) Disks (RAID)[12][13]. Consider the possibility that the database component has a hardware failure and ceases to operate. The damage has compromised not just the replaceable hardware (such as the CPU, motherboard, and RAM), but also some (but not all) storage devices storing the cluster's data. Typically, all data is irreversibly lost, and the cluster must be rebuilt from scratch. However, with a correctly designed disk management system, once the faulty hardware is replaced, all data is automatically recovered and the cluster becomes functional again. The process of rebuilding the data might take a significant amount of time (i.e. calculating the parity bit[13]).
Finally, an optional cache middleware can be implemented between the server and database, enhancing read speed for frequently requested but rarely updated data and reducing disk utilization. The cached data is stored in the main volatile memory.

2.1.4 Registry

The registry component is a private container registry that the K8s orchestrator uses in the cluster to download a specific image or collection of images. A container registry is a stateless, highly scalable server-side application that stores and distributes container-based application images[14].
A registry is regarded as a non-essential component because the whole system may function without it. If an internet connection is available and one of the numerous public container registries (such as Docker Hub[15]) is accessible, the cluster downloads and runs the requested image from the external network. Having a private register in the cluster, on the other hand, may solve a plethora of potential difficulties. Some of them are as follows:

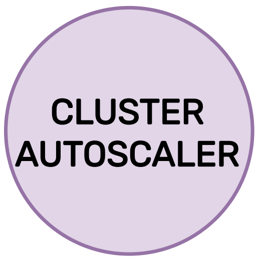
- The K8s orchestrator fails to download the requested images in an Air-Gapped environment when there is no internet connection (see section 3.2.1). As a result, the entire cluster is nearly unusable. The only realistic approach is to make a local duplicate of the remote images for each cluster node. This is hard to accomplish and prone to errors. Furthermore, there is a waste of disk memory, but it also ignores how and where the images are downloaded.
- A local solution is required for an organization that wants complete control over its images and is unwilling to put them on an external service. Furthermore, if the organization takes a security-first approach, a properly configured private registry can improve overall security.
- A newly developed image that must be evaluated before beginning release to production. Moreover, the registry may be used to improve DevOps techniques by preventing potential bugs, security vulnerabilities, and other difficulties that can arise in a real-world environment. Attachment B goes into much depth about DevOps.
- An image created exclusively for the cluster. Uploading it to a public registry is not only pointless due to incompatibility, but it also limits the available namespace because an image name must be unique.
- Registry as a pull-through cache[16]. If multiple cluster nodes require the same external image and it does not exist locally, each of them fetches it from the specified registry. This results in excessive and inefficient network utilization, which might congest the cluster. The private registry can act as a registry mirror, locally caching each externally requested image. Every node points directly to the private registry address. If an image cannot be located locally, the

[9]https://wikipedia.org/wiki/Disk_mirroring
[10]https://wikipedia.org/wiki/Data_striping
[11]https://www.pcmag.com/encyclopedia/term/disk-duplexing
[12]https://wikipedia.org/wiki/RAID
[13]https://wikipedia.org/wiki/Parity_bit
[14]https://docs.docker.com/registry
[15]https://hub.docker.com
[16]https://docs.docker.com/registry/recipes/mirror

request is routed to the local registry, which downloads and caches it. When a node requests the same image again, the registry provides the cached image without needing any further network traffic.

In practice, the registry is used to distribute the Cluster Autoscaler image (see 2.1.5) that is specifically built to function with reCluster.
Even though it is supported out of the box, the component does not need to be instantiated on a specific physical node; rather, it is deployed directly in the K8s environment. The registry may be accessible from both inside and outside the K8s network, thanks to a correctly configured Load Balancer component (further information in section 2.1.6) that exposes the service to a non-K8s-related external network. Exposing the K8s service is critical for conveniently allowing the upload (push) and download (pull) of an image without the user requiring any extra settings. The approach must remain the same as when utilizing a publicly accessible registry, but with the cluster's registry address provided. Because the registry is externally accessible, it must be properly configured to improve overall security. HTTPS, which enables secure communication over a cryptographic channel, and authentication, which permits access only to a limited group of authenticated users, must be set up and/or implemented.
The registry is deployed in the K8s environment, but the K8s orchestrator needs it to fetch the required images, leading to a circular dependence. To avoid the latter, the registry image should be made available on each node, eliminating the need for a registry request.

2.1.5 Cluster Autoscaler



The Cluster Autoscaler[17] component is a program that automatically adjusts the number of nodes in the cluster to guarantee that all deployed services have a location to execute and no nodes are underutilized. It is not in charge of node behavior and registration in the cluster; it only determines if a node should be turned on or off, but not how. As a result, for low-level control, it must rely on the Server component. Nonetheless, the autoscaler is an essential component of the cluster due to its automation, which eliminates the need for human interaction and dramatically lowers resource waste.
The Cluster Autoscaler is deployed in the K8s environment rather than on a separate node. After the registry becomes available, the container image of the Cluster Autoscaler is uploaded and made available to the whole cluster during the initialization phase. Because the registry is defined as a critical K8s service, it may be scheduled on any node, both workers and controllers. Since the cluster may scale to zero workers, this applies to every critical component in the K8s environment, preventing a service outage. Furthermore, the cluster autoscaler must be maintained operational at all times: if a crash or error happens, it must be restarted. The latter is accomplished automatically by the K8s system, removing the need for a custom solution.
The component has both a high-level and a low-level view of the cluster. The high-level view is accomplished by continuously monitoring the cluster's condition to ensure its integrity: workload and active nodes. As a result, it is aware of whether there are services that cannot be deployed owing to a lack of resources, or if there are underutilized active nodes for the present demand, resulting in resource waste. This is accomplished by directly querying the protected API provided by the controller nodes. Far from it, the low-level view is obtained by continually updating a local cache that contains all of the available nodes in the cluster, both active and inactive, to know how many nodes are available and their status. As a result, it is aware of the nodes that can be turned on or off, as well as if the system can be scaled up or down. As previously stated, the latter is accomplished by accessing the server on a series of secured API endpoints that provide direct control over how the cluster's nodes are handled.
The prerequisites and solutions for automatically adjusting the cluster size are outlined below:

- There are some computing units (pods) that are unable to operate in the cluster owing to a lack of resources.

[17] `https://github.com/kubernetes/autoscaler/tree/master/cluster-autoscaler`

If the cluster's current total workload with the present number of active nodes has reached its maximum, it will be unable to support further service deployments. This indicates that newly deployed services are never executed and are always in a state of waiting. In this condition, the controllers continue to monitor the cluster's condition for a suitable node with enough free resources to host a waiting service. Waiting forever for a current active node to free up some resources affects the cluster's overall usability and quality of service (QoS). The solution is to check if there is a suitable inactive node in the local cache that can be turned on. If such a node is identified, a specific request is issued to the server, and the queued deployments are automatically scheduled to the newly bootstrapped node.

- Some nodes are constantly unneeded for an extended period. When a node's resource utilization is low and all of its scheduled deployments can be transferred to another node, it is no longer required.
  There is a waste of resources if the current workload on a node falls below a pre-defined threshold for a certain period. To avoid this, all deployments on the node must be rescheduled to another node with sufficient free resources, raising its overall utilization. Following completion of the latter, the unneeded node has almost zero workloads (non-zero since there may be system deployments running that do not require a migration and may be deleted) and can be switched off and removed. The Cluster Autoscaler makes a specific request to the server with the node identifier; the server removes the node from the K8s environment and subsequently shuts it off altogether. When a node is successfully shut down, its status changes from active to inactive, indicating that it is a candidate for an eventual upscaling.

2.1.6 Load Balancer

The Load Balancer[18] component provides an externally accessible IP address to an internal K8s application. It may be seen as an abstract mechanism of exposing the application as a network service without requiring an unfamiliar service discovery mechanism.
An application can be composed of multiple replicas that are generated and destroyed to maintain the cluster in the desired state. The replicas can be spread over many active nodes, improving overall performance, availability, and fault tolerance. When an application is created in a K8s environment, there is no way to connect from outside the cluster by default. As a result, only other K8s deployments can communicate with it. A load balancer acts as a solution by routing external network traffic to the cluster node where a replica of the application is deployed. To prevent scenarios where some replicas are overloaded and others are not, it does allow different network traffic policies (such as Round-robin[19], random, and others) to homogeneously distribute traffic across all replicas. As a result, it serves as a consistent and fixed entry point for connecting to the potentially numerous application replicas that are intrinsically dynamic due to their non-permanent resources (i.e., internal IP address).
At first, considering the architecture schema, it is unclear if it is an essential or non-essential component. Its utilization, however, is tightly associated with the kind and usage of deployed applications/-components in the cluster. For example, if the cluster is intended to host numerous web applications that must be accessed over the internet, a load balancer component is necessary to expose each application to external network traffic. If the cluster is just used for testing or DevOps techniques (see attachment B), there is no need to expose any applications, leaving the load balancer components unnecessary. As a result, it can be removed, increasing the total available resources.
In data centers operated by huge cloud providers like Microsoft[20], Google[21], or Amazon[22], a load balancer is generally an external, sophisticated, and highly specialized component that is specifically designed to be compatible with the specific cloud provider. The general architecture in question, how-

[18]`https://kubernetes.io/docs/concepts/services-networking/service/#loadbalancer`
[19]`https://wikipedia.org/wiki/Round-robin_item_allocation`
[20]`https://azure.microsoft.com`
[21]`https://cloud.google.com`
[22]`https://aws.amazon.com`

ever, is far simpler and operates on bare-metal servers. A bare-metal server[23] is a physical computer that is exclusively operated by one customer or tenant. As a result, an external load balancer may be substituted with an internal one that is deployed directly in the K8s environment. An internal load balancer minimizes overall resource usage while also improving maintainability. Furthermore, as with the registry and Cluster Autoscaler components, the internal load balancer is kept automatically operational and in good working condition at all times by the K8s orchestrator.

2.1.6.1 Example Schema

Figure 2.2 depicts an example schema of an Internal Load Balancer in a cluster architecture. It is worth noting that the Internal Load Balancer component is presented externally to the nodes to simplify the overall overview but in reality, it is deployed directly on each active node.

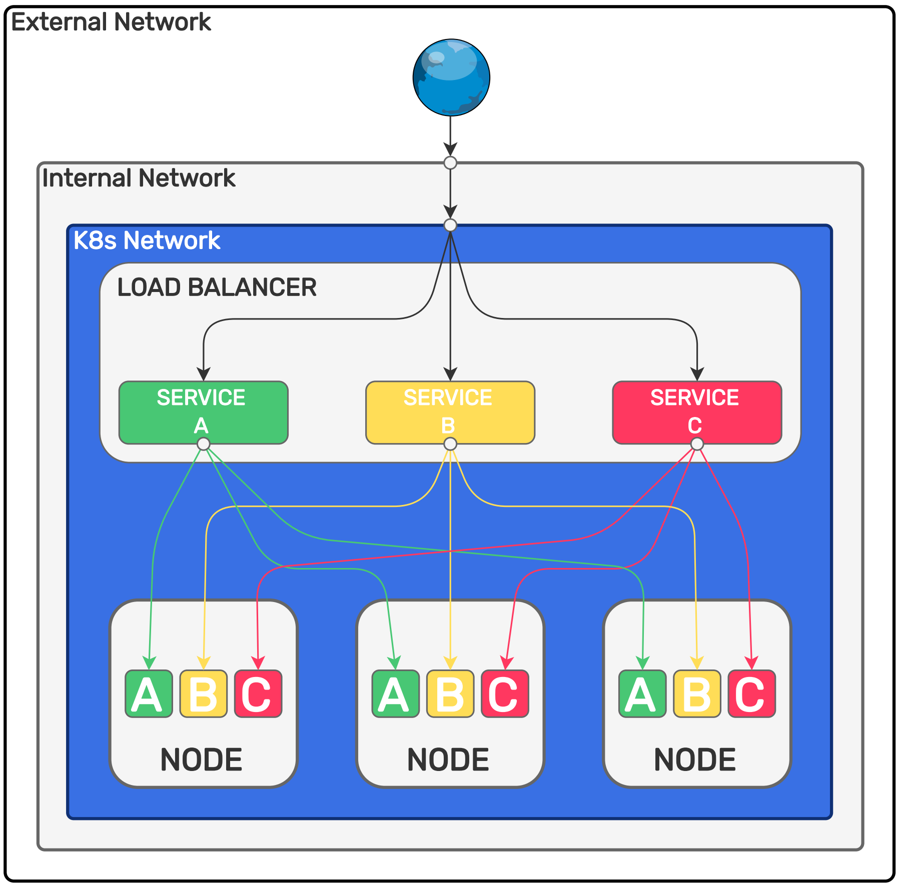


Figure 2.2: Internal Load Balancer schema example

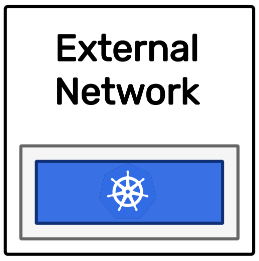
There are three active nodes and one Internal Load Balancer in the cluster. The nodes' roles have not been specified, however as stated in section 2.1.1, each node can be regarded as both a Controller and a Worker. Three applications, `A`, `B`, and `C`, have been deployed in the K8s environment, with three replicas available for each. Concerning the other replicas, each replica is scheduled on a distinct node. The organization in charge of the cluster wishes to expose each application by assigning each one a unique externally accessible IP address from the Internal Network (i.e., a private IPv4 address). It should be noted that the process of mapping/routing an IP address from the Internal Network to an IP address from the External Network is not mentioned. The Load Balancer creates a matching internal service for each application that knows where all the replicas are installed and how to efficiently distribute traffic among them. Assume that application `A` is assigned the IP address `10.0.0.97` and that the Load Balancer receives an external request from a client on the Internal Network. The Load Balancer examines the target address and finds a match; the request is then routed to internal service `A`. Service `A` then redirects the request to one of the active nodes where a replica of application `A` is deployed. Finally, the replica handles the request, and a response is created and sent back to the client.

[23]`https://wikipedia.org/wiki/Bare-metal_server`

2.2 Network

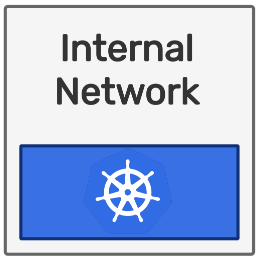
The cluster is composed of several network layers, each of which may interact with components of the same layer but by default, not with components of a different layer. Each network layer is heterogeneous: various software/hardware components and applications correlate to different network layers. The cluster can support communications across layers, but this must be done transparently and accurately to avoid significantly increasing the cluster's complexity and resource waste while lowering its overall security. Both software and hardware components that reside between the two communicating levels enable inter-layer communication. The most fundamental requirement for the latter to work is that there be no potential conflicts between the two communicating layers.
Some network layers of the cluster can be directly compared with `ISO/OSI` model's layers, depending on the requirements and capabilities.
There are three distinct network layers in the cluster architecture seen in figure 2.1. They are covered in the sections that follow, starting with the most external network and progressing to the most internal network in relation to the cluster.

2.2.1 External Network



The external network is any network that does not directly connect to the cluster. It denotes any network that is incompatible with the Internal Network (see the section 2.2.2) and requires a Router or other similar components to communicate with the latter. To put it simply, the External Network is any existing network in the world that is not the Internal Network, ergo the Internet: the network of networks.
The External Network is analogous to Layer 3 of the `ISO/OSI` communication model, defined as the Network Layer, which is in charge of facilitating data transfer between two networks.
If the deployed applications may be accessible by external clients and there is a requirement for external resources, such as public container images, that are not locally available in the cluster, connectivity to the external network is required. Furthermore, if the organization in charge of the overall architecture wishes to have remote control and monitoring of the cluster from anywhere in the world, the only current practical and supported solution is an internet connection. If the latter is not required, the external network can be omitted from the cluster design, resulting in an Air-Gap environment without internet access. As a result, all required resources and components are already available locally, and access to the cluster is restricted to the internal network.
Finally, an External Network is inherently insecure. As a result, high-security protocols, methods, and procedures must be created and effectively set to avoid any possible disservices. Although cluster security should be implemented, it is not currently the primary focus, and only some basic procedures are used as it is such a broad and complicated topic.

2.2.2 Internal Network



The Internal Network uses one or more layer 2 network switches to connect each physical node in the cluster. Simply described, the Internal Network is primarily direct device-to-device communication with no interaction with external networks. Another switch, router, or computer might be considered a device. Because the Internal Network is usually composed of two or more linked devices and is located in a limited geographic area, it is classified as a LAN (Local Area Network).
The Internal Network is analogous to Layer 2 of the `ISO/OSI` communication model, defined as the Data Link Layer, which allows data to be sent between two devices on the same network.
Because of the network design at Layer 2, communication between internal devices in a LAN is based on MAC addresses rather than IP addresses, which are utilized at Layer 3. As a result, network switches forward data based on the destination MAC address. A MAC address (Media Access Control address) is a unique sequence of 12 hexadecimal digits permanently stored in the Network Interface Controller (NIC) that typically encodes the manufacturer's registered identification number. A device

can have multiple NICs, each with its own unique MAC address. One key difference between IP addresses and MAC addresses is the former is assigned dynamically to each device and may change over time, whereas the latter are assigned permanently. This distinction is critical since the device's MAC address may be used to uniquely identify a device in the same network and automatically bootstrap it. The latter is known as Wake-on-LAN (WoL) and is employed by the server component in the cluster. Chapter 3 contains further information regarding automatic cluster upscaling.
A router device is required if the Internal Network wants to communicate with other networks to access various resources or execute external operations. As a result, the router (Layer 3) is directly linked to a switch (Layer 2), which connects every node in the cluster. The latter link between the two layers is regarded as the boundary between the secure Internal Network and the insecure External Network. Furthermore, the total number of switch and/or router devices in the architecture is not predetermined and can be increased to improve overall cluster network performance and fault tolerance. If an external connection is not required, the whole design is reduced and made up of solely L2 network switches, eliminating the requirement for routers. Without them, a considerable amount of resource waste is reduced, not just in terms of pure hardware that can be reutilized as worker nodes, but also in terms of overall cluster power consumption, because a network switch is more power-efficient than a router[2].
Finally, a local domain name may be used to conveniently identify nodes in the Internal Network with a hostname (i.e., `recluster.local`)[11]. As a result, instead of an IP address or MAC address, a Worker node may be recognized with the hostname `worker.recluster.local` and a controller node with the hostname `controller.recluster.local`.

2.2.3 K8s Network

"Kubernetes is all about sharing machines between applications"[24]

Premise: the purpose of this section is not to demonstrate how Kubernetes networking works or how it can be set and configured. Kubernetes networking is a vast and complex topic that necessitates a high degree of competence in a variety of fields. Furthermore, it is always evolving, providing new features and supporting newer technology. As a result, this section only contains fundamental and essential information regarding Kubernetes networking that is considered at its core and will not change in future releases.

The Kubernetes orchestrator networking is represented by K8s Network. It is a virtual network, as opposed to the Internal and External networks, in the sense that there are no physical connections between the various K8s components, only logical ones. The communication between the various K8s components is coordinated and distributed over all active nodes. Unless properly configured via a particular K8s Service, in the cluster architecture represented by the Load Balancer component, by default the various deployed containers in the K8s environment are not allowed to communicate outside the K8s network.
In contrast to the previous two network layers, K8s Network does not map to a single layer of the ISO/OSI communication model. Instead, it covers numerous ISO/OSI layers, starting from the Data Link Layer (Layer 2) up to the Application Layer (Layer 7), which is the highest.
K8s Network is regarded as an overlay network. An overlay network is a network that is established on top of a physical network that already exists. Because the overlay creates the virtual network through encapsulation, it relies on the so-called underlay network for fundamental networking functions such as routing and forwarding. Most overlay networks are created in the ISO/OSI communication model's Application Layer (Layer 7) on top of the TCP/IP networking suite. Overlay technologies can also be utilized to overcome some of the underlay constraints while also providing additional routing and forwarding capabilities without modifying the physical networking infrastructure. An overlay network's nodes, represented as K8s instances, are linked together by logical connections that can span numerous physical links. A connection between two overlay nodes may require multiple hops in the underlying network. In the underlying network, a link between two overlay nodes may require

[24]`https://kubernetes.io/docs/concepts/cluster-administration/networking`

multiple hops. Figure 2.3 compares the physical network to the virtual K8s overlay network that is created on top of it. Take note of the overall network simplicity and the lack of connection hops between each K8s instance.

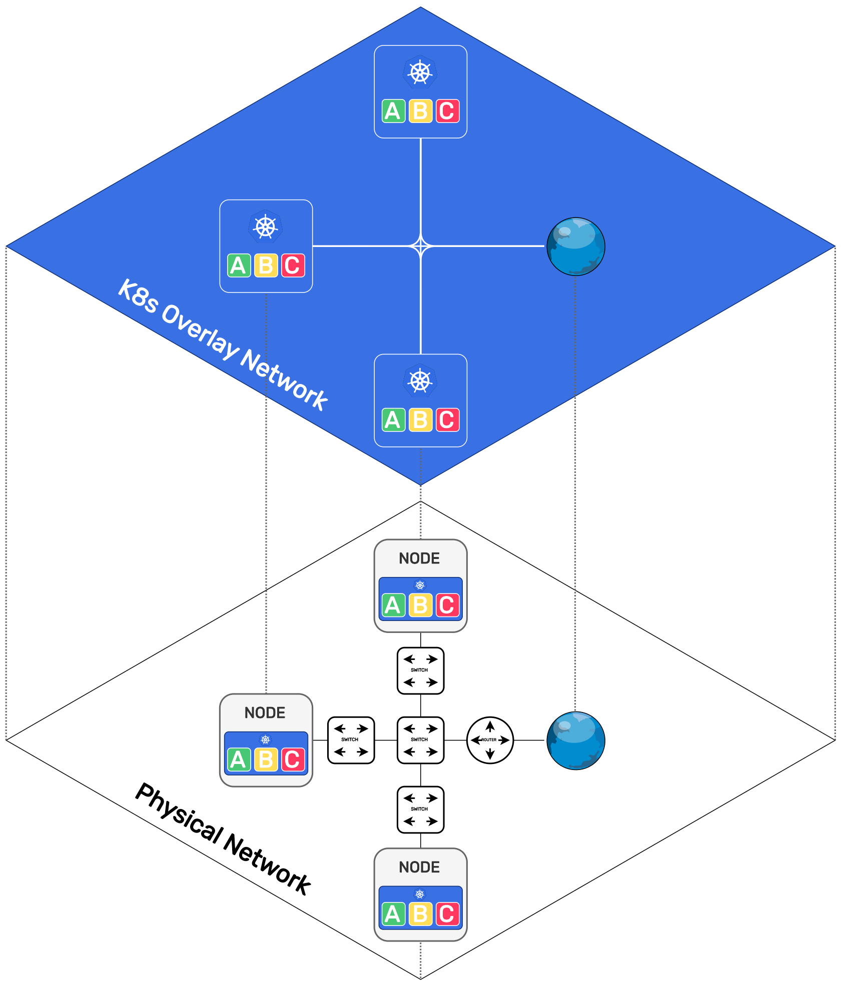


Figure 2.3: Comparison between the Physical Network and the virtual K8s Overlay Network

The Kubernetes networking stack, and, more generally, the Kubernetes ecosystem, is built following a set of standards and specifications. This is critical for two main reasons. It imposes an overall rigorous structure and design that must adhere to the standard's specifications, no more, no less. Finally, before a standard specification can be amended or introduced to the K8s ecosystem, it must go through a sequence of interactions and discussions involving various parties, spanning from Kubernetes developers to single users. As a result of this, the Kubernetes ecosystem is incredibly adaptable, and it can be customized and configured with several plugins and alternative implementations at various levels to represent the end purpose and intent of the organization in charge of the cluster. The latter capability has been employed in a real-world implementation to switch from one network backend to another. The first backend encapsulates and encrypts all packets transferred from one node to another. There is no need for such rigorous security measures in the cluster. Therefore, the backend has been modified to need just direct Layer 2 communication between the nodes without any encryption. Moreover, the latter not only needed fewer dependencies but also improved overall performance because the extra procedures were no longer required.
The following information is essential for understanding the basic principles of K8s networking. They were collected from the official Kubernetes website, which can be found at `https://kubernetes.io`. Sharing computers often necessitates ensuring that two applications do not attempt to use the same ports. Port coordination, also known as dynamic port allocation, is extremely difficult to do at scale and exposes cluster-level issues. Kubernetes, on the other hand, takes a different approach.
Every Pod[25], which are the smallest deployable units of computing that can be created and managed in

[25] `https://kubernetes.io/docs/concepts/workloads/pods`

a cluster, is assigned its cluster-wide IP address. This eliminates the need to explicitly construct connections between Pods and deal with container ports mapping to node ports. As a result, Kubernetes enforces the following basic criteria for any networking implementation:

- Pods can communicate with all other pods on any other node without the use of NAT (Network Address Translation). NAT is a means of transparently mapping an IP address space to another[16].
- Kubernetes agents (such as system daemons, `kubelet`, and so on) on a node can communicate with all pods on that node.

Kubernetes IP addresses exist at the Pod scope; containers within a Pod share their network namespaces, including their IP address and MAC address. This implies that containers within a Pod may all contact each other's ports on `localhost`; requiring port coordination.
Four concerns are addressed by Kubernetes networking:

1. *Highly-coupled container-to-container communications*
   Containers within a Pod interact through networking via loopback.
2. *Pod-to-Pod communications*
   Cluster networking provides connectivity among different Pods.
3. *Pod-to-Service communications*
   Use Service components to publish services that are only accessible within the Kubernetes cluster.
4. *External-to-Service communications*
   Let an application running in Pods be accessible from outside the cluster by exposing it as a Service. The Load Balancer Service component, mentioned in the section 2.1.6, is one example.

The container runtime on each node implements the Kubernetes network model. Container runtime software, such as containerd[26] or CRI-O[27], is responsible for operating the containers. Container runtimes handle their network and security capabilities via the Container Network Interface (CNI). CNI[28] is composed of a specification and libraries for designing plugins that configure network interfaces in Linux containers. CNI is only concerned with the network connectivity of containers and removing allocated resources when the container is destroyed.

2.3 reCluster

The architecture has a real-world implementation in the reCluster project[29].
reCluster employs several hardware components, such as decommissioned University of Trento[30] desktop and laptop computers that were previously used by students in the computer classroom, as well as additional personal hardware components, such as the switch/router, that were left to gather dust due to an upgrade to a newer model.
Figure 2.4 illustrates the reCluster cluster. Worker nodes are the desktop computers and the two small single-board computers on the right, while the (single) Controller node is the laptop computer. All nodes are connected via Ethernet to the switch/router on the left, which is connected to the Internet. Furthermore, the smart plug in the foreground is utilized for power measurements throughout the installation procedure (see section 3.5.2).
The cluster is fully functional and can be scaled down to zero Worker nodes by default to consume the least amount of energy. Furthermore, because the cluster is small, there are (now) no critical services deployed, and to further decrease the overall power consumption, all essential components are directly deployed on the Controller node. As a result, the laptop computer acts as the Controller

[26]`urlhttps://containerd.io`
[27]`https://cri-o.io`
[28]`https://www.cni.dev`
[29]`https://github.com/carlocorradini/reCluster`
[30]`https://www.unitn.it`

node, running instances of the Server and Database, and has the Registry, Load Balancer, and Cluster Autoscaler containers deployed by Kubernetes.
If there is no need to use the cluster for some time, the Controller node (as well as the switch/router) can be terminated. Eventually, to restore the cluster, simply bootstrap the Controller node (along with the switch/router), and the rest is performed automatically by the Server and Cluster Autoscaler components according to the overall workload.

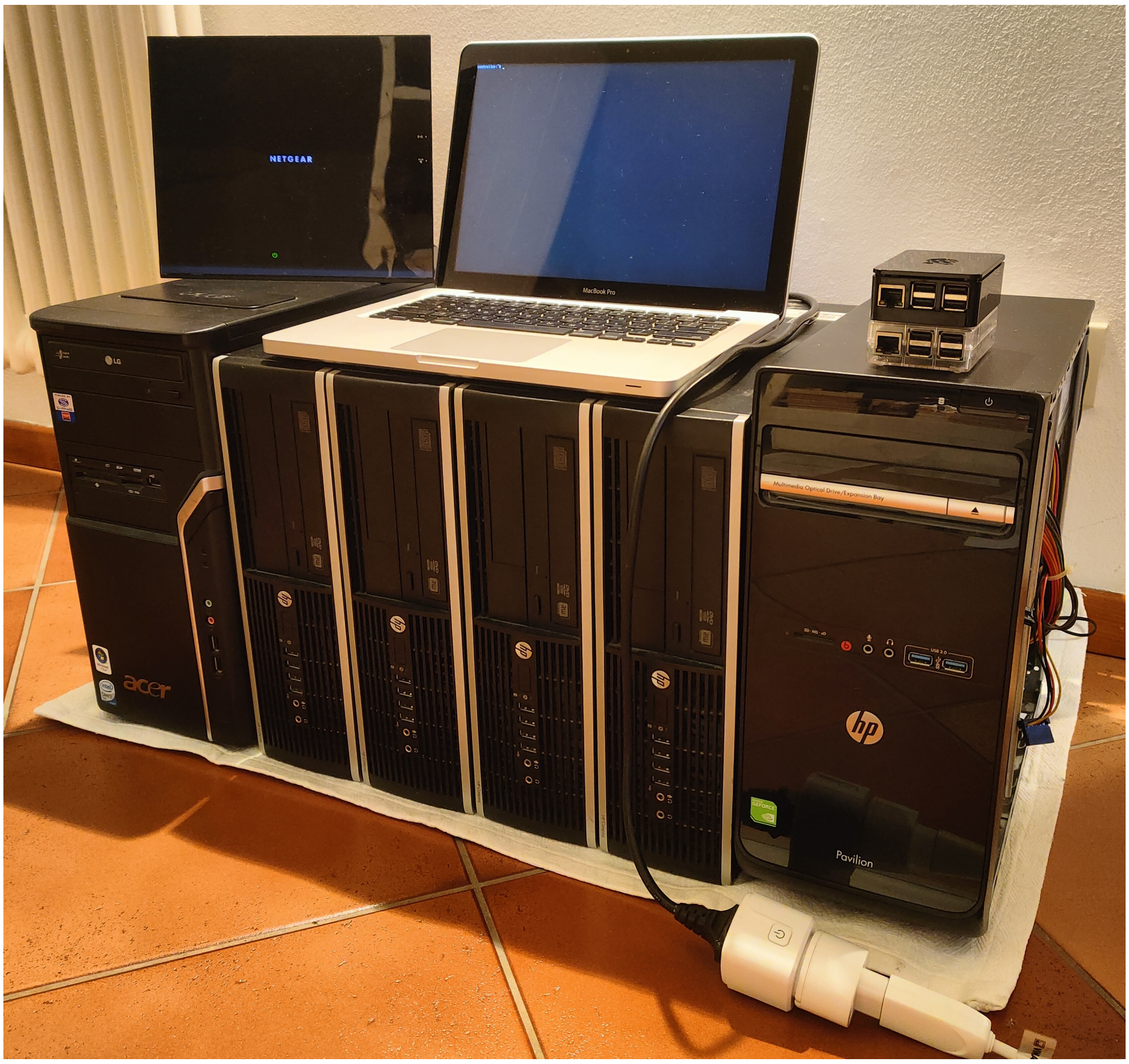


Figure 2.4: reCluster cluster

The following chapter, Implementation, describes the details and information of how reCluster was implemented by following the principles and the theorized architecture, as well as the numerous decisions that were made.

It should be mentioned that anybody can have a local implementation of reCluster since the required components are so minimal and ubiquitous that they can even be refurbished from old and unused components, as in this scenario.

3 Implementation

Ideas are easy, Implementation is hard.

Guy Kawasaki

Most of the implementation details and general decisions made for developing a real-world functional project, based on the design architecture presented in chapter 2, are discussed in this chapter. It is organized in such a way that it starts with the foundation and then progresses to the autoscaling of worker nodes before moving on to example deployments of various applications in the cluster.
The technology and programming languages utilized in the implementation are heterogeneous. Nevertheless, according to a shared set of APIs and pre-defined structures, they interact as a distinct and homogenous entity that maintains the entire cluster in an active and healthy state.
Furthermore, several of the components and/or technologies discussed in this chapter are interchangeable with other solutions. The latter is highly valuable for organizations since it provides more configuration freedom and final control over the cluster while continuously reflecting its ultimate goal.
The main implementation of the cluster and its related applications currently supports the `Linux` platform (see section 2.1.1) together with the architectures `amd64`[1] and `arm64`[2].

3.1 Distributions

Almost all software relies on the necessity for a stable and consistent operating system (OS).
The OS, as specified in section 2.1.1, is installed on every node in the cluster and must be based on `GNU/Linux`. The latter is a critical requirement because most of the technologies employed in the implementation rely on Linux primitives and core functionality to operate properly. To enforce resource management for pods and containers, for example, all Kubernetes distributions require the `cgroup v2`[3] kernel module[4].
To represent the potential of the implementation architecture being compatible with several Linux distributions, the section is titled Distributions rather than Operating System.
To be completely compatible with the implementation architecture, a Linux distribution must meet three major requirements. They are described in the following list:

1. The Linux Kernel version must be compatible with the Kubernetes version.
   Newer versions of K8s may require newer versions of the Linux Kernel to provide newer functions and/or improve overall security.

2. All necessary packages, as explained and documented in the section 3.1.1, must be installed and globally accessible in the system.
   A check on all essential packages is performed before the execution of the installation program, as detailed in section 3.5. If even one package is missing from the system, the installation fails with an error message identifying the missing packages.

3. An Init System that is supported and compatible, as defined in the section 3.1.2.
   OpenRC and Systemd are supported by the implementation. They are currently the two most used init systems, and the majority of available distributions are based on one of them.

Finally, two custom Linux distributions are already configured, built, and packaged as ISO files, as shown in section 3.1.3. They are based on a simple command line with no graphical interface and

[1]`https://wikipedia.org/wiki/X86-64`
[2]`https://wikipedia.org/wiki/AArch64`
[3]`https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v2.html`
[4]`https://kubernetes.io/docs/concepts/architecture/cgroups`

contain only what is required to start a node. This is done to simplify the overall setup by eliminating unnecessary software and technologies, addressing any incompatibilities, and increasing the cluster's implementation and testing efficiency. Both are derived from pre-existing Linux distributions that are known to be stable, easy to configure, and lightweight, requiring just the bare minimum of resources to operate. Furthermore, they are built on separate init systems, requiring the overall implementation to be compatible with and tested with both.

3.1.1 Packages

The packages used in the implementation that must be available in a Linux distribution are specified in this section.
All packages may be installed automatically using the package manager of the respective Linux distribution.
All packages are already included by default in the two custom Linux distributions. As a result, there is no need for any changes or the installation of extra packages.
Each package in table 3.1 contains a brief description of why it has been utilized. Furthermore, the required column indicates with symbols if the package is required (✓), optional (?), or not required (✗). It should be noted that the name of some packages may differ based on the package manager employed by the Linux distribution.

Name	Required	Description
`bc`[5]	✓	The Linux terminal does not support complicated math operations or number comparisons. As a result, the `bc` program is used throughout the installation procedure to guarantee that computed data, such as the node's mean power consumption and Simple Standard Deviation[6], is neither less than nor more than a threshold value.
`coreutils`[7]	✓	By default, minimal Linux distributions only include a portion of the massive list of utilities provided in the standard `coreutils` package. As a result, several of the applications utilized in the implementation are missing. Therefore, the whole `coreutils` package is required.
`docker`[8]	?	During the cluster initialization procedure, `docker` is widely utilized to tag and push various container images corresponding to specific implementation dependencies to the private registry (see section 3.2.5). It should be noted that `docker` is not the only tool for working with OCI containers, hence the optional symbol, but it is the most popular with well-known and simple CLI commands. As an example, `podman`[9] is a Docker-compatible command line front end that can simply alias the Docker CLI (`alias docker=podman`), requiring no modifications to the implementation code.
`ethtool`[10]	✓	If the node's NIC interface supports Wake-on-Lan (WoL), `ethtool` is used to activate it. It's worth noting that the NIC interface may reset to factory settings following a reboot. As a result, `ethtool` is always run before the NIC interface is activated.
`inotify-tools`[11]	✓	Obtaining access information to the underlying Kubernetes cluster is required during the cluster initialization phase. This information is only available in an automatically generated configuration file saved in a specific and well-known directory. As a result, before proceeding to the next phase, the installation script uses the `inotifywait` program to wait for the file to be created.

iproute2[12]	✓	During the installation phase, the command ip is used to gather information about physical network interfaces that are not loopback, such as name and MAC address.
jq[13]	✓	To manipulate JSON data, all scripts make extensive use of jq.
ncurses[14]	✗	The ncurses package includes the tput software, which is a utility for retrieving terminal capabilities in shell scripts. When the spinner process is spawned in a script, tput is used to update its symbols at regular intervals, replacing the old one with the new one and updating the cursor position. Because a spinner is only a decorative element, the package is not required and can be omitted.
nodejs[15]	?	This package is optional since the Server component, see section 2.1.2, may be bundled as a single binary or in a container, removing the need for the nodejs package to be installed. Because the present implementation does not provide any of the aforementioned alternatives, nodejs is included in both of the two custom Linux distributions.
npm[16]	?	This package is optional since the Server component may be bundled as a single binary or in a container, removing the need for the npm package to be installed. Because the present implementation does not provide any of the aforementioned alternatives, npm is included in both of the two custom Linux distributions.
openssh[17]	✓	OpenSSH is critical in the cluster for protecting remote operations, key management, and other tasks. All operations are inherently unsafe without its software and can pose serious security issues, especially if the cluster is accessible from external networks.
postgresql[18]	?	PostgreSQL is the database implementation used for the Database component (see section 2.1.3). This package is optional since the database may be implemented outside of the cluster using other solutions. It is up to the organization in charge of the cluster to decide whether or not the database package should be included in the Linux distribution.
procps-ng[19]	✗	The procps-ng package includes the ps software, which displays information about the system's active processes. When the spinner process is spawned in a script, ps is used to obtain the PID of the parent process. Because a spinner is only a decorative element, the package is not required and can be omitted.
sudo[20]	✓	The installation phase involves actions that need administrative authorization. Before performing any processing, the script checks to see whether the current user already has root privileges; if not, it requests them with sudo.
sysbench[21]	✓	During the installation phase, sysbench is extensively used to analyze a cluster node's performance and, in conjunction with other tools, its power consumption.
tar[22]	✓	tar is primarily used in production during the installation process to extract and manipulate various archive files. tar is used in development to create the bundle archive for distribution, which contains all files and applications.
tzdata[23]	✓	To avoid date and time inconsistencies, all nodes in the cluster must be configured to the same timezone. By default, all nodes are configured to the Etc/UTC timezone.

`util-linux`[24]	✓	The installation script makes use of two `util-linux` utility programs: `lscpu` to display information about the CPU architecture and `lsblk` to list block devices.
`yq`[25]	✓	To manipulate YAML data, all scripts make extensive use of `yq`. `yq` is primarily used to read and save YAML files. However, because of its more advanced functions and solutions, `jq` is used for the majority of complex data processing, at the expense of `yq`. As a result, YAML structures are first transformed to/from JSON, then processed with `jq`, and then converted back to/from YAML. The double conversion should be eliminated and/or simplified in future implementations.

Table 3.1: Packages list

3.1.2 Init System

The Init System[26], short for Initialization System, is the first process that runs when a Unix-based computer operating system is bootstrapped. Init is a daemon process that is normally given Process IDentifier 1 (PID 1) and runs until the system is shut down. It is the direct or indirect ancestor of all other processes and adopts all orphaned processes automatically. The kernel starts Init during the boot process; if the kernel is unable to start it, a kernel panic occurs.
The Init System is logically positioned above the Kernel and is an essential component of every modern system. It frequently integrates sophisticated features over simple ones and is very customizable. Furthermore, it offers active control and monitoring over the running processes, keeping the entire system healthy and active. The latter is critical because it enables automated and continuous monitoring of cluster implementation architecture-related processes operating in the node without the need for a custom solution. For example, if a K8s instance executing in a worker node crashes (i.e. exits with a code different than 0), the Init System detects it and restarts the process automatically, making the node active and able to receive workload again.
When the node is bootstrapped, the Init System is set to start all cluster-related processes (i.e., K8s instance and Metric Server) and run a script that sends a message to the server component notifying it that the node is active and ready. Before the node is shut down, it automatically terminates all cluster-related processes and sends a message to the server component informing that the node is shutting down and will be inactive. As previously stated, the cluster performs these two operations automatically: bootstrap for upscaling and shutdown for downscaling. However, there are scenarios

[5] `https://www.gnu.org/software/bc`
[6] `https://wikipedia.org/wiki/Standard_deviation`
[7] `https://www.gnu.org/software/coreutils`
[8] `https://docs.docker.com/engine`
[9] `https://podman.io`
[10] `https://www.kernel.org/pub/software/network/ethtool`
[11] `https://github.com/inotify-tools/inotify-tools`
[12] `https://wiki.linuxfoundation.org/networking/iproute2`
[13] `https://stedolan.github.io/jq`
[14] `https://invisible-island.net/ncurses`
[15] `https://nodejs.org`
[16] `https://www.npmjs.com`
[17] `https://www.openssh.com`
[18] `https://www.postgresql.org`
[19] `https://gitlab.com/procps-ng/procps`
[20] `https://www.sudo.ws`
[21] `https://github.com/akopytov/sysbench`
[22] `https://www.gnu.org/software/tar`
[23] `https://www.iana.org/time-zones`
[24] `https://github.com/util-linux/util-linux`
[25] `https://mikefarah.gitbook.io/yq`
[26] `https://wikipedia.org/wiki/Init`

in which a node is manually bootstrapped and/or shut down by human action without involving the server component directly. Despite the possibility of the latter, the overall cluster state remains consistent, and all node statuses are correctly updated, thanks to the Init System and its monitoring and configuration capabilities.
The implementation currently supports two init systems: OpenRC (described in section 3.1.2.1) and Systemd (described in section 3.1.2.2). They are the two most popular and widely used Init Systems, and the vast majority of Linux distributions support one or both of them. Their capabilities are nearly identical. However, configuration files and available application(s) that interact with the bare-metal Init System API, are different, necessitating double effort to achieve the same objective.
It should be noted that if the Init System daemon process crashes due to an error, the entire system crashes and becomes inactive, necessitating a manual restart. Even though the likelihood of the latter is extremely low, it is never zero. As a result, as stated in section 2.1.2, a heartbeat mechanism has been established in the cluster to constantly monitor and identify the status and availability of all active nodes. If there are no configuration or networking issues and a node's heartbeat is not received within the predefined interval, it indicates that the heartbeat daemon executing in the node has crashed and has not yet been restarted, or that the entire node has crashed and is not automatically recoverable.
In each of the following sections, where an Init System distribution is described, an example of the related service configuration file is presented to illustrate its differences. A service configuration file contains information about a process that the Init System controls and monitors. The application used as a reference in this example is K3s, a lightweight Kubernetes distribution explained in further detail in section 3.2.3. Before starting, it must clear any temporary files in the `tmp` directory and load environment variables from two distinct directories. Furthermore, anytime the application process crashes, the Init System must restart it within 5 seconds after detection. Even though each Init System has a unique service configuration file, the result is nearly identical. The list below highlights some of the most significant configuration requirements. To aid comprehension, the identification number for each requirement is also provided in the Init System's service configuration file.

