ST. PÖLTEN UNIVERSITY
OF APPLIED SCIENCES

Informatik
& Security

/fh/// st.pölten

Shifting the Orchestration Monoculture

A Practical Evaluation of Alternatives to Kubernetes as Container Orchestration Framework for Distributed Web Systems

Master thesis

for attainment of the academic degree of

Master of Science in Engineering (MSc)

submitted by

Casper De Keyser

52117924

in the

University Course Cyber Security and Resilience at St. Pölten University of Applied Sciences

Supervision

Advisor: FH-Prof. Dipl.-Ing. Dipl.-Ing. Christoph Lang-Muhr, BSc

Assistance: -

Declaration of Honour

First name, Surname: Casper De Keyser

Matriculation number: 52117924

Title of the thesis: Shifting the Orchestration Monoculture

I hereby declare that

- I have written the work at hand on my own without help from others and I have used no other resources and tools than the ones acknowledged
- I have complied with the Standards of good scientific practice in accordance with the St. Pölten UAS' Guidelines for Scientific Work when writing this work.
- I have neither published nor submitted the work at hand to another higher education institution for assessment or in any other form as examination work.

Regarding the use of generative artificial intelligence tools such as chatbots, image generators, programming applications, paraphrasing and translation tools, I declare that

☐ no generative artificial intelligence tools were used in the course of this work.

☒ I have used generative artificial intelligence tools to proof-read this work.

☐ I have used generative artificial intelligence tools to create parts of the content of this work. I certify that I have cited the original source of any generated content. The generative artificial intelligence tools that I used are acknowledged at the respective positions in the text.

Having read and understood the St. Pölten UAS' Guidelines for Scientific Work, I am aware of the consequences of a dishonest declaration.

Abstract

Container orchestration has become a crucial component for deploying distributed applications at scale. Within this domain, Kubernetes' popularity has established a "monoculture", resulting in it becoming the "go-to" solution for many orchestration scenarios. However, Kubernetes' exhaustive feature set and its intrinsic complexity may not align with the specific requirements of every use-case or development team, thereby complicating migration or implementation efforts. Other technologies, with a more significant focus on simplicity and ease of use rather than advanced customization and accompanying configuration, could present a more suitable solution for these specific cases.

This thesis investigates two potential alternatives—Docker Swarm and HashiCorp Nomad—through a practical evaluation based on a predefined use-case containing typical production-grade elements. Both candidates result from a review of the current state of the art in orchestration frameworks, focusing on relevant and open-source technologies that satisfy the use-case's specified requirements. By evaluating each candidate in detail against common orchestration features such as scaling, load balancing, service discovery and networking, we uncover valuable insights, which result in a final verdict on both candidates' suitability for this specific use-case. We find that Swarm's orchestration capabilities meet the requirements with minimal additional configuration required. Nomad demands a more significant investment in terms of configuration and knowledge, although its advanced feature set appears more geared towards deployments on a larger scale. Finally, we acknowledge that future research could result in insights that benefit the broader orchestration landscape, particularly in terms of more advanced application stacks leveraging components—such as a dedicated reverse proxy in case of Swarm, or Consul's service mesh in case of Nomad's ecosystem.

Preface

This thesis marks the conclusion of my academic journey in the master's program "Cyber Security and Resilience" at the St. Pölten University of Applied Sciences. During this period, I had the opportunity to pursue my passion for security and computing, delving deeper in many interesting topics that will only become more relevant in the future. This thesis reflects my background in software engineering, with my growing curiosity in the fields of DevOps and SRE, and I am grateful that I had the opportunity to follow my interests as part of this research project.

I would like to sincerely thank my supervisor, Christoph Lang-Muhr, for his feedback and expertise during the entire process of writing this thesis, from the initial brainstorm phases to his suggestions for final adjustments. Additionally, I would like to thank Raphael Schrittwieser for his support during the practical implementation using the university's infrastructure. I would like to express my gratitude to Simon Tjoa for his guidance and open communications during the span of the entire master degree. Lastly, I would like to thank the people closest to me, in particular my parents, my brother and my partner. Their unwavering support and encouragement kept me going during the more challenging parts of this journey, and I could not have completed this work without their belief in me, for with I am eternally grateful.

Contents

1 Introduction . . . **1**

1.1 Contribution . . . 2

1.2 Thesis Outline . . . 2

1.3 Research Questions . . . 3

1.4 Hypotheses . . . 3

2 Background . . . **5**

2.1 A Brief Overview of Public Cloud Computing . . . 5

2.2 A Brief Introduction to Microservices . . . 6

2.2.1 Monolithic Architecture . . . 6

2.2.2 Microservice Architecture . . . 8

2.2.3 Characteristics of Microservices . . . 9

2.3 Containerization . . . 11

2.3.1 Virtual Machines . . . 11

2.3.2 Containers . . . 12

2.4 Orchestration of Containers . . . 14

2.4.1 Container Orchestration Frameworks . . . 14

2.4.2 Kubernetes . . . 16

3 Related Work . . . **17**

3.1 Challenges Related to Kubernetes . . . 17

3.1.1 Context . . . 17

3.1.2 Related Works . . . 17

3.1.3 Discussion . . . 18

3.2 Managed Kubernetes Cloud Services . . . 19

3.2.1 Context . . . 19

3.2.2 Related Works . . . 19
3.2.3 Discussion . . . 19
3.3 Comparisons between Container Orchestration Frameworks . . . 20
3.3.1 Context . . . 20
3.3.2 Related Works . . . 20
3.3.3 Discussion . . . 21

4 Methodology . . . 23
4.1 State of the Art . . . 23
4.2 Design of Practical Use-Case . . . 23
4.3 Evaluation of Container Orchestration Features and Technologies . . . 24
4.4 Qualitative Analysis of Container Orchestration Frameworks . . . 24
4.5 Overview of Methodology . . . 25

5 Use-Case Definition and Candidate Selection . . . 27
5.1 Use-Case Definition . . . 27
5.1.1 Motivation . . . 27
5.1.2 High-Level Overview and Description . . . 28
5.1.3 Components . . . 31
5.2 Feature Selection . . . 33
5.2.1 Container Orchestration Features . . . 33
5.2.2 Features Scoped to Use-Case . . . 35
5.2.3 Overview of Selected Features . . . 36
5.3 Candidate Selection . . . 36
5.3.1 Additional Criteria . . . 37
5.3.2 Evaluation of Candidates . . . 38
5.3.3 Overview of Selected Candidates . . . 41

6 Implementation . . . 43
6.1 Testing Environment . . . 43
6.2 Initial Use-Case Implementation . . . 44
6.2.1 Databases . . . 44
6.2.2 Web APIs . . . 45

6.2.3 Websites . . . 45
6.2.4 OT Application . . . 46
6.2.5 Monitoring . . . 46
6.2.6 CI/CD . . . 47
6.2.7 Overview . . . 47
6.3 Evaluation Criteria . . . 48
6.3.1 List of Criteria . . . 48
6.3.2 Grading Schema . . . 48
6.4 Docker Swarm . . . 49
6.4.1 Initial Configuration . . . 49
6.4.2 Shared Storage . . . 50
6.4.3 Swarm Setup . . . 51
6.4.4 Scheduling . . . 51
6.4.5 Scaling . . . 52
6.4.6 Service Discovery . . . 53
6.4.7 Fault Tolerance . . . 54
6.4.8 Networking Segmentation . . . 55
6.4.9 Continuous Deployments . . . 56
6.4.10 Load Balancing . . . 57
6.4.11 Encrypted Communication . . . 58
6.4.12 Secret Management . . . 59
6.5 HashiCorp Nomad . . . 60
6.5.1 Initial Configuration . . . 60
6.5.2 Shared Storage . . . 60
6.5.3 Nomad Setup . . . 60
6.5.4 Scheduling . . . 61
6.5.5 Scaling . . . 63
6.5.6 Service Discovery . . . 63
6.5.7 Fault Tolerance . . . 66
6.5.8 Networking Segmentation . . . 67
6.5.9 Load Balancing . . . 69
6.5.10 Continuous Deployments . . . 72

6.5.11 Encrypted Communication 75
6.5.12 Secret Management 78

7 Results and Discussion 81
7.1 Kubernetes' Complexity 81
7.2 Orchestration Features 83
7.3 Orchestration Frameworks 83
7.4 Suitability of Orchestration Candidates to Use-Case 83
7.4.1 Feature Comparison 84
7.4.2 Grading 89
7.4.3 Verdict 90
7.4.4 Review of Hypotheses 91
7.4.5 Review of Implementation Process 92

8 Conclusion 93
8.1 Insights with respect to Kubernetes 93
8.2 Insights with respect to Swarm 94
8.3 Insights with respect to Nomad 94
8.4 Impact of the Findings 95
8.5 Future Work 96

List of Figures 99

List of Tables 100

List of Listings 102

Acronyms 105

Bibliography 109

A Implementation Details 123

1. Introduction

In recent years, the adoption of cloud-native technologies has changed the way software applications are being developed and deployed. Within this movement, containerized workloads have become more prevalent in all kinds of environments, from high-performance machine learning clusters to smaller side-projects and home-lab configurations. Both Docker and Kubernetes have come out on top as the de facto standards when talking about these modern application architectures, with both technologies even becoming synonymous with the concepts of containerization and container orchestration respectively. With this change in application architecture, different issues arise when it comes to the configuration and management of these frameworks, which can become especially challenging when talking about Kubernetes. As a result, the requirements for development teams and consequently their organizations also change, because there is a need for skilled personnel to navigate and manage these complex environments.

As it is often the case with IT projects, it might be beneficial to take a step back and evaluate which tools are available in order to select a suitable solution for a specific task or problem. Choosing the right tool for the job is crucial in today's complex IT landscape. Docker and Kubernetes are both disruptive technologies that have revolutionized the way we operate computing workloads at scale, but that does not mean they are "one-size-fits-all" solutions. Kubernetes is often described as "having a steep learning curve", with the need to dedicate lots of time and resources in order to achieve the desired setup. Moreover, not all production scenarios require a full-fledged Kubernetes cluster in order to achieve their scalability and availability goals. In that case, the extra complexity can be avoided if the requirements and available tools are taken into account prior to the design and implementation. Ultimately, choosing the right tool serves as a crucial step in the design of modern applications—after all, one would not use an axe to hit a nail in a block of wood.

In this thesis, we compare different container orchestration frameworks through a practical use-case that highlights the distinct characteristics of each framework. This comparison aims to provide readers with a clear overview of which technologies may be suitable for their specific orchestration requirements.

1.1. Contribution

This research project proposes the following contributions to the state of the art in the domain of container orchestration:

- A clear understanding of Kubernetes' complexity, where it occurs and how the subtopics are connected.
- A model use-case consisting out of typical elements that need to be addressed by a container orchestration framework.
- An in-depth evaluation of two container orchestration frameworks as potential Kubernetes alternatives for similar use-cases to the one defined in this thesis.

1.2. Thesis Outline

This thesis can be described with the following structure, providing a guiding point to navigate throughout its different facets:

- **Introduction**: As discussed earlier, the introduction (referenced as chapter 1) provides an overview of this thesis' main topic, motivation and problem statement.
- **Background**: This section (referenced as chapter 2) covers necessary prerequisites and fundamental knowledge in the domain of container orchestration.
- **Related Work**: The related work section (referenced as chapter 3) presents existing research and literature relevant to this thesis, enabling us to understand the current container orchestration landscape.
- **Methodology**: In this part (referenced as chapter 4), we delve into the methodology employed in order to present an answer to this thesis' research questions.
- **Use-Case Definition and Candidate Selection**: This section (referenced as chapter 5) discusses the use-case defined in order to support this thesis' evaluation approach. Additionally, it covers the selection of features and candidates subjected to the implementation.
- **Implementation**: The implementation section (referenced as chapter 6) details the approach taken during the practical testing of the selected candidates.
- **Results and Discussion**: We report the results of the experiments in the section labeled chapter 7.
- **Conclusion**: Finally, we draw conclusions and summarize the key findings and contributions of this thesis in chapter 8 and highlight potential future research opportunities.