❶ General information about the service.
Useful for users and/or administrators.

❷ Services on which the service is dependent.
The listed services must be activated before the current service may start.

❸ Remove any temporary files that begin with the name `k3s` and are placed in the temporary (`tmp`) directory before launching the application.

❹ Type of service. Different types correspond to various service behaviors as well as how the application is monitored and regulated.
In this case, monitor the application process and restart it if there is a crash or an anomaly.

❺ Absolute path to the application's binary file.

❻ Application arguments.

❼ Waiting time in seconds before restarting the crashed application.

❽ The maximum number of restarts permitted. When the threshold is met, the program is no longer restarted.
A value of 0 indicates that no threshold exists and that the application has no restart limit.

❾ Load `/etc/environment` and `/etc/rancher/k3s/k3s.env` environment variables files before starting the application.

3.1.2.1 OpenRC

OpenRC[27] is a dependency-based init system for Unix-like systems that maintain compatibility with the system's init system, which is generally found in `/sbin/init`.
At boot, OpenRC starts the appropriate system services in the right sequence, manages them while the system is running, and stops them at shutdown. It can manage installed daemons, optionally monitor the processes it launches, and start processes in parallel when possible to reduce boot time.
OpenRC was developed for Gentoo[28], but it may be used in other Linux distributions and BSD[29] systems as well.

Source: `https://www.gentoo.org/inside-gentoo/artwork/gentoo-logo.html`

Figure 3.1: Gentoo Linux logo

Listing 3.1 illustrates the example of the OpenRC service configuration file for K3s saved in `/etc/init.d/k3s`. Manual control of the service is possible using the `rc-service` command, which locates and runs the OpenRC service with the specified arguments: `rc-service k3s <CMD> [...]`. `<CMD>` can be substituted by `start` to start the service, `stop` to stop the service, `restart` to restart the service, and `status` to acquire the service status. Furthermore, with the command `rc-update add k3s default`, the service may be set to start automatically when the system boots up. The latter examples are merely a very small subset of all the available commands/capabilities that OpenRC is capable of.

```
#!/sbin/openrc-run

name="k3s"                                                                ❶
description="Lightweight Kubernetes"                                      ❶

depend() {
  want cgroups                                                            ❷
  after network-online                                                    ❷
}

start_pre() {
  rm -f "/tmp/k3s.*"                                                      ❸
}

supervisor="supervise-daemon"                                             ❹
command="/usr/local/bin/k3s"                                              ❺
command_args="server"                                                     ❻
output_log="/var/log/k3s.log"
error_log="/var/log/k3s.log"
pidfile="/var/run/k3s.pid"
respawn_delay=5                                                           ❼
respawn_max=0                                                             ❽

set -o allexport
if [ -f "/etc/environment" ]; then sourcex "/etc/environment"; fi         ❾
if [ -f "/etc/rancher/k3s/k3s.env" ]; then sourcex "/etc/rancher/k3s/k3s.env"; fi   ❾
set +o allexport
```

Listing 3.1: `OpenRC` service configuration file for K3s

3.1.2.2 systemd

systemd[30] is a collection of building blocks for a Linux system. It provides a system and service manager that runs as PID 1 and starts the rest of the system.
systemd supports aggressive parallelization, on-demand daemon startup, process tracking through Linux control groups, mount and automount

Source: `https://brand.systemd.io`

Figure 3.2: systemd logo

[27] `https://wiki.gentoo.org/wiki/OpenRC`
[28] `https://www.gentoo.org`
[29] `https://www.bsd.org`
[30] `https://systemd.io`

point management, and a complex transactional dependency-based service control logic.
Other components include a logging daemon, tools to handle basic system configuration such as the hostname, date, and locale, a list of logged-in users and running containers and virtual machines, system accounts, runtime directories and settings, and daemons to manage simple network configuration, network time synchronization, log forwarding, and name resolution.
Listing 3.2 illustrates the example of the systemd service configuration file for K3s saved in `/etc/systemd/system/k3s.service`. The `systemctl` command, which manages the systemd system and the service manager, may be used to manually control the service: `systemd <CMD> k3s [...]`. `<CMD>` can be replaced with `start` to start the service, `stop` to stop it, `restart` to restart it, and `status` to obtain the service status. Furthermore, the service may be enabled to start automatically when the system boots up by using the commands `systemctl enable k3s` followed by `systemctl daemon-reload` to reload the systemd management configuration. The latter examples are only a small portion of all the commands and capabilities provided by systemd.

```
[Unit]
Description=Lightweight Kubernetes                                    ❶
Documentation=https://k3s.io                                          ❶
Wants=network-online.target                                           ❷
After=network-online.target                                           ❷
StartLimitBurst=0                                                     ❽

[Install]
WantedBy=multi-user.target

[Service]
Type=notify                                                           ❹
ExecStartPre=/usr/bin/env sh -c 'rm -f "/tmp/k3s.*"'                  ❸
ExecStart=/usr/local/bin/k3s server                                 ❺ ❻
EnvironmentFile=-/etc/environment                                     ❾
EnvironmentFile=-/etc/rancher/k3s/k3s.env                             ❾
KillMode=process
Delegate=yes
TimeoutStartSec=0
Restart=always                                                        ❹
RestartSec=5s                                                         ❼
```

Listing 3.2: `systemd` service configuration file for K3s

3.1.3 ISO Image

This section briefly describes the two original Linux distributions, Alpine Linux and Arch Linux, which served as the foundation for the two custom Linux distributions. They were both utilized and tested together in the implementation cluster. Furthermore, it is shown how the two custom ISO image files are customized and generated by the use of a simple script.
The ISO image can be burned into physical media such as a CD, or a USB flash drive or can be mounted as an ISO file. The latter shows the high level of flexibility and overall compatibility that an ISO file has. Moreover, it is extremely portable since it is intrinsically only a single file making it the perfect format for a release bundle and very useful in an Air-Gap environment.
After an ISO image of a Linux distribution has been burned to media, it can be easily installed on multiple systems by attaching the media to the system and selecting it as the default boot. It starts a live (volatile) OS that is not currently installed on the primary disk. The live Linux distribution may then be installed directly into the primary disk, making it permanently available on the system without the need for removable media. The latter step may be performed manually, but it is more complex than it appears since it necessitates an understanding of disk formatting, partitioning, as well as other topics. Fortunately, most Linux distributions, including the two custom ones, have simple tools that automate the installation procedure by asking only a few simple questions to the user/administrator.

3.1.3.1 Alpine Linux

Alpine Linux[31] is a lightweight, security-focused, and resource-efficient distribution. In recent years, has grown in popularity as the foundation for the majority of Docker images. It is based on musl[32] (an implementation of the C standard library, an alternative to glibc[33]), BusyBox[34] (a single, small executable that combines tiny versions of many common UNIX utilities), a custom package manager called APK[35] and the OpenRC Init System[7].

Source: `https://alpinelinux.org`

Figure 3.3: Alpine Linux logo

Alpine Linux is the preferred reCluster distribution since it has been widely used and tested throughout the development process.

The official documentation[36] provides a brief guideline on how to build a custom Alpine Linux ISO image. Unfortunately, it is not exhaustive, and it assumes a lot of information and configurations. A simpler, easy-to-use, and portable solution is required for cluster implementation. This resulted in the creation of a POSIX script that handles all of the burdens, as explained below.
Building a custom Alpine Linux ISO image involves three prerequisites: an Alpine Linux system, specific tools/packages to configure and build the image, and a general configuration with a special administrator user and the generation of signing keys. These requirements remain constant over time and can be regarded as essentially static. Furthermore, having a physical Alpine Linux system dedicated just to generating custom images is a waste of resources. As a result, a custom Dockerfile[37][38] based on Alpine Linux has been created to allow building the image in different environments without the need for a physical machine and with all of the previous requirements already fulfilled.
The ISO image is customized using the `mkimg.recluster.sh`[39] file, which defines and integrates a custom recluster profile, based on the basic Alpine Linux profile. This file, whose content is shown in listing 3.3, is not included in the Dockerfile because it is considered dynamic, in the sense that it could be updated and customized over time to meet potential new cluster specifications.
Finally, to create the ISO image, the container image must be launched in detached mode, a local volume mounted to the container, the `mkimg.recluster.sh` file loaded, and the official `mkimage.sh` script executed with extra configuration parameters. When performed manually, the latter tasks are extremely difficult and prone to human error. As a result, all operations have been included in a POSIX script, simply named `build.sh`[40], that does everything automatically, including cleanup procedures when the image(s) has been generated or an error occurs. Moreover, all ISO images created are saved in the current working directory under the directory `iso`. As an example, a custom image based on Alpine Linux version 3.17 and generated with the recluster profile for architecture `x86_64` is saved as `alpine-recluster-v3.17-x86_64.iso` and is just `190MiB` in size.
To configure and install Alpine Linux permanently on the primary disk, simply run the `setup-alpine`[41] program and answer a few easy questions.

```
profile_recluster() {
  profile_base
  title="reCluster"
  desc="reCluster Alpine Linux"
  profile_abbrev="recluster"
  image_ext="iso"
  arch="x86_64 aarch64"
  output_format="iso"
  apks="$apks coreutils docker ethtool inotify-tools iproute2 jq ncurses nodejs npm postgresql
```

[31] https://www.alpinelinux.org
[32] https://musl.libc.org
[33] https://www.gnu.org/software/libc
[34] https://busybox.net
[35] https://wiki.alpinelinux.org/wiki/Alpine_Package_Keeper
[36] https://wiki.alpinelinux.org/wiki/How_to_make_a_custom_ISO_image_with_mkimage
[37] https://docs.docker.com/engine/reference/builder
[38] https://github.com/carlocorradini/reCluster/blob/main/distributions/alpine/Dockerfile
[39] https://github.com/carlocorradini/reCluster/blob/main/distributions/alpine/mkimg.recluster.sh
[40] https://github.com/carlocorradini/reCluster/blob/main/distributions/alpine/build.sh
[41] https://docs.alpinelinux.org/user-handbook/0.1a/Installing/setup_alpine.html

```
        procps sudo sysbench tzdata util-linux yq"
    apkovl="genapkovl-recluster.sh"
}
```

Listing 3.3: Contents of `mkimg.recluster.sh` file which shows the reCluster profile definition

3.1.3.2 Arch Linux

Source: `https://archlinux.org/art`

Figure 3.4: Arch Linux logo

Arch Linux[42] is a general-purpose distribution that focuses on simplicity, minimalism, and code elegance. It is based on Systemd Init System and strives to stay bleeding edge, offering the latest stable versions of most software through Pacman[43] package manager. Uses a rolling release system that allows one-time installation and perpetual software upgrades, allowing to not reinstall or upgrade the system from one version to the next. Notably, Arch Linux is the foundation for several many popular enterprise-grade distributions, such as Manjaro[44] and SteamOS[45], demonstrating its customizability, power, and stability.

The official documentation[46] includes precise guidelines for building a custom Arch Linux ISO image. For cluster implementation, there is a need for only a basic/minimal image with a simple and easy-to-use procedure. As a result, a POSIX script has been written to handle all of the difficulties and configurations (described below). Some techniques are quite similar to those outlined in Alpine Linux. However, the files and overall configurations differ significantly.
Building a custom Arch Linux ISO image requires three prerequisites: an Arch Linux system, specific tools/packages to configure and build the image, and a general configuration with a baseline profile. As with Alpine Linux, these requirements remain constant over time, resulting in the same possible problems. As a result, a custom Dockerfile[47] based on Arch Linux has been developed.
The ISO image is customized with the `profiledef.sh`[48] file, the content of which is displayed in listing 3.4, that defines and integrates a custom profile based on the baseline Arch Linux profile already included in the container image. For the same reasons as in Alpine Linux, this file is not available in the Dockerfile container image.
Finally, similar to the Alpine Linux approach, the container image must be launched, a local volume mounted, the `profiledef.sh` file loaded, and the official `mkarchiso` program executed with extra configuration parameters to generate the ISO image. Because of its inherent difficulties, the latter exposes the same issues as in Alpine Linux. Therefore, a similar POSIX script, also named `build.sh`[49], has been created with similar features but adapted to Arch Linux. As an example, a custom Arch Linux image based on the reCluster profile and built on March 23rd 2023 for architecture `x86_64` is saved as `reCluster-2023.03.23-x86_64.iso` and is `731MiB` in size.
To configure and install Arch Linux permanently on the primary disk, simply run the `archinstall`[50] program and answer a few easy questions.

```
iso_name="reCluster"
iso_label="ARCH_$(date +%Y%m)"
iso_publisher="reCluster"
iso_application="reCluster Arch Linux"
iso_version="$(date +%Y.%m.%d)"
install_dir="arch"
buildmodes=("iso")
bootmodes=("bios.syslinux.mbr" "bios.syslinux.eltorito"
           "uefi-ia32.grub.esp" "uefi-x64.grub.esp"
           "uefi-ia32.grub.eltorito" "uefi-x64.grub.eltorito")
```

[42] https://archlinux.org
[43] https://wiki.archlinux.org/title/Pacman
[44] https://manjaro.org
[45] https://store.steampowered.com/steamos
[46] https://wiki.archlinux.org/title/archiso
[47] https://github.com/carlocorradini/reCluster/blob/main/distributions/arch/Dockerfile
[48] https://github.com/carlocorradini/reCluster/blob/main/distributions/arch/profiledef.sh
[49] https://github.com/carlocorradini/reCluster/blob/main/distributions/arch/build.sh
[50] https://wiki.archlinux.org/title/archinstall

```
arch="x86_64"
pacman_conf="pacman.conf"
airootfs_image_type="erofs"
airootfs_image_tool_options=("-zlz4hc,12 -E ztailpacking")
file_permissions=(
  ["/etc/shadow"]="0:0:400"
)
```

Listing 3.4: Contents of `profiledef.sh` file which shows the reCluster profile definition

3.2 Dependencies

This section is split into three segments, each of which is dependent on the preceding one's knowledge. The first segment defines an Air-Gap environment and why it is critical for overall implementation, and more broadly for specific cluster use cases, to allow scenarios where internet access is highly limited or even non-existent.
The second segment illustrates and explains the most important third-party applications on which the overall architectural implementation is based. Each one is tailored to a distinct function in the cluster and has a direct logical mapping to a component in the architecture depicted in chapter 2. The majority, but not all, of the specified applications are interchangeable with other implementations/distributions. As a result, any application-specific configuration and/or capability must be adjusted and/or reconfigured to be compatible with the new one.
The Server and Autoscaler components are missing because, due to their importance and implementation, they have their dedicated sections, section 3.3 and section 3.4, where they are explained in greater detail. Nevertheless, the name of the related component's implementation appears in various portions of this section. It should be noted that the Server implementation is cloud/cluster provider-specific, and therefore it is completely built from scratch, whereas Autoscaler has an official Kubernetes implementation that is only compatible with a limited number of particular and large cloud providers. As a result, a custom solution has been created that is derived from the original and has been appropriately changed to support the cluster implementation, Server API, and overall use case.
Finally, the last segment discusses how to manage all dependencies and how they are downloaded for a cluster release and to be functional in an Air-Gap scenario, among other things. The latter relies on an intuitive configuration file and a user-friendly POSIX script.

3.2.1 Air-Gap Environment

All devices in an Air-Gap environment have no physical connectivity to the public internet or any other Local Area Network (LAN) that is not itself in an Air-Gap environment. Email clients, browsers, SSH, and other communication applications are all physically and logically isolated from the outside world.
Systems in an Air-Gap environment are permanently separated from the outside world by default. However, if possible, the network can communicate with other physically isolated devices. Any data transfer outside of the network must take place via external hardware that is temporarily connected to the network. USB flash drives, Hotspot devices and other removable media are examples of such hardware. Importantly, these external devices must be physically connected to and disconnected from the Air-Gap environment network by human intervention.
Figure 3.5 depicts an example overview of an Air-Gap environment consisting of two distinct Air-Gap networks that can connect with one another. Because they are on separate networks, each network resembles a hypothetical cluster implementation where communication between the two does not directly involve cluster operations and administration (see section 2.2.2). Every machine in the environment is unable to exchange data, connect to any external network, or connect to the internet in general. The only mechanism to move data between the Air-Gap environment and the outside world is through the use of a USB flash drive, and the only way to connect the Air-Gap networks to an external network and/or the internet is through the use of a personal hotspot. Because no automated mechanisms are involved, both of the latter actions need human interaction.
Air-Gap environments are frequently employed in Network Security scenarios where security is a top priority. In an Air-Gap scenario, a correctly designed network mandates that devices within the net-

work be invisible to and successfully isolated from remote threat actors, who often scan the public internet for vulnerable devices. Similarly, an attacker outside of the Air-Gap environment network cannot directly execute Remote Code Execution (RCE) attacks on potential software vulnerabilities within the environment[27].
Although Air-Gap environments are extensively employed in the security branch, high-security scenarios are not the only ones for which the cluster implementation is designed. In reality, there are situations in which the internet connection, and thus the connectivity to the external global network is either too expensive to maintain or not available at all in the location where the cluster is deployed. To be fully compatible with an Air-Gap environment in various possible scenarios, the cluster implementation is designed to have all external dependencies, including custom Linux Distributions, pre-packaged with each cluster release and without the need, by default, for possible external requests during its utilization. As a result, all that is required is the unique archive file, which is automatically created with each new release and contains all of the files required to bootstrap a functioning cluster instance. Furthermore, even if not in an Air-Gap environment, during the installation procedure, the organization in charge of the cluster's administration can choose to either download all external dependencies from the internet (every time for every node) or to directly use the already available pre-packaged dependencies, reducing network usage as well as overall installation time.

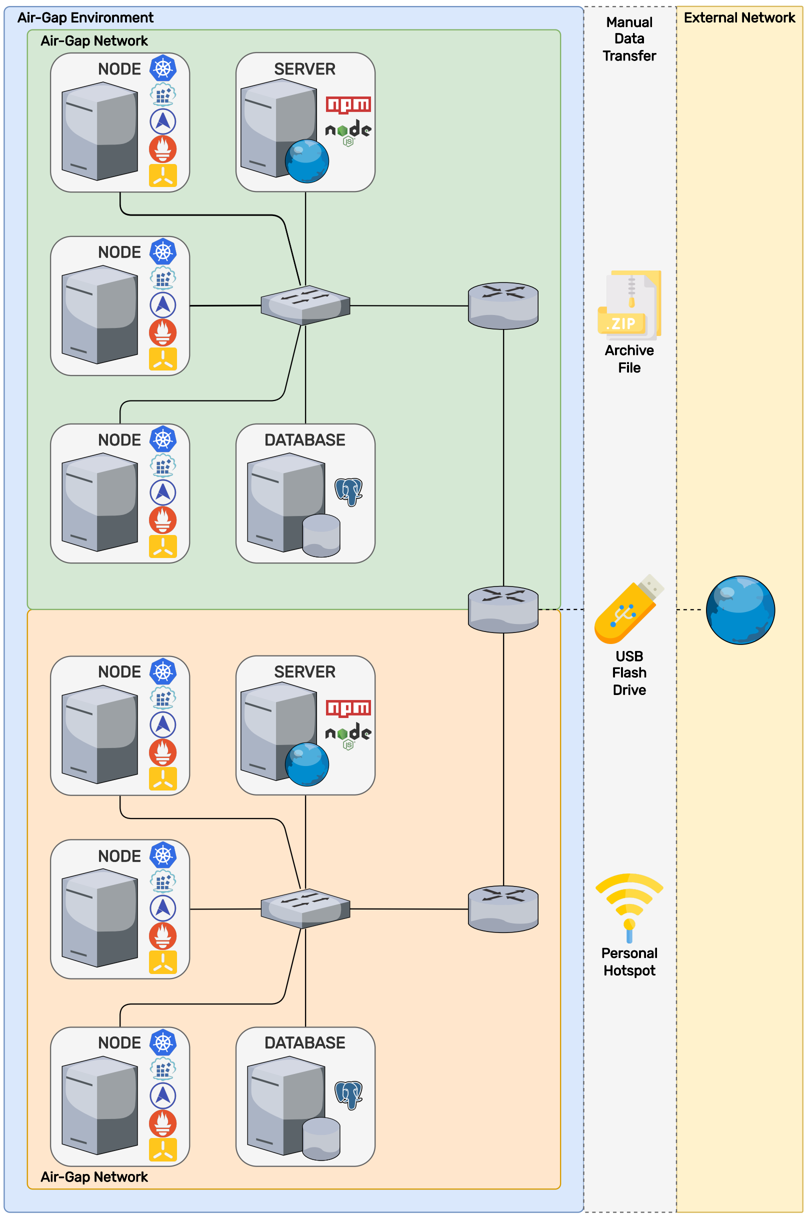


Figure 3.5: An Air-Gap Environment composed of two Air-Gap Networks that are physically and logically isolated by any external data exchange and network

3.2.2 PostgreSQL

Source: `https://wiki.postgresql.org/wiki/Logo`

Figure 3.6: PostgreSQL logo

Premise: This section varies from the other following sections in that it does not intend to go into detail about the PostgreSQL dependency and what and how its core features have been employed in the cluster implementation. PostgreSQL has only been used in the most basic ways for storing the server's data in a persistent and reliable system. Furthermore, there are numerous database systems to choose from, both free and paid, each with unique characteristics that might meet the different requirements of the organization managing the cluster.

PostgreSQL[51] is a powerful, open-source object-relational database system that uses and extends the `SQL` language with several capabilities for properly storing and scaling the most complex data workloads. It is designed to assist developers in building applications, and administrators in preserving data integrity and building fault-tolerant systems.

PostgreSQL has a direct mapping with the cluster architecture's Database component.

PostgreSQL was chosen over all other possible alternatives for three key reasons, which are detailed below:

1. Is one of four databases officially supported by the K3s dependency (see section 3.2.3) that may be utilized as an external data source for persistently storing the Kubernetes cluster's state (see section 3.2.3.1).
2. The Prisma ORM (see section 3.3.1.1) used in the server component implementation officially supports it (see section 3.3).
3. The server implementation mostly employs simple SQL code, with no complex or proprietary functionalities. As a result, it may be easily substituted with MySQL[52] (another database that fulfills the preceding two criteria) without any difficult migration procedures. The latter is particularly valuable since it allows organizations to select from a variety of database systems while being compatible with cluster implementation.

3.2.3 K3s

Source: `https://k3s.io`

Figure 3.7: K3s logo

K3s[53] is a lightweight Kubernetes distribution designed for production workloads in resource-constrained, high-availability, unattended, and remote environments. K3s is the cluster implementation's core, where all Kubernetes-related operations are managed and processed. Because K3s is an element of both a Worker and a Controller Node has no clear mapping to an architectural component.

K3s is distributed as a single small binary (less than 60MiB) that reduces the requirements and procedures required to install, run, and automatically update a production Kubernetes cluster. It is completely compatible with both the `OpenRC` and `systemd` Init Systems and can run on any Linux distribution with a Kernel that satisfies the basic Kubernetes requirements.

3.2.3.1 Enhancements

K3s is a fully compliant Kubernetes distribution that has removed the majority of legacy, alpha, and cloud-provider-specific code to minimize overall application size and hardware requirements. Furthermore, K3s provides the following enhancements over a conventional Kubernetes distribution[54]:

- All required dependencies are pre-packaged within a single binary file, necessitating only a modern Kernel and `cgroup` mounts.
- Lightweight storage backend with `SQlite`[55] as the default storage mechanism. Additional solutions based on `etcd`[56], `MySQL`, or `PostgreSQL` are also available. Both embedded and external

[51] `https://www.postgresql.org`
[52] `https://www.mysql.com`
[53] `https://k3s.io`
[54] `https://docs.k3s.io`
[55] `https://www.sqlite.org`
[56] `https://etcd.io`

storage mechanisms are supported.
The cluster architecture design, as stated in chapter 2, is based on a high availability model rather than a fault tolerance strategy. As a result, it is recommended to use `etcd` (embedded or external) or the already available cluster's database as an external storage mechanism, eliminating needless data replications and relying on a single storage area that can be controlled autonomously.

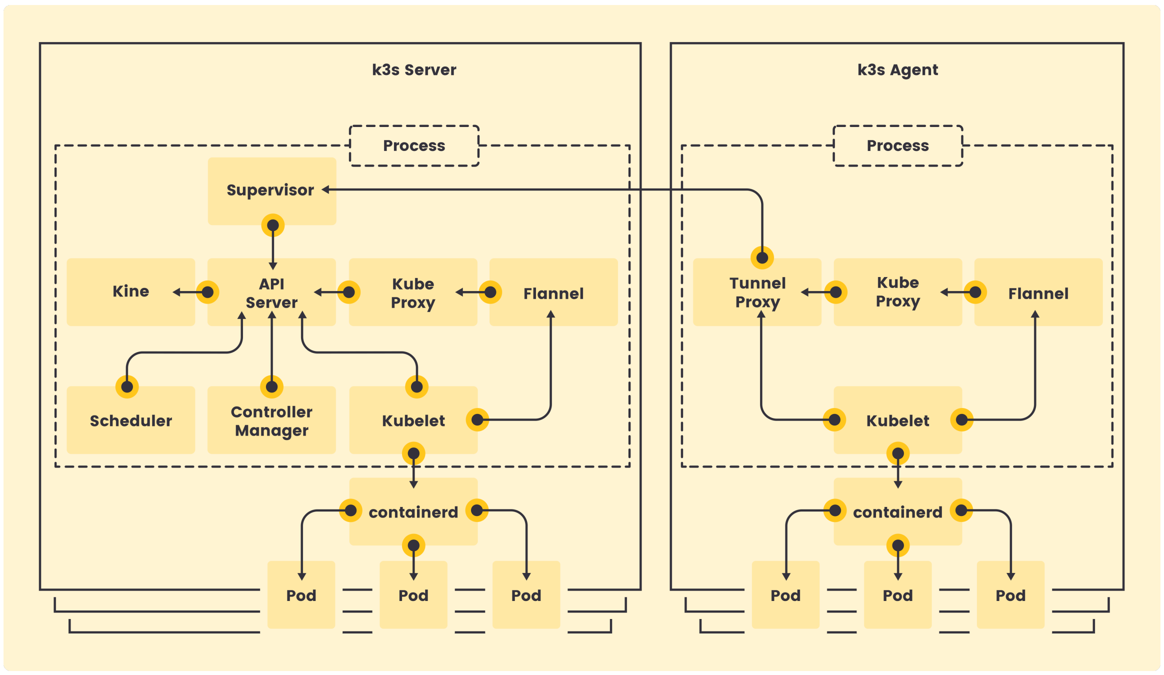
- Secure by default, with appropriate default parameters for lightweight environments. K3s manages automatically the complexity of TLS (Transport Layer Security) like certificate distributions.
If security is not a priority, or if the cluster is in an Air-Gap environment, the security settings can be relaxed to increase the overall cluster's performance while simultaneously decreasing resource utilization.

3.2.3.2 Architecture

K3s architecture[57] clearly distinguishes between two types of Nodes, Server nodes and Agent nodes, which are described below. Furthermore, figure 3.8 depicts the distinction between the two node types by displaying the various components provided on each node.

1. `Server` node
A server node is defined as a host running the K3s server command (`k3s server`), with K3s managing the `control-plane` and `datastore` components.
The `Controller` Node component of the cluster's architecture, depicted in the section 2.1.1.2, is a logical mapping to the K3s Server Node.

2. `Agent` node
An agent node is defined as a host running the K3s agent command (`k3s agent`), that does not have any `datastore` or `control-plane` components running.
The `Worker` Node component of the cluster's architecture, depicted in the section 2.1.1.1, is a logical mapping to the K3s Agent Node.



Source: `https://k3s.io`

Figure 3.8: Difference between K3s Server and K3s Agent nodes

Additionally, the K3s architecture is completely adaptable and may accommodate a variety of different setups, as shown in the list below, depending on the cluster's usage and ultimate goal.

[57]`https://docs.k3s.io/architecture`

1. `Single Server with an Embedded Database`
   A single-node K3s server with a `SQLite` database embedded. Each agent node is associated with the same K3s Server node. By utilizing the K3s API on the Server node, an administrator can directly control the Kubernetes resources.
   This configuration is only recommended for testing purposes, not for real-world production deployments or cluster implementation.

2. `High-Availability Server with an Embedded Database`
   A High-Availability (HA) K3s cluster is composed of two or more Server nodes that serve the Kubernetes API as well as additional control plane services. The database is embedded and operated on the same Server node by the K3s instance.

3. `High-Availability Server with an External Database`
   Similar to the previous setup, however, the datastore is located outside of the Server Node. The K3s Server instance does not execute any database application and instead relies on an externally accessible and operating database.

3.2.3.3 The Choice

The K3s distribution was chosen from among a multitude of various Kubernetes distributions, including the official one, for the four main reasons, listed below.

1. Minimal hardware requirements that are simple to satisfy, even on older systems[58]:

   - A Server node requires 1 GiB of memory and 1 CPU core.
     As stated in the section 2.1.1.2, the Server (Controller) node's minimal hardware requirements must be adjusted to match the cluster's size. A cluster with hundreds of Worker nodes managed by a single Server node with only 1 GiB of memory and 1 CPU core is unrealistic.
   - An Agent node requires 512 MiB of memory and 1 CPU core.
     A Raspberry Pi 3 Model B[59] single-board computer, released in February 2016 (7 years ago at the time of writing), is fully compatible with K3s and can operate as a Worker node without any difficulty.

   In comparison, the official Kubernetes distribution, `kubeadm`[60], necessitates at least 2 GiB of memory and 2 CPU cores.

2. Configurability, modularity, and usability[61].
   As an example, it is extremely simple to bootstrap a small Kubernetes cluster using K3s in a Single Server with an Embedded Database setup:

   - Start the Server node with a unique token (used to join a Server or Agent nodes to the cluster).
     ```
     k3s server --token "<TOKEN>"
     ```
   - Start the Agent node specifying the unique token and the Server address.
     ```
     k3s agent --token "<TOKEN>" --server "<SERVER_ADDRESS>"
     ```

3. K3s can be easily installed in an Air-Gap environment[62]. The only additional steps necessary to have a fully functional cluster in comparison to a standard installation are the deployment of a private registry (see section 3.2.5) and the manual deployment of particular container images on each node.
   By default, the cluster implementation installation (see section 3.5) already meets the latter

[58] `https://docs.k3s.io/installation/requirements`
[59] `https://www.raspberrypi.com/products/raspberry-pi-3-model-b-plus`
[60] `https://kubernetes.io/docs/setup/production-environment/tools/kubeadm`
[61] `https://docs.k3s.io/installation/configuration`
[62] `https://docs.k3s.io/installation/airgap`

two requirements. As a result, an organization that has to deploy a cluster in an Air-Gap environment already has received all of the required files and programs without the need for any additional data transfer involving an internet connection. As a result, an organization that has to deploy a cluster in an Air-Gap environment already has all of the necessary files and applications without the need for any additional data transfer involving an internet connection.

4. On August 19th, 2020, the K3s project was approved into the Cloud Native Computing Foundation[63] (CNCF) with Sandbox maturity level[64]. CNCF is a vendor-neutral cloud native computing, hosting critical components of the global technology infrastructure dedicated to making cloud native ubiquitous. CNCF certifies Kubernetes software compliance, ensuring that every vendor's version of Kubernetes, as well as open source community versions, supports the large set of Kubernetes APIs. The latter means full compatibility with all existing Kubernetes products and software, allowing for the interchangeability of various components within the cluster implementation representing diverse organizations' core aspects.

3.2.4 Node Exporter

Node Exporter[65] provides hardware and OS-related metrics exposed by the `*NIX` Kernel. It is widely used in both testing and production environments thanks to its broad compatibility, simplicity of configuration, and reliability, and it can be regarded as the reference implementation for a Metrics Server. Because it is an element that constitutes both a Worker and a Controller Node, Node Exporter does not have a clear mapping to an architectural component.

3.2.4.1 Collectors

For each Operating System, such as Linux or OpenBSD[66], Node Exporter supports a wide range of different collectors, such as `cpu` for providing CPU statistics or `meminfo` for exposing memory statistics[67]. A collector[68] is an element of an exporter, which represents a collection of metrics. If it is part of direct instrumentation, it may be a single metric, or it may be multiple metrics if it is pulling metrics from another system. Collectors can be enabled (`node_exporter --collector.<NAME>`) or disabled (`node_exporter --no-collector.<NAME>`) based on the data that the node needs to provide for monitoring. The exposed metrics are represented in a standardized format (see section 3.2.7.1), allowing Prometheus (see section 3.2.7) to analyze and process them. Listing 3.5 depicts an example section of a response generated by Node Exporter running on a node with two CPU cores and approximately 4 GiB of system memory. Three metrics are displayed, one from the `cpu` collector and the other two from the `meminfo` collector: `node_cpu_seconds_total` is a counter that keeps track of how many seconds each CPU core (`0` and `1`) spent in each mode, `node_memory_MemTotal_bytes` is the total amount of physical memory (in bytes), and `node_memory_MemFree_bytes` is the total amount of physical memory (in bytes) that is not in use.

```
# HELP node_cpu_seconds_total Seconds the CPUs spent in each mode.
# TYPE node_cpu_seconds_total counter
node_cpu_seconds_total{cpu="0",mode="idle"} 980.6
node_cpu_seconds_total{cpu="0",mode="iowait"} 2.18
node_cpu_seconds_total{cpu="0",mode="irq"} 2.97
node_cpu_seconds_total{cpu="0",mode="nice"} 0
node_cpu_seconds_total{cpu="0",mode="softirq"} 1.47
node_cpu_seconds_total{cpu="0",mode="steal"} 0
node_cpu_seconds_total{cpu="0",mode="system"} 0.59
node_cpu_seconds_total{cpu="0",mode="user"} 423.57
node_cpu_seconds_total{cpu="1",mode="idle"} 954.29
node_cpu_seconds_total{cpu="1",mode="iowait"} 4.44
node_cpu_seconds_total{cpu="1",mode="irq"} 0.98
node_cpu_seconds_total{cpu="1",mode="nice"} 0
```

[63]https://www.cncf.io
[64]https://www.cncf.io/projects/k3s
[65]https://github.com/prometheus/node_exporter
[66]https://www.openbsd.org
[67]https://github.com/prometheus/node_exporter#collectors
[68]https://prometheus.io/docs/introduction/glossary/#collector

```
node_cpu_seconds_total{cpu="1",mode="softirq"} 1.94
node_cpu_seconds_total{cpu="1",mode="steal"} 0
node_cpu_seconds_total{cpu="1",mode="system"} 1.43
node_cpu_seconds_total{cpu="1",mode="user"} 448.79
# HELP node_memory_MemTotal_bytes Memory information field MemTotal_bytes.
# TYPE node_memory_MemTotal_bytes gauge
node_memory_MemTotal_bytes 4.063023104e+09
# HELP node_memory_MemFree_bytes Memory information field MemFree_bytes.
# TYPE node_memory_MemFree_bytes gauge
node_memory_MemFree_bytes 1.88204544e+09
```

Listing 3.5: Example section of a Node Exporter response with `cpu` and `meminfo` collectors enabled

3.2.4.2 Installer

To represent diverse organizations' goals and preferences, the cluster's installation procedure on a system node, as detailed in the section 3.5, requires complete automation and configuration when installing a dependency. Despite being a very popular and widely used application, Node Exporter lacks a fully featured installation script, such as `install.sh`[69] provided by K3s, which can simply automate every operation. As a solution, a side project called Node Exporter Installer was established to address the latter issue(s). The installation script is completely POSIX-compliant, highly customizable, `OpenRC` and `systemd` Init Systems compatible, and requires only basic applications as dependencies that are ubiquitous in almost all `*NIX` systems. The project is completely Open Source, and other users may use it to easily and rapidly configure and install Node Exporter. Attachment A.1 has a detailed description of the Node Exporter Installer project.

3.2.4.3 Graphics Processing Unit metrics

Node Exporter does not support exporting Graphics Processing Unit (GPU) metrics. In general, supporting systems equipped with GPU(s) in a uniform and easily accessible manner is a fairly hard undertaking that necessitates a significant amount of work in terms of initial setup and overall administration. Furthermore, GPU-equipped systems require availability and compatibility with a per GPU model-specific Kernel driver to be efficient without scarifying precious performance and wasting resources. Nevertheless, exporter implementations such as `DCGM-Exporter`[70] and `nvidia_gpu_exporter`[71] are specifically designed to provide just GPU metrics that can be used alongside Node Exporter metrics. However, all current GPU exporters only support NVIDIA GPUs and require extra dependencies to be deployed. Moreover, because of its intrinsic heterogeneity, purpose, and complexity of supporting GPU systems, the cluster implementation currently does not offer any GPU data, metrics, or statistics. Furthermore, because the cluster is primarily composed of consumer hardware rather than high-end enterprise solutions, there is a high possibility that some systems would be equipped with AMD GPUs, which, as previously stated, are currently not supported by any official and/or reliable exporter program.

3.2.5 Docker Registry

Source: `https://github.com/distribution/distribution`

Figure 3.9: Docker Registry logo

Docker Registry[72] is a stateless, highly scalable server-side service for storing and distributing container-based application images. A registry is a storage and content delivery system that holds container images in different tagged versions[73].
The Docker Registry, thus the name, has a direct mapping to the Registry component of the cluster architecture.

[69]`https://github.com/k3s-io/k3s/blob/master/install.sh`
[70]`https://github.com/NVIDIA/dcgm-exporter`
[71]`https://github.com/utkuozdemir/nvidia_gpu_exporter`
[72]`https://docs.docker.com/registry`
[73]`https://docs.docker.com/registry/introduction`

3.2.5.1 Image Naming

`[REGISTRY_HOSTNAME][:REGISTRY_PORT]IMAGE[:TAG]`

An image name is made up of slash-separated name fields that are optionally prefixed with a registry hostname, complying with standard DNS[74] (Domain Name System) rules, and optionally postfixed with a tag name that identifies different versions of the same series of images. If a hostname is specified, it may be followed by a port number in the format `:PORT`. If the registry hostname is not provided, Docker, Kubernetes, Podman, and practically every other software that deals with container images by default utilizes Docker's public registry (Docker Hub[75], which is hosted at `docker.io` at port `443`). If no tag name is supplied, the value `latest` is automatically assigned, indicating that the image is designated as the most recent available release version[76].
For example, the image name `node:18` (short for `docker.io:443/node:18`) indicates the container image of Node.js version 18 accessible on the Docker public registry at port `443`. Instead, the name `registry.recluster.local:5000/node:18` also identifies the container image of Node.js version 18 but available on the local private registry `registry.recluster.local` at port `5000`. Therefore, if there is a requirement in the cluster to refer to container images on a registry different from the default one, the full image name must be provided.

3.2.5.2 Hostname To IP Address Mapping

A local registry deployment requires that there be a known mapping in the cluster between the registry hostname and the IP address where the registry instance is executing. There are three options available, as indicated in the list below:

1. Use a custom Kubernetes distribution configuration file/entry to instruct the container runtime about the mapping. Moreover, if the Kubernetes distribution and runtime support it, the mapping may also be used to rewrite the default Docker Hub registry hostname to link to an alternative local registry hostname. An example image `docker.io/node:18` is transparently rewritten as `registry.recluster.local:5000/node:18`. The latter, allows the cluster to continue operating without any changes to the container image name of the deployments, enabling future less possible maintenance burden because the only change that has to be performed is on the configuration file and not on the cluster deployments.
   This functionality is supported by default in K3s[77] through a configuration file called `registries.yaml` that must be saved in the directory `/etc/rancher/k3s` on each node in the Kubernetes cluster.

2. Add an entry to the `/etc/hosts` file on each cluster node.

3. Deploy a DNS component that must be operational in the cluster at all times.
   This solution requires that all cluster nodes be configured to know about the local DNS, either manually by modifying `/etc/resolv.conf` file or automatically via DHCP (Dynamic Host Configuration Protocol).

Each of the three options has pros and cons. If the cluster size is relatively small and the IP address of the local registry does not vary over time, the first or second (depending on whether the chosen Kubernetes distribution supports the functionality) solutions are recommended. However, if the cluster consists of numerous nodes, or if the IP address of the registry changes regularly, or if further hostname to IP mappings are required, then having a DNS component is by far the best approach. Nonetheless, the organization operating the cluster makes the final decision on what is the best solution: nothing prevents the deployment of a DNS component even in a very small cluster.
Because reCluster is based on K3s, has a modest number of nodes, and the IP address of the registry never changes thanks to MetalLB (see section 3.2.6), the decision fell almost naturally on the first alternative. Section 3.5.5 goes into further detail on this.

[74] https://wikipedia.org/wiki/Domain_Name_System
[75] https://hub.docker.com
[76] https://docs.docker.com/engine/reference/commandline/tag
[77] https://docs.k3s.io/installation/private-registry

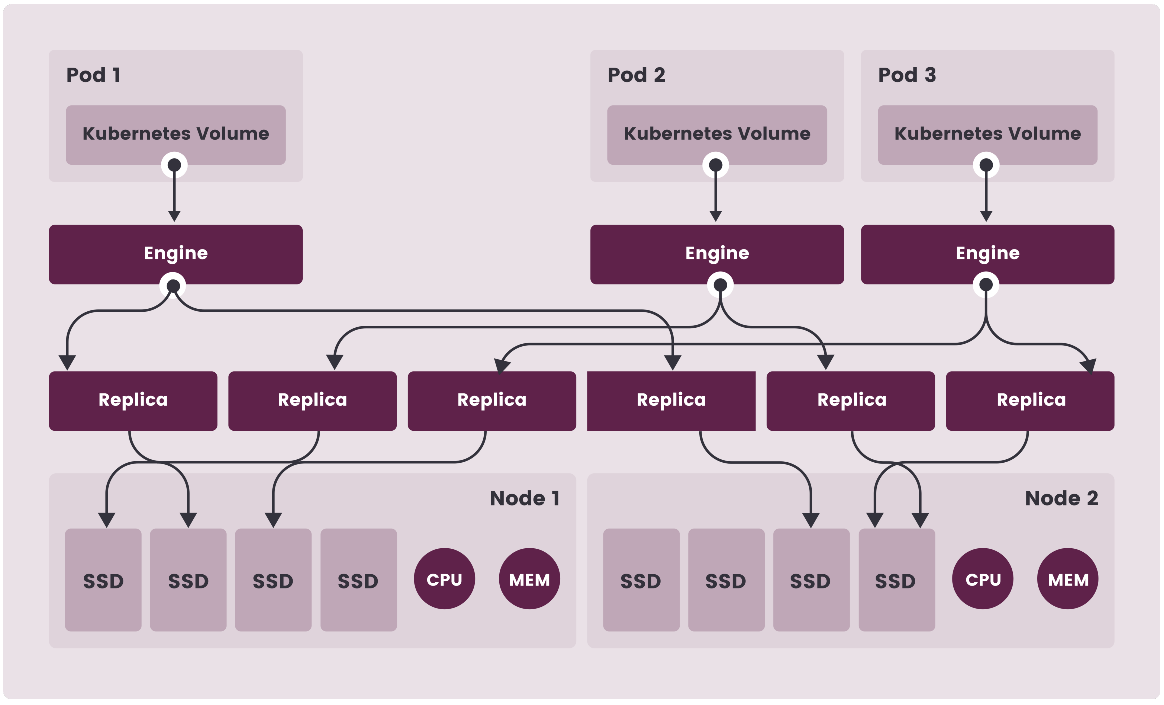
3.2.5.3 Image Storage

The registry must store the available container images in persistent storage to keep them if a failure occurs or the registry instance is restarted. Storage operation and management are not handled by the registry itself but are delegated to specialized drivers, the default driver of which is the local POSIX file system of the node executing the registry instance[78].
Saving all container images on a single node is considered bad practice since it creates a single point of failure that might result in the loss of all images. Therefore, the cluster should be equipped with extra and more powerful storage drivers intended to tolerate multiple storage device failures at the same time as well as automated data recovery. It should be noted, however, that the driver(s) chosen must be compatible with the cluster environment, nodes hardware, and/or meet certain extra requirements. Almost every cloud provider supplies its customized driver that is compatible with the underlying high-performance and distributed data center storage system. Implementing a custom storage driver for the cluster implementation that meets all of the above requirements is extremely difficult owing to the overall complexity. Thankfully, a storage driver, Longhorn, from the same creators of K3s, already exists to address the latter with extra features and customization. Thankfully, a storage driver that addresses the latter problem with extra features and customization already exist: Longhorn, developed by the same team that created K3s.