1.3. Research Questions

This section contains the four research questions (RQ) that encapsulate this research and its different phases. Each RQ attempts to investigate a subdomain of this research, which was discussed in the beginning of this chapter.

- RQ_1: Which aspects of Kubernetes contribute to its perceived complexity?
- RQ_2: Which container orchestration features are essential for this research's defined use-case?
- RQ_3: Which container orchestration frameworks can be considered suitable candidates for this research's defined use-case?
- RQ_4: How do different container orchestration frameworks compare when evaluated against this research's defined use-case?

1.4. Hypotheses

Prior to tackling the aforementioned research questions, we want to establish certain hypotheses in terms of container orchestration (CO) frameworks and their characteristics. For RQ_1, we think that part of Kubernetes' complexity can be attributed to the many versions that exist in the general CO landscape. To the best of our knowledge, we are aware of different versions, such as the original Kubernetes, managed cloud services offered by large cloud service providers (CSP) and some additional offerings aiming at simplifying Kubernetes in some way or form. As possible alternatives, related to RQ_3 and RQ_4, we think that Docker must provide some solution to address CO needs of its users, since it is the most popular containerization platform. We are aware of an offering called OpenShift by Red Hat that leverages Kubernetes as its core technology, while providing certain abstractions and additional services that are supposed to simplify the adoption process of a CO framework in an existing application landscape.

These hypotheses are based on our current knowledge about and limited practical understanding with Kubernetes, Docker and Red Hat OpenShift, and they have no scientific foundation. This knowledge stems mostly from previous industry experience in the domain of software engineering and cloud computing.

2. Background

This chapter contains a concise overview of elements that are mandatory for comprehending the topics discussed in this research. It establishes a foundation upon which we build during the following chapters and serves as the recommended starting point for readers who are new to the domain of container orchestration (CO).

2.1. A Brief Overview of Public Cloud Computing

In 2006, Amazon launched Simple Storage Service (S3) [1] and Elastic Compute Cloud (EC2) [2] on their subsidiary Amazon Web Services (AWS), which can be considered as the first major cloud computing platform. At the time, there were already other providers who offered similar services, such as web hosting and the renting out of virtual private servers (VPS). However, Amazon can be seen a pioneer in this domain, since AWS would become the first of its kind as what we now consider a public cloud service provider (CSP).

Following this release, other leading technology companies followed this trend swiftly. Google released Google App Engine in 2008 [3], which grew into Google Cloud Platform (GCP). In 2010, Microsoft released Azure [4], with other companies such as Oracle [5], IBM [6] and Alibaba [7] launching similar services of their own. The releases of these platforms could be seen as a response to rapidly growing business demands that were incentivized by these new computing possibilities. In 2021, O'Reilly reported a survey where roughly 90% of the 2834 participants indicated that their organizations are using the cloud, with 67% using a CSP [8]. Among these providers, AWS, GCP and Azure were the three biggest providers, which was also supported by Gartner in their Magic Quadrant report on CSPs in 2024 [9]. In an impact analysis report about companies in the Fortune 500[1], 93.6% were reliant on a CSP for at least some of their activity [10]. This evolution brought a shift in the business models of technology companies, but also in the application architecture and development of many of the consuming parties.

[1] `https://fortune.com/ranking/global500/`

The core benefits of the cloud come down to elasticity and availability, where a consumer can utilize a provider's resources on demand. If the consumer's demands change, the provider is able to scale their computing services accordingly, using their more extensive computing infrastructure. This eliminates the need for consuming parties to invest in their own dedicated hardware, and can instead outsource the purchasing, configuration and maintenance to the provider. This process is particularly interesting for smaller businesses and start-ups who want to bring a product to market without making a significant upfront investment. Naturally, this arrangement comes at a certain cost, which is agreed upon in the provider's service level agreement (SLA).

Companies who want to move to the cloud often already have some on-premise computing infrastructure used for hosting their current applications and services. Moving this workload to the cloud using a "lift-and-shift" method, where similar resources are created in the provider's environment so the on-premise infrastructure can be decommissioned, is usually not a cost-effective approach [11], [12]. Prior to this migration step, the application landscape should be evaluated to investigate if applications could be re-architected in such a way that they can leverage the benefits of the cloud more efficiently [13]. Out of the different cloud migration strategies, the refactor or re-architecting step is typically the most complex and resource demanding, but it can yield the most optimal results in the long run. A common architecture pattern that is often leveraged, is the use of so-called "microservices".

2.2. A Brief Introduction to Microservices

A microservice architecture follows an approach where smaller programs, part of a larger application or service, fulfill a specific function. This differs from a traditional monolithic architecture where an application is split up into different layers, depending on their function. The following paragraphs present a concise overview of both application architecture patterns.

2.2.1. Monolithic Architecture

A monolithic architecture or "monolith" often contains three separate application layers: the presentation layer, the business layer and the data access layer [14], [15]. The presentation layer typically contains assets and code that are responsible for providing a user interface (UI). The business layer contains logic that deals with transforming data and other backend related tasks. The data access layer contains code that executes the queries on the database or data storage objects in order to interact with the data used in the application. Various technology stacks provide a template or framework that follows this software pattern, known as a

model-view-controller (MVC) framework. Examples are Django[2] for Python, Ruby on Rails[3], Laravel[4] for PHP, ASP.NET Core MVC[5] for C# and Spring Boot MVC[6] for Java. Figure 2.1 visualizes this three-layer architecture, as described by Tie *et al.* [15].

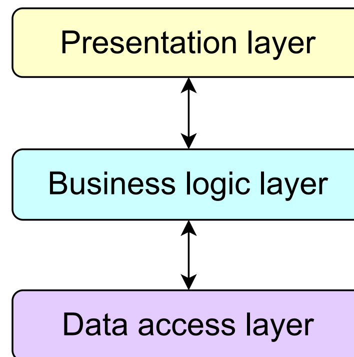

Figure 2.1.: A three layer architecture as derived from Tie *et al.* [15]

This layered architecture forces developers to structure their codebase in a more uniform way. However, the application still gets deployed as one executable or build artifact, which contains the logic for the various functions that it needs to perform. The example scenario presented by Villamizar *et al.* [16, pp. 585–586] can be used in order to illustrate this architecture in a more tangible way.

Suppose an application is being developed for a business venture that offers digital services, for example translations, music recommendations or workout plans. A core requirement for this case is the billing and payment plan management. For the sake of simplicity, it is assumed that the application needs to fulfill two business requirements. As a first functionality, a new payment plan needs to be generated when a user signs up. For a second requirement, users need to be able to get an overview of their current payment plan. Both functions differ in the frequency they are requested and in their response times, because more processing is required for the creation of a payment plan than for fetching it. Table 2.1 shows an overview of both requirements and their varying characteristics.

With the requirements clearly defined, the architecture can be visualized to include other components in fig. 2.2, similarly to the figure presented by Villamizar *et al.* [16, p. 586].

[2] `https://www.djangoproject.com/`
[3] `https://rubyonrails.org/`
[4] `https://laravel.com/`
[5] `https://learn.microsoft.com/en-us/aspnet/core/mvc/overview`
[6] `https://docs.spring.io/spring-boot/index.html`

Functionality	Requests per minute	Avg. response time (ms)	Max. response time (ms)
Creating a payment plan (F_1)	50	1000	5000
Fetching a payment plan (F_2)	1000	250	1150

Table 2.1.: A functionality overview based on example by Villamizar *et al.* [16]

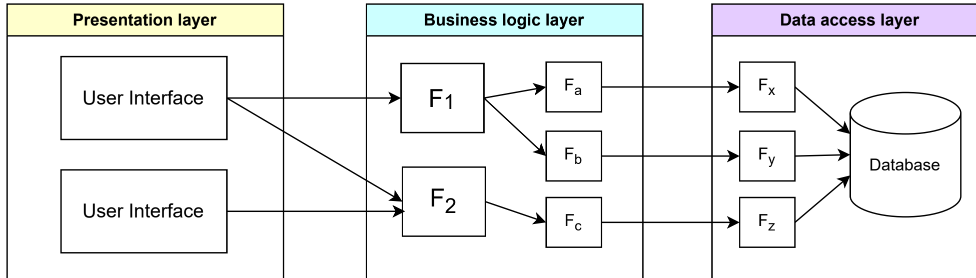

Figure 2.2.: A monolithic architecture based on example by Villamizar *et al.* [16]

In this design, users access the application via the relevant UI component. This component calls the responsible backend function, in this case F_1 or F_2. These functions use other helper functions (i.e. F_a, F_b, F_c in the business logic layer and F_x, F_y, F_z in the data access layer) to accomplish the required task. A possible issue may arise when the fictional business grows in popularity and many new users start creating payment plans. This requires scaling up the relevant part of the application, which can be a complex assignment with the current architecture, since both functionalities are tightly integrated together. Another problem scenario could present itself when the responsible development team for F_2 changes an underlying helper function also used by F_1, resulting in a certain unexpected impact for the owner of F_1.

2.2.2. Microservice Architecture

A microservice is typically a smaller application that is developed and deployed independently [17, p. 2] [18, p. 2]. It fulfills a certain function or business requirement, which should be its main goal and sole reason of existence [19, p. 20359]. Multiple microservices make up a microservice architecture, with each service fulfilling a certain function. Microservices can be configured to communicate with each other, typically using REST APIs[7] over an internal network [16, p. 586], which can be secured with the necessary restrictions such as segmentation and firewalling. In a production scenario, this architecture is often used in conjunction with a gateway service or load balancer, which exposes the services for external access and balance the load between different instances of a service. This topic will be revisited in section 2.4 about

[7] https://ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

orchestration.

This architecture design can be applied to the digital provider use-case which was defined in section 2.2.1. Figure 2.3 visualizes a possible microservice architecture for the previously mentioned example around the digital service provider.

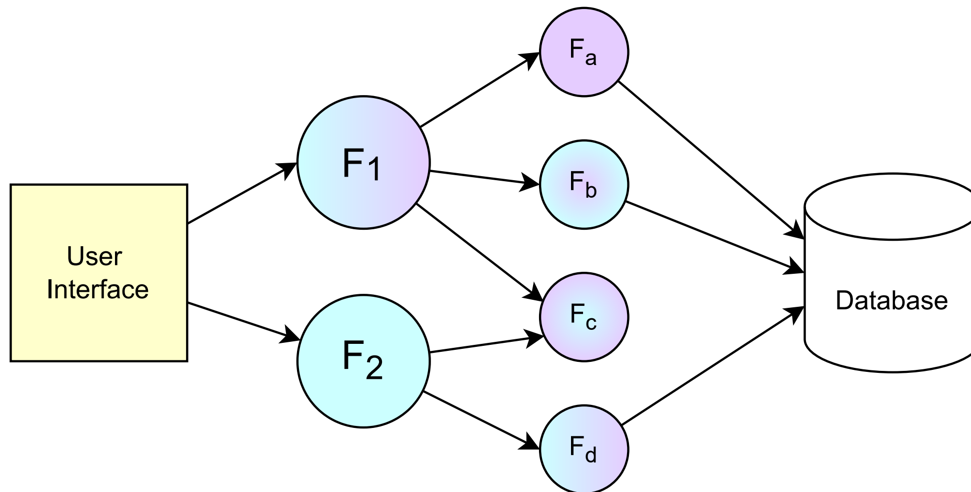

Figure 2.3.: A microservice architecture based on example by Villamizar *et al.* [16]

In this design, every circle represents a function or microservice, where F_1 and F_2 are the main functions which utilize other components in order to complete the required processing steps. These services can be developed, deployed and scaled independently of each other, without impacting the other services. The colors indicate that the different layers are being mixed within one microservice, because they contain all relevant logic to accomplish the required task. This approach supports the application to be more efficient and available in scenarios with varying load, such as during a holiday promotion or a new product launch. In the next section, we dive deeper into the characteristics of microservices.