Longhorn[79] is an Open Source and CNCF Incubating project[80] that provides a lightweight, reliable, and simple distributed block storage system for Kubernetes. Longhorn provides a dedicated storage controller for each volume and replicates it synchronously across multiple replicas stored on multiple nodes. Kubernetes is used to orchestrate and manage the storage controller and replicas. Figure 3.11 depicts Longhorn's architecture overview and Read/Write Data Flow between the Volume, Longhorn Engine, Replica Instances, and Disks.

Source: `https://longhorn.io`

Figure 3.10: Longhorn logo



Source: `https://longhorn.io`

Figure 3.11: Longhorn's architecture overview and Read/Write Data Flow between the Volume, Longhorn Engine, Replica Instances, and Disks

It should be mentioned that having an underlying reliable persistent storage for the whole cluster benefits not just the registry component but also every other Kubernetes application deployment that

[78]`https://docs.docker.com/registry/introduction`
[79]`https://longhorn.io`
[80]`https://www.cncf.io/projects/longhorn`

requires any type of persistent storage, such as database and media services.

3.2.6 MetalLB

Source: `https://metallb.universe.tf`

Figure 3.12: MetalLB logo

MetalLB[81] is a load-balancer implementation for bare-metal Kubernetes clusters that leverages standard networking and routing protocols.
MetalLB has a direct mapping to the cluster architecture's Load Balancer component, and because it is operated directly on and managed by Kubernetes, it is also regarded as an internal Load Balancer that does not require any external system to work.

3.2.6.1 LoadBalancer Service Type

By default, Kubernetes does not provide a fully functional implementation of a network load balancer (Service of type `LoadBalancer`[82]) for bare-metal clusters[83]. The native Kubernetes implementation of network load balancers is just a collection of predefined interfaces that are explicitly designed/mapped to be compatible with the external and sophisticated load balancers running within the different cloud providers' large and energy-hungry data centers. These interfaces are not a real network load balancer implementations, but rather proxies that translate different API/function calls from the Kubernetes ecosystem to the individual cloud provider's load balancer and vice versa. As a result, all Kubernetes Services of type `LoadBalancer` on bare-metal clusters remain in a `pending` state indefinitely, making any deployments that rely on the latter services unavailable from outside the Kubernetes network.
Without the capability of using Kubernetes services of LoadBalancer type, the only alternatives for enabling external traffic to establish connections with internal deployments are primarily two; they are briefly described below.

- `NodePort`[84]
  The Kubernetes Control Plane[85] allocates a port from a predetermined range, by default from port `30000` to port `32767`, and every node in the cluster proxies the same port into a Service instance.
  Every node in the cluster is configured to listen on the same allocated port and forward traffic to one of the Service's ready endpoints.
  Any external traffic that connects to any node in the cluster using the proper protocol (i.e., `TCP` or `UDP`) and port (i.e., `31234` or `32000`) is forwarded to a matching internal service instance.

- `ExternalIPs`[86]
  External IP addresses from outside the Kubernetes network are routed to one or more cluster nodes. Network traffic that enters the cluster using the external IP address (as the destination IP) on the corresponding Service port is then forwarded to one of the Service endpoints.
  External IP addresses are not managed by Kubernetes and are therefore the responsibility of the cluster administrator.

Both of these service types have considerable disadvantages, making bare-metal clusters unsuitable for production environments.
MetalLB provides a network load balancer implementation (service of type `LoadBalancer`) that directly resides within the Kubernetes environment/network and easily integrates with standard network devices and protocols, allowing Kubernetes deployments/services to be accessible from external network traffic.

3.2.6.2 Address Allocation

MetalLB is responsible for allocating and assigning IP addresses to services based on predefined IP address pools that have been manually created by the organization in charge of cluster administration. If the IP addresses in the pool range are already assigned and/or reachable within the (Internal)

[81] `https://metallb.universe.tf`
[82] `https://kubernetes.io/docs/concepts/services-networking/service/#loadbalancer`
[83] `https://kubernetes.io/docs/tasks/access-application-cluster/create-external-load-balancer`
[84] `https://kubernetes.io/docs/concepts/services-networking/service/#type-nodeport`
[85] `https://kubernetes.io/docs/concepts/overview/components/#control-plane-components`
[86] `https://kubernetes.io/docs/concepts/services-networking/service/#external-ips`

cluster network, an IP address conflict occurs[87].
After MetalLB is deployed and configured, it will automatically assign and unassign individual addresses within the pools' range to Kubernetes services of type `LoadBalancer`. Furthermore, a service deployment may be arbitrarily set to be allocated with a certain IP address, ensuring that it is always externally accessible with the same IP address. The latter is only possible if the IP address is available within the range of a MetalLB pool and is not already assigned to another service.
It should be noted that if all available IP addresses have already been allocated, subsequent `LoadBalancer` services remain in the `pending` state until an IP address from a pool becomes available for assignment or a newer pool is created.

3.2.6.3 External Announcement

Once MetalLB has allocated a pool's IP address to a `LoadBalancer` service type, the network outside Kubernetes, i.e. the Internal Network, must be aware that the IP address has been assigned and is accessible ("*lives*" in the network) for any kind of communication to take place. To accomplish this, MetalLB deploys a `DaemonSet`[88] component called `Speaker` on each cluster node, which is in charge of IP address announcements on the network[89].
MetalLB supports two external IP announcement modes[90], which are explained below and are based on common networking or routing protocols.

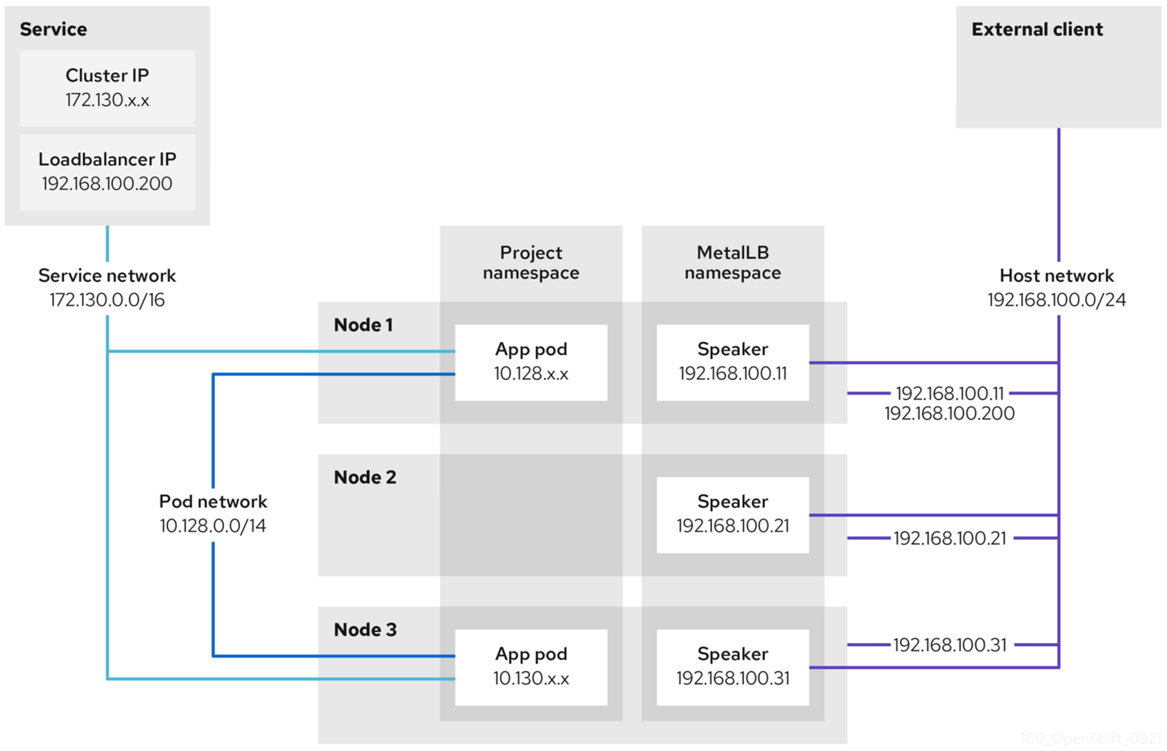
- `BGP mode`[91]
  In `BGP mode`, all nodes in the cluster establish BGP[92] (Border Gateway Protocol) peering sessions with adjacent routers and advise them on how to forward traffic to the service IPs.
  Due to the requirement of BGP-capable routers and a complex configuration, this mode cannot be employed in cluster implementation. Furthermore, BGP is utilized to link with various and external Autonomous Systems[93] (AS).

- `Layer 2 mode`[94]
  In `Layer 2 mode`, one Kubernetes cluster node takes ownership of the `LoadBalancer` service and utilizes standard address discovery protocols (ARP for IPv4, NDP for IPv6) to make the IP address visible on the local network. From the perspective of the network LAN, the announcing node's NIC has multiple IP addresses assigned to it.
  The Kubernetes `kube-proxy`[95] component running on the corresponding node manages all network traffic for a service IP, which subsequently forwards the request to one of the service's pods.
  `Layer 2 mode` does not employ a true load balancer method, but rather a failover strategy in which a new node replaces the current leader node if it fails for any reason.

Due to the inability of using `BGP mode` and its inherent universality to operate on any Ethernet network without the need for additional hardware, configurations, or routers, the cluster implementation is completely based on `Layer 2 mode`. Furthermore, because `Layer 2 mode`, as the name implies, operates directly on Layer 2 of the ISO/OSI model, it is fully compatible with the cluster's Internal Network Architecture (see section 2.2.2). Both MetalLB and the Internal Network's Layer 2 capabilities meet the criteria for a fully functional Wake-on-LAN (WoL) feature (see section 3.3.1.7).

Figure 3.13 depicts an example schema of a MetalLB cluster with distinct Kubernetes namespaces, services, and pods, as well as network connections.

[87] `https://www.linksys.com/support-article?articleNum=132159`
[88] `https://kubernetes.io/docs/concepts/workloads/controllers/daemonset`
[89] `https://metallb.universe.tf/installation`
[90] `https://metallb.universe.tf/concepts/#external-announcement`
[91] `https://metallb.universe.tf/concepts/bgp`
[92] `https://wikipedia.org/wiki/Border_Gateway_Protocol`
[93] `https://wikipedia.org/wiki/Autonomous_system_(Internet)`
[94] `https://metallb.universe.tf/concepts/layer2`
[95] `https://kubernetes.io/docs/concepts/overview/components/#kube-proxy`



Source: `https://docs.openshift.com/container-platform/latest/networking/metallb/about-metallb.html`

Figure 3.13: MetalLB cluster with distinct Kubernetes namespaces, services, and pods, as well as network connections

3.2.7 Prometheus

Source: `https://prometheus.io`

Figure 3.14: Prometheus logo

Prometheus[96] is an Open Source system that is specifically designed for monitoring and alerting. It is the only system that Kubernetes natively supports and the de facto standard across the Cloud Native ecosystem. Prometheus joined CNCF in 2016[97] as the second hosted project with a Graduated maturity level, only after Kubernetes itself. Prometheus has no mapping to a cluster component, and the architecture overview indicates no monitoring component at all. This is because having Prometheus deployed and continually operating in the cluster is a choice of the organization operating the cluster rather than a hard necessity; even if the Metrics Server is required on each Node. Furthermore, Prometheus may be deployed outside of the cluster environment and configured to access node metrics from the external network, or it can be run exclusively for certain periods, allowing monitoring of the cluster for testing or performance evaluation. Nonetheless, Prometheus remains an important tool for monitoring and analyzing how the cluster is behaving, which is why it is specified as a dependency and is also included in the final archive bundle.

As previously stated, Prometheus' primary function is to collect metrics data generated by exporters and query/interpolate this data. Moreover, Prometheus can monitor itself since, like Node Exporter, it provides its metrics via an HTTP endpoint. There is a plethora of available exporters and possible integrations[98]. In addition to the default exporter installed on each node in the cluster implementation, an organization managing the cluster can choose to add as many exporters as the number of components and applications running on it, resulting in a better, clearer, and finer-grained overview of the entire cluster status both in real-time and over a period. For example, in addition to the Database component, it is possible to install its corresponding exporter, which allows the monitoring of Database metrics such as health, performance and resource usage.

[96] `https://prometheus.io`
[97] `https://www.cncf.io/projects/prometheus`
[98] `https://prometheus.io/docs/instrumenting/exporters`

3.2.7.1 Features

Prometheus has several features[99], but the two most essential for the cluster implementation are described below:

1. A multidimensional data model that contains time series data that is uniquely identified by metric name and optional key-value pairs known as labels. Time series data are streams of timestamped values from the same metric and set of labeled dimensions[100].

   (a) `Metrics`
   The metric name, which is sometimes complemented by a short description (`# HELP ...`), defines the overall system characteristic being monitored.
   For example, `node_memory_MemFree_bytes` metric defines the total amount of physical memory (in bytes) that is not in use.
   Each metric has a distinct type[101] (`# TYPE ...`). There is a total of four different types of metrics, which are briefly outlined below:

      i. `Counter`[102]
      A cumulative metric that depicts a single monotonically increasing counter, the value of which can only increase or be reset to zero.

      ii. `Gauge`[103]
      A metric that represents a single numerical value that can vary arbitrarily up and down.

      iii. `Histogram`[104]
      Samples and counts observations in variable buckets, providing also the total sum of all observed values.

      iv. `Summary`[105]
      Similar to `Histogram` type, it also calculates configurable quantiles over a time window.

   (b) `Labels`
   Labels allow Prometheus' dimensional data model: each given combination of labels for the same metric name identifies a unique dimensional instantiation of that metric. These dimensions can be used to filter and aggregate data by the `PromQL` query language.
   For example, `node_cpu_seconds_total{cpu="0",mode="idle"}`, determines the number of seconds spent in idle mode by CPU core 0.

2. Prometheus includes a functional and extensible query language known as `PromQL`[106] (Prometheus Query Language) that allows the selection and aggregation of time series data in real-time. The outcome of an expression can be shown as a graph, tabulated data, or consumed by external systems (such as Grafana, see section 3.2.7.2) via the HTTP API.

Two examples of `PromQL` queries that have been widely used for cluster monitoring are shown below. Note how the various metrics and labels are used for filtering and aggregation.

Listing 3.6 illustrates an example of a `PromQL` query to obtain the overall CPU usage in percent (`0` to `100`). Because the metric `node_cpu_seconds_total` is of type `counter`, the considered value is limited to the last one minute (`1m`).

```
100 * (avg without (mode, cpu) (1 - rate(node_cpu_seconds_total{mode="idle"}[1m])))
```

Listing 3.6: `PromQL` query to obtain overall CPU usage in percent

[99] https://prometheus.io/docs/introduction/overview/#features
[100] https://prometheus.io/docs/introduction/overview
[101] https://prometheus.io/docs/concepts/metric_types
[102] https://prometheus.io/docs/concepts/metric_types/#counter
[103] https://prometheus.io/docs/concepts/metric_types/#gauge
[104] https://prometheus.io/docs/concepts/metric_types/#histogram
[105] https://prometheus.io/docs/concepts/metric_types/#summary
[106] https://prometheus.io/docs/prometheus/latest/querying/basics

Listing 3.7 illustrates an example of a `PromQL` query to obtain the amount of memory usage in percent (`0` to `100`).

```
100 * ((node_memory_MemTotal_bytes - node_memory_MemFree_bytes) / node_memory_MemTotal_bytes)
```

Listing 3.7: `PromQL` query to obtain the amount of memory usage in percent

3.2.7.2 Grafana

Source: `https://grafana.com`

Figure 3.15: Grafana logo

Prometheus is frequently used in conjunction with Grafana[107], an interactive data visualization platform.
Prometheus collects metrics and provides the sophisticated `PromQL` query language; Grafana then translates these metrics and/or query results into relevant charts and graphs that may be consolidated into one or more dashboards.
It should be noted that Grafana is not considered a dependency in the cluster implementation and therefore is not distributed in the release archive bundle.

3.2.8 Management

To create a cluster release bundle, all of the dependencies must be managed and downloaded (see section B.3). Furthermore, as stated in previous sections, having all of the required dependencies locally during cluster initialization without the need for an internet connection is essential for an Air-Gap environment but can also be beneficial for speeding up the overall installation time because there is no need to download them from the Internet. It should be noted that if the cluster is made up of many systems with various architectures, such as `amd64` or `arm64`, the necessary dependencies must be downloaded for all architectures, otherwise, the installation procedure will fail with an error.
The management of dependencies should be as straightforward as possible, with as few manual tasks as possible. As a result, the entire procedure has been entirely automated using a simple configuration file called `dependencies.config.yaml` and detailed in the section 3.2.8.1, as well as a POSIX script called `dependencies.sh` and explained in section 3.2.8.2.
It should be mentioned that dependencies management has been designed to be as compatible as possible, with no restrictions on what dependency and version must be downloaded. The procedure is compatible with any publicly accessible software on `GitHub`[108] that has at least one release[109]. Because compatibility with just GitHub is due to certain API calls, future versions will also include support for custom solutions as well as both `GitLab`[110] and `Bitbucket`[111].

3.2.8.1 Configuration

The dependencies management configuration file, `dependencies.config.yaml`, is written in YAML[112] format.
The attributes for each dependency declared as a root object, are defined in table 3.2.

Name	Type	Description
`url`	`string`[113]	The `URL` of the dependency project's `GitHub` repository.
`assets`	`string sequence`	A list of assets (files) to be downloaded. Regular Expression[114] may be used in the asset string name. This is especially useful for matching files that may not always have the same name across releases. `Node Exporter` and `Prometheus`, for example, always include the version name in each release asset.

[107]`https://grafana.com/grafana`
[108]`https://github.com`
[109]`docs.github.com/repositories/releasing-projects-on-github/managing-releases-in-a-repository`
[110]`gitlab.com`
[111]`https://bitbucket.org`
[112]`https://yaml.org`

`releases`	`string sequence`[115]	A list of releases (versions) to download. All of the files specified in the `assets` attribute are downloaded for each release. As a result, the specified release values must be precisely selected, otherwise, the number and total size of the downloaded files might quickly become unmanageable. For example, if three `releases` and five `assets` are provided, the total number of files downloaded is 15: 5 for each release. All `assets` are saved in the same directory as the corresponding release name. A special release value named `latest` is supported, which denotes the most recent available release version. A particular `GitHub` API[116] request is used to obtain the corresponding version name of a `latest` release.
`files`	`string mapping`[117]	A list of extra files to be downloaded. Each file has a `name` (key) and a string `URL` (value). A file is downloaded from the specified `URL` and saved with the specified `name`. Because these files lack a matching release value, they are saved in the directory with the same name as the dependency root name (the same where release directories are saved)

Table 3.2: Dependency attributes

Listing 3.8 displays the content of the dependencies configuration file, `dependencies.config.yaml`[118], which is used in the cluster implementation. It is worth noting the use of the `latest` value in the `releases` attribute, as well as the use of Regular Expression in the `prometheus` and `node_exporter` `assets` attribute. Furthermore, the `files` attribute specifies the corresponding `LICENSE` file for each dependency.

```yaml
---
autoscaler:
  url: 'https://github.com/carlocorradini/autoscaler'
  assets:
    - 'cluster-autoscaler.amd64.tar.gz'
    - 'cluster-autoscaler.arm64.tar.gz'
  releases:
    - 'latest'
  files:
    LICENSE: 'https://raw.githubusercontent.com/carlocorradini/autoscaler/master/LICENSE'

---
k3s:
  url: 'https://github.com/k3s-io/k3s'
  assets:
    - 'k3s'
    - 'k3s-arm64'
    - 'k3s-airgap-images-amd64.tar.gz'
    - 'k3s-airgap-images-arm64.tar.gz'
  releases:
    - 'latest'
    - 'v1.26.1+k3s1'
    - 'v1.25.6+k3s1'
  files:
    LICENSE: 'https://raw.githubusercontent.com/k3s-io/k3s/master/LICENSE'
```

[113] https://yaml.org/spec/1.2.2/#10113-generic-string
[114] https://wikipedia.org/wiki/Regular_expression
[115] https://yaml.org/spec/1.2.2/#10112-generic-sequence
[116] https://docs.github.com/rest
[117] https://yaml.org/spec/1.2.2/#10111-generic-mapping
[118] https://github.com/carlocorradini/reCluster/blob/main/dependencies/dependencies.config.yaml

```yaml
    install.sh: 'https://raw.githubusercontent.com/k3s-io/k3s/master/install.sh'

---
node_exporter:
  url: 'https://github.com/prometheus/node_exporter'
  assets:
    - 'node_exporter-[0-9]+\.[0-9]+\.[0-9]+\.linux-amd64.tar.gz'
    - 'node_exporter-[0-9]+\.[0-9]+\.[0-9]+\.linux-arm64.tar.gz'
  releases:
    - 'latest'
    - 'v1.5.0'
  files:
    LICENSE: 'https://raw.githubusercontent.com/prometheus/node_exporter/master/LICENSE'
    install.sh: 'https://raw.githubusercontent.com/carlocorradini/node_exporter_installer/main/
                install.sh'

---
prometheus:
  url: 'https://github.com/prometheus/prometheus'
  assets:
    - 'prometheus-[0-9]+\.[0-9]+\.[0-9]+\.linux-amd64.tar.gz'
    - 'prometheus-[0-9]+\.[0-9]+\.[0-9]+\.linux-arm64.tar.gz'
  releases:
    - 'latest'
    - 'v2.42.0'
  files:
    LICENSE: 'https://raw.githubusercontent.com/prometheus/prometheus/main/LICENSE'
```

Listing 3.8: Content of dependencies configuration file

3.2.8.2 Script

A POSIX script named `dependencies.sh`[119] is used to automate all dependencies-related procedures. Its primary function is to read the dependencies configuration file and synchronize (download) the dependencies listed therein.
The script requires some `coreutils` package's utility programs and the `yq` application to correctly handle `YAML` file and syntax.
Dependencies' script behavior is customizable by using argument flags, which begin with a double dash followed by the property name and an optional value (`--NAME [VALUE]`).
The accepted configuration parameters are listed in the table below.

Name	Description	Default Value
`config-file`	Path to the dependencies configuration file (`<FILE>`). Both relative and absolute paths are supported. It should be noted that the configuration file name does not have to be `dependencies.config.yaml`, but may be any name and extension. The sole condition is that it must be in `YAML` format and that the attributes structure is respected.	`dependencies.config.yaml`
`sync`	Synchronize all dependencies specified in the configuration file. Dependencies are saved in the current working directory. If a file is already present in the respective directory, it is not downloaded (skipped).	

[119] `https://github.com/carlocorradini/reCluster/blob/main/dependencies/dependencies.sh`

`sync-force`	The same as `--sync`, except that all dependencies are downloaded even if they are already present locally. The downloaded file replaces the local one (if exists).	
`log-level`	Logging level (`<LEVEL>`). Attachment C provides additional information regarding logging and logging levels. The following logging levels are supported (listed in descending order of importance): ⑤ `fatal` ④ `error` ③ `warn` ② `info` ① `debug`	`info`
`help`	Display a help message and terminate (successfully).	

Table 3.3: Dependencies script parameters

To synchronize (force) all of the dependencies listed in the configuration file with the script, just execute: `./dependencies.sh --sync-force`

3.3 Server

Server[120] is a completely custom implementation that manages all low-level cluster operations, such as turning on and off nodes, as well as user operations, such as authentication and authorization. The server is written entirely in `TypeScript`[121], a strongly typed programming language that is compiled/-transpiled[122] to `JavaScript`, and executed by the `Node.js` runtime. When dealing with particular cluster processes, it has a strong policy of resource-waste minimization, attempting to reduce overall cluster power consumption while also enhancing the resource utilization of active nodes. It is intended to be completely interoperable with a Kubernetes cluster composed of heterogeneous systems. It provides a rich, type-safe, and secure `GraphQL API` (see section 3.3.2) that can be utilized manually by administrators and basic users or automatically by scripts and other cluster components, such as the Cluster Autoscaler (see section 3.4.3). Server, as the name implies, has a direct mapping to the cluster architecture's Server component.
To query, monitor (see section 3.3.5), and manipulate the state of the Kubernetes cluster, Server implementation mainly relies on the Kubernetes API[123]. The Kubernetes API server, which is operating on a Controller node, listens on port `6443` and is secured with TLS. TLS certificates can be signed using a private Certificate Authority (CA) or with a Public Key Infrastructure (PKI) linked to a well-known CA. Because the cluster implementation supports deployment in an Air-Gap environment, without any connection to an external network, and leveraging a fully-fledged certificate infrastructure for most of the conceivable use-case scenarios might be regarded a waste of resources and an unnecessary effort, it is recommended to utilize the solution with the private Certificate Authority, which is also the default one. If the Kubernetes cluster implementation is configured to work with a certificate signed by a private CA, then any K8s client, including the one employed by the Server, must have a local copy of the certificate, which is available within the corresponding `kubeconfig` file generated by the Controller, to mutually trust the connection (client-controller and controller-client) and be confident that it is not intercepted. The overall cluster implementation is meant to operate with a certificate signed by a private CA by default, hence the Server requires a copy of the `kubeconfig` file. If the

[120] `https://github.com/carlocorradini/reCluster/tree/main/server`
[121] `https://www.typescriptlang.org`
[122] `https://wikipedia.org/wiki/Source-to-source_compiler`
[123] `https://kubernetes.io/docs/concepts/security/controlling-access`

Kubernetes Server and the Server are deployed on the same node, as is the case with reCluster, the latter is completely automated by the installation script (see section 3.5).
The server resource-waste reduction strategy is maintained automatically and in conjunction with the Cluster Autoscaler component implementation, without any human/physical interaction. As stated in chapter 2, the Server and Cluster Autoscaler are required and operate as a single entity. Because it only has a low-level understanding of the cluster, the Server does not know when to scale (automatically) without the Cluster Autoscaler, which monitors the overall cluster workload and each active node resource utilization. Vice versa, without the Server, which has global low-level knowledge of the whole cluster, including active and inactive nodes, the Cluster Autoscaler only knows when to scale nodes up or down but not how to do so.

This section is split into five segments, each of which requires the knowledge provided by the previous segment to be completely understood. The first segment depicts the Database structure, its tables, and their relationships. The schema, which contains the core knowledge of the cluster and users, is constantly updated and persistently stored. The second segment depicts the GraphQL API and its queries (queries and mutations). The API provides advanced knowledge, the foundation of which is the database and its schema, in that certain queryable data are not available in the database but are calculated using multiple raw data from persistent storage. The latter is prominent throughout the autoscaling procedure. Certain queries are secured by authentication, authorization, or both, while others are exposed to any entity making the request (users, nodes, etc...). The third segment focuses on the autoscaling technique, illustrating how nodes are automatically powered on (from an inactive to active state) or powered off (from an active to inactive state). The latter are fully automated procedures that are triggered via specific GraphQL queries and consumed by either automatic entities like the Cluster Autoscaler or human entities like an administrator. The fourth segment demonstrates how the server continuously monitors the Kubernetes cluster using a specialized node informer that checks to see whether a node resource is added, deleted, or updated. The latter aids in keeping the cluster's knowledge consistent and up to date by using the Kubernetes heartbeat and resource systems. The final segment is about server configuration: how it may be configured and the many available parameters.

3.3.1 Database

This section is concerned with the Database, its structure (schema), and the interactions between it and the Server.
To interface with the Database, the Server implementation does not directly involve any raw SQL queries but instead depends on a kind of middleware layer that automatically translates any Database specific `TypeScript` functions and data structures into raw `SQL` queries and vice versa. Section 3.3.1.1 explains the latter, as well as the procedures and technologies involved.
Section 3.3.1.2 depicts the Database schema, which is a pure mapping of the cluster's low-level knowledge in the form of raw data, its many nodes (both active and inactive), and the various registered users. The schema not only illustrates how the various data are defined and organized in the Database, but it also reflects the cluster's knowledge and what data and information constitute it.

3.3.1.1 Object-Relational Mapping

Prisma ORM[124][125] manages all database-related operations in the Server implementation, removing the need for raw `SQL` queries and replacing them with more natural, easy-to-use, and maintainable `TypeScript` functions and data structures.
Consider a database with a table that represents the nodes in the cluster and has a structure similar to the one shown in section 3.3.1.2. The `name` and `memory` properties of the node with the identifier `e2b87848-bba0-46d5-923b-403f7141564f` should be selected. The raw `SQL` query looks like the following code:

```
const id = "e2b87848-bba0-46d5-923b-403f7141564f";
const result = await query("SELECT name, memory FROM node WHERE id = $1", [id]);
```

[124] https://wikipedia.org/wiki/Object-relational_mapping
[125] https://www.prisma.io

Listing 3.9: Raw SQL query to retrieve the `name` and `memory` attributes of the node with the identifier `e2b87848-bba0-46d5-923b-403f7141564f`

In contrast, the same query using the Prisma ORM looks like the following code:

```
const id = "e2b87848-bba0-46d5-923b-403f7141564f";
const result = await prisma.node.findUnique({
  select: {
    name: true,
    memory: true
  },
  where: {
    id: id
  }
});
```

Listing 3.10: Prisma ORM query to retrieve the `name` and `memory` attributes of the node with the identifier `e2b87848-bba0-46d5-923b-403f7141564f`

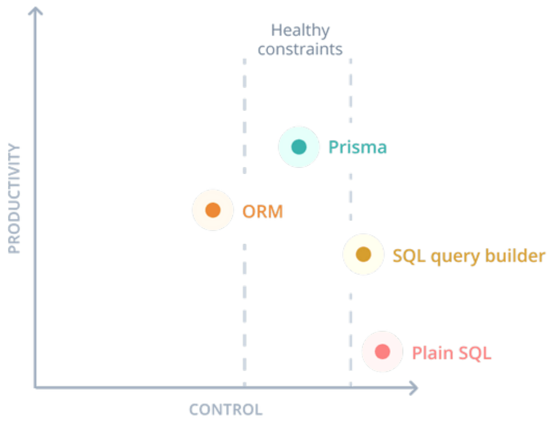
Take note of the differences, particularly how the ORM query is simpler, type-safe, and does not directly involve any `SQL` code but rather pure programming language primitives.
ORMs are not the only tool for interfacing with a Relational Database; alternative options, such as Raw SQL and SQL query builders, are available; nonetheless, ORMs provide the finest balance between application productivity and database control. Raw SQL provides complete control over database operations, but productivity suffers since interacting with raw SQL in code is time-consuming, error-prone, and produces a lot of overhead and maintainability issues. SQL query builders maintain a high level of control over database operations and give somewhat higher productivity, but they do it by relying on logical thinking in SQL rather than programming objects, functions, and primitives[126].
Figure 3.17 depicts the productivity vs control tradeoff of the various database tools, including Prisma.

Prisma

Source: `https://www.prisma.io`

Figure 3.16: Prisma logo

Prisma is an open-source next-generation ORM that defines the database data model structure using a Prisma Schema file[127][128], which is then used to construct the mapping between the database and the programming language with a rich and user-friendly API. Furthermore, because Prisma is not designed as a standard ORM but instead employs an intermediary Schema file, it can improve overall productivity and control over traditional ORMs[129]. Figure 3.17 reflects the latter.



Source: `https://www.prisma.io/docs/concepts/overview/why-prisma`

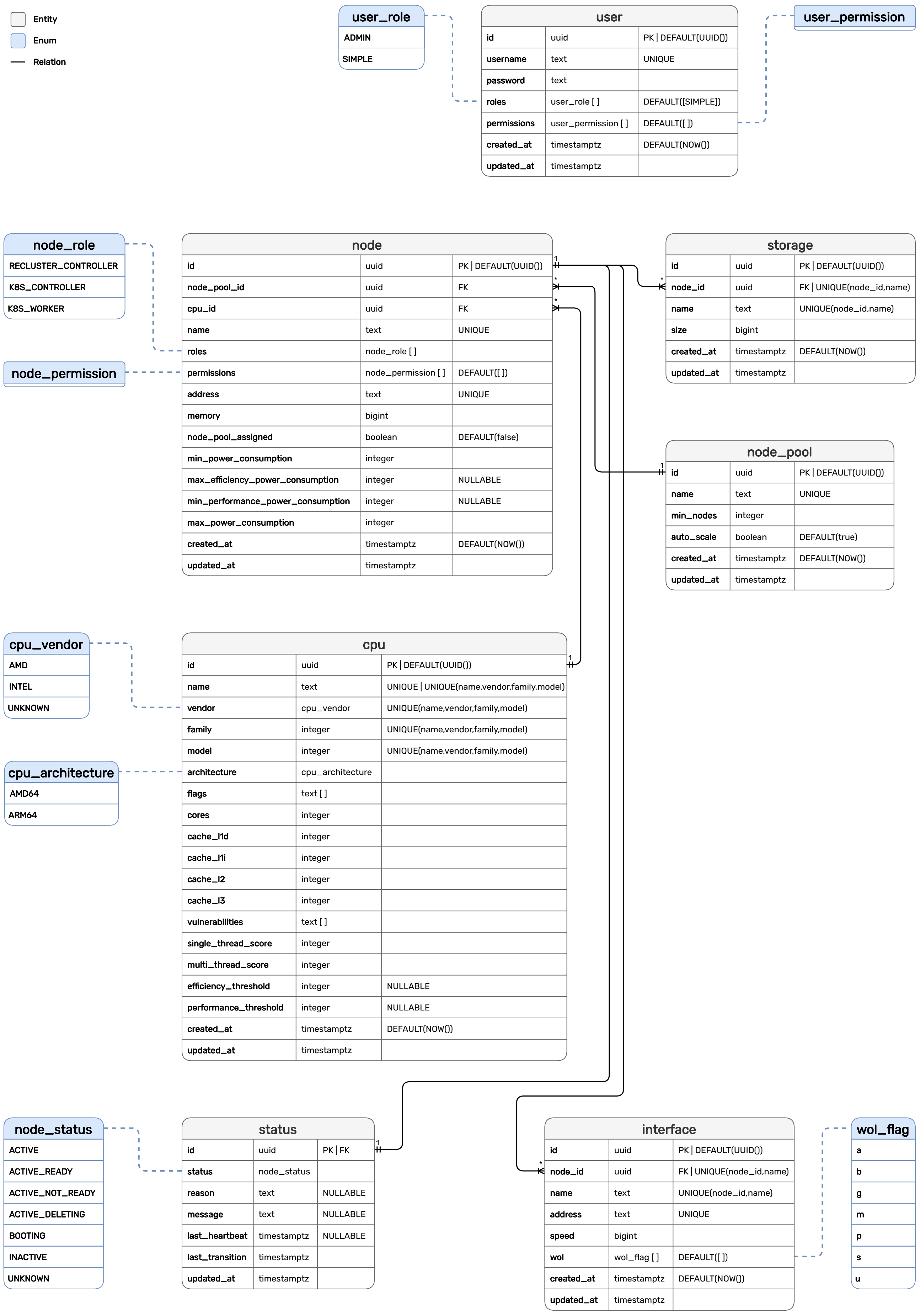
Figure 3.17: `Productivity` vs. `Control` tradeoff between various database tools

[126]`https://www.prisma.io/docs/concepts/overview/why-prisma`
[127]`https://www.prisma.io/docs/concepts/components/prisma-schema`
[128]`https://github.com/carlocorradini/reCluster/blob/main/server/prisma/schema.prisma`
[129]`https://www.prisma.io/docs/concepts/overview/what-is-prisma`

3.3.1.2 Schema

This section depicts and describes the database schema structure, as well as how the various entities (tables) are connected. Figure 3.18 displays the overall high-level representation of the database schema as an Entity Relationship Model[130] (ERM), where ⬜ represents an Entity, 🟦 an Enum, and — a relation between two entities. An Entity has a header and a body, with the header representing the entity's name and the body representing the attributes that make it up. The body is divided into three columns: the first column provides the attribute names, the second column is the type of the related attributes, and the third column contains extra information and context about the corresponding attribute (e.g. `PK` indicates that the attribute is a Primary Key[131] of the entity). An enum has the same form as an entity, with a header and a body, but the body has a single column that represents all of the Enum's possible values.
Following the schema, each entity is extensively described, as are the attributes that comprise it, as well as the relation(s) cardinality and why it is important.
This part's information is crucial for gaining a better understanding of the data that the server requires and the installation script (see section 3.5) collects from a node. Additionally, the latter is necessary for comprehending the raw data provided and processed by the GraphQL API (see section 3.3.2).

[130] `https://wikipedia.org/wiki/Entity-relationship_model`
[131] `https://wikipedia.org/wiki/Primary_key`



user_role
ADMIN
SIMPLE

user		
id	uuid	PK | DEFAULT(UUID())
username	text	UNIQUE
password	text	
roles	user_role [ ]	DEFAULT([SIMPLE])
permissions	user_permission [ ]	DEFAULT([ ])
created_at	timestamptz	DEFAULT(NOW())
updated_at	timestamptz	

user_permission

node_role
RECLUSTER_CONTROLLER
K8S_CONTROLLER
K8S_WORKER

node_permission

node		
id	uuid	PK | DEFAULT(UUID())
node_pool_id	uuid	FK
cpu_id	uuid	FK
name	text	UNIQUE
roles	node_role [ ]	
permissions	node_permission [ ]	DEFAULT([ ])
address	text	UNIQUE
memory	bigint	
node_pool_assigned	boolean	DEFAULT(false)
min_power_consumption	integer	
max_efficiency_power_consumption	integer	NULLABLE
min_performance_power_consumption	integer	NULLABLE
max_power_consumption	integer	
created_at	timestamptz	DEFAULT(NOW())
updated_at	timestamptz	

storage		
id	uuid	PK | DEFAULT(UUID())
node_id	uuid	FK | UNIQUE(node_id,name)
name	text	UNIQUE(node_id,name)
size	bigint	
created_at	timestamptz	DEFAULT(NOW())
updated_at	timestamptz	

node_pool		
id	uuid	PK | DEFAULT(UUID())
name	text	UNIQUE
min_nodes	integer	
auto_scale	boolean	DEFAULT(true)
created_at	timestamptz	DEFAULT(NOW())
updated_at	timestamptz	

cpu_vendor
AMD
INTEL
UNKNOWN

cpu_architecture
AMD64
ARM64

cpu		
id	uuid	PK | DEFAULT(UUID())
name	text	UNIQUE | UNIQUE(name,vendor,family,model)
vendor	cpu_vendor	UNIQUE(name,vendor,family,model)
family	integer	UNIQUE(name,vendor,family,model)
model	integer	UNIQUE(name,vendor,family,model)
architecture	cpu_architecture	
flags	text [ ]	
cores	integer	
cache_l1d	integer	
cache_l1i	integer	
cache_l2	integer	
cache_l3	integer	
vulnerabilities	text [ ]	
single_thread_score	integer	
multi_thread_score	integer	
efficiency_threshold	integer	NULLABLE
performance_threshold	integer	NULLABLE
created_at	timestamptz	DEFAULT(NOW())
updated_at	timestamptz	

node_status
ACTIVE
ACTIVE_READY
ACTIVE_NOT_READY
ACTIVE_DELETING
BOOTING
INACTIVE
UNKNOWN

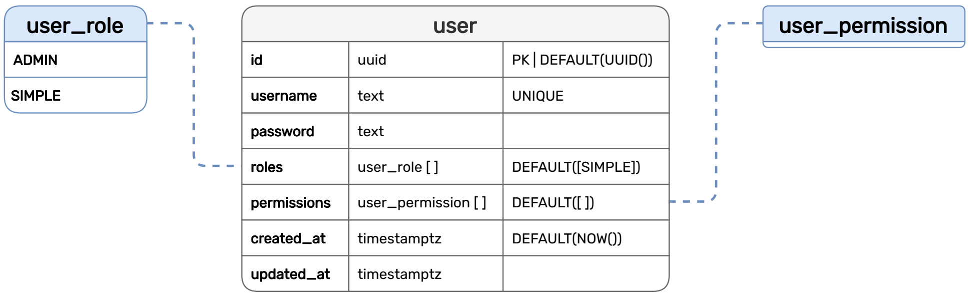
status		
id	uuid	PK | FK
status	node_status	
reason	text	NULLABLE
message	text	NULLABLE
last_heartbeat	timestamptz	NULLABLE
last_transition	timestamptz	
updated_at	timestamptz	

interface		
id	uuid	PK | DEFAULT(UUID())
node_id	uuid	FK | UNIQUE(node_id,name)
name	text	UNIQUE(node_id,name)
address	text	UNIQUE
speed	bigint	
wol	wol_flag [ ]	DEFAULT([ ])
created_at	timestamptz	DEFAULT(NOW())
updated_at	timestamptz	

wol_flag
a
b
g
m
p
s
u

Figure 3.18: Database Entity Relationship Model (ERM)

3.3.1.3 User



user		
id	uuid	PK | DEFAULT(UUID())
username	text	UNIQUE
password	text	
roles	user_role [ ]	DEFAULT([SIMPLE])
permissions	user_permission [ ]	DEFAULT([ ])
created_at	timestamptz	DEFAULT(NOW())
updated_at	timestamptz	

The user entity represents all of the cluster's registered users.
A user can logically be a real person, such as an administrator, or an automation system, such as the Cluster Autoscaler.
The user entity is currently solely used for authentication and authorization in the GraphQL API. Certain queries are only available to authenticated users, while others are only available to users with specified roles and/or permissions, or both. Yet, nothing prevents the future implementation of additional capabilities or the extension of the entity with extra attributes and/or relationships with other (new) entities. Because of the latter, this entity differs from the others in that it exists independently of other entities.
The `user` entity attributes are as follows:

- `id`
  Uniquely identify (`PK`) a user record.
  The attribute is of type Universally Unique Identifier[132] (`UUID`) version 4, and it is a 128-bit alphanumeric value generated by a (secure) random number generator. Unless otherwise specified, the UUID value is generated automatically whenever a new user record is added to the entity.
  The `UUID` type is widely used in database schema to uniquely identify all entity records.