2.2.3. Characteristics of Microservices

Microservices have certain characteristics that contribute to solving common challenges related to software development and deployment. Typically, microservices have the following traits [16]–[19]:

- Independently developed and deployed
- Loosely coupled with other microservices
- Aimed at solving one task or function
- Ownership lies with one development team or entity

Because microservices have a loose coupling with other microservices, they can be developed and deployed independently of each other. This enables a team, which owns a certain service, to implement features and to make changes without being dependent on other components or teams. This decreases the amount of challenges and effort it takes to update parts of an application or system, because there are fewer parties involved that might need to synchronize their systems with the updated components, resulting in faster and more frequent releases. A microservice can also be scaled independently of other services, which is often a necessary requirement when certain parts of an application are experiencing a heavy load. This offers more flexibility to application owners, because they are able to scale their application both horizontally (i.e. by adding more instances of a service) or vertically (i.e. by assigning more computing resources to a service)[8]. This is one of the main advantages of microservices over monoliths, because monoliths are mainly bound to vertical scaling due to the tight integration of their different components, although horizontal scaling can be achieved with some effort [20, p. 806]. Another advantage of a microservice is that it contains all dependencies necessary to run the application, which increases portability, thereby enabling deployments to virtually any environment, as long as the underlying containerization technology is supported. Due to the high coupling of components in a monolith, a monolith does not support the usage of different programming languages. In contrast, microservices can be considered heterogeneous, because they allow for different technology stacks to be used in conjunction, giving developers the freedom to choose their preferred tool for the task at hand.

Several leading technology companies such as Amazon [21], Netflix [22], [23], Spotify [24] and Uber [25] were early adopters of this emerging architecture. In most cases, they were migrating from a monolithic architecture to a microservice architecture because they needed to address demanding scaling requirements and had to be able to adapt to rapidly changing and competitive markets.

Microservices should also be aimed at solving a particular task or function, which supports a certain business capability. This independent service can be supported by a dedicated team and the relevant business stakeholders. Naturally, there are also challenges that come with the adoption of microservices, such as the need to secure communications between each microservice and the added complexity of managing and coordinating all components of a large application [26, p. 465],[27, p. 2]. However, a comprehensive evaluation of microservices is not in scope for this chapter or research in general. In the next section, we discuss an essential technology which enables the use of microservices on a more technical level.

[8] Horizontal and vertical scaling: `https://www.digitalocean.com/resources/articles/horizontal-scaling-vs-vertical-scaling`

2.3. Containerization

In order to determine how a microservice architecture can be effectively supported, two essential concepts from the computing world need to be examined: virtual machines and containers. They are both related to the virtualization of different infrastructure components, but have some key differences which are important to grasp moving forward.

2.3.1. Virtual Machines

A virtual machine (VM) is an established concept in the domain of computing. A VM is a virtual instance of a system or machine that runs isolated on a host system. It enables machines, with different resource allocation and even operating systems (OS), to run on the same physical server [28]. The technology that facilitates this is referred to as the hypervisor[9]. A hypervisor can be installed directly on a physical server, which is called a type 1 hypervisor (i.e. bare-metal virtualization), or it can be run as a typical software application on the host machine, which is known as a type 2 hypervisor (i.e. host OS virtualization) [29]. It manages the different hardware components (e.g. CPU, RAM, storage, NIC) of the host and assigns them as a shared or dedicated resource to the VMs. Figure 2.4 is based on the figures presented by Morabito *et al.* [29, p. 387] in a comparison between hypervisor and other virtualization techniques, which is covered in section 2.3.2.

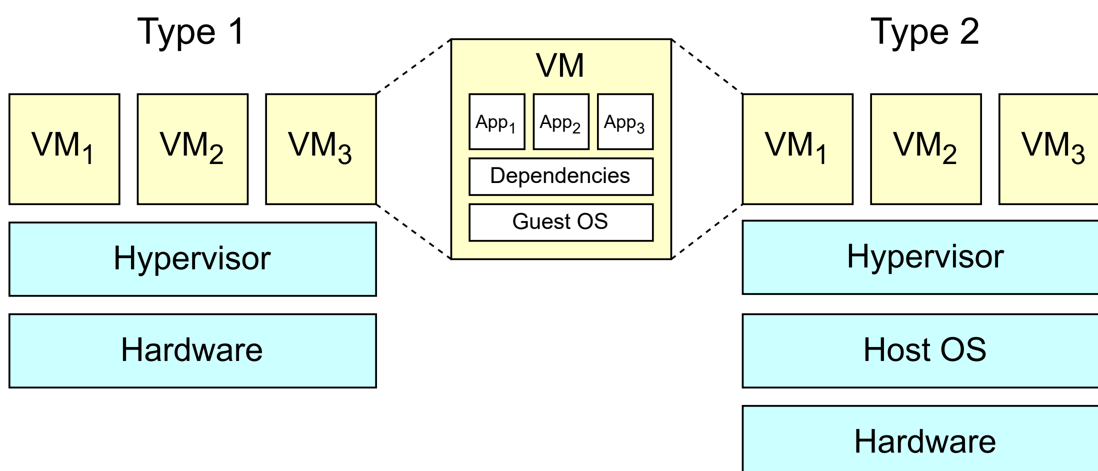

Figure 2.4.: A high-level comparison between bare-metal (type 1) and host OS (type 2) hypervisors

In the above figure, it can be observed that a type 1 hypervisor needs one fewer layer than a type 2 hypervisor, because it is installed on the hardware and can access the resources directly. A type 2 hypervisor requires a

[9] https://www.vmware.com/topics/hypervisor

host OS to be installed upon. In this particular example, each hypervisor hosts three VMs, which each have their own guest OS, dependencies and applications. By default, the VMs are fully isolated from each other and the host OS, however, connectivity between them can be achieved if required for specific scenarios.

VMs offer several advantages to system administrators, because physical hardware can be utilized with more flexibility and for different types of workloads—often in parallel and isolated from each other—which are essential features in scenarios where security and cost optimization are of crucial importance. However, for some applications, to process of running a hypervisor with VMs introduces a significant overhead. In case of a microservice architecture, it would be inefficient to deploy a VM for running each microservice. A more lightweight alternative, discussed in the next section, would be a more optimal solution for a microservice architecture.

2.3.2. Containers

A container presents a more lightweight virtualization approach than a VM, often resulting in a better performance [28], [30]. Instead of leveraging virtualized hardware components as is the case with a VM, containers rely on virtualization on the OS level. A container requires a container engine, which can be installed on the host OS. Applications running in containers share parts of the underlying OS (e.g. kernel, binaries, and libraries), which results in smaller deployments than in a hypervisor scenario [31, p. 82]. Because of their smaller footprint, physical hardware can host a much greater amount of containers than VMs. They can also be created and deleted with better performance than VMs, allowing for more flexible horizontal scaling. As a result of this architecture, containers are by design not as isolated as VMs, which can result in a larger attack surface [32]. Figure 2.5 shows the architecture of a container compared against the architecture of hypervisors from section 2.3.1.

In fig. 2.5, the difference between a VM and a container architecture can be observed. A container engine and a select set of dependencies are required for containers to run on the host. Because of this shared design, containers are smaller and consume fewer resources, resulting in improved performance. Note that in most production scenarios, both VMs and containers are used in conjunction, where the VM acts as the host for the container engine.

Various container engines exist in today's landscape, with Docker[10] being the most popular [33, p. 3],

[10] https://www.docker.com/

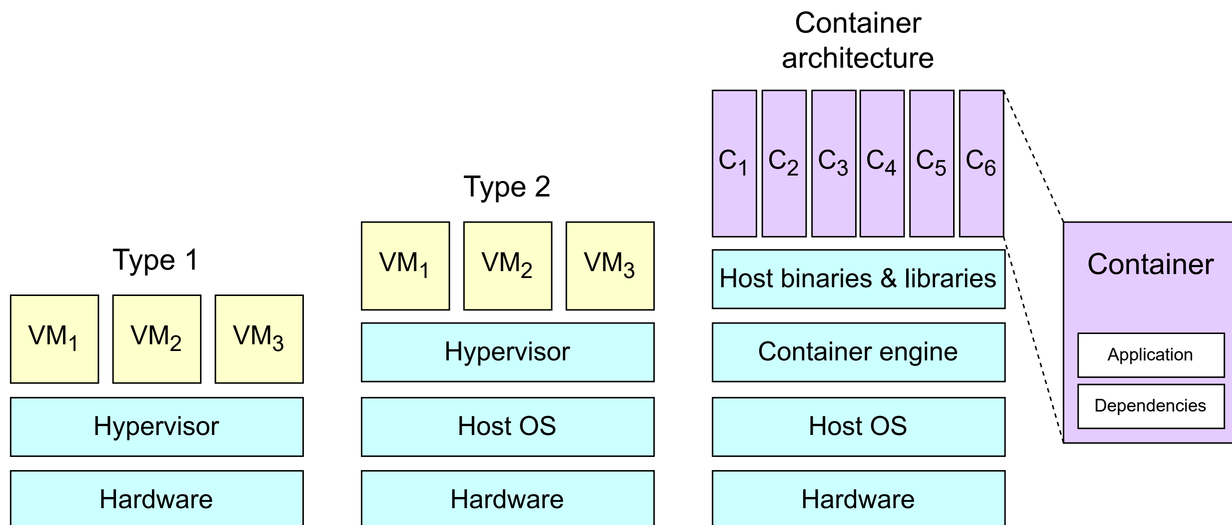

Figure 2.5.: A high-level comparison between virtual machine and container architecture

but other technologies such as Podman[11], containerd[12] and LXC[13] also being prominent options. As a result of these different solutions, the Open Container Initiative[14] (OCI) was established in order to enable interoperability between different container technologies, so that developers can build and migrate their applications between different OCI-compliant container engines.

Some essential concepts exist in the domain of Docker and containers in general [31], [32], [34]. Containers are running instances of a certain container image. An image is the actual package or artifact that contains the files, libraries and binaries necessary to run a certain container. This image can be a lightweight Linux container which serves a base to build an application upon, or it can define a specialized environment to handle specific application requirements.

An image is immutable and consists out of several layers, allowing developers to extend existing images for their specific needs. In the case of Docker, images can be defined with so-called "Dockerfiles". This file contains a base image and a set of instructions or commands that are executed when building the image. It enables developers to configure their specific application environments, installing certain dependencies and exposing certain ports for their application to run as intended. For example, an Ubuntu image can be used as a base, onto which additional applications—in this case a web server such as NGINX[15]—can be installed

[11] `https://podman.io/`
[12] `https://containerd.io/`
[13] `https://linuxcontainers.org/`
[14] `https://opencontainers.org/`
[15] `https://nginx.org/en/`

in order to host a website. However, a specific NGINX image could be used as a base image instead, which contains all necessary components to run the web server, without the need to install anything manually. In the Dockerfile, the required commands can be added to copy the application's build files and configure the NGINX server as needed. Detailed examples of Dockerfiles can be found later in this thesis during the practical implementation, discussed in section 6.2.

Container images, like other build artifacts, are typically stored in a central repository in order to be accessible by different teams or in different environments. In the case of images, this repository is often referred to as a registry. These registries can be public services or privately-hosted services only accessible in an enterprise's internal environment. Examples of registries are Docker Hub[16], Azure Container Registry (ACR)[17] and Amazon Elastic Container Registry (ECR)[18], which offer various managed features on top of storing the container images.

In the earlier example from fig. 2.5, a scenario with six running containers on one host was demonstrated. While this could be applicable in a development or testing scenario, in most production environments where microservices are implemented, a much larger number of containers and hosts are utilized. Bernstein [31, p. 84] mentions that the real power of containers comes from implementing distributed systems. With a larger amount of instances, a need arises for managing these larger amounts of nodes in a scalable and dynamic way, as it is not feasible to manage the process of building, deploying and updating containers and hosts manually. A form of management or orchestration is required, which is covered in the final section of this chapter.