- `username`
  User's username.
  A string of characters that uniquely (`UNIQUE`) identifies a user record.
  The `username` and `id` attributes are logically interchangeable since they both uniquely identify the same user record. Yet, memorizing a personalized and simple sequence of characters (`username`) is significantly simpler than remembering a 128-bit alphanumerical (`UUID`).
  During the authentication method in the GraphQL API, the `username` attribute is utilized in combination with the `password` attribute to guarantee that the user exists and is who it claims to be.

- `password`
  User's password.
  A (secret) string of letters, numbers, and symbols used to identify the user during the authentication procedure.
  The password has rigorous criteria that can be changed in the server configuration (see section 3.3.6): A minimum of eight characters (inclusive), one capital letter, one number, and one symbol are required.
  The password stored in the database is not in plain text; rather, it is derived from the original one provided by the user during the registration process using a password-hashing function[133].
  The password-hashing function employed in the implementation is `bcrypt`[134]. which is based on the `Blowfish`[135] cipher and includes a `salt`[136] (random data used as an additional input).
  The `bcrypt` function is an industry-standard that can withstand brute-force attacks and other

[132] https://wikipedia.org/wiki/Universally_unique_identifier
[133] https://wikipedia.org/wiki/Password-hashing_function
[134] https://wikipedia.org/wiki/Bcrypt
[135] https://wikipedia.org/wiki/Blowfish_(cipher)
[136] https://wikipedia.org/wiki/Salt_(cryptography)

threats[29].
It should be noted that, for security reasons, the `password` attribute is never returned by the GraphQL API (there is no mapping at all in the GraphQL schema) or by default by the Prisma ORM unless specifically provided in the `select` object.

- `roles`
  User's roles.
  Roles are used to logically group users into established categories, with various roles indicating varying levels of privilege.
  The `roles` attribute is of type `array` (`[]`) of `user_role` enum since a user might have multiple/different responsibilities at the same time.
  The `user_role` enum has the following possible values/constants:

  - `ADMIN`
    Identifies administrators and/or automation systems.
    It is the highest currently available role that permits the execution of any operation (query).
  - `SIMPLE`
    Identifies users who do not have any extra privileges.
    When a new user is added to the database, the role assigned by default is `SIMPLE`.

  It should be noted that a `SIMPLE` user cannot perform any operations that an ADMIN can't. Also, the `roles` attribute must always include at least one value.

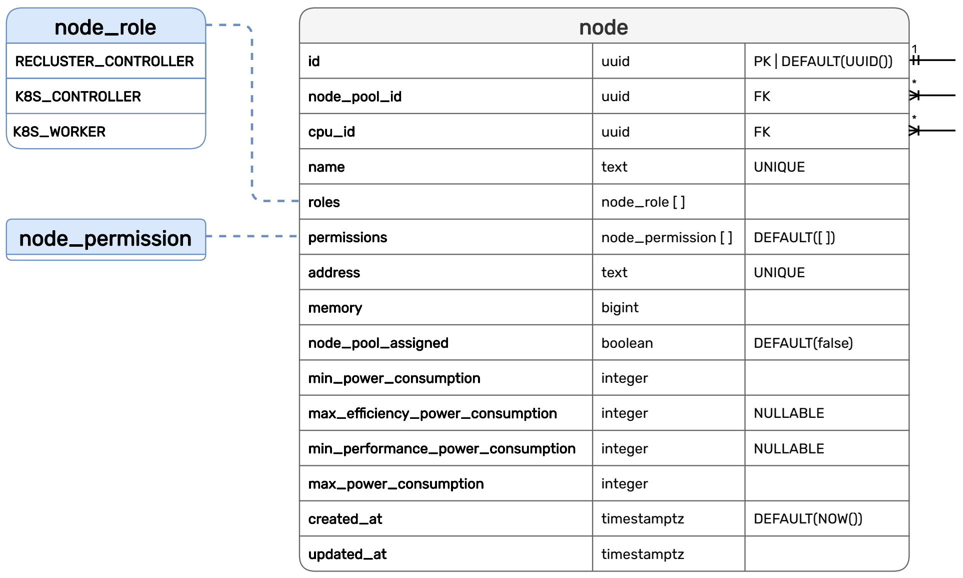
- `permissions`
  User's permissions.
  Identifies the possible actions that a user is allowed to execute.
  The `permissions` attribute is directly tied to the `roles` attribute and the context in which a query is executed. For example, two `ADMIN` users can change data on a node in the cluster (both have the `UPDATE` permission), but only the first user can delete a node record from the entity since it has the `DELETE` permission, while the second does not.
  The type of the attribute is an `array` of `user_permission` enum since a user might have multiple/different permissions depending on the role(s) and context.
  The `user_permission` enum is currently empty (hence the default value as an empty array) since no queries require rigorous authentication/authorization. Still, future implementations are completely supported.

- `created_at`
  Date and time of creation.
  The attribute indicates when the user record was created.
  The type is `timestamptz`[137] because it specifies a timestamp that also includes timezone information for the database's deployment location. It should be noted that the timezone is set to Coordinated Universal Time[138] (UTC) by default, therefore it is not tied to a specific timezone and hence the value may be readily converted to a specific timezone.
  The attribute is automatically set thanks to `DEFAULT` which gets the value from the `NOW()` function that returns the current timestamp and timezone.

- `updated_at`
  Date and time of the most recent update.
  The attribute specifies the most recent timestamp the user record was updated.
  The attribute is of type `timestamptz`.
  The value is controlled automatically by the server rather than the database. The latter is owing to differing database implementations, some of which support the feature natively while others do not.

[137]https://www.postgresql.org/docs/current/datatype-datetime.html
[138]https://wikipedia.org/wiki/Coordinated_Universal_Time

3.3.1.4 Node



node_role
RECLUSTER_CONTROLLER
K8S_CONTROLLER
K8S_WORKER

node_permission

node		
id	uuid	PK | DEFAULT(UUID())
node_pool_id	uuid	FK
cpu_id	uuid	FK
name	text	UNIQUE
roles	node_role [ ]	
permissions	node_permission [ ]	DEFAULT([ ])
address	text	UNIQUE
memory	bigint	
node_pool_assigned	boolean	DEFAULT(false)
min_power_consumption	integer	
max_efficiency_power_consumption	integer	NULLABLE
min_performance_power_consumption	integer	NULLABLE
max_power_consumption	integer	
created_at	timestamptz	DEFAULT(NOW())
updated_at	timestamptz	

The node entity represents all of the cluster's physical nodes, both active and inactive. It holds critical information about a node, and from this entity, other information like the current status or the hardware components may be obtained.
Certain GraphQL API queries can only be performed by an authorized node. The node lacks a `password` attribute like the user entity, both because it is bad practice keeping the plain password in the node system and also because it is an entirely new kind of object that must handle the various operations differently. When the node record is created in the database for the first time, the auth token (in the user returned when it is successfully authenticated after providing the correct `username` and `password` attributes, see section 3.3.2.2) is returned only once and it cannot be obtained again. As a result, the auth token is securely stored only in the corresponding node, and if it is lost for any reason, indicating that the node is faulty or compromised, the only way to retrieve a new token and perform secure operations on the specific node is to register the node in the cluster again (removing the old record).
The `node` entity attributes are as follows:

- `id`
  Uniquely identify (`PK`) a node record.
  The attribute is of type `UUID`.
  The `id` attribute is related to multiple entities. A node can only have one status (see section 3.3.1.8), and the status record belongs to only one unique node, hence the relation is one (`1`) to one (`1`). A node can have several storages (see section 3.3.1.6), but each storage corresponds to a single and distinct node, hence the relation is one (`1`) to many (`*`). Lastly, like with the previous relation, a node might have several interfaces (see section 3.3.1.7), but each interface corresponds to a single and unique node, therefore the relation is one (`1`) to many (`*`).

- `node_pool_id`
  Node pool identifier.
  Uniquely identify a node pool record (`FK`).
  The attribute is of type `UUID`.
  When a new node joins the cluster, it is automatically assigned to a node pool (see section 3.3.1.9) based on its roles and hardware. If there is no node pool for the node, one is automatically created. The relation is many (`*`) to one (`1`), indicating that a node may only be allocated to one node pool, but a node pool can be assigned to several nodes.

- `cpu_id`
  Cpu identifier.
  Uniquely identify a CPU record (`FK`).
  The attribute is of type `UUID`.

During the registration phase for a new node in the cluster, it is first verified to determine if the same CPU type is already registered in the database (see section 3.3.1.5); if not, a new CPU record is created using the information supplied by the node and then assigned to it.
The relation is many (`*`) to one (`1`), indicating that a node may only have one CPU, although the same CPU can be present on several nodes.

- `name`
  Node's name.
  A string of characters that uniquely (`UNIQUE`) identifies a node record.
  It serves the same purpose as the user entity's `username` attribute: it allows users/administrators to easily identify a node by using a shorter name rather than the 128-bit alphanumerical identifier (`id`). As a result, the `name` and `id` attributes are interchangeable because they both uniquely identify the same node entry.
  A default value is automatically assigned by the Server during the node registration procedure and consists of two sections separated by a dot (`.`). The first section provides a high-level overview of the node's role, which can be either `Controller` or `Worker` (if a node is both, `Controller` is preferred since it has the highest logical importance). The node's identifier is the second element. As an example, a node with the id `e2b87848-bba0-46d5-923b-403f7141564f` and both roles will be given the name `controller.e2b87848-bba0-46d5-923b-403f7141564f`.
  Because this attribute is only deemed as a helper, its value can be modified by administrators (via the `updateNode` GraphQL query) to fit a more appropriate name that better represents the node in the cluster.

- `roles`
  Node's roles.
  This attribute is nearly similar to that of the user entity, but it is unique to nodes rather than users. Roles are used to logically categorize nodes, with different roles indicating various degrees of privilege.
  Because a node may have multiple/different responsibilities at the same time, the `roles` attribute is of the form array (`[]`) of `node_role` enum.
  The `node_role` enum has the following possible values/constants:

  - `RECLUSTER_CONTROLLER`
    The corresponding node has an instance of a reCluster component that controls (monitors) the cluster and thus must be operational at all times.
    The Server component is currently the only reCluster component that can be deployed outside of the K8s environment and needs to be always active.
  - `K8S_CONTROLLER`
    The node is a Kubernetes Controller that must be maintained functioning at all times.
    It must not be autoscaled by the Cluster Autoscaler component and the only entities authorized to deactivate it are administrators via a specific protected GraphQL query.
  - `K8S_WORKER`
    The node is a Kubernetes Worker that can be autoscaled by the Server component in combination with the Cluster Autoscaler component.
    It is important to note that if a node is also a Controller, it is not autoscaled.

A node can have multiple roles assigned to it, but the `roles` attribute must always have at least one value. A node can be a Kubernetes Controller (`K8S_CONTROLLER`) and Worker (`K8S_WORKER`) that monitors the K8s cluster but also accepts workload, as well as a reCluster Controller (`RECLUSTER_CONTROLLER`) that enables physical node autoscaling and provides the GraphQL API.
The generic Controller component in reCluster is both a Kubernetes Controller and a reCluster Controller, but not a Worker.

- `permissions`
  Node's permissions.
  This attribute is comparable to that of the user entity, but it is specific to nodes rather than users. The `roles` attribute and the context in which a query is performed are both directly related to the `permissions` attribute. Furthermore, and this is where the `permissions` in the node entity differ, it can be used within the node during some predefined actions such as before termination or after bootstrap.
  Because a node may have multiple/different `permissions` depending on the role(s), context, and action, the attribute's type is an array of `node_permission` enum.
  Even though the cluster implementation supports it out of the box, there are currently no available permissions for nodes (hence the empty array as the default value). The latter enables future implementations to easily add and modify the `node_permission` enum and `permissions` attribute without changing the implementation.

- `address`
  Node's IP address.
  The IP address of the node in the Internal Network. There must be no nodes with the same IP address in the Internal Network. The latter is reflected in the database schema, which specifies that the attribute value must be `UNIQUE`, and thus two nodes cannot have the same IP address. As a result, this attribute can also be used to uniquely distinguish a node record in the nodes entity.
  Because each database implementation has its own specific/custom datatype for encoding an IP address, the attribute is of type text.
  During the downscaling procedure (see section 3.3.4), the node's IP address is used to establish an SSH connection (from Server to Node) and remotely terminate the node. Furthermore, the IP address is available on multiple GraphQL queries and can be used by users or scripts. The latter is especially helpful when using Prometheus because it requires the IP addresses of the nodes where the metrics are published to monitor them.
  The attribute's value is constantly updated using the K8s informer provided by the Kubernetes API (see section 3.3.5). Because of this, the Database and thus the Server always have the most recent available IP address of the node, even though (technically) it should never change.

- `memory`
  Node's memory size in bytes.
  Because a large amount of memory does not fit into the smaller `integer` type, the attribute is of type `bigint`[139].
  The unit is in `byte` because it is the smallest on which memory can be represented (`bit` unit is almost useless and the conversion is straightforward). The latter is especially useful when dealing with conversions (which are directly supported by the GraphQL API) and comparisons because the unit is always the same and thus no further adjustments are needed.
  Every attribute in the database that defines a quantity is represented in its smallest unit. The installation script or the server automatically converts a larger value to its smallest unit.

- `node_pool_assigned`
  A Boolean flag that indicates whether the node is allocated to the corresponding node pool.
  The attribute is of type `boolean`, with `true` indicating that the node is allocated to the node pool (visible to Kubernetes) and `false` indicating that the node is not allocated to the node pool (not visible to Kubernetes).
  As previously stated, a node is always related to a node pool upon registration. This attribute is used in combination with the status entity (see section 3.3.1.8) to decide whether the node is appropriate for autoscaling. A node in the `inactive` state that is not allocated to a node pool can be selected for automatic bootstrap. A node that is not allocated to the node pool but is in a state other than `inactive` indicates that the node is bootstrapping, terminating, or

[139]`https://www.postgresql.org/docs/current/datatype-numeric.html`

experiencing a problem.
The default value (`DEFAULT`) is `false` because the node has not yet initiated any Kubernetes-related processes and thus cannot be allocated to the pool when it is registered for the first time in the cluster. Only when the installation procedure has been completed successfully and the node sends a request to a specific and protected GraphQL API along with the auth token the value can be set to `true`.

- `min_power_consumption`
  Minimum power consumption in `Watt` (`W`).
  The node's minimum power consumption when there is no workload and the running processes are the bare minimum.
  The unit is in `Watt` rather than a lower one because the device used to measure the power consumption of the different nodes has a minimum sensitivity of `Watt` (see section 3.5.2). Furthermore, because the read values are always integers (i.e., no decimal point/float), the attribute's type can be securely set to `integer`. To represent that the device can read units well below the `Watt`, the type can be set to `bigint` in different implementations to prevent overflow. Nonetheless, because the maximum value of an `integer` is approximately `2` or `4` billions (depending on whether the sign can be ignored or not depending on the database implementation) and there are no systems that utilize this much, the `integer` type is well above the needs for the implementation.
  The latter affects every attribute in the database that denotes a value for power consumption.

- `max_efficiency_power_consumption`
  Maximum power consumption in `Watt` (`W`) when in a power-efficient state.
  Modern systems are beginning to include hybrid components that, when under low workload conditions, are more power efficient, consuming less energy at the expense of performance, while when under high workload conditions, are more energy-hungry (less power efficient), but with higher performance and reactiveness. With efficiency and performance cores, the latter is becoming more prevalent in the CPU ecosystem (see section 3.3.1.5).
  Because hybrid systems are still in their early stages and the implementation does not yet recognize when and how the threshold is reached, causing the system to transition from an efficiency to a performance state, the attribute is `NULLABLE`. Furthermore, a `NULL` value can determine whether or not the node is hybrid.

- `min_performance_power_consumption`
  Minimum power consumption in `Watt` (`W`) when in a performance state.
  It is closely related to the previous attribute `max_efficiency_power_consumption`, but in this case, the attribute reflects the minimum power consumption when the hybrid system is in the performance state and has the lowest workload. Simply put, it is the minimum power consumption when switching from a power-efficient to a performance state.
  This attribute, like the prior one, is `NULLABLE` due to the present lack of tools and the effort needed to monitor hybrid systems, and a `NULL` value can identify whether or not the node is hybrid.
  It is important to note that whenever the node is registered or updated, both the attributes `max_efficiency_power_consumption` and `min_performance_power_consumption` must be provided; otherwise, an error is generated and the operation is terminated.

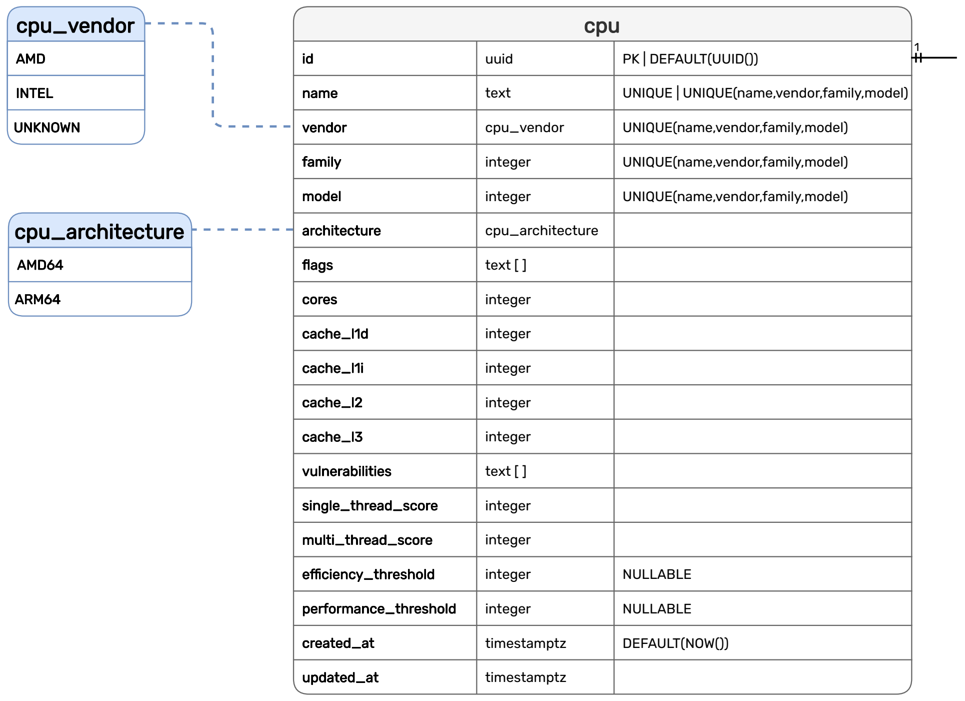
- `max_power_consumption`
  Maximum power consumption in `Watt` (`W`).
  The node's maximum power consumption when under full workload, and thus its hardware resources are under intense stress.
  It is the opposite of the `min_power_consumption` attribute, and its value is read when the node is under extreme stress.

- `created_at`
  Date and time of creation.

The attribute indicates when the node record was created.
The attribute is of type `timestamptz`.

- `updated_at`
  Date and time of the most recent update.
  The attribute specifies the most recent timestamp the node record was updated.
  The attribute is of type `timestamptz`.

3.3.1.5 Cpu



cpu_vendor
AMD
INTEL
UNKNOWN

cpu_architecture
AMD64
ARM64

cpu		
id	uuid	PK | DEFAULT(UUID())
name	text	UNIQUE | UNIQUE(name,vendor,family,model)
vendor	cpu_vendor	UNIQUE(name,vendor,family,model)
family	integer	UNIQUE(name,vendor,family,model)
model	integer	UNIQUE(name,vendor,family,model)
architecture	cpu_architecture	
flags	text [ ]	
cores	integer	
cache_l1d	integer	
cache_l1i	integer	
cache_l2	integer	
cache_l3	integer	
vulnerabilities	text [ ]	
single_thread_score	integer	
multi_thread_score	integer	
efficiency_threshold	integer	NULLABLE
performance_threshold	integer	NULLABLE
created_at	timestamptz	DEFAULT(NOW())
updated_at	timestamptz	

The cpu entity contains information about the CPU (Central Processing Unit).
The entity includes the most important information about a particular CPU model obtained from the Linux Kernel data available at the virtual path `/proc/cpuinfo`[140].
Because a particular CPU model can be present in numerous nodes (as is the case with worker nodes in reCluster), it is checked before registering a node that the CPU is already listed in the database. If not, it is registered using the provided information; otherwise, the information is combined with certain attributes to minimize potential deviation errors (i.e. `benchmarks`) or to update the attribute with the most recent known information (i.e. `vulnerabilities`).
As stated in the node entity, modern Processors, such as Arm Big.LITTLE[141] or Intel Performance Hybrid Architecture[142], are beginning to include hybrid technology (heterogeneous processing architecture) which employs two kinds of processors. The first type of processor is intended for maximum power efficiency, which reduces overall performance, whereas the second type of processor is designed for maximum compute performance, which increases power consumption. This form of Processor can instantly adapt to the variable workload by switching from one type to the other[5][15]. Even though the latter technology is new and difficult to monitor (understanding when the CPU decides to transition from one type to the other and vice versa), it is represented in the entity with two dedicated attributes (`efficiency_threshold` and `performance_threshold`).
The `cpu` entity attributes are as follows:

- `id`
  Uniquely identify (`PK`) a CPU record.
  The attribute is of the `UUID` type.
  Because multiple nodes can share the same CPU type, the id attribute is associated with the node entity. As a result, the relation is one (`1`) to many (`*`).

[140]`https://www.kernel.org/doc/html/latest/filesystems/proc.html#kernel-data`
[141]`https://www.arm.com/technologies/big-little`
[142]`https://www.intel.com/content/www/us/en/gaming/resources/how-hybrid-design-works.html`

- `name`
  CPU name.
  The Processor's common name, including its project name.
  The `name` attribute, like some of the previous attributes, can be used by users and administrators to easily and uniquely (`UNIQUE`) identify the processor without relying on one or more complicated alphanumerical attribute(s).
  This attribute, along with the `vendor`, `family`, and `model` attributes, specifically defines a CPU record (`UNIQUE(name,vendor,family,model)`). The quadruplet is used during the node registration procedure to uniquely identify if the same CPU has already been registered. It should be noted that if a CPU has the same `name` attribute value but a different value for the other attributes, registration fails because something in the received data is incorrect.
  An example `name` attribute value is `Intel(R) Core(TM) i7-6700HQ CPU @ 2.60GHz`, which identifies the model `i7-6700HQ` with a basic frequency of `2.60GHz` manufactured by `Intel Corporation`.

- `vendor`
  CPU vendor identifier.
  Identifies the Processor manufacturer identifier.
  The attribute is of type `cpu_vendor` enum. It was decided to use a preset set of values because almost every desktop CPU processor is manufactured by a very limited number of vendors, and knowing the possible values ahead of time aids in the development of a more powerful and robust GraphQL API. Nonetheless, if a CPU has a manufacturer that is not available, an `UNKNOWN` value exists, enabling future implementations to easily add and support more manufacturers.
  The `cpu_vendor` enum has the following possible values/constants:

  - `AMD`
    Advanced Micro Devices (AMD).
  - `INTEL`
    Intel Corporation.
  - `UNKNOWN`
    Identifies an unknown manufacturer.

  Due to historical reasons, the original manufacturer identifier string value read from `/proc/cpuinfo` is slightly different[143]: `INTEL` is `GenuineIntel` and `AMD` is `AuthenticAMD`. They have been changed/standardized to be more easily identified, with a shorter name as well.

- `family`
  CPU family.
  Identifies the Processor's microarchitectural lineage.
  Following the example CPU in the `name` attribute, it has a family value of `6`, indicating that it is a descendant of the Pentium Pro and thus uses the same microarchitecture.

- `model`
  CPU model.
  Identifies the model of the Processor as specified by the manufacturer.
  It should be noted that the unique quadruplet (`UNIQUE(name,vendor,family,model)`) must include the `name` attribute because the other three attributes (`vendor`, `family`, `model`) distinguish only a collection of processors rather than a single specific one.
  Following the example CPU in the `name` attribute, it has a model value of `94` indicating that it is a `Skylake` generation (model) Processor.

- `architecture`
  CPU architecture.
  Identifies the Processor Instruction Set Architecture implementation.

[143] `https://wikipedia.org/wiki/CPUID`

The attribute is of type `cpu_architecture` enum. The possible values reflect the CPU architectures supported by the cluster. The latter is critical because some components only support specific architectures, and downloaded dependencies ignore files that are not present in the configuration (commonly unsupported architectures). It should be noted that the installation script detects if it is operating on an unsupported architecture. Nonetheless, the verification occurs also in the GraphQL API during the registration process.
The `cpu_architecture` enum has the following possible values/constants:

  - `AMD64`
    `64-bit` version of the `x86` instruction set.
    This architecture is also known as `x64`, `x86_64`, `x86-64`, and `Intel 64`.
  - `ARM64`
    `64-bit` extension of the ARM architecture family.
    This architecture is also known as `AArch64`.

  It should be noted that the names of the enum values are subjective, and thus they can be (easily) changed to better fit the organization administering the cluster's favored choice.

- `flags`
  CPU feature flags.
  Determines the features that the Processor implements (identified by a distinct flag name). One example is the `fpu` (floating-point unit) flag, which indicates that the CPU has a dedicated coprocessor for working with floating-point numbers.
  The attribute is of the type array (`[]`) of text, and it contains all of the feature flags. The latter enables a robust and complex GraphQL API for filtering and searching particular CPUs with a specified set of flags.
  An application can be built to support a particular set of feature flags (normally improving overall performance thanks to compiler optimizations) and then scheduled only on worker nodes whose Processor implements these flags. The selection of nodes is based solely on the node(s) identification number (`id` attribute) that is available in the Kubernetes environment under the label `recluster.io/id=<ID>`.

- `cores`
  CPU core count.
  Almost every modern CPU is a multi-core processor which combines multiple processing units (two or more) to enable parallel work. This attribute indicates how many core units the CPU has.
  It should be noted that this value takes into consideration the CPU's multithreading capability, so it is common for this value to be twice the real physical number of cores.
  Following the example CPU in the `name` attribute, the attribute has a value of `8`, even though the actual physical cores are `4`.

- `cache_l1d`
  CPU `L1d` (data) cache size in bytes (`B`).
  Because of their small sizes, all cache attributes are of the type `integer` rather than `bigint`.

- `cache_l1i`
  CPU `L1i` (instructions) cache size in bytes (`B`).

- `cache_l2`
  CPU `L2` cache size in bytes (`B`).

- `cache_l3`
  CPU `L3` cache size in bytes (`B`).

- `vulnerabilities`
  Known CPU vulnerabilities.

A collection of known vulnerabilities that affect the Processor.
Because the Processor is vulnerable to numerous vulnerabilities, the attribute's type is an array (`[]`) of `text`. If the CPU is already registered, the previous value of the `vulnerabilities` attribute is combined with the new value obtained during the registration procedure, with the result that if newer vulnerabilities are discovered, the database and server components are immediately updated with the information. Furthermore, because it is an array type, the GraphQL API allows advanced filtering and searching.
The vulnerabilities attribute can be used to evaluate applications on known vulnerable dedicated nodes. The latter is particularly helpful in an offensive security class, where students can experiment with various vulnerabilities to take control of the node or execute arbitrary code.

- `single_thread_score`
  CPU single-thread score.
  This attribute indicates the performance score of a single CPU core.
  It should be noted that the evaluation, as well as the final score for this attribute and `multi_thread_score` attribute, are carried out using the `sysbench` application (see section 3.5.1)

- `multi_thread_score`
  CPU multi-thread score.
  This attribute indicates the performance score of all CPU cores combined.
  It should be noted that hyperthreading capability is taken into consideration, and the tests are performed on highly parallelizable algorithms.

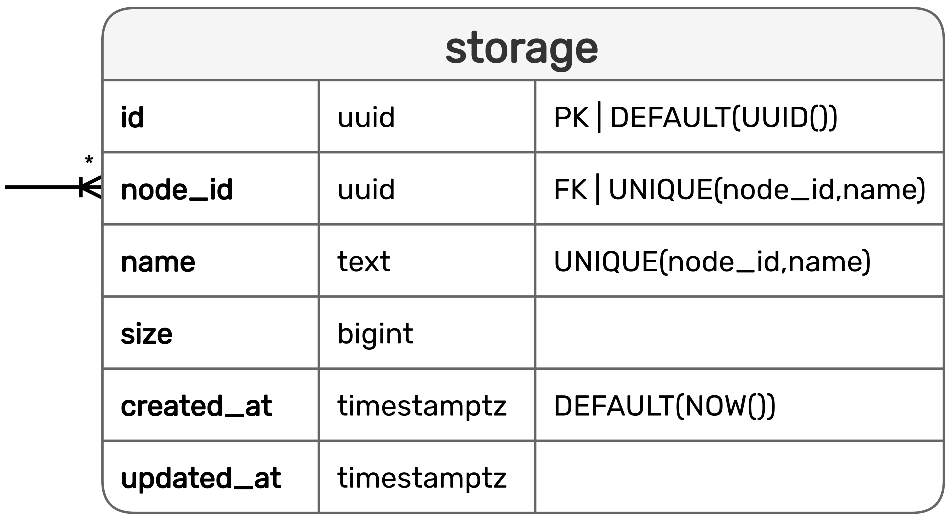
- `efficiency_threshold`
  CPU power-efficiency threshold (inclusive).
  The workload threshold value (upper) beyond which it is no longer certain that the Processor will stay in a power-efficient condition.
  The threshold value is a number between `1` (inclusive) and `99` (inclusive), where `1` indicates that the CPU is executing at `1%` of its total capacity (almost idle) and `99` indicates that the CPU is operating at `99%` of its total capacity (full load).
  Due to the difficulty of monitoring the threshold value, the attribute allows a `NULL` value (`NULLABLE`).

- `performance_threshold`
  CPU performance threshold (inclusive).
  The workload threshold value (lower) beyond which it is no longer certain that the Processor will stay in a performance condition.
  By combining the attributes `efficiency_threshold` and `performance_threshold`, a workload range is obtained that indicates an area of uncertainty where the Processor can transition from one state to the other or remain in the current one. For example, if a CPU has the attribute `efficiency_threshold` set to `45` and the attribute `performance_threshold` set to `55`, then the workload range between `45` and `55` (`10` workload range) is uncertain, and the Processor can transition or not.
  Due to the difficulty of monitoring the threshold value, the attribute allows a `NULL` value (`NULLABLE`).

- `created_at`
  Date and time of creation.
  The attribute indicates when the CPU record was created.
  The attribute is of the `timestamptz` type.

- `updated_at`
  Date and time of the most recent update.
  The attribute specifies the most recent timestamp the CPU record was updated.
  The attribute is of the `timestamptz` type.

3.3.1.6 Storage



storage		
id	uuid	PK | DEFAULT(UUID())
node_id	uuid	FK | UNIQUE(node_id,name)
name	text	UNIQUE(node_id,name)
size	bigint	
created_at	timestamptz	DEFAULT(NOW())
updated_at	timestamptz	

The storage entity represents the persistence storage(s) accessible on the associated node. The information is obtained by reading the block devices identified by the Linux kernel and made available via the virtual path `/sys/block`[144]. Block devices[145], such as a Hard Disk, SSD, CD-ROM, and RAM disk, are distinguished by random access to data organized in fixed-size blocks. Because the implementation only requires persistent storage(s), the collection of block devices is filtered to retrieve only Disk(s)/SSD(s) block devices that are not removable devices (i.e. USB flash drive). The latter is done automatically by the installation script while reading the node information.
A node can have zero or multiple storage devices. Nonetheless, it is usual for a consumer system to include at least one storage device, and modern (consumer) systems may include a combination of a Hard Disk (cheap with large capacity but slow) and an SSD (expensive but faster than a Hard Disk). Using a multi-SSD configuration can significantly improve cluster performance. Furthermore, if certain nodes have three or more storage devices, Longhorn (see section 3.2.5.3) can utilize them to create a distributed block storage system.
Even though `Sysbench` (see section 3.5.1) supports it, the application currently does not support a scoring system for a storage device that can help determine its performance under various conditions.
The `storage` entity attributes are as follows:

- `id`
  Uniquely identify (`PK`) a storage record.
  The attribute is of type `UUID`.

- `node_id`
  Node identifier.
  Uniquely identify a node record (`FK`).
  The attribute is of type `UUID`.
  A node can have multiple storage devices, but each storage device is associated with one unique node. As a result, the relation is many (`*`) to one (`1`).

- `name`
  Storage name.
  The name of the storage device on the associated node. In GNU/Linux, typical storage device names are `sda`, `sdb`, `sdc`, and so on for Hard Disks/SSDs and `nvme0`, `nvme1`, `nvme2`, and so on for NVMEs. Nonetheless, the organization in charge of the cluster can choose to modify the default name by using arbitrary names to better identify the storage device.
  Despite its appearance as a helper for users or administrators, this attribute is also used in conjunction with the `node_id` attribute to uniquely (`UNIQUE(node_id,name)`) identify the storage

[144] `https://www.kernel.org/doc/html/next/filesystems/sysfs.html`
[145] `https://linux-kernel-labs.github.io/refs/heads/master/labs/block_device_drivers.html`

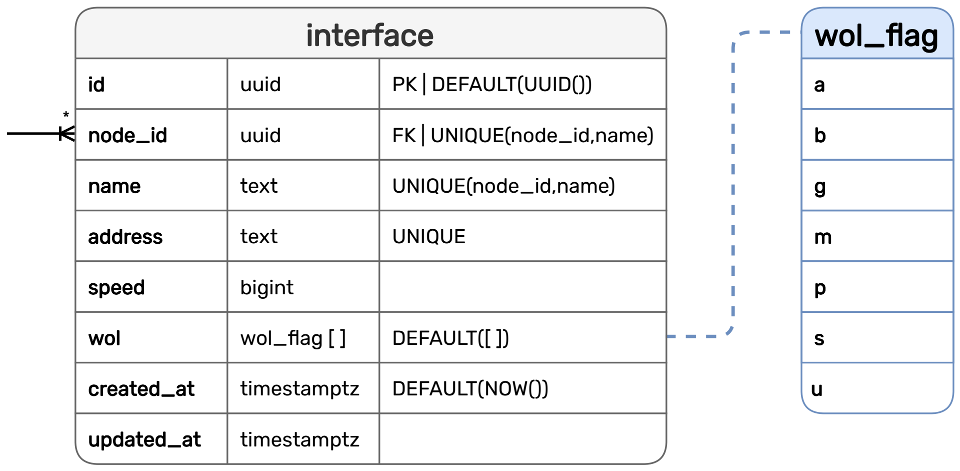
device on the node and to prevent the registration of two devices with the same name on the same node.

- `size`
  Storage memory size in bytes (`B`).
  The total quantity of memory that the storage device can store.
  Because the overall memory of common storage devices is quite large and their counterpart in bytes is quite massive, the attribute is of type `bigint`.

- `created_at`
  Date and time of creation.
  The attribute indicates when the storage record was created.
  The attribute is of type `timestamptz`.

- `updated_at`
  Date and time of the most recent update.
  The attribute specifies the most recent timestamp the storage record was updated.
  The attribute is of type `timestamptz`.

3.3.1.7 Interface



interface		
id	uuid	PK | DEFAULT(UUID())
node_id	uuid	FK | UNIQUE(node_id,name)
name	text	UNIQUE(node_id,name)
address	text	UNIQUE
speed	bigint	
wol	wol_flag [ ]	DEFAULT([ ])
created_at	timestamptz	DEFAULT(NOW())
updated_at	timestamptz	

wol_flag
a
b
g
m
p
s
u

The interface entity depicts the network interface(s) accessible within the cluster's Internal Network that are associated with the node.
The `ip`[146] command/application is used to obtain a node's available network interfaces (`ip -details link show`). However, the returned list contains both logical interfaces (such as `lo`[147]) and physical interfaces (such as `eth0` or `wlan0`), and must be filtered to retrieve only the physical one. The latter action is performed automatically by the installation script that discards all network interfaces that have the `info_kind` information object and is of type `loopback`.
A physical interface is a Network Interface Card (NIC) that is installed on the node and can thus be used to acquire information about the NIC as well as be configured to modify some key parameters (i.e. Wake-on-Lan flags) to function properly in the cluster environment. The information obtained is critical for understanding which nodes are available for automated bootstrapping by the Server (see section 3.3.3) and to which specific physical address (MAC address) the magic wake-up message must be sent.
A node must have at least one NIC to communicate with/from it. The latter verification is done by both the installation script and the Server component when the registration query is executed. Nonetheless, (almost) every system is equipped with at least one NIC and, if possible/needed, can have also extra NICs. It is standard practice in data centers and almost every enterprise environment to have a system with multiple NICs.
The `interface` entity attributes are as follows:

[146] `https://linux.die.net/man/8/ip`
[147] `https://tldp.org/LDP/nag/node66.html`

- `id`
  Uniquely identify (`PK`) an interface record.
  The attribute is of type `UUID`.

- `node_id`
  Node identifier.
  Uniquely identify a node record (`FK`).
  The attribute is of type `UUID`.
  A node can have multiple network interfaces, but each network interface is associated with a single and unique node. As a result, the relation is many (`*`) to one (`1`).

- `name`
  Interface name.
  The network interface name on the associated node. In GNU/Linux, common network interface names for wired network interfaces are `eth0`, `eth1`, and so on, and for wireless network interfaces are `wlan0`, `wlan1`, and so on. The network interface name, like block devices, can be changed by the organization in control of the cluster from the one automatically assigned by the Kernel. The latter procedure, while supported, is strongly discouraged to prevent potential confusion. Furthermore, modifying it necessitates reconfiguring and restarting all applications/programs that are configured to interact with the particular interface (i.e. `SSH`).
  The interface `name` attribute is used in conjunction with the `node_id` attribute to uniquely identify (`UNIQUE(node_id,name)`) an interface record and to avoid two interfaces with the same name from being registered for the same node. Moreover, identifying a NIC by its name for a related node is also helpful to users and administrators.

- `address`
  Interface MAC address.
  The MAC address assigned to the NIC and used for Internal Network communication (see section 2.2.2). There must be no interfaces in the Internal Network with the same MAC address. The latter is reflected in the database schema, which states that the attribute value must be `UNIQUE`, implying that no two interfaces can have the same MAC address. This attribute can be used to uniquely identify an interface record as a result of the latter, and it is known to be particularly helpful when dealing with networking problems in understanding from which node certain packets are sent/lost in the cluster.
  Each database implementation, like the `address` attribute in the node entity, has its own specific/custom datatype for encoding a MAC address, therefore the attribute type is defined as plain text.
  Each database implementation, like the address attribute in the node entity, has its own specific/custom datatype for encoding a MAC address, therefore the attribute type is defined to plain text.
  It should be noted that the NIC's MAC address is assigned by the corresponding manufacturer. Nonetheless, there are software tools, such as `macchanger`[148], that enable the MAC address to be arbitrarily modified with any (valid) value.

- `speed`
  Interface speed in bit-per-second (`b/s`).
  The NIC's speed/performance in bit-per-second. The value is read with the `ethtool` (see section 3.1.1) program by specifying the network interface name and is thus used in conjunction with the `ip` program output. Because the obtained value/unit is not in `b/s` but in a base 10 multiple unit, such as `100 Mb/s` or `1 Gb/s` (common speed values/units for consumer NICs), the installation script converts to `b/s` using the `numfmt`[149] application, which understands the unit and applies the corresponding multiplication value.
  Although some speed values of consumer NICs can fit in an `integer` type, this is not always the

[148]`https://www.kali.org/tools/macchanger`
[149]`https://www.gnu.org/software/coreutils/manual/html_node/numfmt-invocation.html`

case (for example, enterprise NICs, which can easily reach `10 Gb/s` and beyond), therefore the attribute's type has been set to `bigint`.

- `wol`
  Interface Wake-on-Lan (`WoL`) flags supported.
  An interface can be triggered to bootstrap the node by multiple distinct `WoL` activities and this attribute determines which `WoL` flags (each flag mapping to a particular trigger) are supported by the interface.
  The `wol` attribute is an array (`[]`) of `wol_flag` enum because an interface can accept different `WoL` triggers/flags. The array is empty if an interface does not support `WoL`. The latter is reflected in the schema by the default value (`DEFAULT([])`) provided by the database to the attribute.
  The `wol` attribute is intrinsically tied to the `address` attribute because both are used during the Server's (up)scaling procedure (see section 3.3.3). To begin, the `wol` attribute is used to filter inactive nodes to obtain only a list of nodes that can be automatically bootstrapped using `WoL` (search where `wol` is not an empty array). Then, for each node that needs to be bootstrapped (select `n` nodes from the list), the associated MAC address of the interface is used as the destination address of the `WoL` special message. It should be noted that the current implementation only allows basic `WoL`, even though some nodes in the cluster may support advanced `WoL` configurations. Nevertheless, future implementations may support more sophisticated logic and deterministically apply specific configurations based on the interface's supported `WoL` flags.
  The node `wol_flag` enum has the following possible values/constants:

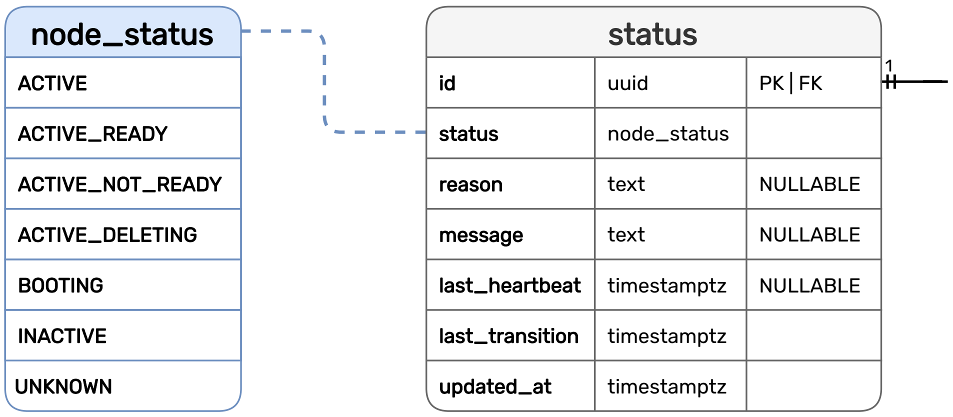
  - `a`
    Wake on `ARP` activity.
  - `b`
    Wake on `Broadcast` activity.
  - `g`
    Wake on `MagicPacket` activity.
  - `m`
    Wake on `Multicast` activity.
  - `p`
    Wake on `PHY` (Physical) activity.
  - `s`
    Enable `SecureOn` password for `MagicPacket`.
  - `u`
    Wake on `Unicast` activity.

  It is worth noting that there is an additional possible value for the `WoL` flags, which is the value `d`, which indicates that `WoL` is disabled on the interface. Because the installation script checks to determine whether or not all `WoL` capable NICs have `WoL` enabled (if not, an error is generated), the `d` value cannot be accepted by the GraphQL API and is thus omitted.
  The `g` flag is the basic configuration for a `WoL` interface, allowing the node to be automatically bootstrapped with `WoL`. As previously stated, the latter is the current (only) supported configuration.

- `created_at`
  Date and time of creation.
  The attribute indicates when the interface record was created.
  The attribute is of type `timestamptz`.

- `updated_at`
  Date and time of the most recent update.
  The attribute specifies the most recent timestamp the interface record was updated.
  The attribute is of type `timestamptz`.

3.3.1.8 Status



The status entity, as the name implies, represents the node's current status, as updated with the most recently available information.
The status entity's information is critical for gaining a better understanding of the node, its condition, and if any potential problems, such as a crash or an anomaly/unprogrammed shutdown, arise. Furthermore, it is widely used in autoscaling procedures (see section 3.3.3 and section 3.3.4). When upscaling, the Server determines what are the list of nodes that are in an inactive status and, vice versa, when downscaling, the Server determines what are the list of nodes that are in an active status. It should be emphasized that the status entity is composed of multiple attributes, not just one, each of which specifies a particular knowledge that can also be correlated to other attributes.
A node must have only one (unique) status record, and a status record must identify only one (unique) node record. When a node successfully registers, the status record is automatically created, and it is continuously updated while the node is operational. Because there are no status information updates sent by the node when it is inactive, all of the attribute's values reflect the most recent status information before the node is shut down.
The status information is automatically updated by leveraging the Kubernetes API, which provides access to the node's status information, which is constantly updated by periodic Heartbeats sent by the Kubernetes node to the API Server (Controllers). The information (attributes) of the status entity is only a subset of the multitude made available by the Kubernetes API, as most of them are unnecessary for low-level cluster operation and administration. Furthermore, some status updates are performed without leveraging the Kubernetes API, either because the latter is unavailable or because the node autonomously sends a request (i.e. after successfully bootstrapping and before starting all cluster services) to specific protected GraphQL queries, generating intrinsic knowledge that the node is somewhat active and performing a status update.
It should be noted that this entity lacks the `created_at` attribute because it is created automatically when the associated node is created, and thus the two timestamps will have the same value.
The `status` entity attributes are as follows:

- `id`
  Uniquely identify (`PK`) a status record.
  The attribute is of type `UUID`.
  A node must have only one status, and a status must be assigned to only one/unique node. As a result, the relation is one (`1`) to one (`1`).
  The `id` attribute is also the Node identifier, which allows a node record to be uniquely identified (`FK`). The `id` and node identifier (`node_id`) attributes are separated in the previous entities to represent the fact that the corresponding node can be associated with multiple records of the entity. In the status entity, however, the `id` attribute serves as both the primary key (`PK`) and the foreign key (`FK`) to a distinct node record. The latter ensures that a status record is unique, easy to identify, and related to a single/unique node record automatically at the database schema level (without the need for Server checks). When a status record is created (during the installation process), its `id` is not derived from a default value (in fact, the `DEFAULT(UUID)` metadata is missing), but it is the same as the associated node. The latter is also very useful for preventing

complex and expensive joins between the two entities because identifying a node's status record requires only its id because they are the same value.

- `status`
  Node status.
  Identify the node's high-level status.
  It is one of the most essential data that correlates to a node, and it is continuously updated when the node is powered on. It can be used by users/administrators to gain a better understanding of the node's current state, such as determining whether or not the node is properly working/healthy. It is widely used in autoscaling procedures in conjunction with other attributes to identify which nodes are inactive (upscaling) or active (downscaling).
  The attribute is of type `node_status` enum. Understanding each status is critical, and the general perspective resembles a finite-state machine machine. Some statuses are only used to transition from one to the next, whereas others designate a permanent status that varies only in response to specific actions.
  The `node_status` enum has the following possible values/constants:

  – `ACTIVE`
    Node is active and healthy, with no Kubernetes orchestrator operating.
    It is a transitory status that is assigned in one of the two situations listed below:

    1. `BOOTING` ➔ `ACTIVE` ➔ `ACTIVE_NOT_READY`
       The node has successfully booted (`BOOTING`) and has sent an update status request to the Server's protected GraphQL API (`ACTIVE`). After receiving a successful response from the Server, the Node begins to initialize all necessary cluster services, such as Node Exporter and K3s, by leveraging the corresponding installed Init System (`ACTIVE_NOT_READY`).
    2. `ACTIVE_DELETING` ➔ `ACTIVE` ➔ `INACTIVE`
       The node has been successfully removed (`ACTIVE_DELETING`) from the Kubernetes cluster (`ACTIVE`), and the Server begins the shutdown procedure (`INACTIVE`).

  – `ACTIVE_READY`
    Node is active, healthy and the Kubernetes orchestrator is operating and accepting pods.
    It is a permanent status that is constantly monitored by the Heartbeat system.
    The status is assigned in the following situation:

    1. `ACTIVE_NOT_READY` ➔ `ACTIVE_READY`
       The Kubernetes orchestrator on the node (`ACTIVE_NOT_ READY`) has finished the initialization and is ready to accept workloads (`ACTIVE_READY`).

    The status is unassigned in the following situations:

    1. `ACTIVE_READY` ➔ `ACTIVE_DELETING`
       The node (`ACTIVE_READY`) should be turned off (`ACTIVE_DELETING`) because the Cluster Autoscaler has monitored it and determined that its overall workload is less than the predefined threshold, or because an administrator has sent a request to the GraphQL API.
    2. `ACTIVE_READY` ➔ `ACTIVE_NOT_READY`
       The node has not sent any Heartbeat messages for more than the predefined duration threshold value. When the threshold time value is reached, the node status changes from `ACTIVE_READY` to `ACTIVE_NOT_READY`, and Kubernetes stops scheduling workloads (pods) on the node.

  – `ACTIVE_NOT_READY`
    Node is active, healthy and the Kubernetes orchestrator is initializing (it cannot accept pods).
    It is a transitory status that is assigned in the following situations:

    1. `ACTIVE` ➔ `ACTIVE_NOT_READY` ➔ `ACTIVE_READY`
       The node has received a successful response from the Server (`ACTIVE`) and is starting all

    cluster-related services via the node's corresponding Init System. When most services start, they are still in a transitory phase known as initialization, in which they are not immediately ready to accomplish their primary purposes (`ACTIVE_NOT_ READY`). After initialization, the services are available to be utilized and fulfill their purposes (`ACTIVE_READY`).
    For example, the Kubernetes orchestrator takes nearly `15` to `30` seconds to initialize, and during this time it is unable to receive any workload from the Controller nodes because all of its components are also initializing.

  2. `ACTIVE_READY` → `ACTIVE_NOT_READY` → `ACTIVE_READY`
     Because the node has not sent any Heartbeat messages for more than the threshold time value, its status is changed from `ACTIVE_READY`, indicating that the node is working and accepting pods, to `ACTIVE_NOT_READY`, indicating that the node is not working/unknown, and Controller nodes cease scheduling pods to the node. If the node resumes sending Heartbeat messages periodically, the status returns to `ACTIVE_READY`. Otherwise, if the node's most recent Heartbeat message has been received for more than a second (longer) threshold, the node status changes to `UNKNOWN` with the attributes `reason` and `message` properly set, indicating that the node is unreachable and thus in error.

– `ACTIVE_DELETING`
  Node is active and healthy and it is being removed from the Kubernetes cluster.
  It is a transitory status that is assigned in the following situation:

  1. `ACTIVE_READY` → `ACTIVE_DELETING` → `ACTIVE`
     After receiving the request to power off the node, the Server instructs the Kubernetes cluster to drain[150] the node (safely remove/evict all pods from the node) so that it can be successfully removed from the Kubernetes cluster (`ACTIVE_DELETING`). After successfully removing the node from the Kubernetes cluster, the associated Kubernetes operator on the deleted node is terminated (`ACTIVE`).
     Since this status is executed because the node needs to be turned off, the Server automatically shuts down the node after the node status is changed to `ACTIVE`. The latter is to indicate that if the `ACTIVE` status is assigned after this status, it will only be effective for a very short time before the node becomes `INACTIVE`.

– `BOOTING`
  Node is booting.
  It is a transitory status that is assigned in the following situation:

  1. `INACTIVE` → `BOOTING` → `ACTIVE`
     The Server has received a GraphQL request from the Cluster Autoscaler or an administrator to turn on a server in the cluster (`INACTIVE`). The Server then sends the associated `Wake-on-Lan` magic message to the corresponding `MAC` address of the node's interface to automatically boot it up (`BOOTING`). After the booting period, which differs depending on the hardware of the node, the node sends the previously mentioned update status message, which changes its status to `ACTIVE`.
     It should be noted that if a node is powered on manually by a user or an administrator using the power button, the `BOOTING` status is omitted in the transition because the Server has no awareness of the action until the node sends the update status request message transitioning from an `INACTIVE` to `ACTIVE` status.

– `INACTIVE`
  Node is inactive (powered off).
  It is a permanent status.
  The status is assigned in the following situation:

  1. `ACTIVE` → `INACTIVE`
     The node is in `ACTIVE` status, and the Server terminated it remotely via SSH (`INACTIVE`).

[150] `https://kubernetes.io/docs/tasks/administer-cluster/safely-drain-node`

It should be noted that the latter can also be executed manually by a user/administrator using the power button, but the Server does not know anything about the current status because there is no predefined transition between the various statuses, and thus, after a period of time is set to an error status (`UNKNOWN`).

The status is unassigned in the following situation:

1. `INACTIVE` ➜ `BOOTING`
   The node is in the `INACTIVE` status until the Server (via `Wake-on-Lan`) or a user/administrator (via the power button) turns it on (`BOOTING`).
   As previously stated, if the node is started using the power button, the status transitions directly from `INACTIVE` to `ACTIVE`, skipping the `BOOTING` status because the Server does not know anything about the manual operation.

– `UNKNOWN`
  Unknown/Error status.
  This status is special in that it identifies a node that is in an error condition due to an issue that must be resolved.
  All statuses can transition to (error discovered) and from (error resolved) this status.
  For example, when this status is set for a node, it means that the Kubernetes orchestrator has crashed and no Heartbeats have been received by the Controllers for a specified amount of time.

Figure 3.3.1.8 depicts all of the possible statuses and the transitions between them. Take note of the manual transition from `INACTIVE` to `ACTIVE`, as well as the bidirectional transition between `ACTIVE_NOT_READY` and `ACTIVE_READY`.

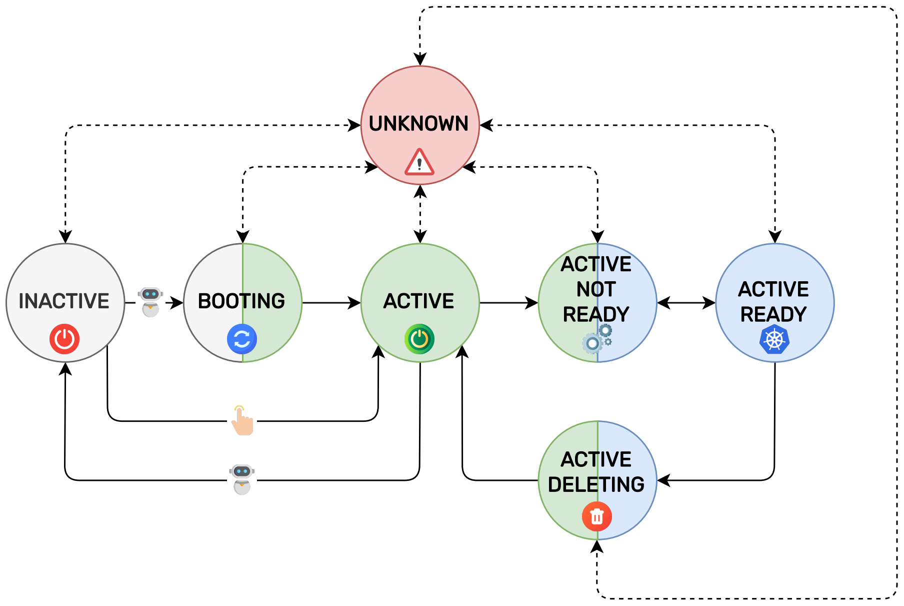


Figure 3.19: Statuses and their transitions

- `reason`
  Status' reason.
  Determine briefly why the `status` attribute has the corresponding value. The `reason` attribute value is typically one word or a sequence of very few words that briefly describe the cause for the node's `status` attribute value. In transitional statuses (i.e. no persistent) that transition from one status to another, the value of this attribute can be assigned with various and distinct values, defining the logical path to the final status. The latter is particularly obvious during the Kubernetes initialization (from `ACTIVE` to `ACTIVE_READY`), which continues to change the attribute's value to represent the fact that the various Kubernetes components are starting.
  For example, if the node's status value is `ACTIVE_READY` and the Kubernetes process is running smoothly, the value of this attribute is set to `KubeletReady`, indicating that the Kubernetes

kubelet[151] component is functioning properly.
Because the reason why the status has the corresponding value is not always known, this attribute can take a NULL value (NULLABLE), indicating that the reason is undetermined. This is prevalent if the status value is UNKNOWN and the cause of what happened is obscure.

- message
  Status' message.
  Describes why the status/reason attributes have the corresponding value using a longer string. This attribute is intrinsically tied to the other two attributes; if the value of one of them changes, this attribute will probably be updated as well. It can be considered an extended description message for the reason value, and it is particularly useful for users and administrators who want to understand the current node's status. If the node status is set to UNKNOWN and the cause is known, the latter can be extremely beneficial: the reason attribute describes the issue in brief with a high-level perspective, whereas the message attribute is more detailed in explaining the cause of the problem with a low-level perspective. As for the reason attribute, the value of the message attribute during transitional statuses can vary arbitrarily depending on the logical path between the two statuses, whereas it remains nearly constant during persistent statuses.
  For example, if the node's status is ACTIVE_READY and the Kubernetes process is functioning properly, this attribute's value is set to kubelet is posting ready status, indicating that the Kubernetes kubelet is accepting pods (working) and sending periodic Heartbeat messages to the corresponding Controller node. The combination of the reason and message attributes indicate that the Kubernetes node is operating properly and why.
  Because the content of the message attribute is not always known, it can accept NULL values (NULLABLE). If the value of the reason attribute is NULL, it is almost certain that the value of the message attribute is also NULL.

- last_heartbeat
  Date and time of the most recent Heartbeat message[152].
  The attribute indicates the most recent timestamp at which a Controller node received a HeartBeat message sent by the associated node (Kubernetes kubelet). The attribute is of type timestamptz.
  A HeartBeat message is sent whenever the node's Kubernetes status changes or there is no update in the defined HeartBeat interval. During the initialization phase, Kubernetes bootstraps numerous components, and the reason and message attributes are constantly changed, resulting in frequent updates to the last_heartbeat attribute. Far from it, when the node is finally ready/working and accepting pods (ACTIVE_READY status), the frequency of Heartbeat messages is much slower, but constant, and follows the specified interval, defined as node-status-update-frequency. If the node's Heartbeat messages are not received after a second time threshold, defined as node-monitor-grace-period and higher than the node-status-update-frequency, the node status is changed from ACTIVE_READY to ACTIVE_NOT_READY, and the Controller cease assigning workload (pods) to the node. If the node becomes reachable again and resumes transmitting periodic Heartbeat messages, the node status is immediately revert to ACTIVE_READY, and the Controller node resume scheduling pods to the node. If no Heartbeat messages are received after a third time threshold, defined as pod-eviction-timeout and higher than node-monitor-grace-period, the Controller begins evicting[154] all deployments (pods) that are scheduled on the node and the node's status is changed to UNKNOWN. Note that the pods scheduled on the node may still operate, and because the Controller is unable to communicate with the disconnected node, the pods status is also set to Terminating or Unknown.
  To summarize, the previously mentioned threshold time values are outlined below (along with their respective default value):

[151] https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet
[152][153]
[154] https://kubernetes.io/docs/concepts/scheduling-eviction/api-eviction

1. `node-status-update-frequency`[155]
   The frequency at which the `kubelet` sends node status (heartbeat message) to the Controller.
   The default value is `10` seconds (`10s`).
2. `node-monitor-grace-period`[156]
   Amount of time after which an operating node is set to be unresponsive, indicating that it is unhealthy. Must be `N` times greater than the `node-status-update-frequency` of the node's `kubelet`, where `N` is the number of retries.
   The default value is `40` seconds (`40s`).
3. `pod-eviction-timeout`[157]
   The duration after which the pods scheduled on the unresponsive node begin to be removed.
   The default value is `300` seconds (`5m`).

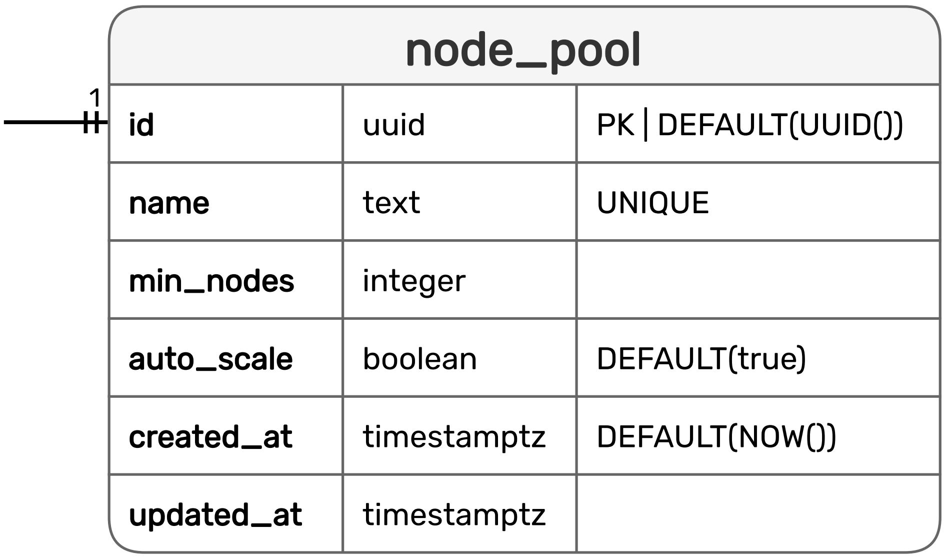
It should be noted that the latter threshold values can be tailored to cluster-specific use-case scenarios, yielding approximately three configurations[158]: `Fast Update/Fast Reaction`, `Medium Update/Average Reaction` and `Low Update/Slow reaction`.
Because the reception of the most recent Heartbeat message is not always known, this attribute can be `NULL` (`NULLABLE`). Furthermore, the value is `NULL` when the `status` is `INACTIVE`.

- `last_transition`
  Date and time of the most recent status transition.
  This attribute indicates the last time the `status` attribute has been changed. It is very helpful for users and administrators to understand how long the node has been in the current status.
  The attribute is of type `timestamptz`.
  The value cannot be `NULL` because the status attribute cannot be `NULL` (and thus it is known when the most recent transition occurs) and also because the Server is in charge of updating this attribute. Therefore, the Server is the only component with both low-level knowledge (GraphQL API requests sent by the node itself) and high-level knowledge (Kubernetes Heartbeats).
  For example, if the node's status attribute is set to `ACTIVE_READY` and the difference between the current timestamp and the value of the last transition attribute is `48` hours, which implies that the node has been participating in cluster operations for two consecutive days. The same logic can be applied to determine how long the node is powered off (`INACTIVE` status) and thus unused.

- `updated_at`
  Date and time of the most recent update.
  The attribute specifies the most recent timestamp the status record was updated.
  The attribute is of type `timestamptz`.

[155] `https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet/#node-status-update-frequency`
[156] `https://kubernetes.io/docs/reference/command-line-tools-reference/kube-controller-manager/#node-monitor-grace-period`
[157] `https://kubernetes.io/docs/reference/command-line-tools-reference/kube-controller-manager/#pod-eviction-timeout`
[158] `https://github.com/kubernetes-sigs/kubespray/blob/master/docs/kubernetes-reliability.md`

3.3.1.9 Node Pool



node_pool		
id	uuid	PK | DEFAULT(UUID())
name	text	UNIQUE
min_nodes	integer	
auto_scale	boolean	DEFAULT(true)
created_at	timestamptz	DEFAULT(NOW())
updated_at	timestamptz	

The node pool entity represents a collection of nodes with similar hardware characteristics. This entity is critical for autoscaling, and the Cluster Autoscaler component makes extensive use of the available registered node groups, their respective information, and the corresponding nodes. Indeed, the Cluster Autoscaler relies on both high-level knowledge obtained from the Kubernetes API to continuously monitor the nodes and all scheduled workloads across the cluster, and low-level knowledge obtained from this entity (via the GraphQL API) to determine whether there are node pools with inactive nodes that can be booted up (upscaling) or nodes that can be turned off (downscaling) if some nodes are underutilized.
The current implementation combines nodes based on their respective number of CPU cores and memory (RAM) amount, which is also reflected in the `name` attribute. It should be noted that in future implementations, the GPU model may also be used. Because the number of CPU cores is relatively small, dealing with comparisons is relatively simple. However, for two primary causes, the latter is far from true for memory quantity. Firstly, the memory quantity unit is in byte (`B`), the data type is `bigint`, and therefore comparisons are performed on extremely large integers. Secondly, even if two nodes are from the same manufacturer and have the same hardware components, it is not always guaranteed that the memory quantity in bytes is identical. The latter has happened in reCluster, where the four Worker nodes are the same model from the same manufacturer and equipped with the same hardware, but their memory quantities in bytes vary slightly. One potential explanation is that some memory cells are faulty (this is more likely with older hardware, on which reCluster is designed), and the memory controller hides them from the Operating System, causing the total to be slightly different. To address the latter issues, the memory amount is first converted to GigaBytes (`GiB`) and then rounded to the closest half, yielding `0.5` or a multiple of it. It is now simple to organize the nodes in groups. It should be noted that the `512 MiB` (`0.5 GiB`) value is not random, but rather reflects the minimal hardware requirements of the Kubernetes Orchestrator (K3s). As an example, consider the following four nodes: the first has a `4` core CPU and `4 GiB` of memory, the second has an `8` core CPU and `7.5 GiB` of memory, the third has a `4` core CPU and `4 GiB` of memory, and the fourth has an `8` core CPU and `8 GiB` of memory. The first and third nodes are assigned to the same node pool, whereas the second and fourth nodes are assigned to separate node pools.
The System has two logical node pools: one for Controller nodes and one for Worker nodes. Worker node pools function as previously explained, and they're employed for autoscaling by default. While all Controller nodes are merged into a single node pool called `controllers`, no hardware components are examined, and auto-scaling is disabled. The latter is done because, by design, the cluster cannot auto-scale Controller nodes, leaving hardware comparisons meaningless. If a controller node needs to be converted to a Worker node, it must first be deleted by the Kubernetes cluster, then removed by the Server/Database (via a single GraphQL API call made by an administrator), and finally re-registered in the cluster as a Worker node.
The node must be assigned to a node pool during the registration process. If the node pool does not

exist, it is created automatically; otherwise, the node is assigned to the appropriate node pool with the same amount of CPU cores and transformed memory quantity.
The node pool entity has the attributes `min_nodes` and `auto_scale`, which can be modified by an administrator via a GraphQL API node pool update request to change the associated node pool's autoscaling behavior.
As previously stated, a node is assigned to a single and unique node pool, but a node pool can be assigned to zero or multiple nodes.
The `node_pool` entity attributes are as follows:

- `id`
  Uniquely identify (`PK`) a node pool record.
  The attribute is of type `UUID`.
  A node must be assigned to a single and unique node pool, and a node pool can have zero or more nodes assigned to it. As a result, the relation is one(`1`) to many(`*`).

- `name`
  Node pool's name.
  A string of characters that uniquely (UNIQUE) identifies a node pool record.
  The `name` and `id` attributes are conceptually interchangeable because they both uniquely identify the same node pool record. Furthermore, identical to the user entity's `username` attribute, the `name` attribute is mostly used for users and administrators to easily and quickly identify a node pool record without having to remember a 128-bit alphanumerical string (`UUID`).
  When the Server needs to create an additional node pool, it automatically assigns a value to the `name` attribute based on the node type, the number of CPU cores, and the transformed memory quantity value. If the node pool is dedicated only to Controller nodes, the name is set to `controllers` without any further information (this can be changed in the Server configuration). However, the name of a node pool dedicated to Worker node(s) is a combination of the two hardware values separated by a dot (`.`). It should be noted that the latter can be easily modified in the Server implementation with different values/logic.
  For example, `cpu8.memory7.5` is the name of a Worker nodes pool that combines nodes with `8` CPU cores and `7.5 GiB` of memory.

- `min_nodes`
  Minimum number of active (`ACTIVE_READY` status) nodes in the node pool.
  The attribute specifies a minimal number (inclusive) of active nodes with the `ACTIVE_READY` status in the node pool.
  The Cluster Autoscaler uses this information to determine the minimal number of nodes in the node pool that must always be available in the Kubernetes cluster. Until the number of active nodes in the node pool equals the value of the `min_nodes` attribute, the Cluster Autoscaler is allowed to downscale the nodes in the node pool.
  The architecture is particularly focused on minimizing resource loss and energy usage. Therefore, to enable the Cluster Autoscaler to completely downscale all Worker nodes (or particular node pools), all node pools that are dedicated to Worker nodes by default have the value of this attribute set to `0`. Contrarily, the Cluster Autoscaler automatically disables autoscaling procedures in the node pool reserved for controller nodes (`controllers`), where the number of the `min_nodes` attribute is set to the total number of nodes assigned to the node pool. Since there are no active Worker nodes when the cluster's overall workload is extremely low or nonexistent, with just one or a very small number of Controller Nodes requiring minimal resources, this enables the least amount of power consumption and resource loss.
  Administrators can modify this attribute based on the cluster's requirements and/or increase the cluster's minimal capacity in preparation for a high volume of requests. However, whenever the attribute is changed, it is verified that the new value is not less than `0` or greater than the total number of nodes assigned to the node pool. The two latter checks prevent a node pool from having the minimum number of nodes below `0` (impossible) and also ensure that it does not exceed the node pool's maximum allowed capacity, as any autoscaling capabilities or procedures

are automatically disabled when a value is greater than the number of nodes assigned to the node pool.
It should be noted that the additional knowledge of the number of active nodes (`count`) and the maximum number of nodes (`max_nodes`) available in the node pool are calculable values that reflect a higher degree of knowledge concerning the Database schema and are, as a result, calculated by the Server and made available via internal services and the GraphQL API. The two values are also used by the Cluster Autoscaler to determine the number of active nodes currently present in the node pool as well as the highest number of nodes that can be provided for upscaling.

- `auto_scale`
  A Boolean flag that indicates whether or not the node pool can be employed for autoscaling.
  The attribute is of type `boolean`, where `true` indicates that the node pool can be used for autoscaling and `false` indicates that it cannot be handled for autoscaling and all nodes assigned to the node pool must be ignored.
  For all node pools that identify Worker nodes, the default value (`DEFAULT`) is `true`, so the Server omits the attribute whenever it needs to create a new node pool for Worker nodes. The flag is set to `false` by the Server during the initial creation of the node pool designated to Controller nodes since any autoscaling procedures must ignore it.
  When combined with the `min_nodes` attribute, this attribute prevents the node pool designated for Controller nodes from autoscaling. Additionally, administrators have the same flexibility in changing the flag temporarily as with the prior attribute to meet specific cluster requirements.

- `created_at`
  Date and time of creation.
  The attribute indicates when the node pool record was created.
  The attribute is of type `timestamptz`.

- `updated_at`
  Date and time of the most recent update.
  The attribute specifies the most recent timestamp the node pool record was updated.
  The attribute is of type `timestamptz`.

3.3.2 GraphQL API

This description shows the GraphQL API made available by the Server, which enables users and nodes to interact with both the high-level cluster, which represents the Kubernetes cluster and its overall workload, and the low-level cluster, which represents the bare-metal knowledge stored inside the Database. As previously mentioned, the API can provide more advanced capabilities and calculated data/attributes that are not present in the pure Database schema, such as the `count` and `max_nodes` attributes of the Node Pool entity.

GraphQL[159] is a query language and server-side runtime for running queries using a type system specified in a schema. A GraphQL service is built by specifying types and fields on those types, then providing methods (resolvers) for each field on each type. Before any operation is made, it is first verified that it only pertains to the types and variables specified in the schema. Declarative data fetching with GraphQL allows clients to define precisely which data are needed, eliminating over- and under-fetching. Additionally, a GraphQL server only provides a single API endpoint rather than multiple endpoints that yield fixed data structures, as is the case with `REST` (REpresentational State Transfer). The GraphQL endpoint in the Server implementation is accessible at `/graphql`, but it can be modified to better suit the requirements of the cluster's organization. It requires considerable effort/time to understand how GraphQL operates and how its overall structure/architecture is composed of all the various types, fields, queries, mutations, etc. The latter, however, is outside the scope of this document, so it is

Source: `https://graphql.org/brand`

Figure 3.20: GraphQL logo

[159] `https://graphql.org`

strongly advised to visit the official website at `https://graphql.org` for a better comprehension of how GraphQL works.
It is nearly unfeasible to provide a detailed explanation of the *generated* GraphQL schema for the Server implementation because it consists of almost 2000 lines of code. Therefore, the explanations in this section (hence the titles of the two subsections) are limited to the queries and mutations that are exposed by the GraphQL API and the majority of them can be considered as an extension of the knowledge that is contained in the database schema that was previously discussed. Additionally, the word *generated* at the beginning of this paragraph was not written at random but rather to highlight the fact that the GraphQL API is implemented using a code-first approach rather than a schema-first approach, establishing a single source of truth by defining the schema using classes and decorators.
The majority of GraphQL queries and mutations provided by the Server implementation do not require any authentication or authorization, allowing anyone to utilize them. Far from it; the remaining queries and mutations are protected by a customized GraphQL Auth Directive (`@auth`) that prevents unauthenticated and/or unauthorized accesses. Since the GraphQL Auth Directive is a standard feature that can be used and shared by other GraphQL APIs and projects, it is not restricted to cluster implementation and enables protection for all possible GraphQL APIs. To make the GraphQL Auth Directive accessible to the public as a library that can be easily integrated with other projects, it was developed as a side project alongside the cluster implementation. Attachment A.3 includes an in-depth overview correlated with examples of the GraphQL Auth Directive. Compared to the implementation of the latter directive, the directive used in the cluster implementation only differs by one small detail: `type`, an extra argument used to identify the type of the entity making the request (`USER` or `NODE`). Requests are rejected if the entity sending them does not match the one listed in the directive. The value `USER` is assigned by default if the `type` parameter is missing.
A personal security token that is generated and provided by the Server for either a Node or a User is required from the entity performing the request to access a protected GraphQL query or mutation. When a Node registers to the cluster for the first time, it automatically obtains the token and is unable to obtain a new one ever again. A User is more flexible and can obtain a new token whenever it is successfully authenticated using the appropriate GraphQL mutation (`signIn`). The token implementation and management rely entirely on JSON Web Tokens[160] (JWT). JWT is an open standard, which specifies a compact and self-contained method for safely transferring data between parties as a JSON (JavaScript Object Notation) object. Because it is digitally signed with a secret (using the HMAC algorithm) or a public/private key pair (using RSA or ECDSA), this information can be verified and trusted[6]. The JWT token is digitally signed by `RS256` in the Server implementation, but this can be easily modified in the configuration settings with the appropriate public/private key pair files and corresponding encryption algorithm. The content of the token comprises 4 pieces of encoded information in addition to the JWT's data. The first of these is the `type`, which specifies the type of the entity (`USER` or `NODE`), the second is the `id`, which is used to uniquely identify the entity in queries and mutations implementation, and the final two are the `roles` and `permissions` of the entity, which are used in combination with the `type` to ensure protection. The token must be provided in the request's `Authorization` header using the `Bearer` authentication scheme[161]: `Authorization: Bearer <TOKEN>`. An error response is generated if the entity attempts to access a protected API but the token is either absent, in an invalid format, or the verification procedure fails.
It should be noted that in the implementation, all GraphQL queries/mutations that return a list (`[]`) support pagination, which includes both offset pagination and cursor-based pagination, via the arguments `cursor`, `skip` (default value set to `0`), and `take` (default value set to `8`). This allows the entities to divide the returned resources into manageable chunks improving the overall search experience and the overall Server performance. Additionally, the latter queries/mutations support advanced sorting, via the `orderBy` argument, and filtering, via the `where` argument. As previously stated, the GraphQL API is built on top of the database and its schema, so the API can be considered an extension of the database in terms of both the parameters that are accepted and the resources/objects that are returned.

[160] `https://jwt.io`
[161] `https://www.rfc-editor.org/rfc/rfc6750`

3.3.2.1 Queries

GraphQL queries (`Query`[162]) are used to query/fetch data without any server-side modification. `Query` fields can be executed in parallel by the GraphQL engine.
A GraphQL `Query` is the logical equivalent of a `GET` request in `REST`.
All GraphQL `Query`(s) supported by the Server implementation are listed below:

- `cpu(id: ID!): Cpu`
  `cpu` query returns the `Cpu`[163] object that matches the specified identifier (`id` argument).
  Because of the `!` symbol, the `id` argument cannot be `null` and is thus required. Whereas the returned `Cpu` object is nullable (there is no `!` symbol) implying that the query returns `null` if no CPU matches the provided identifier.

-
  ```
  cpus(cursor: ID, orderBy: OrderByCpuInput, skip: NonNegativeInt! = 0, take: Int! = 8,
       where: WhereCpuInput): [Cpu!]!
  ```
  `cpus` query returns a non-nullable list (`[]!`) of non-nullable `Cpu` objects.
  The query allows for pagination (`cursor`, `skip`, and `take` arguments), filtering (`where` argument of type `WhereCpuInput`[164]), and sorting (`orderBy` argument of type `OrderByCpuInput`[165]). It is worth noting that the argument `skip` is of the type `NonNegativeInt`, indicating that it can accept values of `0` (inclusive) or greater and that it is required (`!` symbol), but because it has a default value of `0`, it can be omitted in the request. The `take` argument is similar, but it can accept values less than, equal to, or greater than `0` and has a default value of `8`. The latter arguments and requirements are present in all queries that return a list, with the only variation being the type of the associated `where` and `orderBy` arguments.

- `interface(address: MAC, id: ID): Interface`
  `interface` query returns the `Interface`[166] object that matches the specified identifier (`id` argument) or MAC address (`mac` argument).
  Because they both uniquely designate an `Interface` object, both arguments are nullable. If the query is executed with both arguments set to `null`, an error is returned because the corresponding resolver function is unable to uniquely identify the record.

-
  ```
  interfaces(cursor: ID, orderBy: OrderByInterfaceInput, skip: NonNegativeInt! = 0, take: Int! = 8,
             where: WhereInterfaceInput): [Interface!]!
  ```
  `interfaces` query returns a non-nullable list of non-nullable `Interface` objects.
  The `where` argument is of type `WhereInterfaceInput`[167], and the `orderBy` argument is of type `OrderByInterfaceInput`[168].

- `node(id: ID, address: IP, name: String): Node`
  `node` query returns the `Node`[169] object that matches the specified identifier (`id` argument), IP address (`ip` argument) or name (`name` argument).

-
  ```
  nodes(cursor: ID, orderBy: OrderByNodeInput, skip: NonNegativeInt! = 0, take: Int! = 8,
        where: WhereNodeInput): [Node!]!
  ```
  `nodes` query returns a non-nullable list of non-nullable `Node` objects.

[162]http://spec.graphql.org/draft/#sec-Query
[163]https://github.com/carlocorradini/reCluster/blob/main/server/src/graphql/entities/Cpu.ts
[164]https://github.com/carlocorradini/reCluster/blob/main/server/src/graphql/inputs/where/WhereCpuInput.ts
[165]https://github.com/carlocorradini/reCluster/blob/main/server/src/graphql/inputs/orderby/OrderByCpuInput.ts
[166]https://github.com/carlocorradini/reCluster/blob/main/server/src/graphql/entities/Interface.ts
[167]https://github.com/carlocorradini/reCluster/blob/main/server/src/graphql/inputs/where/WhereInterfaceInput.ts
[168]https://github.com/carlocorradini/reCluster/blob/main/server/src/graphql/inputs/orderby/OrderByInterfaceInput.ts
[169]https://github.com/carlocorradini/reCluster/blob/main/server/src/graphql/entities/Node.ts

The `where` argument is of type `WhereNodeInput`[170], and the `orderBy` argument is of type `OrderByNodeInput`[171].

- `nodePool(id: ID, name: String): NodePool`
  `nodePool` query returns the `NodePool`[172] object that matches the specified identifier (`id` argument) or name (`name` argument).
  As mentioned in section 3.3.1.9, the `Node Pool` object in the GraphQL API adds two additional attributes/fields to the basic knowledge provided by the Database schema. The first field, `max_nodes`, defines the total number of nodes that the node pool can handle. The field has the same value as the number of nodes that are associated with the corresponding node group. The second field, `count`, defines the amount of nodes that are associated with the respective node pool, have the `node_pool_assigned` attribute set to `true`, and are operational (active status). The latter field is used to determine the amount of Kubernetes nodes of the corresponding node pool that are operational and actively participating in the cluster. Both of these attributes are critical for autoscaling purposes and are extensively used by the Cluster Autoscaler component.

- `nodePools(cursor: ID, orderBy: OrderByNodePoolInput, skip: NonNegativeInt! = 0, take: Int! = 8,`
  `where: WhereNodePoolInput)): [NodePool!]!`
  `nodePools` query returns a non-nullable list of non-nullable NodePool objects.
  The `where` argument is of type `WhereNodePoolInput`[173], and the `orderBy` argument is of type `OrderByNodePoolInput`[174].

- `status(id: ID!): Status`
  `status` query returns the `Status`[175] object that matches the specified identifier (`id` argument).

- `statuses(cursor: ID, orderBy: OrderByStatusInput, skip: NonNegativeInt! = 0, take: Int! = 8,`
  `where: WhereStatusInput): [Status!]!`
  `statuses` query returns a non-nullable list of non-nullable `Status` objects.
  The `where` argument is of type `WhereStatusInput`[176], and the `orderBy` argument is of type `OrderByStatusInput`[177].

- `storage(id: ID!): Storage`
  `storage` query returns the `Storage`[178] object that matches the specified identifier (`id` argument).

- `storages(cursor: ID, orderBy: OrderByStorageInput, skip: NonNegativeInt! = 0, take: Int! = 8,`
  `where: WhereStorageInput): [Storage!]!`
  `storages` query returns a non-nullable list of non-nullable `Storage` objects.
  The `where` argument is of type `WhereStorageInput`[179], and the `orderBy` argument is of type `OrderByStorageInput`[180].

[170] https://github.com/carlocorradini/reCluster/blob/main/server/src/graphql/inputs/where/WhereNodeInput.ts
[171] https://github.com/carlocorradini/reCluster/blob/main/server/src/graphql/inputs/orderby/OrderByNodeInput.ts
[172] https://github.com/carlocorradini/reCluster/blob/main/server/src/graphql/entities/NodePool.ts
[173] https://github.com/carlocorradini/reCluster/blob/main/server/src/graphql/inputs/where/WhereNodePoolInput.ts
[174] https://github.com/carlocorradini/reCluster/blob/main/server/src/graphql/inputs/orderby/OrderByNodePoolInput.ts
[175] https://github.com/carlocorradini/reCluster/blob/main/server/src/graphql/entities/Status.ts
[176] https://github.com/carlocorradini/reCluster/blob/main/server/src/graphql/inputs/where/WhereStatusInput.ts
[177] https://github.com/carlocorradini/reCluster/blob/main/server/src/graphql/inputs/orderby/OrderByStatusInput.ts
[178] https://github.com/carlocorradini/reCluster/blob/main/server/src/graphql/entities/Storage.ts
[179] https://github.com/carlocorradini/reCluster/blob/main/server/src/graphql/inputs/where/WhereStorageInput.ts
[180] https://github.com/carlocorradini/reCluster/blob/main/server/src/graphql/inputs/orderby/OrderByStorageInput.ts

- `user(id: ID, username: NonEmptyString): User`
  `user` query returns the `User`[181] object that matches the specified identifier (`id` argument) or username (`username` argument).

- `users(cursor: ID, orderBy: OrderByUserInput, skip: NonNegativeInt! = 0, take: Int! = 8,`
  `        where: WhereUserInput): [User!]!`
  `users` query returns a non-nullable list of non-nullable `User` objects.
  The `where` argument is of type `WhereUserInput`[182], and the `orderBy` argument is of type `OrderByUserInput`[183].

3.3.2.2 Mutations

GraphQL mutations (`Mutation`[184]) are used for operations that modify any server-side data. `Mutation` top-level fields are executed in serial by the GraphQL engine.
A GraphQL `Mutation` is the logical equivalent of a `POST`, `PUT`, `PATCH`, or `DELETE` request in `REST`.
All GraphQL `Mutation`(s) supported by the Server implementation are listed below:

- `createNode(data: CreateNodeInput!): JWT!`
  `createNode` resolver registers a new `Node` object in the Database using the data provided in the `data` input argument (of type `CreateNodeInput`[185]) and returns a `JWT` security token if successful.
  The associated Node will use the returned `JWT` token for any future operations involving any of the protected GraphQL APIs.
  It should be noted that if the registration fails for any reason, no `JWT` token is returned and an error is generated.
  Because neither the `data` argument nor the returned `JWT` token can be `null`, a value must/is always supplied.

- `createUser(data: CreateUserInput!): User!`
  `createUser` resolver registers a new `User` object in the Database using the data provided in the `data` input argument (of type `CreateUserInput`[186]) and returns the created `User` object if successful.
  The returned `User` object includes extra fields related to the given data, such as the generated `id` and designated `roles`/`permissions` fields.
  The plain text `password` is hashed using the `bcrypt` password-hashing function before being saved in the Database. Furthermore, there is no `password` field mapped in the GraphQL schema that is assigned to the `User` object, so it cannot be retrieved via any GraphQL queries or mutations. The latter is done as a security precaution to prevent sensitive information from leaking outside of the Server.
  The difference between this mutation and the previous one is that no `JWT` security token is returned because the user must use the specific mutation `signIn` to successfully authenticate and acquire the `JWT` security token.
  Because neither the `data` argument nor the returned `User` object can be `null`, a value must/is always supplied.