2.4. Orchestration of Containers

As established in the previous sections, orchestration of containers forms an essential part of running microservices at scale for production-grade workloads. This section briefly covers which functionalities can be expected from an orchestrator and which technologies traditionally have been used to achieve this.

2.4.1. Container Orchestration Frameworks

Managing containers which are distributed over multiple hosts or nodes (i.e. a cluster) is not a trivial task. Different components need to be in place in order for containers to be orchestrated properly. As a minimum,

[16] https://hub.docker.com/

[17] https://azure.microsoft.com/en-us/products/container-registry/

[18] https://aws.amazon.com/ecr/

the following requirements, which are based on the capabilities of a CO framework mentioned by Straesser *et al.* [35] and Khan [36], need to be addressed by a CO framework:

- Scheduling of containers
- Scaling of containers
- Monitoring the health of containers and hosts
- Providing high-availability features
- Load balancing between containers

This list of requirements should be handled by one or more services that form a CO framework. Its main purpose is to abstract the underlying compute resources to the applications and containers that are running on them. It should function as the connecting layer between them, providing functionality in terms of scheduling, lifecycle management and networking as mentioned above. Figure 2.6 visualizes this CO layer between physical infrastructure and applications. In this example, the VMs are divided into multiple clusters which each contain a number of applications, consisting out of one or more containers. Note that the hypervisor layer has been omitted for the sake of simplicity.

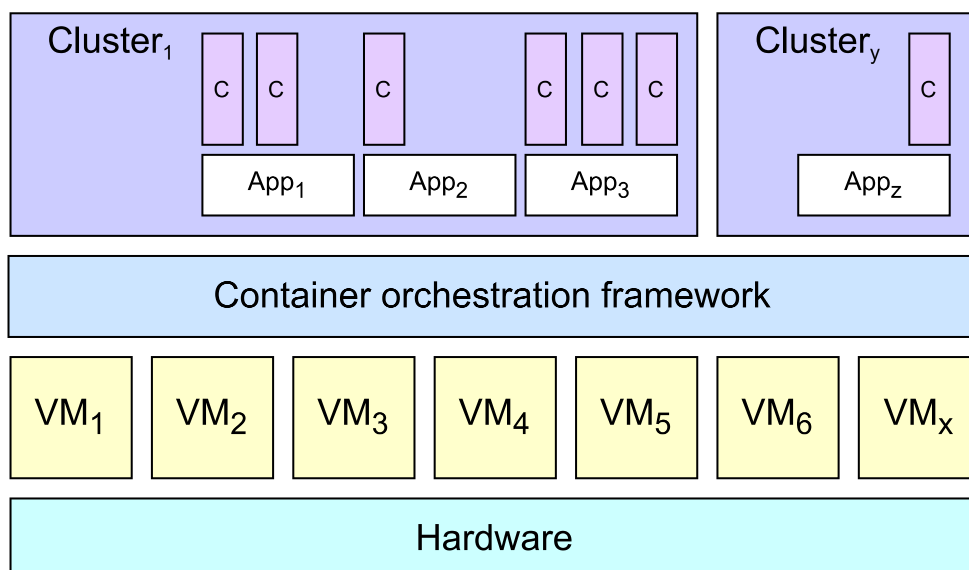

Figure 2.6.: A high-level overview of a container orchestration architecture

2.4.2. Kubernetes

In 2014, Google announced Kubernetes[19] as an open source cluster manager for Docker [31, p. 84] [37]. Kubernetes was presented as a scheduler for containers on raw resources, which could be provided by GCP. Following this announcement, other major players such Microsoft, VMware and Red Hat also announced their support for Kubernetes. In the following year, Google transferred control of the project to a newly-formed foundation—the Cloud Native Computing Foundation[20] (CNCF) [38]. Since the launch of Kubernetes, it has steadily become the de facto standard for managing containers at scale [39, p. 57174]. In their annual survey, the CNCF and The Linux Foundation[21] reported that 84% of participants were either using or evaluating Kubernetes [40]. In annual surveys organized by OpenStack[22], Kubernetes also remained the top CO framework used to manage OpenStack deployments [41]–[43]. This trend continues in the domain of CO at the time of writing.

Kubernetes can provide a solution to virtually all CO issues, because of its vast feature set. It includes capabilities for service discovery, auto-scaling, load balancing, storage management, automatic deployments and rollbacks, self-healing, secret and configuration management and more [44]. It introduces so-called "pods", which provide an abstraction for one or more containers [31]. This allows the usage of different containerization engines, giving users total flexibility and minimal vendor lock-in. Because of its open-source nature, other entities have created their own flavors or managed services that are based on Kubernetes, all providing a solution for a specific problem statement.

In the subsequent chapters, we dive deeper into Kubernetes and what its common issues or complexities are. We attempt to evaluate alternative CO frameworks for a specific use-case and provide insights based on practical reviews.

[19] https://kubernetes.io/
[20] https://www.cncf.io/
[21] https://www.linuxfoundation.org/
[22] https://www.openstack.org/

3. Related Work

In this chapter we highlight the existing research efforts that were realized in the domain of container orchestration. The related works are divided into three groups, which can be observed in the following sections.

3.1. Challenges Related to Kubernetes

3.1.1. Context

This section groups six works that are related to different aspects of Kubernetes and Kubernetes-flavors, and the complexity that comes with them. It includes theoretical and practical experiments with Kubernetes and comparable platforms.