- `signIn(username: NonEmptyString!, password: NonEmptyString!): JWT!`
  `signIn` resolver requires a user's `username` and `password` to authenticate it. A `JWT` security

[181] https://github.com/carlocorradini/reCluster/blob/main/server/src/graphql/entities/User.ts
[182] https://github.com/carlocorradini/reCluster/blob/main/server/src/graphql/inputs/where/WhereUserInput.ts
[183] https://github.com/carlocorradini/reCluster/blob/main/server/src/graphql/inputs/orderby/OrderByUserInput.ts
[184] http://spec.graphql.org/draft/#sec-Mutation
[185] https://github.com/carlocorradini/reCluster/blob/main/server/src/graphql/inputs/create/CreateNodeInput.ts
[186] https://github.com/carlocorradini/reCluster/blob/main/server/src/graphql/inputs/create/CreateUserInput.ts

token identifying the user entity is returned if the authentication procedure succeeded.
The `username` argument is used to uniquely identify a `User` record to acquire the associated `password` hashed using the `bcrypt` password-hashing algorithm during the registration procedure. A `bcrypt` function compares the provided plain text `password` argument with the saved hashed `password` and returns `true` if the password matches and `false` otherwise.
If the authenticated procedure fails due to an invalid `username` and/or `password`, the mutation's error provides the generic message `"Username or password is incorrect"` to prevent possible information leakage that an attacker could leverage.
It should be noted that the current implementation lacks an advanced token system for withdrawing tokens or verifying the current available active sessions. The latter may be employed in future implementations that depend on a more advanced architecture.

- `unassignNode(id: ID!): Node! @auth(type: USER, roles: [ADMIN])`
  `unassignNode` mutation unassign and deactivate the node identified by the `id` attribute.
  The mutation is widely used in downscaling procedures to downscale the cluster's specific node, which is identified by the `id` argument. The procedure drains the Kubernetes node first, then deletes it from Kubernetes, and when the Server monitoring receives confirmation of the deletion, it remotely turns off the node via `SSH`. Section 3.3.4 explains the latter procedure in better detail.
  An equivalent mutation for upscaling procedures does not exist because the Cluster Autoscaler only has high-level knowledge of the cluster and thus only knows about the current Kubernetes cluster and its corresponding active nodes, rather than all available nodes and their respective low-level information (i.e. identifiers, statuses and more).
  The Auth directive (`@auth`) is used to protect the mutation from unauthenticated and unauthorized access. It specifies that only an entity of type `USER` with the `ADMIN` role is authorized to unassign and turn off a node (as indicated by the directive arguments).

- `updateNodePool(id: ID!, data: UpdateNodePoolInput!): NodePool! @auth(type: USER, roles: [ADMIN])`
  `updateNodePool` mutation is used to update the node pool with the `data` argument (of type `UpdateNodePoolInput`[187]) identified by the `id` argument.
  It should be noted that the `data` argument contains the `count` field that is not available in the Database and is thus considered marker information. The cluster is autoscaled whenever the `count` field is not `null` and the difference between it and the current count is not zero. If the specified `count` is less than the current `count`, the cluster must be downscaled, and the difference indicates how many nodes in the corresponding node pool must be downscaled (deleted from Kubernetes and powered off remotely). The downscale procedure is the same as the one outlined in the previous mutation, but the nodes that must be downscaled are selected with the strict policy of minimizing total cluster power consumption. As a result, the selected nodes are those that consume more energy than the others that are currently operational in the Kubernetes cluster. If the specified `count` is greater than the current `count`, the cluster must be upscaled, and the difference indicates how many nodes in the corresponding node pool must be upscaled (remotely powered on). The list of nodes that need to be upscaled is generated by searching for nodes that are currently in the `INACTIVE` status and have the appropriate interface with `Wake-on-Lan` support (`wol` attribute array non-empty) to allow remote bootstrapping. Then, the energy-saving policy is applied again, and the list is sorted in ascending order of power consumption: nodes at the beginning of the list are more power-efficient, while nodes at the end are less power-efficient. Starting from the beginning of the list, the Server selects the same number of nodes as the difference between the two count values (is always positive since it is an upscale procedure), and for each node, the Server sends a Wake-on-Lan message on the associated interfaces' MAC address, remotely bootstrapping it. Section 3.3.3 explains the latter procedure in better detail.
  If the specified `count` field is less than zero or higher than the total number of nodes associated

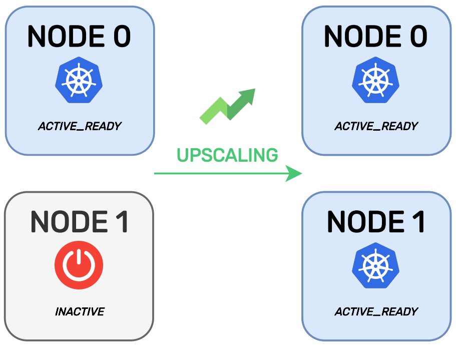
[187] `https://github.com/carlocorradini/reCluster/blob/main/server/src/graphql/inputs/update/UpdateNodePoolInput.ts`

with the corresponding node pool (`max_nodes`), an error is returned because the value provided is incorrect. Furthermore, if the identified node pool has disabled the autoscaling feature by having the `auto_scale` flag attribute set to `false`, an error is returned indicating that the operation is not allowed for the given node pool.
Identical to the previous one, this mutation is protected by the Auth directive (`@auth`), which allows only a `USER` entity with the `ADMIN` role to perform the action.

- `updateStatus(data: UpdateStatusInput!): Status! @auth(type: NODE)`
  `updateStatus` mutation updates the Status with the `data` argument (of type `UpdateStatusInput`[188]) of the corresponding Node identified in the `JWT` security token provided. If the update procedure is successful, the updated `Node` object is returned.
  Because the mutation is protected by the Auth directive (`@auth`), and a `JWT` token (of a `NODE` entity type) must always be provided to perform the mutation, there is intrinsic knowledge of who is requesting the update. The `JWT` token encodes the Node identifier (`id` field), which is used to uniquely identify the status record that necessitates the update with the provided `data` argument. As stated in section 3.3.1.8, the Status record identifier and the Node identifier are the same value because the Status entity's `id` attribute is both a Primary Key (`PK`) and a Foreign Key (`PK`) that is uniquely related to the corresponding Node identifier (`id` field). If the two IDs are different, a join operation between the Node entity and the Status entity is required, necessitating of more resources and increasing the total execution time of the mutation.

- `updateUser(data: UpdateUserInput!): User! @auth(type: USER)`
  `updateUser` mutation updates the User with the `data` argument (of type `UpdateUserInput`[189]) of the corresponding User identified in the `JWT` security token provided. If the update procedure is successful, the updated `User` object is returned.
  This mutation, like the previous one, relies on the identifier encoded in the provided `JWT` token (of a `USER` entity type) to uniquely identify the User record and update it with the `data` argument provided.
  It should be noted that if the `password` field in the `data` argument is not `null`, it must be verified that it matches the minimal password criteria outlined in the section 3.3.1.3. If the `password` meets the minimal criteria, it is hashed using the `bcrypt` password-hashing function and then saved in the Database.

3.3.3 Upscaling



This section illustrates how the Server implementation performs the Upscaling procedure to enable `INACTIVE` nodes to be remotely bootstrapped employing `Wake-on-Lan`.
`Wake-on-Lan` (`WoL`) is an Ethernet standard protocol used to power on a machine using a specifically crafted network message known as a Magic Packet[1]. It should be noted that to be awakened, the system's NIC must support and enable `WoL`. The Magic Packet is a frame sent in broadcast (destination `MAC` address `FF:FF:FF:FF:FF:FF`) containing the NIC's `MAC` address of the target machine that needs to be awakened. When the Magic Packet is received by the corresponding NIC and `WoL` is supported and enabled, the system bootstraps as if it were turned on via the power button. Because the Magic Packet is broadcasted, every NIC in the cluster receives it, but it is simply ignored because the target `MAC` address does not match. `WoL` configuration and flags have already been thoroughly discussed in section 3.3.1.7.
When the `data` argument of the `updateNodePool` GraphQL mutation has the `count` field set to a value larger than the current one, the Upscaling process is executed. It should be noted that if the `count`'s value is less than the current `count`, the process is reversed and a Downscale procedure is performed.

[188] `https://github.com/carlocorradini/reCluster/blob/main/server/src/graphql/inputs/update/UpdateStatusInput.ts`
[189] `https://github.com/carlocorradini/reCluster/blob/main/server/src/graphql/inputs/update/UpdateUserInput.ts`

The difference between the `count` argument and the current `count` is the number of `INACTIVE` nodes in the associated node pool that must be bootstrapped. The Server implementation examines invalid or unfeasible values (i.e. the `count` is higher than the total number of nodes assigned to the associated node pool) and potential errors during the overall procedure.
Because the Server only has a low-level knowledge of the cluster and does not continuously monitor the entire Kubernetes cluster, its overall workload, and resource consumption, it must be instructed to execute the action by another entity. The entity that initiates the Upscaling procedure is typically the Cluster Autoscaler, which, unlike the Server, has a high-level knowledge of the cluster and can thus use the previously missing information to determine when and for how many nodes to perform the Upscaling procedure. A human Administrator is the second entity that physically forces the cluster to be Upscaled. The latter is less common, but it can be the consequence of different motivations and or requirements, such as preparing the cluster for an upcoming high volume of traffic that cannot be sustained with the previous number of active nodes.
The decision about which `INACTIVE` nodes should be bootstrapped adheres to the strict policy of power consumption reduction. The collection of `INACTIVE` nodes is ordered in ascending order of power consumption, first by the `max_power_consumption` attribute, then by the `min_power_consumption` attribute if the previous values are the same. If two nodes have both attributes equal, they are compared against their performance in descending order, with the most performant nodes appearing first. The final resulting list begins with all of the most power-efficient nodes and ends with the most energy-hungry nodes. Starting at the top of the list, the number of nodes that must be bootstrapped is determined by the difference between the count data argument and the current count. It should be noted that the Cluster Autoscaler's Automatic Upscaling request is typically not aggressive and only involves one node. As a result, only the first node at the top of the list is typically selected and involved in the bootstrapping process.
The preceding information and strategies for cluster Upscaling are condensed in the following steps and visually depicted in Figure 3.21:

1. A Cluster Autoscaler or Administrator entity submits a request to the GraphQL mutation `updateNodePool` with the node pool's identifier (`id` argument) and a `count` value higher than the current `count`. To allow the entity to perform the action, a valid `JWT` token must be supplied in the `Authorization` header.
   As an example, the identified node pool is `460d17d5-96ff-4e56-815c-e3367c60ae0d`, the `count` value is `2`, and the current count is `1`.

2. The `count` value is compared to the current one to determine whether the number of nodes in the corresponding node group should be `Increased` (`Upscaled`) or `Decreased` (`Downscaled`). The requested operation is to `Increment` the number of nodes in the node pool because the `count` value is higher. Moreover, it is also calculated the difference between the two values to determine the number of nodes to bootstrap.
   Because the provided `count` is higher than the existing count, the `Increase` operation is selected. The difference (`N`) number is `1`.

3. The Database is queried to obtain the list of `INACTIVE` nodes in the associated node pool that also supports `WoL`.

4. By comparing the `max_power_consumption` and/or `min_power_consumption` attributes, the list is ordered in ascending order of power consumption. If the latter two attributes for some nodes are identical, they are sorted by performance score in descending order.

5. `N` nodes are selected starting at the top of the list, where the most power-efficient nodes are located. A Magic Packet in broadcast is sent for each of these `N` nodes, targeting the `MAC` address of the associated node's NIC. After that, the status of each node is updated to `BOOTING`.
   The single most power-efficient node is selected (`NODE 1`) and the Magic Packet targeting its NIC `MAC` address is sent. The status is then updated to `BOOTING`.

6. The requested `N` nodes are successfully bootstrapped, and following the initialization procedure, they join the Kubernetes cluster.

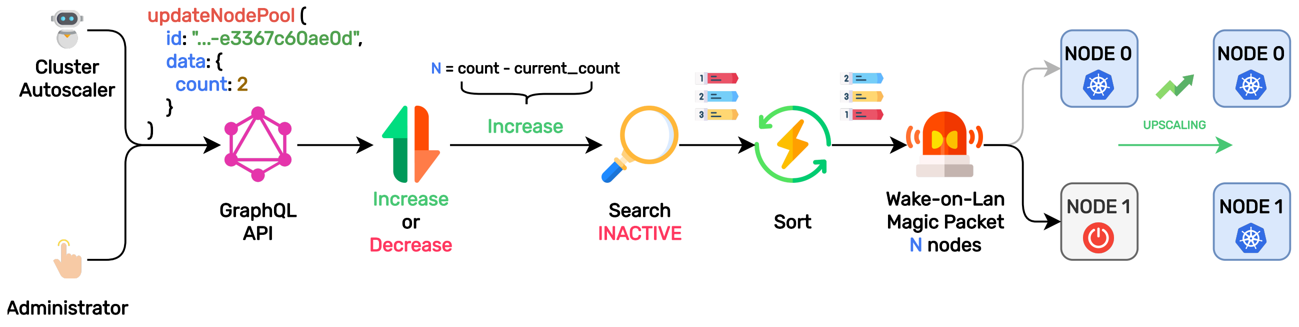
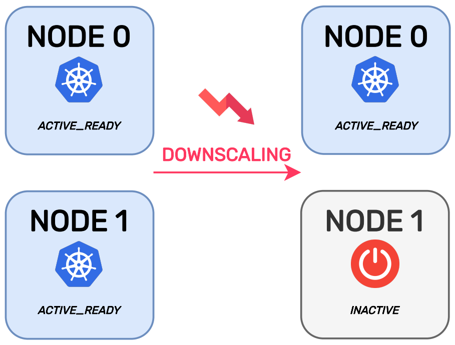


Figure 3.21: Upscaling scheme

3.3.4 Downscaling



This section illustrates how the Server implementation performs the Downscaling procedure to enable `ACTIVE` nodes to be remotely powered off employing `SSH`.
`SSH` (Secure Shell) is a protocol for secure remote login from one machine to another. `SSH` is used in the cluster implementation to remotely issue the `poweroff`[190] command to the corresponding node that needs to be Downscaled. The `SSH` remote connection is immediately closed (automatic disconnection) by the Server as soon as the command is successfully issued to prevent potential errors because the Downscaled node begins the shutdown procedure, which includes killing all processes, including the `SSH` Server process itself. Every node in the cluster has an `SSH` Server that accepts remote connections. The `SSH` Server process is continuously monitored by the corresponding Init System deployed on the node, and if it fails for any reason, it is resumed automatically. To issue the `poweroff` command, the Server implementation involves an `SSH` Client that establishes a secure and remote connection with the associated `SSH` Server by specifying the node's `IP` address. Both the `SSH` Server on the nodes and the `SSH` Client on the Server component must be properly configured with correct parameters, certificates, and keys, or the request for a connection from Client to Server will be promptly denied, resulting in the failure of the Downscaling procedure. During the installation procedure, the installer script configures the `SSH` Server, replacing any previous configuration with the one provided. The `SSH` Client configuration, on the other hand, is part of the Server and is thus performed with the same approach (see section 3.3.6). The current cluster implementation configures `SSH` in accordance with the best practices outlined in the article "`Secure Secure Shell`"[191]: Only the `Ed25519`[192] high-speed, high-security digital signature scheme is allowed, and only the most secure `Ciphers` and `MACs` are enabled. Because the latter are only configuration files, the organization can easily change and customize them to reflect different requirements. Having all cluster nodes pre-configured with `SSH` allows extra and diverse capabilities: Administrators can use an `SSH` Client to remotely connect to a node and perform various operations such as interactive and/or automated file transfers, whereas the use of automation tools such as Ansible[193] or Terraform[194] allow performing operations in parallel and in the same programmatic way, enabling another solution of configuring/managing the cluster via an Infrastructure as a Code. Finally, if the cluster is deployed in an Air-Gap environment and/or relying on `SSH` for remotely issuing the `poweroff` command is deemed unnecessary or too verbose for an organization, other non-protected and unsecure login protocols, such as `telnet` or `rlogin`, exist and can be used to achieve the same result. The cluster implementation currently only supports `SSH`, but extending support for other alternatives and/or unsecure protocols should be relatively simple.
Two GraphQL mutations support the Downscaling procedure:

[190] `https://linux.die.net/man/8/poweroff`
[191] `https://stribika.github.io/2015/01/04/secure-secure-shell.html`
[192] `https://ed25519.cr.yp.to/ed25519-20110926.pdf`
[193] `https://www.ansible.com`
[194] `https://www.terraform.io`

1. `updateNodePool`
   The first mutation is very similar to the process outlined for Upscaling. The Downscaling procedure is triggered when the `count` field in the `data` argument is set to a value smaller than the current one. The number of `ACTIVE` nodes in the associated node pool that must be terminated is the difference between the current `count` and the `count` argument. The collection of current `ACTIVE` nodes is arranged in the same way as the Upscaling procedure, using the same attributes and the same priority order. The sorting order, however, is inverted, so the `max_power_consumption` and `min_power_consumption` attributes are ordered in descending order while the `performance` is ordered in ascending order. The final list starts with the most energy-hungry (least power-efficient) nodes and ends with the most power-efficient (least energy-hungry) nodes. The number of nodes that must be terminated, starting at the top of the list, is equivalent to the previously stated difference between the current `count` and the `count` field.
   Before completely terminating a node, it must be drained (automatically performed by the Cluster Autoscaler before calling the GraphQL mutation) and deleted from the Kubernetes cluster (status `ACTIVE_DELETING`), resulting in all of its deployments being safely migrated without outages and the node being completely removed by the Kubernetes cluster, stopping the K8s Agent and its monitoring. The latter action is asynchronous, and once completed, the Server retrieves the `IP` address of the associated node, initiates an `SSH` remote connection to the `SSH` Server, and issues the `poweroff` command, effectively terminating the node. When the latter is finished, the node's status is updated to `INACTIVE`. The node's `ACTIVE` status lasts only a few seconds, from when the Server is notified of the Kubernetes removal to when the command is properly issued via `SSH`.
   It should be noted that the Server is notified of the node's successful removal from Kubernetes by utilizing the same monitoring system discussed in section 3.3.5, which is also used to keep the node's status updated. As a result, the implementation meets two requirements with a single efficient solution.

2. `unassignNode`
   The second mutation is much simpler than the first because it does not require the entire section of Increase/Decrease, filtering, and sorting because the node to be terminated is already provided as an argument (`id`).
   The complete termination procedure, which includes draining and deleting the node from Kubernetes and then remotely terminating via `SSH`, is the same as previously described.

The important distinction between the two mutations is that `updateNodePool` is generic in that it is most concerned about how many and which nodes need to be terminated, whereas `unassignNode` is specific in that it allows to specifically identify which node needs to be terminated. Because most Downscale operations are performed on a single and known a priori node, the mutation `unassignNode` is the obvious and preferred option. The latter is especially noticeable in the Cluster Autoscaler, which knows which nodes are underutilized and for how long thanks to continuous monitoring. If one of the latter nodes exceeds a predefined time threshold, the Cluster Autoscaler makes a request to the `unassignNode` mutation, specifying the identifier of the underutilized node. As previously mentioned, when the Cluster Autoscaler sends the Downscaling request, the Drain operation is much quicker because it has almost already been performed by the Cluster Autoscaler itself and/or the residual number of deployments are extremely low.
The preceding information and strategies for cluster Downscaling are condensed in the following steps and visually depicted in Figure 3.22:

1. A Cluster Autoscaler or Administrator entity submits a request to one of the two GraphQL mutations with a valid `JWT` token supplied in the `Authorization` header:
   - `updateNodePool`
     (a) The request includes the node pool's identifier (`id` argument) and a `count` value lower than the current `count`.
     As an example, the identified node pool is `460d17d5-96ff-4e56-815c-e3367c60ae0d`, the `count` value is `1`, and the current count is `2`.

(b) The Database is queried to obtain the list of `ACTIVE_READY` nodes in the associated node pool.
(c) By comparing the `max_power_consumption` and/or `min_power_consumption` attributes, the list is ordered in descending order of power consumption. If the latter two attributes for some nodes are identical, they are sorted by performance score in ascending order.
(d) `N` nodes are selected starting at the top of the list, where the less power-efficient nodes are located.
(e) For each selected node: Continue to parent step `2`.

- `unassignNode`
  (a) The request includes the node identifier (`id` argument).
  As an example, the identified node is `e64fd01e-03c4-435b-9b02-32dfed936507`.
  (b) The database is queried to obtain information about the corresponding node.
  (c) Continue to parent step `2`.

2. The corresponding Kubernetes node is drained and deleted.
   The node's status is updated to `ACTIVE_DELETING`.

3. The previous operation is completed and the Server is notified that the node was successfully deleted from Kubernetes.
   The node's status is updated to `ACTIVE`.

4. The Server retrieves the corresponding node's `IP` address, establishes an `SSH` remote connection to the `SSH` Server, and issues the `poweroff` command.
   The node's status is updated to `INACTIVE`.

5. The node has been successfully terminated, and the Kubernetes cluster now has one fewer node.

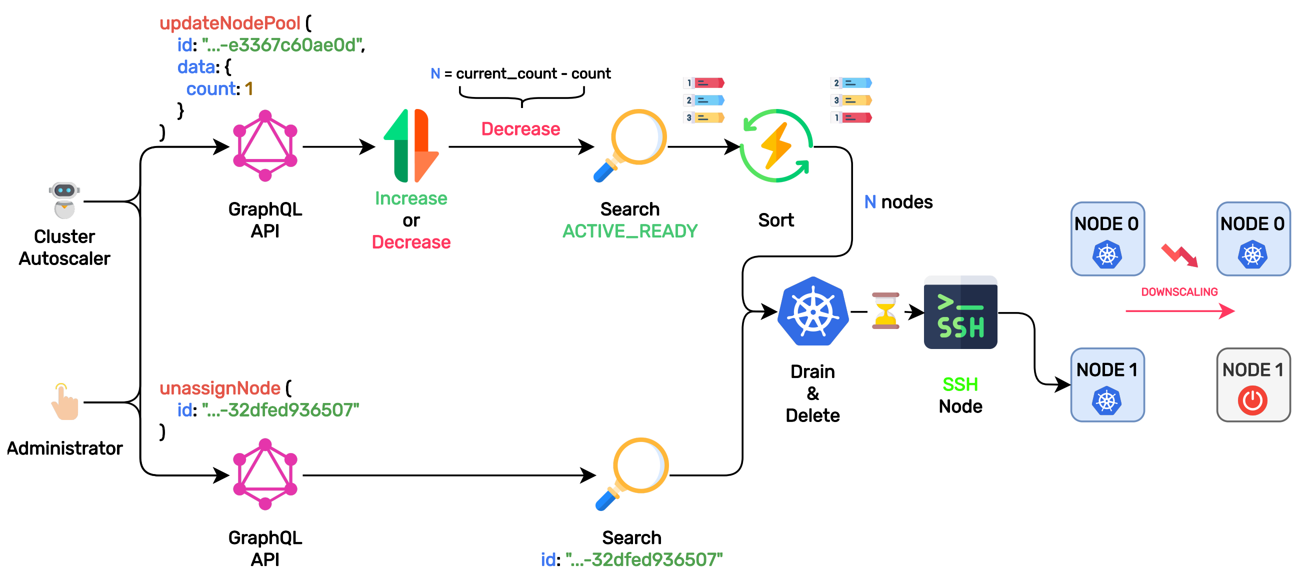


Figure 3.22: Downscaling scheme

3.3.5 Monitoring

The Server component must continuously monitor the currently active nodes to grasp their `status` and determine whether a node has been successfully `Added`, `Updated`, or `Deleted`.
Implementing a monitoring system from scratch that is based on periodic updates via Heartbeat messages and is also reliable, scalable, performant, and simple to use is a challenging task.
Because the cluster is built around Kubernetes, a monitoring system with periodic Heartbeat messages is already in place, satisfying all of the preceding requirements. As a result, the Server implementation takes advantage of the latter, thanks to the use of a Kubernetes `Informer`[195].

[195] `https://github.com/kubernetes-client/javascript/blob/master/src/informer.ts`

A Kubernetes `Informer` employs the Kubernetes API to `List` and `Watch` a specific resource or a group of resources. To prevent performance degradation and potentially useless requests (the resource has not been changed since the last request), the `Informer` does not constantly query (polling) the Kubernetes API. Instead, the `Informer` queries the resource(s) and stores the result in a local/internal cache, and an event (`Create`, `Update`, or `Delete`) is only triggered/generated when the cached resource(s) differs from the one available through the Kubernetes API. The latter outlined Kubernetes `Informer` is only an oversimplification of how it works and how it is internally implemented, but it is critical to understand that it provides a powerful, yet simple-to-use and configure, mechanism for monitoring cluster nodes.
The Kubernetes `Informer` can be considered as a subscription mechanism in which the Server component subscribes to the Node resource and whenever the resource changes, an event callback (`Add`, `Update`, or `Delete`) is triggered.
The monitoring system is implemented by the Server through a specialized class called `NodeInformer`[196], which encapsulates the Kubernetes Informer and is subscribed to all possible events concerning the Node resource. When the Kubernetes `Informer` triggers an event, the `NodeInformer` class's associated callback method is called. A callback function is only a frontend interface to a Kubernetes event, with the actual business logic delegated to one of the many Services based on the event that triggered the callback. The `NodeInformer` class, and thus the Kubernetes `Informer`, are only instantiated once when the Server bootstraps and terminated when the Server itself terminates.
Figure 3.23 depicts the monitoring scheme and the interaction of the previously mentioned components.

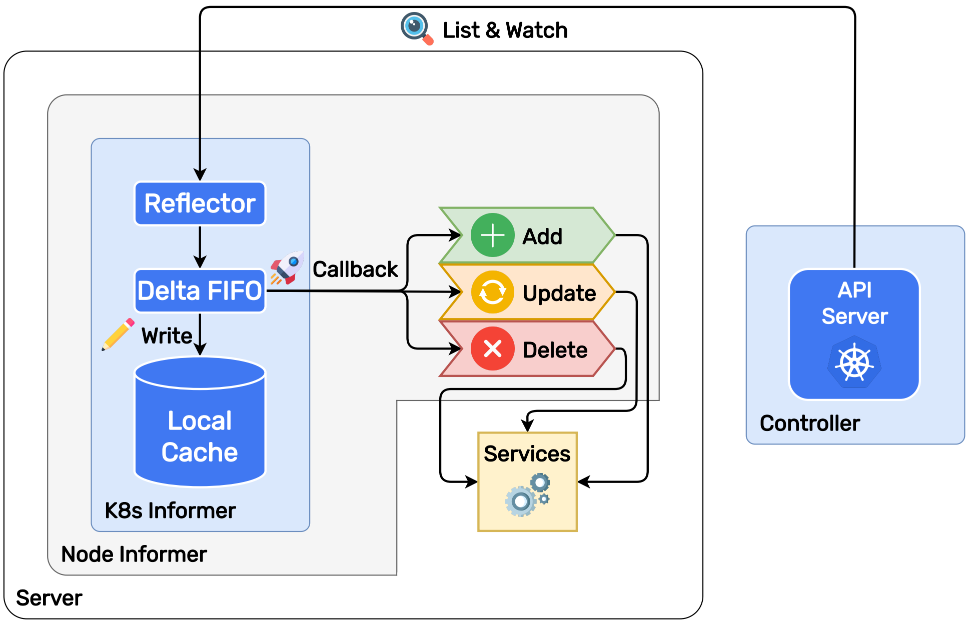


Figure 3.23: Monitoring scheme

The following list depicts the various NodeInformer callbacks and their purposes:

- `onAdd(node)`
  A new `node` resource has been added.
  Update the associated `node` record with the acquired `IP` address and `status`.

- `onUpdate(node)`
  A `node` resource has been updated.
  Update the associated `node` record with the acquired `IP` address and `status`.

- `onDelete(node)`
  A `node` resource has been deleted.
  Terminate (`poweroff`) the associated `node`. The latter procedure also updates its `status`.

[196] `https://github.com/carlocorradini/reCluster/blob/main/server/src/k8s/NodeInformer.ts`

- `onConnect()`
  `Informer` has successfully connected and is started monitoring the associated resource (`Node`).

- `onError(error)`
  Informer has encountered an `error`.
  After `3` seconds, it automatically restarts. The latter value is configurable.

3.3.6 Configuration

The Server implementation is fully customizable, with two methods available.
The first method is to use a `TypeScript` configuration file, named `config.ts`[197], that contains all of the setting parameters and structures used by the Server. Because it is a pure `TypeScript` file, it can be imported and used directly by other `TypeScript` source files with no additional steps.
The second method, which is more commonly used for application configuration, employs the use of environment variables[198]. Because polluting the environment with a multitude of variables is not a good practice, only the most significant and/or required configuration parameters are available. If an environment variable is provided, it overrides the default value assigned in the first method's `TypeScript` configuration file (`config.ts`). As a result, even though it only supports a subset of all possible configuration parameters, the second approach has a higher precedence than the first.
It would take too much space to explain all of the available configuration parameters that the Server implementation can be customized with, and some are self-explanatory, such as the `name` parameter, which specifies the application name, while others are almost immutable and rarely change. As a result, the following table only describes all of the configuration parameters and thus the environment variables that can be customized using the second method.

Name	Required	Description	Default Value
`NODE_ENV`	✖	`Node.js` environment. Different environments enable or disable certain Server components or features. For example, in `production`, the GraphQL Studio Explorer[199] is disabled, whereas it is enabled in all other environments. The following environments are supported: • `development` • `production` • `test`	`production`
`HOST`	✖	Listening address(es).	`0.0.0.0`
`PORT`	✖	Listening port.	`80`

[197] https://github.com/carlocorradini/reCluster/blob/main/server/src/config/config.ts
[198] https://github.com/carlocorradini/reCluster/blob/main/server/src/config/env.ts

`LOGGER_LEVEL`	✗	Logging level. Attachment C provides additional information regarding logging and logging levels. The following logging levels are supported (listed in descending order of importance): ⑦ `silent` ⑥ `fatal` ⑤ `error` ④ `warn` ③ `info` ② `debug` ① `trace`	`info`
`DATABASE_URL`	✓	Database URL. The current implementation is dependent on `PostgreSQL` and therefore the `URL` must be a valid `PostgreSQL` connection string.	
`SSH_USERNAME`	✗	`SSH` username.	`root`
`SSH_PRIVATE_KEY`	✓	`SSH` private key (identity file).	
`SSH_PASSPHRASE`	✗	`SSH` passphrase. The passphrase used to encrypt the `SSH` private key's sensitive section. Because the default cluster implementation does not involve any passphrase, it is not required.	
`TOKEN_PRIVATE_KEY`	✓	`JWT` private key. The private key used to sign the `JWT`.	
`TOKEN_PUBLIC_KEY`	✓	`JWT` public key. The public key used to verify the `JWT`.	
`TOKEN_PASSPHRASE`	✗	`JWT` passphrase. The passphrase used to encrypt the `JWT` private key's sensitive section. Because the default cluster implementation does not involve any passphrase, it is not required.	

Table 3.4: Server configuration environment variables

3.4 Autoscaling

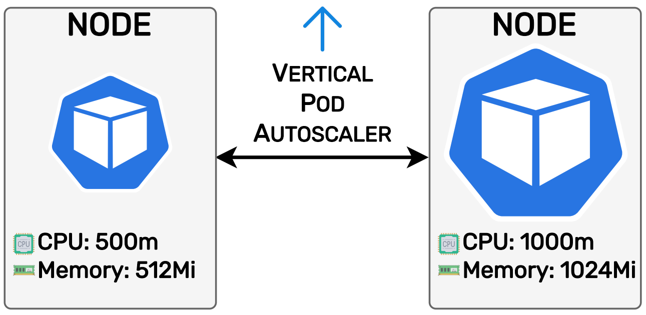
Autoscaling dynamically increases (upscale) or decreases (downscale) the number of cluster resources (limits, replicas, and nodes) that are allocated.

The cluster must be able to automatically perform autoscaling procedures to independently and autonomously change the resource limits of pods, number of replicas (pods), and number of nodes, requiring no or minimal human/manual intervention.
Kubernetes includes three Autoscaler components that perform automatically and autonomously the previous autoscaling procedures: `Vertical Pod Autoscaler` (see section 3.4.1) modifies the resource limits of the pods, `Horizontal Pod Autoscaler` (see section 3.4.2) modifies the number of replicas (pods), and `Cluster Autoscaler` (see section 3.4.3) modifies the number of nodes in the cluster.

[199] `https://www.apollographql.com/docs/graphos/explorer/explorer`

Because autoscaling (`Vertical`, `Horizontal`, and `Cluster`) is not the primary emphasis of the document, this section is not dedicated to explaining it in depth and how it is implemented. Rather, it is intended to provide a high-level overview of the three autoscaling mechanisms and their Kubernetes implementations. The only exception is the `Cluster Autoscaler`, because it necessitates a custom implementation built on top of the existing one to be compatible with the specific cluster environment (cloud provider).

3.4.1 Vertical Pod Autoscaler

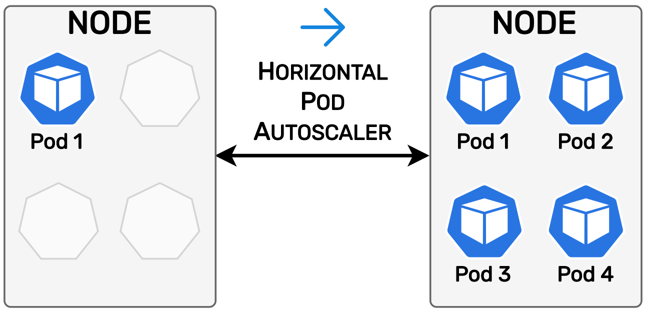


Vertical Pod Autoscaler[200] (VPA) adjusts a Pod's resource attributes (requests and limits) automatically.

Pods are executed with unbounded resource constraints by default and can thus utilize as much as the amount of available resources on the Node. To address this and other potential issues, a global policy can be defined via a Limit Range[201], which allows particular types of objects (such as `Pod` or `PersistentVolumeClaims`) to be limited in the quantity of allocable resources. The latter only partially solves the problem because it does not allow for a finer-grained resource definition for each possible scheduled object deployment/instance, because some Pods may require fewer resources, resulting in resource waste, while others may require/need more resources, resulting in performance degradation and/or service outages.
Kubernetes enables the fine-grained definition of the amount of resources that a Pod necessitates, commonly `CPU` and `Memory`. There are two kinds of resource definitions available: `request` and `limit`[202].
The resources `request` is the bare minimum of resources that the Pod necessitates. The `kube-scheduler`[203] component, which monitors for newly created Pods with no Node designated, uses this information to determine which Node is the best choice for the Pod to be scheduled on. When a Pod is deployed to a Node, the `kubelet`[204] component reserves the Node's desired resources `request` exclusively for the Pod.
The resources `limit` is the highest amount of resources that the Pod can use. The `kubelet` component enforces these constraints on a Pod, preventing it from utilizing more than what has been defined. These limits can be applied either `reactively` (when a violation is detected) or by `enforcement` (prevents the container from ever exceeding the limits).
If the Node where the Pod is deployed has sufficient free resources, the Pod may use more resources than the defined `request` amount while not exceeding the defined `limit`.
Manually defining a Pod's resources `request` and resources `limit` is a challenging task because it can result in resource waste if the values are set too high, as well as potential service outages and/or performance degradation if the values are set too low. The latter issues are the same as when the `Limit Range` approach is used.
Vertical Pod Autoscaler (VPA) relieves users from the burden of manually, accurately, and consistently updating the resources `limit` and `request` of a Pod. The VPA automatically sets the values of resources `request` and resources `limit` based on Pod utilization over time, providing for an improved scheduling mechanism that ensures the proper resource amount is available for each Pod scheduled on a Node. Moreover, the VPA can maintain the same resource ratio between `request` and `limit` that the user initially defined. As a result, the VPA downscales Pods that over-request resources and upscales Pods that under-request resources.

[200] https://github.com/kubernetes/autoscaler/tree/master/vertical-pod-autoscaler
[201] https://kubernetes.io/docs/concepts/policy/limit-range
[202] https://kubernetes.io/docs/concepts/configuration/manage-resources-containers
[203] https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler
[204] https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet

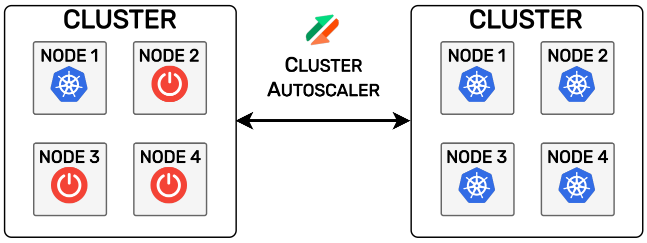
3.4.2 Horizontal Pod Autoscaler



Horizontal Pod Autoscaler[205] (HPA) adjusts the number of Pods (replicas) automatically.

Manually defining the number of Pods for a deployment is a difficult task that, like in the VPA, if not properly configured can result in resource waste and/or service disruptions.
The previously described resource `request` and `limit` must be specified when defining the Pod configuration. They are employed to establish the single resource or average resource utilization measurements of all Pods and whether an upscale (add replicas) or downscale (remove replicas) action is required.
The HPA configuration must target a specific application deployment and must include a minimal number of replicas and a maximum number of replicas. The minimal number of replicas is used to determine a lower bound (inclusive) level below which the HPA terminates any downscale operation. The maximum number of replicas is used to determine an upper bound (inclusive) level that cannot be exceeded, and the HPA stops any upscale operation. The final required parameter for the HPA is to specify a collection of metrics that indicate when to perform any autoscaling operation based on what resource types and their corresponding values to monitor. The `CPU` and `Memory` resources, as well as their average value, are the most common metric types. As an example, suppose the HPA is set to monitor the CPU metric with an average resource utilization of 50%. The HPA adds or removes Pods until the deployment's average CPU utilization is near 50%. If the average utilization is greater than 50%, the HPA would then upscale by adding more Pods, whereas if the average utilization is less than 50%, the HPA would then downscale by removing Pods.
Although the resource attributes used are similar to those employed by the VPA, the HPA behavior, execution, and algorithm are significantly different because the VPA scales the resources of a Pod while the HPA scales the number of Pods. It should be noted that HPA cannot be used alongside VPA on the same resources, and horizontal scaling is only suitable for stateless applications.
Horizontal Pod Autoscaler (HPA) relieves users from the burden of manually, accurately and consistently updating the number of Pods for a specific deployment by leveraging metrics monitoring and utilization.

3.4.3 Cluster Autoscaler



Cluster Autoscaler[206] (CA) adjusts the number of Nodes automatically.

Because the CA has already been extensively described in section 2.1.5, this section is devoted to showing how it is intended to interface with the Server component (and, more broadly, with the cluster implementation) for upscaling (see section 3.3.3 and section 3.4.3.2) or downscaling (see section 3.3.4 and section 3.4.3.3) operations. Finally, are mentioned some of the most important configuration parameters that can be modified to adjust the CA and its general autoscaling responsiveness/behavior to better fit the use-case situation of the organization maintaining the cluster.
Unlike the prior autoscalers, where only VPA or HPA could be used to monitor the same resources, CA can be used alongside both of them because it depends on different metric objects and has a different end goal. The combination of HPA and CA in the cluster is especially recommended. If the HPA attempts to schedule a number of Pods that exceed the current cluster capacity, the CA responds by increasing the total number of active nodes in the cluster, allowing un-schedulable/waiting Pods (Pods waiting to be assigned to a Node with enough free resources) to be scheduled on the newly booted

[205]`https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale`
[206]`https://github.com/carlocorradini/autoscaler/tree/master/cluster-autoscaler`

node. If the HPA decreases the number of Pods, reducing the overall cluster workload, and a Node remains underutilized, the CA responds by migrating the Pods scheduled on the Node to another Node, decreasing the total number of active nodes in the cluster[10].

3.4.3.1 Cloud Provider

The CA implementation differs from the previous two autoscalers in that it is not restricted to the Kubernetes cluster boundaries but also involves the external cluster managed by the so-called Cloud Provider. As a result, the CA must have a dedicated implementation built on top of the core CA implementation to support the specific Cloud Provider that manages the underlying cluster in order to automatically upscale (increase the number of nodes) and/or downscale the cluster (decrease the number of nodes).
The distinction between the two boundaries has already been extensively discussed using the word knowledge. The CA has a high-level knowledge of the cluster (Kubernetes) and information about the active Nodes and Pods, so it can determine when to perform an autoscaling operation, but it lacks a low-level understanding of how to do so. As a result, it must rely on the Cloud Provider's Server implementation, which has complete low-level knowledge of how to bootstrap or terminate the Nodes in the cluster but lacks the high-level knowledge of when to do so. As a result, the CA and the Server cooperate to accomplish the common autoscaling objective by combining their knowledge.
The official CA implementation is only compatible with big private and enterprise-oriented Cloud Providers like AWS, Azure, and GCP. As a result, a custom implementation of a Cloud Provider, known as `recluster`[207], has been developed that is compatible with the architecture outlined in this document. Furthermore, the `recluster` implementation is smaller in size than the official CA distribution because the official one needs to be compatible with every Cloud Provider (hence more code), whereas the custom implementation is only compatible with the `recluster` architecture and excludes the other Cloud Providers (hence reducing the code).
The Cloud Configuration (`cloud-config`) file is required by the CA to configure the custom Cloud Provider. The recluster Cloud Configuration is a `JSON` file that contains the two following attributes:

1. `token`
   `JWT` security token.
   It is included in the Authorization header of every request to the Server's GraphQL API.

2. `url`
   Server `URL`.
   The complete Server `URL`, including potential routes such as `/graphql`.

3.4.3.2 Upscaling

When the Cluster Autoscaler core needs to perform an upscaling operation, it employs the `increaseSize` function callback from the associated Cloud Provider implementation.

The `increaseSize` function takes two parameters. The first parameter, `nodePool`, is a pointer to the node pool on which the upscaling operation must be performed. The second parameter, `delta`, is a value that specifies the number of nodes that must be bootstrapped. The most common value of `delta` is `1`.
Listing 3.11 illustrates the pseudocode of the `recluster` Cloud Provider's upscaling function callback, and the following summary outlines its main steps:

❶ Check that `delta` is strictly greater than zero; otherwise, an error is thrown.
A `delta` value less than `0` indicates a downscaling operation rather than an upscaling operation, while a `delta` value identical to `0` indicates no scaling operation and thus both are invalid values.

❷ Calculate the new `count` value (`newCount`).
The value is determined by adding the current `count` value of the node pool to the `delta` value.

[207]`https://github.com/carlocorradini/autoscaler/tree/master/cluster-autoscaler/cloudprovider/recluster`

❸ Check that `newCount` is not greater than the maximum number of nodes allowed in the corresponding node pool (`max_nodes`); otherwise, an error is thrown.
It is important to note that if the value is identical to the maximum, it is accepted.

❹ Send a GraphQL request to the mutation `updatedNodePool` with the appropriate node pool identifier (`id` argument) and the updated count value in the `data` argument. The variable `updatedNodePool` holds the updated node pool.
Because it is an external request, it can fail, but this is omitted from the pseudocode for clarity.

❺ Compare the updated count value (`updatedNodePool.count`) to the new count value (`newCount`) computed to ensure that the updated node pool has been correctly updated. An error is thrown if the two values are not identical.

❻ Update the provided node pool's count (`nodePool.count`) value with the updated node pool's count (`updatedNodePool.count`) value.

```
function increaseSize(nodePool, delta) {
  if (delta <= 0) throw new Error();                                                    ❶

  const newCount = nodePool.count + delta;                                              ❷

  if (newCount > nodePool.max_nodes) throw new Error();                                 ❸

  const updatedNodePool = GraphQL.updateNodePool(nodePool.id, { count: newCount });     ❹

  if (updatedNodePool.count != newCount) throw new Error();                             ❺

  nodePool.count = updatedNodePool.count;                                               ❻
}
```

Listing 3.11: Pseudocode of `increaseSize` function callback

3.4.3.3 Downscaling

When the Cluster Autoscaler core needs to perform a downscaling operation, it can use one of the two function callbacks from the associated Cloud Provider implementation, `decreaseSize` and `deleteNodes`.
As previously described in section 3.3.4, when performing a downscaling operation in the cluster, it can be very general (`updateNodePool`) by specifying only the number of Nodes belonging to a node pool that need to be downscaled, or very specific (`unassignNode`) by specifying the exact Node identifier of the Node that needs to be downscaled. In this section, the more general approach implemented by the function `decreaseSize` is explained first, followed by the more specific approach implemented by the function `deleteNodes`. It should be noted that in `recluster`, probably due to its small size, the CA is very specific about which Node(s) should be downscaled, therefore it consistently employs the more specific callback function `deleteNodes`.

The `decreaseSize` function takes two parameters and is very similar to the `increaseSize` function in that it does the exact opposite. The first parameter, `nodePool`, is a pointer to the node pool on which the downscaling operation must be performed. The second parameter, `delta`, is a value that specifies the number of nodes that must be terminated. The most common value of delta is `-1`.
Listing 3.12 illustrates the pseudocode of the recluster Cloud Provider's downscaling function callback, and the following summary outlines its main steps:

❶ Check that `delta` is strictly lower than zero; otherwise, an error is thrown.
A `delta` value greater than `0` indicates an upscaling operation rather than a downscaling operation, while a `delta` value identical to `0` indicates no scaling operation and thus both are invalid values.

❷ Calculate the new `count` value (`newCount`).
The value is determined by adding the current `count` value of the node pool to the `delta` value. Even though the operation is identical to that of the `increaseSize` function, the resulting value is less than the current `count` value because `delta` is a negative value and thus sum over a negative number is a subtraction: $X + (-Y) = X - Y$.

❸ Check that `newCount` is not lower than the minimum number of nodes allowed in the corresponding node pool (`min_nodes`); otherwise, an error is thrown.
It is important to note that if the value is identical to the minimum, it is accepted.

❹ Send a GraphQL request to the mutation `updatedNodePool` with the appropriate node pool identifier (`id` argument) and the updated count value in the `data` argument. The variable `updatedNodePool` holds the updated node pool.
Because it is an external request, it can fail, but this is omitted from the pseudocode for clarity.

❺ Compare the updated count value (`updatedNodePool.count`) to the new count value (`newCount`) computed to ensure that the updated node pool has been correctly updated. An error is thrown if the two values are not identical.

❻ Update the provided node pool's count (`nodePool.count`) value with the updated node pool's count (`updatedNodePool.count`) value.

```
function decreaseSize(nodePool, delta) {
  if (delta >= 0) throw new Error();                                                    ❶

  const newCount = nodePool.count + delta;                                              ❷

  if (newCount < nodePool.min_nodes) throw new Error();                                 ❸

  const updatedNodePool = GraphQL.updateNodePool(nodePool.id, { count: newCount });     ❹

  if (updatedNodePool.count != newCount) throw new Error();                             ❺

  nodePool.count = updatedNodePool.count;                                               ❻
}
```

Listing 3.12: Pseudocode of `decreaseSize` function callback

The `deleteNodes` function takes two parameters. The first parameter, `nodePool`, is a pointer to the node pool on which the downscaling operation must be performed. The second parameter, `nodes`, is an array of the (active) Nodes that need to be terminated. Because the array usually contains only one Node, it has a corresponding length of `1`.
Listing 3.13 illustrates the pseudocode of the recluster Cloud Provider's downscaling function callback, and the following summary outlines its main steps:

❶ Check that the node's array `length` (number of Nodes) is strictly greater than zero; otherwise, an error is thrown.
A `length` value lower than zero `0` indicates an unexpected value because the length of an array must always be larger than or equal to `0`, whereas a length value equal to `0` indicates no scaling operation and thus both are invalid values.

❷ Calculate the new `count` value (`newCount`).
The value is determined by subtracting the current `count` value of the node pool from the `length` of the nodes' array.

❸ Check that `newCount` is not lower than the minimum number of nodes allowed in the corresponding node pool (`min_nodes`); otherwise, an error is thrown.
It is important to note that if the value is identical to the minimum, it is accepted.

❹ Iterate through the array's nodes.
The variable `i` represents the index of the array's ith node.

❺ Send a GraphQL request to the mutation `unassignNode` with the appropriate Node identifier (`id` argument). The variable `deletedNode` holds the deleted/terminated node.
Because it is an external request, it can fail, but this is omitted from the pseudocode for clarity.

❻ Decrement the provided node pool's count (`nodePool.count`) value by one (-1).
After repeating the same procedure for each Node, the node pool's count value is identical to the new count (`newCount`) value.

```
function deleteNodes(nodePool, nodes) {
  if (nodes.length <= 0) throw new Error();                         ❶

  const newCount = nodePool.count - nodes.length;                   ❷

  if (newCount < nodePool.min_nodes) throw new Error();             ❸

  for (let i = 0; i < node.length; i = i + 1) {                     ❹
    const deletedNode = GraphQL.unassignNode(nodes[i].id);          ❺

    nodePool.count = nodePool.count - 1;                            ❻
  }
}
```

Listing 3.13: Pseudocode of `deleteNodes` function callback

3.4.3.4 Configuration

The most significant configuration parameters of the Cluster Autoscaler that can be adjusted to better meet the requirements of the organization in charge of the cluster are listed in table 3.5. These parameters can be modified to increase the CA's responsiveness, making it more aggressive for any autoscaling procedure, or to decrease the CA's responsiveness, making it less aggressive for any autoscaling procedure.
The complete list of CA parameters supported is accessible at `https://github.com/carlocorradini/autoscaler/blob/master/cluster-autoscaler/FAQ.md#what-are-the-parameters-to-ca`.

Name	Description	Default Value
`cloud-provider`	Cloud Provider (`<NAME>`).	`recluster`
`cloud-config`	Path to the Cloud Configuration file (`<FILE>`).	
`kubeconfig`	Path to the `kubeconfig`[208] file (`<FILE>`). If the parameter is not provided, the CA attempts to acquire the `kubeconfig` directly from the Kubernetes cluster where it was deployed. The latter implies that the CA has been configured with the appropriate permissions (this is the case in `recluster`).	
`scale-down-delay-after-add`	The time frame during which the CA is logically disabled from executing a downscaling operation after having successfully executed an upscaling operation (`<TIME>`).	`10m`

`scale-down-delay-after-delete`	The time frame during which the CA is logically disabled from executing a downscaling operation after having successfully executed another downscaling operation (`<TIME>`). The default value is the same as the value of the `scan-interval` parameter.	`scan-interval`
`scale-down-delay-after-failure`	The time frame during which the CA is logically disabled from executing a downscaling operation after having unsuccessfully executed another downscaling operation (`<TIME>`).	`3m`
`scale-down-unneeded-time`	The duration that a Node should be unneeded (underutilized) before it is suitable for downscaling (`<TIME>`).	`10m`
`scale-down-unready-time`	The duration that a Node should be unready (`ACTIVE_NOT_READY` status and/or `UNKNOWN` status) before it is suitable for downscaling (`<TIME>`).	`20m`
`scale-down-utilization-threshold`	Node utilization level (`<LEVEL>`). It is defined as the total of requested resources divided by the capacity, below which a node is suitable for downscaling.	`0.5`
`scan-interval`	Time interval after which the cluster is evaluated for upscaling or downscaling (`<TIME>`).	`10s`
`skip-nodes-with-system-pods`	If `true`, never downscale Nodes with Pods from the `kube-system`[209] namespace (except for `DaemonSet` or mirror pods).	`true`
`skip-nodes-with-local-storage`	If `true`, never downscale Nodes with Pods that have persistency storage of type local storage (`EmptyDir` or `HostPath`).	`true`

Table 3.5: Cluster Autoscaler parameters

3.5 Installer

The installer is a POSIX script named `install.sh`[210] that contains approximately 3000 lines of code and is specifically designed to help administrators install the cluster implementation on a Node. It is highly configurable and can bootstrap a cluster from scratch if the `--init-cluster` parameter is specified. When the script is properly configured, all actions are performed automatically, exonerating the administrator of all responsibility. The overall installation procedure can take up to 10 minutes, depending on what components need to be installed and whether or not it is deployed in an Air-Gap environment.
Many previously described concepts and components are found in this section and are merged to

[208] https://kubernetes.io/docs/concepts/configuration/organize-cluster-access-kubeconfig
[209] https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces
[210] https://github.com/carlocorradini/reCluster/blob/main/install.sh

create a final and functional cluster implementation. It should be noted that the resulting cluster can be considered a unique entity in which each component collaborates with other components to keep the cluster healthy, stable and consistent.
This section begins by briefly explaining the benchmark and power consumption procedures, followed by a high-level view of the script's steps, and concludes by listing all of the supported configuration parameters and the configuration files employed during the installation procedure.

3.5.1 Benchmarks

Specific benchmarks are required to evaluate all of the different hardware components of a Node. As stated in previous sections, the resulting data scores are then used to understand the overall performance of the Node as well as the individual performance of the hardware components that comprise it.
The program selected to evaluate the system is `sysbench`[211], which allows benchmarking the CPU in single-thread and multi-thread, the memory in random/sequential read/write modes, and the different persistent storages in read/write modes.
A benchmark has a fixed time duration, which can be specified using the `--bench-time` parameter. Moreover, the `sysbench` application takes a configurable amount of time to warm the system before executing a benchmarking operation.
It should be noted that the final score number is specific to the `sysbench` application; thus, if the company desires to use a different application after having deployed the cluster, all benchmarks data must be retaken because the scores of the two applications may be incompatible.

3.5.2 Power Consumption

To determine the power consumption of the Node and to attempt to evaluate the power consumption of each hardware component, an external device capable of detecting the system's current draw is required. Furthermore, the device must include a basic but powerful API that enables the script to query for the current power consumption to obtain meaningful data over a specified period.
Figure 3.24 depicts the device used in the cluster, which is a `CloudFree`[212] EU Smart Plug. Even though numerous smart plug devices can perform the same tasks as the latter, the `CloudFree` device, as the name implies, does not require an internet connection to function and is thus cloud-free. Furthermore, it is pre-flashed with `Tasmota`[213], an Open Source firmware for `ESP` devices.

Figure 3.24: CloudFree EU Smart Plug

Because the overall cluster implementation is built on the `Tasmota` firmware, the installation script supports almost all devices that have been flashed with the `Tasmota` firmware in addition to the `CloudFree` devices. The latter is because the API endpoint and returned `JSON` schema for querying the current power consumption is the same for all `Tasmota` devices.
The script has a dedicated common function for reading the power consumption over a specified period. Before starting the readings, there is a brief warming period. Following that, a request is sent to the device with a sleep period between each request. The latter is repeated for the duration of the total period of power consumption readings. Then, all of the readings are examined and the mean and Sample Standard Deviation are computed. An error is thrown if the Sample Standard Deviation value exceeds a defined tolerance.

3.5.3 Installation Procedure

The installation procedure comprises the following steps, which are executed sequentially by the installation script. It should be noted that the name of each step corresponds to a specific function implemented in the script.

1. `parse_args`
   Process the script's input arguments.

[211]`https://github.com/akopytov/sysbench`
[212]`https://cloudfree.shop`
[213]`https://tasmota.github.io`

Section 3.5.4 lists the supported parameters.

2. `verify_system`
   Verify that the current system meets all of the criteria for installing `recluster`.
   Example requirements include verifying that all necessary programs are installed and the current Init System is supported.

3. `setup_system`
   Set up the system for the next steps.
   To perform determine operations, the script needs a temporary directory, which is created in this step.

4. `read_system_info`
   Read all of the system hardware information.
   All system information read is referred to as node facts (`NODE_FACTS`).

5. `run_benchmarks`
   Run benchmark tests on each hardware component and update `NODE_FACTS` with the results.

6. `read_power_consumptions`
   Read the overall power consumption of the system, both in idle and maximum load, and try to calculate the power consumption of each hardware component.
   The corresponding read values are appended to `NODE_FACTS`.

7. `finalize_node_facts`
   Adjust some values to finalize `NODE_FACTS`.

8. `install_k3s`
   Install K3s.

9. `install_node_exporter`
   Install Node Exporter.

10. `cluster_init`
    Install and activate the Database and Server components.
    This procedure must be run only once when the cluster is initialized with the argument `--init-cluster`.

11. `install_recluster`
    Install recluster on the node by registering the node via the Server component.
    After the installation process is completed successfully, the system is configured with the Server's returned data and other specifications.

12. `start_recluster`
    Start the cluster-related Node components.

13. `configure_k8s`
    Set up the Kubernetes cluster.
    It includes the installation of all Kubernetes components, such as the Load Balancer, Registry, and Cluster Autoscaler. It is important to note that each component is installed in a specific order because one component may require another component.
    This procedure must be run only once when the cluster is initialized with the argument `--init-cluster`.

After the installation procedure has been completed, `recluster` is up and running.
It should be noted that the installation script also installs an uninstall script, named `recluster.uninstall.sh`, that is used to completely remove all installed software and files.

3.5.4 Configuration Parameters

All of the installation script's supported configuration parameters are listed in the table below:

Name	Description	Default Value
`admin-username`	Administrator username (`<USERNAME>`).	`admin`
`admin-password`	Administrator password (`<PASSWORD>`).	`Password$0`
`airgap`	Boolean flag that indicates whether the current is an Air-Gap environment.	`false`
`autoscaler-username`	Autoscaler username (`<USERNAME>`).	`autoscaler`
`autoscaler-password`	Autoscaler password (`<PASSWORD>`).	`Password$0`
`autoscaler-version`	Autoscaler version (`<VERSION>`).	`latest`
`bench-time`	Benchmarking duration (`<TIME>`).	`30s`
`certs-dir`	Certificates directory (`<DIR>`).	`configs/certs`
`config-file`	`recluster` configuration file path (`<FILE>`).	`configs/recluster/config.yaml`
`disable-color`	Disable any color from the output.	`false`
`disable-spinner`	Disable spinner animation.	`false`
`help`	Display a help message and terminate (successfully).	
`init-cluster`	Boolean flag that indicates whether to initialize the cluster.	`false`
`k3s-config-file`	K3s configuration file path (`FILE`).	`configs/k3s/config.yaml`
`k3s-registry-config-file`	K3s registry configuration file path (`FILE`).	`configs/k3s/registries.yaml`
`k3s-version`	K3s version (`<VERSION>`).	`latest`

`log-level`	Logging level (`<LEVEL>`). Attachment C provides additional information regarding logging and logging levels. The following logging levels are supported (listed in descending order of importance): ⑤ `fatal` ④ `error` ③ `warn` ② `info` ① `debug`	`info`
`node-exporter-config-file`	Node Exporter configuration file path (`<FILE>`).	`configs/node_exporter/config.yaml`
`node-exporter-version`	Node Exporter version (`<VERSION>`).	`latest`
`pc-device-api`	Power consumption device's API URL (`<URL>`).	`http://UNKNOWN/cm?cmnd=status%2010`
`pc-interval`	Interval between the readings of power consumption (`<TIME>`).	`1s`
`pc-time`	Power consumption readings duration (`<TIME>`).	`30s`
`pc-warmup`	Power consumption warmup (`<TIME>`).	`10s`
`server-env-file`	Server environment file path (`<FILE>`).	`configs/recluster/server.env`
`spinner`	Spinner animation (`<SPINNER>`). This is regarded as an Easter Egg[214]. The following spinners are supported: 1. `dots` 2. `grayscale` 3. `propeller`	
`ssh-authorized-keys-file`	SSH authorized keys file path (`<FILE>`).	`configs/ssh/authorized_keys`
`ssh-config-file`	SSH configuration file path (`<FILE>`).	`configs/ssh/ssh_config`

`sshd-config-file`	SSH server configuration file path (`<FILE>`).	`configs/ssh/sshd_config`
`user`	Current system user to install recluster files and programs (`<USER>`).	`root`

Table 3.6: Installer script parameters

3.5.5 Configuration Files

This section highlights the most essential portions of the configuration and Kubernetes deployment files employed in the cluster implementation. Some portions have been removed, and the comment `...` indicates where this has occurred.

The majority of the parameters and/or configuration attributes revert to what has been explained and illustrated in all of the previous chapters and sections, therefore it is strongly advised to revise the corresponding argument of the respective configuration/deployment file if it is unclear.
The files may require identical properties, such as the Server URL or Registry URL, with other files. It has developed a utility POSIX script called `configs.sh`[215] that reads a configuration file named `configs.config.yaml`[216] that includes all of the most frequent and important configuration parameters. The script then substitutes an identification marker in the configuration/deployment files with the value read from the specified configuration file. This was done to prevent a high degree of duplication in the files as well as potential misconfiguration due to simple mistakes. The marker, represented by the color 🟪, begins with the symbols `${{` and ends with the symbols `}}`, and in the middle, there is a dot-separated string that identifies the corresponding parameter read from the script's configuration files. If there is no valid mapping between the identifier and the value, an error is thrown. The comment beneath the marker indicates the resulting line with the substitute value after the script has processed it. It should be noted that this document does not describe how the script has been implemented and configured.
Lastly, keep in mind that these files represent the configurations of the recluster implementation and can thus be adjusted or updated to better match the needs and/or requirements of another cluster deployment's organization. As an example, the IP addresses of the Load Balancer component are based on the private network `10.0.0.0/24`, but this setup may not be appropriate for other installations (it may clash with other already deployed networks), and thus it must/can easily be modified.

3.5.5.1 K3s

K3s registries configuration file (`registries.yaml`[217]) is shown in Listing 3.14. Take note of the relationship between the registry `mirror` name used in the Kubernetes deployment files and the actual local IP address `endpoint`.

```yaml
mirrors:
  ${{ k8s.registry.mirror.host }}:${{ k8s.registry.mirror.port }}:
  # registry.recluster.local:5000:
    endpoint:
      - "https://${{ k8s.registry.endpoint.host }}:${{ k8s.registry.endpoint.port }}"
        # "https://10.0.0.200:5000"

configs:
  ${{ k8s.registry.mirror.host }}:
  # registry.recluster.local:
    tls:
```

[214] https://wikipedia.org/wiki/Easter_egg
[215] https://github.com/carlocorradini/reCluster/blob/main/scripts/configs.sh
[216] https://github.com/carlocorradini/reCluster/blob/main/scripts/configs.config.yaml
[217] https://github.com/carlocorradini/reCluster/blob/main/configs/k3s/registries.yaml

```
      ca_file: '/usr/local/share/ca-certificates/registry.crt'
      key_file: '/usr/local/share/ca-certificates/registry.key'
```

Listing 3.14: K3s registries configuration file

K3s server configuration file (`config.controller.yaml`[218]) is shown in Listing 3.15. Take note of the `token`, cluster initialization flag, `server` URL, and node-taint `CriticalAddonsOnly=true:NoExecute`. The `server` parameter is only provided if the Node is not the first to be bootstrapped. If the `taint` is commented, it indicates that the Node is both a Server and an Agent, and thus can accept workload; otherwise, the Node cannot accept any workload.

```yaml
token: "${{ k3s.token }}"
# token: "4646f99bc4cbae3d5eceed856b337c9d3284be0d4056a3909f780c0c385fbf93"
cluster-init: true
# server: "https://${{ k3s.server.host }}:${{ k3s.server.port }}"
# server: "https://10.0.0.10:6443"
cluster-domain: 'recluster.local'
flannel-backend: 'host-gw'
disable:
  - 'servicelb'
node-taint:
  # - 'CriticalAddonsOnly=true:NoExecute'
```

Listing 3.15: K3s server configuration file

K3s agent configuration file (`config.worker.yaml`[219]) is shown in Listing 3.16. The parameters `token` and `server` are always required.

```yaml
token: "${{ k3s.token }}"
# token: "4646f99bc4cbae3d5eceed856b337c9d3284be0d4056a3909f780c0c385fbf93"
server: "https://${{ k3s.server.host }}:${{ k3s.server.port }}"
# server: "https://10.0.0.10:6443"
```

Listing 3.16: K3s agent configuration file

3.5.5.2 K8s

Cluster Autoscaler deployment file (`deployment.yaml`[220]) is shown in Listing 3.17. Take note of the Cloud Configuration, as well as the associated `url` and `token` attributes, in addition to the deployment definition, which specifies the Cluster Autoscaler container image from the respective local Registry and the command arguments.

```yaml
# ...

---
apiVersion: 'v1'
kind: 'Secret'
metadata:
  name: 'cluster-autoscaler-secret'
  # ...
stringData:
  cloud-config: |-
    {
      "url": "http://${{ recluster.server.host }}:${{ recluster.server.port }}
                /${{ recluster.server.path }}",
      # "url": "http://10.0.0.10:8080/graphql",
      "token": "${{ __.token }}"
      # "token": "${{ __.token }}"
    }

---
apiVersion: 'apps/v1'
```

[218] `https://github.com/carlocorradini/reCluster/blob/main/configs/k3s/config.controller.yaml`
[219] `https://github.com/carlocorradini/reCluster/blob/main/configs/k3s/config.worker.yaml`
[220] `https://github.com/carlocorradini/reCluster/blob/main/configs/k8s/autoscaler/ca/deployment.yaml`

```yaml
kind: 'Deployment'
metadata:
  name: 'cluster-autoscaler'
  # ...
spec:
  # ...
  replicas: 1
  template:
    # ...
    spec:
      # ...
      containers:
        - name: 'cluster-autoscaler'
          image: "${{ k8s.registry.mirror.host }}:${{ k8s.registry.mirror.port }}/recluster
                    /cluster-autoscaler"
          # image: "registry.recluster.local:5000/recluster/cluster-autoscaler"
          command:
            - './cluster-autoscaler'
            - '--v=4'
            - '--stderrthreshold=info'
            - '--cloud-provider=recluster'
            - '--cloud-config=/config/cloud-config'
            - '--cluster-name=recluster'
            - '--scale-down-unneeded-time=5m'
            - '--skip-nodes-with-local-storage=false'
            - '--skip-nodes-with-system-pods=false'
          # ...
      volumes:
        - name: 'cloud-config'
          secret:
            secretName: 'cluster-autoscaler-secret'
```

Listing 3.17: Cluster Autoscaler deployment file

MetalLB configuration file (`config.yaml`[221]) is shown in Listing 3.18. Take note of the IP address pool and the fact that it is used in `Layer 2 Advertisement` mode. Moreover, because both extremes are inclusive, the example pool, which is set from `10.0.0.200` to `10.0.0.250`, has `51` available addresses and not `50`.

```yaml
---
apiVersion: 'metallb.io/v1beta1'
kind: 'IPAddressPool'
metadata:
  name: 'default'
  namespace: 'metallb-system'
spec:
  addresses:
    - "${{ k8s.loadbalancer.ip.from }}-${{ k8s.loadbalancer.ip.to }}"
      # "10.0.0.200-10.0.0.250"

---
apiVersion: 'metallb.io/v1beta1'
kind: 'L2Advertisement'
metadata:
  name: 'default'
  namespace: 'metallb-system'
spec:
  ipAddressPools:
    - 'default'
```

Listing 3.18: MetalLB configuration file

Docker Registry deployment file (`deployment.yaml`[222]) is shown in Listing 3.19. Take note of the `TLS` parameters, persistent storage, deployment definition, and service configuration. The persistent

[221]https://github.com/carlocorradini/reCluster/blob/main/configs/k8s/loadbalancer/config.yaml
[222]https://github.com/carlocorradini/reCluster/blob/main/configs/k8s/registry/deployment.yaml

storage is of the type `local-path` or `longhorn` and has a capacity of `32 GiB`. The Registry `image` and its associated `port` are specified in the deployment. Lastly, the Service configuration of type `LoadBalancer` with all associated parameters is specified to expose the Registry. In addition to the latter, take notice of the `loadBalancerIP` parameter, which specifies a fixed IP address for the Registry that is within the Load Balancer IP address pool.

```yaml
# ...

---
apiVersion: 'v1'
kind: 'Secret'
type: 'kubernetes.io/tls'
metadata:
  name: 'registry-secret'
  # ...
data:
  tls.crt: "${{ k8s.registry.tls.crt }}"
  # tls.crt: "LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUZaekNDQTArZ0F3SUJBZ0lVVWg5T01vV0k1SEFG..."
  tls.key: "${{ k8s.registry.tls.key }}"
  # tls.key: "LS0tLS1CRUdJTiBQUklWQVRFIEtFWS0tLS0tCk1JSUpRd0lCQURBTkJna3Foa2lHOXcwQkFRRUZBQVND..."

---
apiVersion: 'v1'
kind: 'PersistentVolumeClaim'
metadata:
  name: 'registry-data-pvc'
  # ...
spec:
  accessModes:
    - 'ReadWriteOnce'
  storageClassName: 'local-path'
  # storageClassName: 'longhorn'
  resources:
    requests:
      storage: '32Gi'

---
apiVersion: 'apps/v1'
kind: 'Deployment'
metadata:
  name: 'registry'
  # ...
spec:
  # ...
  replicas: 1
  template:
    # ...
    spec:
      # ...
      containers:
        - name: 'registry'
          image: 'registry:2'
          ports:
            - containerPort: 5000
          # ...
      volumes:
        - name: 'registry-certs'
          secret:
            secretName: 'registry-secret'
        - name: 'registry-data'
          persistentVolumeClaim:
            claimName:' registry-data-pvc'

---
apiVersion: 'v1'
kind: 'Service'
```

```yaml
metadata:
  name: 'registry-service'
  # ...
spec:
  type: 'LoadBalancer'
  selector:
    app: 'registry'
  ports:
    - name: 'registry-port'
      protocol: 'TCP'
      port: ${{ k8s.registry.endpoint.port }}
      # port: 5000
      targetPort: 5000
  loadBalancerIP: "${{ k8s.registry.endpoint.host }}"
  # loadBalancerIP: "10.0.0.200"
```

Listing 3.19: Docker Registry deployment file

3.5.5.3 Node exporter

Node Exporter configuration file (`config.yaml`[223]) is shown in Listing 3.20. Take note that all collectors are disabled, except for the `CPU` and `memory` collectors.
The Node Exporter application does not allow a configuration file by default, and it can only be customized when it is executed via arguments. To simplify the process, the cluster implementation allows configuring Node Exporter with a file, and then all the parameters are transformed into arguments when it is installed via the installation script.

```yaml
collector:
  disable-defaults: true
  cpu: true
  meminfo: true
```

Listing 3.20: Node Exporter configuration file

3.5.5.4 reCluster

reCluster Controller configuration file (`config.controller.yaml`[224]) is shown in Listing 3.21.

```yaml
kind: 'controller'

recluster:
  server: "http://${{ recluster.server.host }}:${{ recluster.server.port }}"
  # server: "10.0.0.10:8080"
```

Listing 3.21: reCluster Controller configuration file

reCluster Worker configuration file (`config.worker.yaml`[225]) is shown in Listing 3.22.
The only distinction between the Controller and Worker configuration files is the `kind` parameter, which instructs the installation script which K3s and other potential component configuration file to read.

```yaml
kind: 'worker'

recluster:
  server: "http://${{ recluster.server.host }}:${{ recluster.server.port }}"
  # server: "10.0.0.10:8080"
```

Listing 3.22: reCluster Worker configuration file

Server environment configuration file (`server.env`[226]) is shown in Listing 3.23.

[223] https://github.com/carlocorradini/reCluster/blob/main/configs/node_exporter/config.yaml
[224] https://github.com/carlocorradini/reCluster/blob/main/configs/recluster/config.controller.yaml
[225] https://github.com/carlocorradini/reCluster/blob/main/configs/recluster/config.worker.yaml
[226] https://github.com/carlocorradini/reCluster/blob/main/configs/recluster/server.env

```
NODE_ENV='production'
HOST='0.0.0.0'
PORT=${{ recluster.server.port }}
# PORT=8080
LOGGER_LEVEL='info'
DATABASE_URL="postgresql://${{ recluster.database.user }}:${{ recluster.database.password }}
                @${{ recluster.database.host }}:${{ recluster.database.port }}
                /${{ recluster.database.db }}"
# DATABASE_URL="postgresql://recluster:password@10.0.0.10:5432/recluster"
SSH_USERNAME='root'
SSH_PRIVATE_KEY='/etc/recluster/certs/ssh.key'
SSH_PASSPHRASE=
TOKEN_PRIVATE_KEY='/etc/recluster/certs/token.key'
TOKEN_PUBLIC_KEY='/etc/recluster/certs/token.crt'
TOKEN_PASSPHRASE=
```

Listing 3.23: Server environment configuration file

3.5.5.5 SSH

SSH authorized keys configuration file (`authorized_keys`[227]) is shown in Listing 3.24.

```
${{ ssh.authorized_key }}
# ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIK4mrab03qgmkmWKSP+WcBDEWGGqJUBoiQdK3yAbKry9
```

Listing 3.24: SSH authorized keys file

SSH configuration file (`ssh_config`[228]) is shown in Listing 3.25. Take note of the `Ed25519` digital signature scheme, as well as the `Ciphers` and `MACs`.

```
Host *
  AddKeysToAgent yes
  IdentitiesOnly yes
  PasswordAuthentication no
  ChallengeResponseAuthentication no
  PubkeyAuthentication yes
  HostKeyAlgorithms ssh-ed25519-cert-v01@openssh.com,ssh-ed25519
  Ciphers chacha20-poly1305@openssh.com,aes256-gcm@openssh.com,
          aes128-gcm@openssh.com,aes256-ctr,aes192-ctr,aes128-ctr
  MACs hmac-sha2-512-etm@openssh.com,hmac-sha2-256-etm@openssh.com,
       umac-128-etm@openssh.com,hmac-sha2-512,hmac-sha2-256,umac-128@openssh.com
  UseRoaming no
  # ...
```

Listing 3.25: SSH configuration file

SSH server configuration file (`sshd_config`[229]) is shown in Listing 3.26.

```
Ciphers chacha20-poly1305@openssh.com,aes256-gcm@openssh.com,
        aes128-gcm@openssh.com,aes256-ctr,aes192-ctr,aes128-ctr
MACs hmac-sha2-512-etm@openssh.com,hmac-sha2-256-etm@openssh.com,umac-128-etm@openssh.com,
     hmac-sha2-512,hmac-sha2-256,umac-128@openssh.com
PubkeyAuthentication yes
PasswordAuthentication
ChallengeResponseAuthentication no
AllowTcpForwarding no
GatewayPorts no
X11Forwarding no
Subsystem sftp /usr/lib/ssh/sftp-server
# ...
```

Listing 3.26: SSH server configuration file

[227]https://github.com/carlocorradini/reCluster/blob/main/configs/ssh/authorized_keys
[228]https://github.com/carlocorradini/reCluster/blob/main/configs/ssh/ssh_config
[229]https://github.com/carlocorradini/reCluster/blob/main/configs/ssh/sshd_config

4 Conclusions

Current cluster architectures are mainly concerned with maximizing service performance and achieving near-instantaneous responsiveness while ignoring the energy impact and the various resources involved. As a result, it is becoming more and more necessary — almost a requirement — to reconsider and transform existing designs to create architectures that are more resource-aware and sustainable. These latter requirements must be considered just as vital as performance and responsiveness.
The context and fundamental principles on which the new and more sustainable cluster architectures should either completely or partially rely on have been initially explained in this document. Moreover, due to the outsourcing phenomenon that shifts local and physical architectures towards remote, intangible, and virtual architectures, users tend to forget the numerous implications of this decision that not only can have privacy and/or security concerns but also, and most importantly in this case, completely removes the direct control and the resource awareness of the overall system.
Following the introduction, a possible theoretical design that is more sustainable has been put forward, prioritizing the reduction of resource waste and energy consumption while maintaining a certain degree of adaptability and configurability. Every component that constitutes the architecture has been thoroughly explained, highlighting its main purpose and different capabilities. A cluster is not a cluster if the different components are not connected. In the design, there are three major network groups: K8s, Internal and External; where the External Network may even be removed if external resources and connections are not required. The reCluster project, which is a practical and sustainable cluster implementation of the theorized architecture, has also been presented. All of its hardware components have been recycled following decommissioning.
The implementation thoroughly explains how the reCluster was put into practice while adhering to both the fundamental requirements and the theoretical architecture behind the networks and components. It begins with the foundations, which are composed of GNU/Linux, a collection of programs, and the Init System, all of which are packaged together as a custom distribution into a portable and simple-to-install ISO image file. Then, each theorized component is mapped to a real-world application, and some of its most important features used in reCluster are explained. The way the Server and Cluster Autoscaler components are implemented is different because the first one was entirely built from scratch, whereas the second one used an existing core that had been modified to be compatible with the cluster architecture. The Server component represents the low-level knowledge of the cluster and is deeply described, including the database structure, GraphQL queries and mutations API, how the upscaling and downscaling procedures are handled, and how to exploit Kubernetes monitoring without the need for a custom implementation. The Cluster Autoscaler, on the other hand, represents the high-level knowledge of the cluster, and it is used in conjunction with the Server component to perform the upscaling or downscaling operations. The Cluster Autoscaler only scales the number of nodes and, because it is not the only method of autoscaling in a cluster, the other two approaches are also explained: Vertical Pod Autoscaler, which automatically adjusts the resources of a Pod, and Horizontal Pod Autoscaler, which automatically adjusts the number of Pods. The combination of the Horizontal Pod Autoscaler and the Cluster Autoscaler is critical for improving total automation and minimizing the need for human intervention, as well as lowering overall power consumption while maintaining scalability.
Lastly, because manually installing the cluster and providing all of the necessary information is tedious, the installation script that automatically performs the installation operation while following a specific set of configuration parameters has been discussed and proposed. Moreover, it is demonstrated how the node's benchmarks and power consumption data are measured. Finally, the most essential elements of the configuration and deployment files of the various components and/or technologies used are illustrated and discussed, so that the employed features of each can be recognized.
With all of the prior information and technology, building and deploying a more sustainable and

resource-aware cluster is feasible without incurring into breaking trade-offs; while it may be regarded as a tiny step towards sustainable computing, it, along with other initiatives, attempts to bring the result closer to realization.

4.1 Limitations And Future Work

This section depicts some of the difficulties encountered, existing criticalities and limitations, and relevant future ideas that can enhance and extend the current architecture and implementation. Because one of the project's main points is FLOSS (Free/Libre and Open Source Software) compliance, anyone from anywhere can contribute to it. The latter not only can increase the overall number of features but also improve overall stability thanks to intrinsic testing in various use-case scenarios and heterogeneous environments, which allows for the discovery and correction of previously unknown bugs and/or errors. External contributions from a wide variety of users are so valuable that they are frequently undervalued and underestimated in current contexts.

The combination of the installation script and the external device for measuring power consumption only conducts simple and basic readings on the instantaneously drawn energy. This is ideal for desktop computers and servers that require an external energy source, but it is incompatible with devices that use an internal energy source (batteries), such as laptop computers.
If the existing implementation of the installation script is executed on a laptop computer, the resulting power consumption readings are inaccurate since it does not take into consideration the current absorbed energy from both the external (plug) and internal (battery) sources, but only the first. As a result, the obtained final power consumption data are always lower than what are in reality, leading to general inefficiency and erroneous selection by the algorithm in upscaling and downscaling operations, because the Server recognizes the node as a much more power-efficient system than it is.
The script should determine whether the current node has internal energy sources and, when conducting power consumption readings and calculations, adjust with a more sophisticated but efficient and compatible algorithm that takes into account for both external and internal absorbed energy. Moreover, further testing on devices with an internal energy source is required to better understand how the system is powered, as it can solely use the external source, only use the internal source, or even employ both.

Another of the project's most significant difficulties has been developing the recluster Cloud Provider for the Cluster Autoscaler. There are three primary reasons for this.
First off, the Cluster Autoscaler and the majority of the Kubernetes development environment are built using the Go[1] programming language. I had never written a line of Go code before developing the `recluster` Cloud Provider, so I had to learn the language from the beginning as well as the package ecosystem, code style, and syntax. Although difficult at first, the effort paid off in the end.
Second, neither official nor unofficial documentation exists that explains how to create a Cloud Provider for the Cluster Autoscaler. My entire knowledge of developing the `recluster` Cloud Provider is based on studying and evaluating the code developed by the other Cloud Providers. Additionally, because each Cloud Provider has a unique implementation, distinct Server logic, and a different API, I had to determine the similarities and fundamental elements that almost all of them share to transform it and make it compatible with reCluster. The lack of documentation is understandable given that only a small number of entities utilize and develop this technology, making it possible for them to access experts and/or specialized consulting from the main CA developers themselves. Keep in mind that the latter needs familiarity with the Go programming language, which was covered in the preceding point.
Finally, it took longer than I anticipated to test the Cluster Autoscaler in conjunction with the Server and the general architecture since almost every time a new, minor error or bug that I hadn't considered appeared. However, the latter was not only beneficial for reCluster; it additionally helped in identifying an issue[2] in the Cluster Autoscaler's main code, which was submitted to the maintainers

[1] `https://go.dev`
[2] `https://github.com/kubernetes/autoscaler/issues/5378`

and promptly fixed within a week.

Currently, the only way for a user or administrator to interact with the cluster is to use applications such as `curl`, `wget`, or similarities to directly use the various set of GraphQL queries and mutation API, or use the GraphQL Studio Explorer, which is not available in production and is only a nicer interface for low-level interactions with the GraphQL API. As a result, there is a need to create a UI dashboard, such as the one from Kubernetes[3], where all of the information provided by the Server is visually available without involving the user in low-level GraphQL API interactions. The majority of operations should be simple and intuitive, without needing the user to better understand the requirements and outcomes.
Because the web application must consider the authentication mechanism, only authorized and/or authenticated users may access specific information or do certain operations. Furthermore, to provide a better overview of all the information accessible to the user, the dashboard should be capable of generating faster-to-understand graphs and/or charts. Consider a pie chart in which the number of active/working nodes (green color) is visually compared against the number of inactive nodes (gray color). The user can immediately understand the current cluster capacity/demand, where if the chart is almost entirely green, there is an overall high workload in the cluster, whereas if the chart is almost entirely gray, there is an overall low workload in the cluster, and thus the number of inactive nodes predominates.
With so many tools, technologies, and frameworks available for developing a web application nowadays, the most difficult and critical phase is choosing which one to use before even beginning development. Also, there may be a desire to improve the simplicity with which each node is managed without directly requiring `SSH` or similar technologies.
The latter can be accomplished by directly integrating a protected graphical dashboard on each node that displays the various information of the associated node as well as the ability to install, upgrade, or delete packages and perform additional operations.
`YunoHost`[4], which leverages a simple and easy-to-use, but incredibly powerful, web UI dashboard to completely administer the node, is a starting point for what may be accomplished or developed.

Computers are the only systems that are currently used and supported in the cluster. But, there are an enormous number of unused smartphone and tablet devices that might be employed in the cluster. Most of these devices are ideal for becoming worker nodes in a cluster because the previous owners replaced them due to an upgrade to a newer model or because there is hardware damage, such as on the screen or camera, but this has not affected important components such as the main board. Furthermore, these devices have become so powerful in recent years that they may be compared to the same performance as medium/low-end PCs while having a significantly lower power consumption thanks to different internal architecture and a prior design on energy consumption optimizations.
Developing for mobile devices might be challenging since there may be incompatibilities between the software used for the cluster related to the internal Operating System and missing libraries or Kernel requirements. A viable alternative is to uninstall the existing operating system (`Android` or `iOS`) and replace it with a pure GNU/Linux distribution designed exclusively for mobile devices, such as `postmarketOS`[5]. It should be noted that the overall compatibility of the Operating System with the device varies from device to device, and the OS may need to be patched. Mobile development should not be hard, however, this is the current status and/or only method for quickly porting applications designed for GNU/Linux on PCs to mobile devices.
As previously stated, the power consumption of mobile devices is extremely small compared to what they can provide. When compared to desktop computers and even laptop computers, the performance-to-power-consumption ratio is enormous. There are no simple techniques for remotely powering on a mobile device that does not involve custom solutions. However, it should be mentioned that the sleep mode of mobile devices is so efficient that a device in sleep mode consumes almost zero energy and may

[3]`https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard`
[4]`https://yunohost.org`
[5]`https://postmarketos.org`

last for more than a week on a single charge. As a result, if the remote wake-up solution is too expensive or unfeasible, the concept of keeping a mobile device constantly in sleep mode might be considered.

The cluster currently supports only `amd64` and `arm64` architectures. Because the majority of the employed software and technologies support a wide range of different architectures by default, such as `mips`[6] or `mipsel`, the remaining incompatible components, such as the Cluster Autoscaler, can be ported and/or transformed to be compatible with additional and different architectures.
Expanding the number of supported architectures allows more devices to be incorporated into the cluster, giving them new life while also improving the cluster's overall workload capacity.

The existing Node selection algorithm is effective, but it is a simplistic implementation because it just depends on maximum and minimum power consumptions and, if both are identical, falls back to performance data.
The existing algorithm works, despite its simplicity and limited accessible data to perform all conceivable computations and evaluations. Nevertheless, there are several aspects where it may be refined to accomplish a more suitable node's final decision. As an example, while upscaling, the chosen node may have a lower power consumption and possibly greater performance, whereas when downscaling, it may have higher power consumption and potentially worse performance. Furthermore, in future versions, the more efficient and sophisticated algorithm should take into consideration other factors with a more advanced internal logic. There may be nodes that are more efficient while under medium workload conditions, while others that are less efficient when under the same conditions but with lower minimum and maximum power consumption coefficients. The existing approach is deterministic, always preferring the second node type when upscaling and the first node type when downscaling. The new algorithm should perform extra computations and evaluations on which node type should bootstrap or terminate, and it's expected to follow the current overall cluster state as well as its overall power consumption.
The algorithm may also be enhanced with a Kalman Filter[7] for estimating the combination of power consumption and workload to significantly reduce total (real) power consumption while also improving overall Quality of Service (QoS).
The latter are merely possible ideas for what the new enhanced algorithm should include and what possible advancements, such as lowering power consumption while increasing overall performance, must be addressed.

There are no graphs or charts in this document to demonstrate how the generally existing cluster implementation performs on different workloads and/or hardware. This is because the time and resources required to conduct such measurements accurately and precisely are presently insufficient for a single person.
The cluster's performance management can be proposed as an extra document/paper that not only illustrates the various data collected, but also enhances and extends the existing design and algorithms with improvements determined from the performance management results. This task is currently assigned to another CRIT team member who employs this document as a foundation to better comprehend how the general architecture is designed and how to obtain correct and precise performance measurements.
Even though this document lacks data on performance measurements, the current cluster implementation by default exposes data that can be used for analysis of general performance and resource utilization. Every node in the cluster provides real-time hardware and OS-related data thanks to the Node Exporter component, which is described in section 3.2.4. These metrics are then collected and analyzed by the Prometheus component, as described in section 3.2.7.1, which allows for monitoring and querying/interpolating them with various results that can cover a specified period.
Furthermore, because the overall architecture is already established and deployed automatically, control of overall performance may be employed not only occasionally or during testing but always. This

[6]`https://wikipedia.org/wiki/MIPS_architecture`
[7]`https://wikipedia.org/wiki/Kalman_filter`

not only gives reacher statistics, but it additionally has the potential to provide real-time alerts on the overall health of the cluster. Besides that, custom notifications that monitor the total cluster power consumption can be established. This can be critical in a variety of use-case situations. Consider the following scenario: the cluster is powered solely by batteries that are charged during the day by solar panels, similar to the Low-Tech magazine[8], and there is a known overall threshold amount of energy consumption that, if exceeded, can cause the entire cluster to go offline until the next recharge. Employing an accurate performance management system to compute the threshold, as well as a monitoring and alerting system to notify an administrator, is critical; enabling the cluster to be online and active as much as possible while also performing its main routine of service orchestration.
Nevertheless, this point is critical, and it must be addressed and implemented in future versions of reCluster.

In addition to the preceding point, there is a missing section where the various use-case scenarios are analyzed and tested owing to the same reasons of a single person's lack of time and resources.
Currently, only two use-case scenarios have been successfully evaluated in a real-world context using the reCluster cluster architecture. The first scenario is a one-to-one mapping to a potential cluster used for hosting various services that need to be exposed to an external network or, more broadly, the Internet. It has a stable connection, and most of the components and their configurations may be downloaded, guaranteeing the most recent and up-to-date versions are used in the cluster. Having the most recent versions available might be beneficial since there may be bugs or security vulnerabilities that have been addressed, whereas the bundled ones may contain issues due to being outdated. The first scenario is the most common, in which practically anybody can deploy a cluster without difficulty. The second scenario, on the other hand, is an Air-Gap environment in which there is no Internet connection and all components and processes must be available and performed offline. The latter has been well detailed throughout the document, and it may also be regarded as a solid foundation/requirement for future environmental and availability crises in which having an internet connection is no longer obvious. Using Air-Gap measurements only during the installation procedure, while the normal execution is in normal conditions with an Internet connection, can be advantageous because it does not require any download, reducing overall installation time and also avoiding the involvement of networking equipment, which can consume unnecessary energy, and the installed software behavior is always predictable.
In addition to the preceding two scenarios, the cluster might be deployed in two additional scenarios. The first instance is when the cluster may be used in education to evaluate students' work. The latter can be employed to test their Kubernetes abilities, as well as teach them basic and advanced web apps or APIs that include various software components, such as a server, a database, and a cache server. The second scenario is for redundancy, in which reCluster is being used only if the main cluster, which is more performant but consumes more energy, is no longer accessible due to a failure. reCluster uses decommissioned main cluster hardware and has just one Controller node active, whereas all worker nodes are inactive. If there is workload that has to be scheduled while the main cluster is unavailable, the worker nodes start to boot up. It is worth noting that, even if there are hundreds of worker nodes, the overall power consumption when the main cluster is healthy is that of the single Controller node, which is negligible.

In future releases, the Server implementation must be improved.
Replace any interpreted languages, such as `JavaScript` or `Python`, in the Server with a compiled/low-level programming language, such as `C`, `C++`, or `Rust`. Using these programming languages has two major advantages over the existing implementation. First, because the program must be compiled to produce the final executable, there is no need to distribute an archive or a folder containing many files and then download and install all of the required dependencies. The required dependencies must be available only on the machine performing the compilation procedure, and if the program

Source: `https://foundation.rust-lang.org/policies/logo-policy-and-media-guide`

Figure 4.1: Rust logo

[8] `https://solar.lowtechmagazine.com`

is compiled statically, all used libraries, such as the OpenSSL library, are directly bundled inside the final executable, at the expense of being larger than a dynamically linked program. Furthermore, and this is critical, [22] reports that applications that are implemented with a system programming language and are compiled directly to machine code are significantly more efficient and demand fewer resources and energy to accomplish the same result as an interpreted language. I would prefer `Rust`[9] over the other two system languages because it guarantees memory safety and thread safety, increasing overall reliability and productivity, even though it has a steep learning curve at first, thanks to the various official tools, such as a package manager and auto formatter. Although the development and effort required by a compiled language is greater than that required by an interpreted language, the overall benefits in this circumstance are too significant to be ignored.
Another area that needs to be enhanced is general error management and node administration, since there is currently just a simple management of the two that heavily relies on Kubernetes. Updated versions should improve error alerts to administrators as well as possible recovery mechanisms. The latter can also be extended to use various approaches based on the overall circumstances and deployment environment.
Undoubtedly, many aspects of the Server could be improved but are not covered in this section. However, because the project is completely FLOSS compliant, users can contribute and improve the code with ideas that I hadn't even considered before.

4.2 Closing Remarks

Adapting existing tools that are specifically developed for big and energy-hungry environments to a more sustainable and resource-aware design while maintaining nearly the same capabilities for the computing industry is not only feasible, but also necessary for a more sustainable future.

[9] `https://www.rust-lang.org`

Bibliography

Icons in all Figures, unless otherwise specified, are from `www.flaticon.com`. They were created by one or more of the following authors: `Fathema Khanom`, `Freepik`, `Iconpro86`, `juicy_fish`, `kerismaker`, `Kiranshastry`, `Pixel perfect`, `Roundicons`, `Smashicons`, `Uniconlab`, `Vectors Market`, and `Vitaly Gorbachev`.

[1] Advanced Micro Devices, Inc. Magic Packet Technology. Technical Report 20123, Advanced Micro Devices, Inc, November 1995.

[2] Aleksić S. Analysis of Power Consumption in Future High-Capacity Network Nodes. *J. Opt. Commun. Netw.*, 1:245–258, August 2009.

[3] Amazon Web Services, Inc. What is DevOps? `https://aws.amazon.com/devops/what-is-devops`. Accessed on 20/01/2023.

[4] Angeli L., Okur Ö., Corradini C., Stolin M, Huang Y., Brazier F. and Marchese M. Conceptualising Resources-aware Higher Education DigitalInfrastructure through Self-hosting: a Multidisciplinary View. In *Eighth Workshop on Computing within Limits 2022*, June 2022.

[5] Arm Limited. Arm Big.LITTLE. `https://www.arm.com/technologies/big-little`. Accessed on 24/01/2023.

[6] Bradley J., Jones M. and Sakimura N. JSON Web Token (JWT). RFC 7519, May 2015.

[7] Castro J. D. *Introducing Linux Distros.* Apress, 1 edition, 2016.

[8] Cox R. Surviving Software Dependencies. *Commun. ACM*, 62(9):36–43, August 2019.

[9] Decker D. K. The monster footprint of digital technology. `https://www.lowtechmagazine.com/2009/06/embodied-energy-of-digital-technology.html`.

[10] DigitalOcean, LLC. Autoscale Cluster With Horizontal Pod Autoscaling. `https://docs.digitalocean.com/tutorials/cluster-autoscaling-ca-hpa`. Accessed 04/02/2023.

[11] Eastlake D. E. 3rd and Panitz A. R. Reserved Top Level DNS Names. RFC 2606, June 1999.

[12] Enes J., Fieni G., Expósito R. R., Rouvoy R. and Touriño J. Power Budgeting of Big Data Applications in Container-based Clusters. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 281–287, 2020.

[13] Groth D., Lammle T. and Tedder W. *Network +.* Sybex, Inc., 1 edition, 2003.

[14] IBM Corporation. High availability versus fault tolerance. `https://www.ibm.com/docs/en/powerha-aix/latest?topic=aix-high-availability-versus-fault-tolerance`. Accessed on 06/01/2023.

[15] Intel Corporation. What Is Performance Hybrid Architecture? `https://www.intel.com/content/www/us/en/gaming/resources/how-hybrid-design-works.html`. Accessed on 24/01/2023.

[16] Javvin Technologies, Inc. *Network Protocols Handbook.* Javvin Technologies, Inc., 2 edition, 2004.

[17] Muhammad B. A., Shahin M. and Zhu L. Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. *IEEE Access*, 5:3909–3943, 2017.

[18] Nardi B., Tomlinson B., Patterson D. J., Chen J., Pargman D., Raghavan B. and Penzenstadler B. Computing within Limits. *Commun. ACM*, 61(10):86–93, September 2018.

[19] Ong D., Moors T. and Sivaraman V. Comparison of the energy, carbon and time costs of videoconferencing and in-person meetings. *Computer Communications*, 50:86–94, 2014.

[20] Pasek A., Vaughan H. and Starosielski N. The world wide web of carbon: Toward a relational footprinting of information and communications technology's climate impacts. *Big Data & Society*, 10:205395172311589, February 2023.

[21] Paya A. and Marinescu D. C. Energy-Aware Load Balancing and Application Scaling for the Cloud Ecosystem. *IEEE Transactions on Cloud Computing*, 5(1):15–27, 2017.

[22] Pereira R., Couto M., Ribeiro F., Rua R., Cunha J., Paulo J. F. and Saraiva J. Ranking programming languages by energy efficiency. *Science of Computer Programming*, 205:102609, 2021.

[23] Permacomputing contributors. Permacomputing. `https://permacomputing.net`. Accessed 26/02/2023.

[24] Radovanović A., Koningstein R., Schneider I., Chen B., Duarte A., Roy B., Xiao F., Haridasan M., Hung P., Care N., Talukdar S., Mullen E., Smith K., Cottman M. and Cirne W. Carbon-Aware Computing for Datacenters. *IEEE Transactions on Power Systems*, 38(2):1270–1280, 2023.

[25] Schwarz N., Lungu M. F. and Nierstrasz O. SEUSS: Decoupling responsibilities from static methods for fine-grained configurability. *The Journal of Object Technology*, 11:3:1, April 2012.

[26] Selwyn N. Ed-Tech Within Limits: Anticipating educational technology in times of environmental crisis. *E-Learning and Digital Media*, 18(5):496–510, 2021.

[27] SentinelOne. What is an "Air Gap" in Network Security? `https://www.sentinelone.com/blog/air-gapped-networks-a-false-sense-of-security`. Accessed on 15/01/2023.

[28] Silberman S. M. and Tomlinson B. Precarious infrastructure and postapocalyptic computing. In *Examining Appropriation, Re-use, and Maintenance for Sustainability, workshop at CHI 2010*, 2010.

[29] Sriramya P. and Karthika R. A. Providing password security by salted password hashing using Bcrypt algorithm. *ARPN Journal of Engineering and Applied Sciences*, 10:5551–5556, January 2015.

[30] Vanderbauwhede W. Frugal Computing. `https://limited.systems/articles/frugal-computing`. Accessed on 15/02/2023.

Appendix A Corollary Projects

During the development, I noticed that several portions of the code might be turned into independent libraries and utility scripts that may be valuable to other programmers as well as my single use-case scenario. Even though it is now external to reCluster, this software is still an essential part of it, which is now open to the entire community.
Shortly after the publication, I began receiving some feedback, contributions, and, most importantly, appreciation in the form of GitHub stars[1].
The three projects derived from the development of reCluster are briefly illustrated and explained in the sections that follow. As stated in the Philosophy section, everything is completely Open Source and available under the MIT license.

A.1 Node Exporter Installer

Available at `https://github.com/carlocorradini/node_exporter_installer`
Used while installing reCluster on a Node (see section 3.5).
Node exporter (see section 3.2.4) does not provide any installation script and the default procedure (see `https://github.com/prometheus/node_exporter#installation-and-usage`) is far from user-friendly and easily configurable.
Inspired by K3s (see section 3.2.3) `install.sh`[2] script, Node exporter installer helps the user by automatically installing Node exporter on the machine. Condensed in a single `install.sh` POSIX script, it is easily configurable (see section A.1.1) and can be downloaded and executed directly by `sh` (see example A.1.2.1).

A.1.1 Configuration

Node exporter installer accepts environment variables only as configuration parameters. This is done to avoid any potential conflict with the default behavior of Node exporter, which employs argument flags. See `https://github.com/prometheus/node_exporter#collectors` for a comprehensive list of Node exporter configuration parameters.
The configuration settings allowed by Node exporter installer are shown in the table below.

Name	Description	Default Value
`INSTALL_NODE_EXPORTER_SKIP_DOWNLOAD`	Skip downloading Node exporter. A local executable binary must already exist at `<BIN_DIR>/node_exporter` Useful in an Air-Gapped environment.	`false`
`INSTALL_NODE_EXPORTER_FORCE_RESTART`	Force restarting Node exporter service.	`false`

[1] `https://docs.github.com/en/get-started/exploring-projects-on-github/saving-repositories-with-stars`
[2] `https://github.com/k3s-io/k3s/blob/master/install.sh`

`INSTALL_NODE_EXPORTER_SKIP_ENABLE`	Do not enable Node exporter service at startup.	`false`
`INSTALL_NODE_EXPORTER_SKIP_START`	Do not start Node exporter service.	`false`
`INSTALL_NODE_EXPORTER_SKIP_FIREWALL`	Do not apply any firewall rules. Supported firewalls are: `iptables`[3], `firewalld`[4] and `UFW`[5].	`false`
`INSTALL_NODE_EXPORTER_SKIP_SELINUX`	Do not change `SELinux`[6] context for Node exporter binary.	`false`
`INSTALL_NODE_EXPORTER_VERSION`	Node exporter version to download.	`latest`
`INSTALL_NODE_EXPORTER_BIN_DIR`	Directory where to install Node exporter binary and uninstall script.	`/usr/local/bin` or `/opt/bin`
`INSTALL_NODE_EXPORTER_SYSTEMD_DIR`	Directory where to install Systemd service files.	`/etc/systemd/system`
`INSTALL_NODE_EXPORTER_EXEC`	Node exporter configuration flags.	

Table A.1: Node exporter installer configuration parameters

A.1.2 Example

This section shows some examples of how to use Node exporter installer. In both, we use `curl`[7] to download the script from `https://raw.githubusercontent.com/carlocorradini/node_exporter_installer/main/install.sh` and pipe the output to `sh` for execution.

A.1.2.1 Basic

Install Node exporter using the default configuration parameters for both installer and binary.

```
curl \
  --silent \      # Do not show progress meter or error messages
  --show-error \  # Show an error message if it fails
  --fail \        # Fail fast with no output at all on server errors
  --location \    # If the requested resource has moved, redo the request to the new location
  "https://raw.githubusercontent.com/carlocorradini/node_exporter_installer/main/install.sh" \
  | sh -
```

Listing A.1: Basic installation with default configuration parameters

A.1.2.2 Advanced

Install Node exporter v1.5.0 without starting the service automatically. Disable all default collectors, leaving only CPU and memory statistics enabled.

```
curl \
  ... \
```

[3] `https://www.netfilter.org/projects/iptables`
[4] `https://firewalld.org`
[5] `https://wiki.ubuntu.com/UncomplicatedFirewall`
[6] `https://selinuxproject.org`
[7] `https://curl.se`

```
| INSTALL_NODE_EXPORTER_VERSION="v1.5.0" \   # Download and install Node exporter v1.5.0
  INSTALL_NODE_EXPORTER_SKIP_START="true" \  # Do not start Node exporter service
  sh - \
    --collector.disable-defaults \             # Disable default collectors
    --collector.cpu \                          # Expose CPU statistics
    --collector.meminfo                        # Expose memory statistics
```

Listing A.2: Advanced installation with custom configuration parameters

A.2 Inline

Available at `https://github.com/carlocorradini/inline`
Used while generating the final release bundle for distribution (see section B.3).
It is useful to be able to split a large script into many files to make it easier to work with while still being able to distribute it as a single script. This program reads an input file and produces an output file with all of the sources inlined.
The source command[8], `. <FILE>` for POSIX shells or `source <FILE>` for non-POSIX shells (e.g. Bash[9]), read, and executes commands from the file specified in the current shell environment. It is useful to load functions, variables, and configuration files into the current shell context.
Inline is a static tool that does not execute the input script. As a result, it cannot determine the value of a variable dynamically if it is used inside the source command path (i.e. `source "$DIR/path/to/script/sh"`). To avoid this, inline requires a hint on where to find the specified file.

A.2.1 Features

Many helpful features of Inline are described below.

- POSIX standard-compliant.
- Sourcing with quotes, spaces, and more.
- Sourcing from global variable `$PATH`.
- Sourcing from `ShellCheck` (a shell script static analysis tool[10]) source directive[11].
  This is considered a hint to Inline and is used only if the source path is invalid.

```
# shellcheck source=path/to/script.sh
. "$DIR/path/to/script.sh"

# shellcheck source=path/to/script.sh
source "$DIR/path/to/script.sh"
```

- Recursive sources. If a sourced file contains additional source commands, they are also inlined and included in the final output script.
  It should be noted that these can cause infinite recursion.
- Recursion detection. To avoid infinite recursion, an exception is thrown if a file is sourced multiple times. This is accomplished through the use of an internal cache that saves the absolute path to each sourced file. If the path to a script file in a source command is already in the cache, a recursion is detected.

[8]`https://linuxize.com/post/bash-source-command`
[9]`https://www.gnu.org/software/bash`
[10]`https://www.shellcheck.net`
[11]`https://www.shellcheck.net/wiki/Directive`

- Shebang removal in sourced files.
  A shebang[12] is the character sequence consisting of the hashtag symbol and exclamation mark (`#!`) at the beginning of a script.
  The shell interpreter only allows one shebang per script. As a result, the shebang is only allowed in the input script file, while it is automatically removed in all sourced script files.

- Avoid inlining a certain source file. If the inline skip comment directive (`# inline skip`) is present before a source command, the latter is ignored and not inlined. As a consequence, the final output script includes the original command, unaltered and unaligned.
  It is worth noting that the skip directive also works with ShellCheck. The sole requirement is that there are no blank lines between the directives and the source command.

```
# inline skip
# shellcheck source=path/to/script.sh
. "$DIR/path/to/script.sh"

# shellcheck source=path/to/script.sh
# inline skip
source "$DIR/path/to/script.sh"
```

A.2.2 Configuration

Inline's behavior is easily customizable by using argument flags.
The accepted configuration parameters are listed in the table below.

Name	Description	Default Value
`in-file`	Input script file (`<FILE>`). If the file does not exist, an error is thrown.	
`out-file`	Output script file (`<FILE>`). If a file with the same name and path already exists, an error is thrown. Unless the `--overwrite` argument is given, the latter is always true (see option below). If no value is specified, the name of the output file is determined by examining the original input file. The first portion is the original name, followed by a `.inlined` string, and finally, if present, the extension (usually `.sh`).	`<NAME>.inlined[EXTENSION]`
`overwrite`	Replace the input file. The inlined result replaces the original file content. This is the same as setting the output file's value as the original input file, except that the existence check is bypassed (see option above).	`false`

[12] `https://wikipedia.org/wiki/Shebang_(Unix)`

`log-level`	Logging level (`<LEVEL>`). Attachment C provides additional information regarding logging and logging levels. The following logging levels are supported (listed in descending order of importance): ⑤ `silent` ④ `fatal` ③ `warn` ② `info` ① `debug`	`info`
`disable-color`	Disable terminal colors (enabled by default).	`false`
`help`	Display a help message and terminate (successfully).	

Table A.2: Inline configuration parameters

A.2.3 Example

This section provides an example of how Inline works and how it may be used.
There are two script files, both of which begin with a shebang. There is a print command in the first file, `hello.sh`, that outputs `Hello` and an empty space, followed by a source command that includes the script file `world.sh`. The format of the second file, `world.sh`, is the same as the first, except that the print command outputs `World!` and the newline character (`\n`), and there are no source commands.
The following two listings show the contents of the two files:

```sh
#!/usr/bin/env sh

printf "Hello "

. "world.sh"
```

Listing A.3: Input script `hello.sh`

```sh
#!/usr/bin/env sh

printf "World!\n"
```

Listing A.4: Sourced script `world.sh`

The goal is to inline the script file `hello.sh` and produce an output file that has no source commands. When the command `./inline.sh --in-file "hello.sh"` is executed, the following result is obtained:

```sh
#!/usr/bin/env sh

printf "Hello "

# . "world.sh"

printf "World!\n"
```

Listing A.5: Inlined script `hello.inlined.sh`

Note the unique shebang as well as the source command that has been commented out with the symbol `#`. Furthermore, because no `--out-file` option is used, the final output file is named `hello.inlined.sh`, which does not override the original input file.

A.3 GraphQL Auth Directive

Available at `https://github.com/carlocorradini/graphql-auth-directive`
Used in the `GraphQL` API (see section 3.3.2).

A custom GraphQL directive[13] that protects resources from unauthenticated and unauthorized access in high-security contexts. It is available in all major Node.js registries as graphql-auth-directive. A directive is an identifier preceded by a @ character, optionally followed by a list of named arguments, which can appear after almost any form of syntax in the GraphQL query or schema languages.
The GraphQL context[14] holds all important information about the current request. A GraphQL context is an object that is shared by all resolvers in a given execution. It helps store data such as authentication information, the current user, database connections, data sources, and other information required to operate the business logic. It is important to note that the context does not have to follow a predefined structure; rather, it is highly flexible to the user's implementation.

A.3.1 Configuration

Before applying the directive to the GraphQL schema, it must be built/configured. The library exposes a utility function called `buildAuthDirective` that accepts a configuration object with the values provided in the table below and returns the type definition (see section A.3.2) as well as a transformer function that applies the auth logic to the executable GraphQL schema.

Name	Description	Default Value
`name`	Directive name. If a name different from the default is specified, it must be reflected in the schema where the directive is used, otherwise, an error is thrown.	`auth`
`auth`	A function or class that handles authentication and authorization. The current context (which contains information about the current request) and the roles and permissions required by the requested resource must be accepted as arguments by the implementation. If access to the requested resource is granted, the boolean value `true` is returned; otherwise, `false` is returned if access is denied. A default basic auth function is already implemented and checks for the existence of an authorized applicant and that its roles and permissions overlap with those of the requested resource.	

[13] https://spec.graphql.org/October2021/#sec-Language.Directives
[14] https://the-guild.dev/graphql/modules/docs/essentials/context

`authMode`	Auth mode if access is not granted. Methodology for informing the requestor that access to the resource has been denied. It is often desirable to hide the important information that the request is failing due to an auth check and instead deliver an error/informational response stating that the requested resource simply does not exist. The first mode, `ERROR`, throws an authentication or authorization error (see below), whereas the second mode, `NULL`, returns the value `null` (empty).	`ERROR`

`roles`	Roles configuration.
	An object with two properties:
	• `enumName`
	Defines the array type, which is by default a `String`.
	It is standard practice in `GraphQL` and in programming languages to map a set of values to an enum. An enumeration type[15] is a special kind of scalar that is restricted to a particular set of allowed values.
	This option restricts the allowed values of roles to a specific set rather than a general string.
	• `default`
	Roles that are required by default.
	No roles are required by default. As a result, access to a protected resource can only be provided with authentication. Overriding this option enables authorization by default, requiring the requestor to have at least one matching role.
		`enumName: String`
		`default: []`

`permissions`	Permissions configuration.
	An object with the same properties as roles configuration (see above).
		`enumName: String`
		`default: []`

`authenticationError`	Authentication error class. The error is thrown if there is no authenticated requestor in the current context. By default, a generic error is thrown. The class must extend the `Error` class.	`AuthenticationError`
`authorizationError`	Authorization error class. The error is thrown if the roles and/or permissions of the requestor do not overlap with those of the requested resource. By default, a generic error is thrown. The class must extend the `Error` class.	`AuthorizationError`
`container`	Dependency Injection Container. Dependency injection is a design pattern that shifts the responsibility of resolving dependencies to a dedicated dependency injector that knows which dependent objects to inject into application code[25]. It should be noted that dependency injection is only available if `auth` is a class type.	`IOCContainer`

Table A.3: GraphQL auth directive configuration parameters

A.3.2 Type Definition

GraphQL has its language for writing GraphQL Schemas: the GraphQL Schema Definition Language[16] (SDL). SDL is incredibly powerful and expressive while being simple and intuitive to use. The syntax is well-defined and is included in the official GraphQL specification[17].
The Type Definition of the GraphQL auth directive generated with default settings is shown below:

```
"""
Protect the resource from unauthenticated and unauthorized access.
"""
directive @auth(
  """
  Allowed roles to access the resource.
  """
  roles: [String!]! = []
  """
  Allowed permissions to access the resource.
  """
  permissions: [String!]! = []
) on FIELD | FIELD_DEFINITION | OBJECT
```

Listing A.6: GraphQL auth directive Type Definition

The above definition must be available in every `GraphQL` schema that utilizes the directive, otherwise, an error is thrown during the server bootstrap operation.
As stated in section A.3.1, the definition is generated dynamically by a set of configuration options or their default value. The name (`@auth`), `roles` and `permissions` type (non-nullable `String` array), and default value (an empty array `[]`) are easily distinguishable.

[15] https://spec.graphql.org/October2021/#sec-Enum-Value
[16] https://www.prisma.io/blog/graphql-sdl-schema-definition-language-6755bcb9ce51
[17] https://spec.graphql.org

A.3.3 Example

Consider a GraphQL backend application that requires authentication and authorization. The sections of the GraphQL schema where the auth directive is used, as well as the enums that compose the different roles and permissions mappings are listed below:

```
enum Role {
  ADMIN
  SIMPLE
}

enum Permission {
  VIEW
  EDIT
}

directive @auth(
  roles: [Role!]! = []
  permissions: [Permission!]! = []
) on FIELD | FIELD_DEFINITION | OBJECT

type Query {
  unprotected: String!
  protected(arg: Boolean): Int! @auth
  secret: Float! @auth(roles: [ADMIN], permissions: [VIEW])
}
```

Listing A.7: GraphQL schema with auth directive and mappings

The available roles are represented by the enum `Role`: `ADMIN` for administrators and `BASIC` for simple users. The supported permissions are represented by the enum `Permission`: `VIEW` to allow resource visualization and `EDIT` to allow resource modification.
The definition of `@auth` directive accepts as roles an array of `Role` whose default value is empty and as permissions an array of `Permission` whose default value is empty. Notice the difference from the default type definition shown in section A.3.2.
Three queries are supported:

1. `unprotected: String!`
   `unprotected` does not accept any arguments and returns a non-nullable string. Does not require any authentication or authorization. As a result, it may be called by anyone.

2. `protected(arg: Boolean): Int! @auth`
   `unprotected` accepts a boolean optional parameter named `arg` and returns a non-nullable integer number. Because no roles or permissions are specified in the directive, it requires only authentication but no authorization. As a consequence, everyone who has been authenticated is allowed to call this query.

3. `secret: Float! @auth(roles: [ADMIN], permissions: [VIEW])`
   The most secure of the three, `secret`, does not take any parameters and returns a non-nullable float integer. The caller must have an admin role with view permission to use this query. As a consequence, the query is secured by both authentication and authorization, resulting in higher requirements to perform the desired action on the resource.

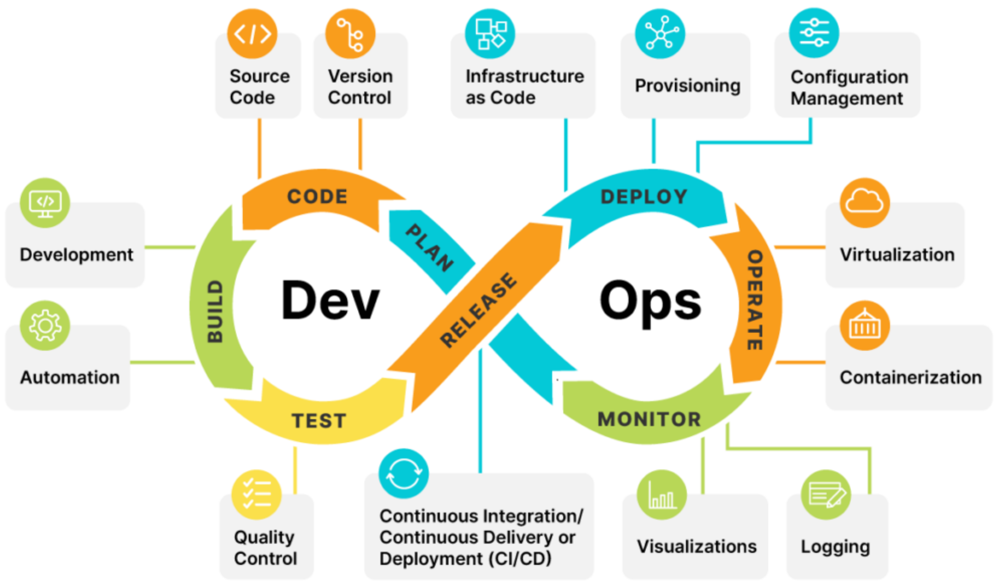
Appendix B Good Practices

Many common software techniques, processes, and applications were used throughout cluster implementation to improve overall code quality and to better and easily manage the Git[1] repository automatically. It would be virtually impossible to list and explain them all in detail. Also, there are several guides and publications available on them. Therefore, this attachment will just briefly cover the two most well-known and widely used techniques that have been used together, without going into much detail.
Finally, it is explained how the cluster implementation's final release archive bundle is generated using a YAML configuration file and a POSIX script.

B.1 DevOps

DevOps is a technique that integrates and automates Software Development (Dev) and IT Operations (Ops). The combination of cultural philosophies, practices, and tools improves an organization's capacity to provide applications and services at high velocity: changing and enhancing products at a quicker rate than traditional software development and infrastructure management procedures. Implementing a DevOps paradigm for a project can result in significant benefits such as increased speed, rapid delivery, reliability, scale, improved collaboration, and security[3]. Figure B.1 depicts the cyclical collection of DevOps phases. Depending on their primary goal, organizations might prioritize different aspects of the DevOps paradigm.

DevOps is essential in modern software development and is intrinsically linked to the Continuous Practices discussed in the next section.



Source: `https://www.edureka.co/blog/what-is-devops`

Figure B.1: Cyclical collection of DevOps phases

[1]`https://git-scm.com`

B.2 Continuous Practices

Continuous Practices also referred to as Continuous Integration, Delivery, and Deployment, are software development industry strategies that enable organizations to release new features and products on a regular and consistent basis while simultaneously keeping the code repository in a consistent state[17]. When a new change is made to the code and pushed to the main repository, it is automatically checked, compiled, and tested to guarantee code standards, consistency, and that there are no potentially breaking changes in the prior application's behavior. The relationship between the three Continuous Practices is depicted in figure B.2.

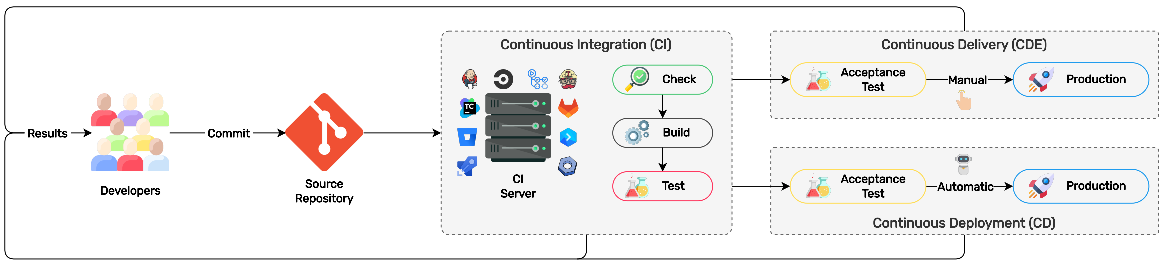


Figure B.2: Relationship between Continuous Integration, Delivery and Deployment

Two automatic workflows to manage Continuous Practices on the source repository have been implemented in the cluster implementation using GitHub Actions[2]. The first workflow[3] executes CI and CDE operations, while the second workflow[4] scans the code and generates security alerts if any known security vulnerabilities are discovered. Furthermore, Dependabot[5] has been added to keep all project dependencies up to date and to inform when a security vulnerability is identified on a certain installed version.
Because of the use of Git Hooks[6], a kind of local CI pipeline that examines every file before committing has also been developed[7]. Because it may be skipped with a configuration flag, this procedure does not substitute a true CI server.
The previously mentioned CI/CDE workflow fully automates the creation of a new release[8]. When a new Git Tag[9] is pushed to the main repository and the CI pipeline succeeds, the current cluster implementation code is bundled (see section B.3) and released with the same name as the Git Tag. In the future, a CD pipeline will be implemented that automatically produces a prerelease anytime a new change in the code is made, ensuring that the newest nightly code is always available, even if it is not as stable as a tagged release.
It should be noted that all prior implementations of Continuous Practices rely on the GitHub platform. Yet, there are equivalent implementations for practically every other coding platform.

B.2.1 Continuous Integration

Continuous Integration (CI) is a well-established development process in the software development industry in which team members regularly integrate and merge development work (e.g., code). CI allows software organizations to have shorter and more frequent release cycles, enhance software quality, and raise overall productivity. This practice involves automated software checking, building and testing[17].

[2]`https://docs.github.com/actions`
[3]`https://github.com/carlocorradini/reCluster/blob/main/.github/workflows/ci.yml`
[4]`https://github.com/carlocorradini/reCluster/blob/main/.github/workflows/codeql.yml`
[5]`https://github.com/dependabot/dependabot-core`
[6]`https://git-scm.com/book/en/v2/Customizing-Git-Git-Hooks`
[7]`https://github.com/carlocorradini/reCluster/blob/main/.husky/pre-commit`
[8]`https://github.com/carlocorradini/reCluster/releases`
[9]`https://git-scm.com/book/en/v2/Git-Basics-Tagging`

B.2.2 Continuous Delivery

Continuous Delivery (CDE) aims to ensure that an application is always ready for production after passing automated tests and quality checks. CDE leverages a collection of processes, such as CI and deployment automation, to deliver software automatically to a production-like environment. This method has various advantages, including fewer deployment risks, cheaper costs, and faster user feedback. Having a Continuous Delivery approach necessitates a Continuous Integration process.[17].

B.2.3 Continuous Deployment

Continuous Deployment (CD) deploys an application to production or customer environments automatically and continuously. What distinguishes Continuous Deployment from Continuous Delivery is the presence of a production environment (i.e., actual customers): the purpose of Continuous Deployment practice is to automatically and consistently deploy every update into the production environment. It's important to note that CD practice implies CDE practice, but not the opposite. While the final deployment in CDE is a manual process, there should be no manual steps in CD, where changes are pushed to production as soon as developers commit them via a deployment pipeline.
CDE is a pull-based approach in which an organization determines what and when to deploy; CD is a push-based approach. In other words, CDE's scope excludes frequent and automatic release, and CD is therefore a continuation of CDE. While CDE may be applied to all kinds of systems and organizations, CD may only be appropriate for particular types of organizations or systems[17].

B.3 Bundle

To distribute a release of the cluster implementation, a mechanism that bundles all required files and directories in a single archive file have been implemented. The bundle is created automatically by the CDE workflow pipeline and uploaded on GitHub with the name `recluster.tag.gz`. The archive file is composed of a configuration file (see section B.3.1) that lists all necessary files and directories, and a POSIX script (see section B.3.2) reads the configuration file and generates the archive file, which may then be published.
It should be noted that the CDE pipeline is not required to build a bundle. The latter enables organizations that do not wish to use the workflow to generate a bundle or to locally test a cluster with a different implementation without involving any workflow.

B.3.1 Configuration

The bundle configuration file, `bundle.config.yaml`, is written in `YAML` format.
The attributes structure is a mapping that adheres to the development project's tree hierarchy organization of files and directories[10]. The key of an attribute is the file or directory name, and the value can be of two types:

1. `boolean`
   The `true` value indicates that the file or directory should be copied (recursively), whereas the `false` value indicates that the file or directory should be ignored.
   When the value is `true` and it is a directory, the included files and directories are copied in the order specified by the Git index and working tree[11].

2. `mapping`
   Applicable exclusively to a directory (parent), defines a mapping of files and directories that are contained in the parent directory.
   A special metadata attribute, defined as `__`, does not reflect a directory tree mapping and is instead employed by the script for specific operations and management that are applied to the current directory where the special attribute is located. At the moment, the only attribute

[10] `https://github.com/carlocorradini/reCluster`
[11] `https://git-scm.com/docs/git-ls-files`

that can be defined in the metadata is `run`, which provides an inline POSIX command. When generating the bundle file, all metadata attributes are examined and, if a `run` is identified, it is executed. It should be noted that the working directory for `run` is always the directory containing the metadata attribute. If the first two characters are `./`, the absolute path of the project's root directory is substituted. This facilitates the execution of scripts from various directories.

Listing B.1 displays the contents of the `bundle.config.yaml`[12] bundle configuration file. It is worth noting the use of the metadata attribute `__` in combination with the `run` attribute for executing applications and/or programs that provide various outputs required for the release bundle.

```yaml
__:
  run: './scripts/inline.sh --in-file ./install.sh --overwrite'
install.sh: true
LICENSE: true
README.md: true

configs:
  README.md: true
  certs: true
  k3s: true
  k8s:
    README.md: true
    autoscaler: true
    loadbalancer:
      __:
        run: 'wget --output-document=deployment.yaml https://raw.githubusercontent.com/metallb
                /metallb/v0.13.7/config/manifests/metallb-native.yaml'
      config.yaml: true
      deployment.yaml: true
      README.md: true
    registry: true
  node_exporter: true
  recluster: true
  ssh: true

dependencies:
  __:
    run: './dependencies/dependencies.sh --sync-force'
  autoscaler: true
  k3s: true
  node_exporter: true
  prometheus: true

distributions:
  alpine:
    __:
      run: './distributions/alpine/build.sh'
    README.md: true
    logo.png: true
    iso: true
  arch:
    __:
      run: './distributions/arch/build.sh'
    README.md: true
    logo.png: true
    iso: true

docs: true

scripts:
  __:
    run: './scripts/inline.sh --in-file ./certs.sh --overwrite
            && ./scripts/inline.sh --in-file ./configs.sh --overwrite'
```

[12] https://github.com/carlocorradini/reCluster/blob/main/scripts/bundle.config.yaml

```yaml
  README.md: true
  certs.sh: true
  configs.sh: true
  configs.config.yaml: true

server:
  __:
    run: 'npm run build'
  README.md: true
  package.json: true
  package-lock.json: true
  build: true
  prisma:
    schema.prisma: true
    migrations: true
```

Listing B.1: Content of bundle configuration file

B.3.2 Script

To automate all bundle-related procedures, a POSIX script named `bundle.sh`[13] is used. Its primary function is to read the bundle configuration file, read metadata attributes, and generate a bundle archive file containing all files and directories specified therein.
The script requires some `coreutils` package's utility programs and the `yq` application to correctly handle `YAML` file and syntax.
application.
The script behavior of the bundle can be changed using parameter flags. The acceptable setup settings are provided in the table below.

Name	Description	Default Value
`config-file`	Path to the bundle configuration file (`<FILE>`). Both relative and absolute paths are supported. It should be noted that the configuration file name does not have to be `bundle.config.yaml`, but may be any name and extension. The sole condition is that it must be in `YAML` format and that the attributes structure is respected.	`bundle.config.yaml`
`out-file`	Name of the file path where the generated bundle archive file should be saved (`<FILE>`). Absolute and relative paths are both supported. By default, it is saved in the current working directory. The extension name should always be `.tar.gz` to distinguish a compressed tar archive file.	`bundle.tar.gz`
`skip-run`	Skip all commands in the metadata attribute `run`.	`false`
`log-level`	Logging level (`<LEVEL>`). Attachment C provides additional information regarding logging and logging levels. The following logging levels are supported (listed in descending order of importance): ⑤ `fatal` ④ `error` ③ `warn` ② `info` ① `debug`	`info`

[13] `https://github.com/carlocorradini/reCluster/blob/main/scripts/bundle.sh`

`help`	Display a help message and terminate (successfully).	

Table B.1: Bundle script parameters

To generate an archive bundle file, named `recluster.tar.gz`, that contains all files and directories listed in the configuration file, just execute: `./bundle.sh --out-file "recluster.tar.gz"`

Appendix C Logging

Logging[1] is the process of preserving a record of events that occur in a computer system, such as issues, errors, or just information on current operations. These events might happen in the operating system or other applications. For each such event, a message or log entry is recorded. These log messages may subsequently be used to monitor and understand the system's operation, troubleshoot problems, or during an audit. Logging is very crucial in multi-user applications to provide a centralized view of the system's functionality.
It is critical in the architecture implementation to log the numerous operations that occur in each component of the cluster. The latter not only assists in understanding how the general architecture performs and its status, but also the potential causes of an incident. During the cluster's development, if any components failed or did not behave as they should (e.g., no autonomous upscaling or downscaling), monitoring and analyzing the numerous log files created was the only method to solve the problems and understanding what and why happened. Furthermore, a logging system is available for the multitude of utility scripts and applications that are not directly employed with the cluster but rather during its general development.
Log interpretation is a challenging process. Log management system collects data from a variety of sources with varying forms, purposes, and granularities. Section C.1 describes common logging levels that are also employed in cluster development and implementation.

C.1 Levels

A log level[2] is a piece of information that indicates the importance of a log message. It is a basic yet effective method of identifying log events from one another. Maintaining defined log levels helps give each entry meaning and better understand the significance of the related log message. Furthermore, it is utilized to filter crucial information regarding the system state to only those that are solely informational.
During the initialization phase, a level is chosen from the list below (sorted in descending order of importance), with the expectation that the lower levels are ignored in favor of the higher ones.

⑦ `SILENT`
The highest possible logging level is meant to completely disable logging.

⑥ `FATAL`
The system experienced a very severe error, which caused the system to abort.

⑤ `ERROR`
Employed when the system encounters a problem that prevents one or more operations from functioning properly but allows it to continue operating.

④ `WARN`
Designates potentially dangerous conditions that do not cause the system to crash.

③ `INFO`
The standard log level indicates informational messages about the progress at a coarse-grained

[1] `https://wikipedia.org/wiki/Logging_(computing)`
[2] `https://sematext.com/blog/logging-levels`

level. The information reported using the `INFO` log level should be strictly informative, without the need for any essential information to be lost.

② `DEBUG`
Used for information that may be required for diagnosing and debugging issues, or for running in a test environment to ensure that everything is working properly.

① `TRACE`
The most fine-grained level of information is only employed in rare circumstances where full visibility of what is occurring is required. `TRACE`'s logging level is quite verbose.

Table C.1, shows the correlation between the logging level and the related logging message. The default logging level in various implementations is often set to `INFO` level.

Logging Level \ **Logging Message**	`FATAL`	`ERROR`	`WARN`	`INFO`	`DEBUG`	`TRACE`
⑦ `SILENT`	🟥	🟥	🟥	🟥	🟥	🟥
⑥ `FATAL`	🟩	🟥	🟥	🟥	🟥	🟥
⑤ `ERROR`	🟩	🟩	🟥	🟥	🟥	🟥
④ `WARN`	🟩	🟩	🟩	🟥	🟥	🟥
③ `INFO`	🟩	🟩	🟩	🟩	🟥	🟥
② `DEBUG`	🟩	🟩	🟩	🟩	🟩	🟥
① `TRACE`	🟩	🟩	🟩	🟩	🟩	🟩

Table C.1: Correlation between logging level and logging message
🟩 Logging message is recorded and saved
🟥 Logging message is completely ignored