3.1.2. Related Works

- **Abdollahi Vayghan *et al.* [45]** investigated the high availability (HA) functionalities that Kubernetes offers, through experiments in different failure scenarios. In their conclusion, they stated that the HA requirements are not automatically satisfied by deploying an application or microservice with Kubernetes. Furthermore, Abdollahi Vayghan *et al.* mentioned that the reconfiguration of Kubernetes' reaction to node failures can be complicated and requires great effort.
- **Geerling [46]** evaluated his experience with Kubernetes using a cluster consisting of Raspberry Pi single-board computers (SBC). He identified a number of reasons why Kubernetes is considered to be "complex", such as the networking and ingress configurations when setting up Kubernetes on your own, without using a managed service. Geerling concluded that this complexity is necessary, because Kubernetes abstracts the underlying infrastructure, so the total application development and architecture becomes simpler, especially when scaling up to a larger amount of containers.
- In an empirical study of security misconfigurations in Kubernetes manifests, **Rahman *et al.* [47]** identified 11 categories of issues. They analyzed 1,051 issues in 2,039 manifests from 92 open-source repositories, using a static analysis tool called Security Linter for Kubernetes Manifests (SLI-KUBE)

which was built by the authors for this study. Rahman *et al.* underlined the necessity of security-focused code reviews and static analysis to prevent malicious actors from exploiting the identified misconfigurations.

- **Telenyk *et al.* [48]** and **Koziolek *et al.* [49]** compared different lightweight Kubernetes-compatible platforms, such as MicroK8s[1], k3s[2], k0s[3] and MicroShift[4]. These distributions provide easier deployment and support for operating Kubernetes in environments with limited resources, such in an IoT scenario. Telenyk *et al.* found that k3s performed better in terms of disk utilization and time consumption, even being comparable to original Kubernetes. This is mostly due to the fact that MicroK8s is optimized for a single-node cluster, and thus did not perform as well as the other candidates. Koziolek *et al.* concluded that k3s and k0s show marginally higher control plane throughput.
- **Jawarneh *et al.* [50]** evaluated several container orchestration frameworks in terms of functionality and performance. The relevant technologies that were covered include Kubernetes, Docker Swarm, Apache Mesos and Cattle. They found that Kubernetes boast the broadest feature support, making it the most complete orchestrator. However, they mentioned the associated complexity with Kubernetes' architecture, which could result in significant overhead hindering its performance.

3.1.3. Discussion

The above bodies of research accentuate the complexity that comes with Kubernetes, especially when it comes to "vanilla" Kubernetes and not using a simplified distribution or a managed service. They also show the fragmentation within the domain of Kubernetes, with different kinds of distributions being formed and optimized for specific use-cases. This does not make the decision process for software developers or architects any easier. On top of this, misconfigurations can have impactful consequences and lead to data breaches, as shown by Rahman *et al.* [47].

While the aforementioned studies mostly concentrate on performance evaluation and stress tests, this thesis focuses on a more qualitative approach using a specific use-case in order to provide insights to technology decision-makers choosing the right technology for their containerization orchestration needs.

[1] `https://microk8s.io/`
[2] `https://k3s.io/`
[3] `https://k0sproject.io/`
[4] `https://github.com/openshift/microshift`

3.2. Managed Kubernetes Cloud Services

3.2.1. Context

This section groups three works that discuss managed Kubernetes services, which present a viable approach to leverage the benefits of Kubernetes while avoiding some of the configuration and management that comes with it.

3.2.2. Related Works

- **Singh *et al.* [20]** conducted a review of Docker and containers in general, while noting the need for Kubernetes for container administration and management. They touched briefly on containers as a service (CaaS), which is a form of container-based virtualization where container engines, orchestrators and underlying compute resources are provided by a CSP, such as AWS, IBM or GCP.
- In an evaluation of different managed Kubernetes services, **Pereira Ferreira *et al.* [51]** subjected the offerings of different CSPs—such as Amazon Elastic Kubernetes Service[5] (EKS), Azure Kubernetes Service[6] (AKS) and Google Kubernetes Engine[7] (GKE)—to performance tests in terms of CPU, RAM, disk and network usage. They found that the performance was mostly impacted by the choice of underlying resources (i.e. type and size of VM instances) and not by the overhead associated with the managed service itself.
- **Singh *et al.* [52]** compared container performance in different Kubernetes environments. They evaluated two popular container engines—Docker and containerd—in terms of CPU, disk I/O, memory and network latency. This evaluation was performed on two managed Kubernetes services, GKE and EKS. Singh *et al.* found that containerd performed better because Docker contains more overhead. They also concluded that a manually deployed Kubernetes cluster outperformed a managed cloud service, attributing this to the additional pods that were running in the managed cluster in order to provide more capabilities.

3.2.3. Discussion

The studies mentioned above are related to managed Kubernetes services, which provide another approach of using Kubernetes. These findings indicate that, when using a managed service, different choices have

[5] https://docs.aws.amazon.com/eks/latest/userguide/what-is-eks.html
[6] https://learn.microsoft.com/en-us/azure/aks/what-is-aks
[7] https://cloud.google.com/kubernetes-engine

to be made, such as which CSP to use and how to configure the underlying infrastructure that make up the cluster.

This research project focuses more on solutions that can be hosted in an on-premise environment and that are considered to be open-source, instead of commercial products. We attempt to provide insights for teams that do not have the resources or the freedom of technology choice to opt for a managed cloud service, although the aim of these services align with this thesis' main target, which is to simplify the configuration and operation of an orchestration platform.

3.3. Comparisons between Container Orchestration Frameworks

3.3.1. Context

This section groups six works that compare CO frameworks in different scenarios. Various other technologies are mentioned, which are interesting to take into account for the candidate selection in this research project.

3.3.2. Related Works

- **Shah *et al.* [53]** highlighted both Docker and Kubernetes as key technologies for building modern applications on the cloud, using GCP in this case. They gave an overview of how Kubernetes functions and which features it offers by deploying a WordPress web application. At the end of the paper, Shah *et al.* briefly mentioned Docker Swarm[8] in a feature comparison with Kubernetes. They concluded that using Kubernetes allows for deployment and management of applications running on multiple hosts, saving costs and time.
- **Ileana *et al.* [54]** presented Docker Swarm as an approach for improving performance for distributed web systems. They leveraged Docker Swarm to deploy a PHP application with MySQL database and noted that proper security mechanisms—such as authentication and authorization, TLS encryption and network segmentation—are required to prevent unauthorized access and data breaches. Ileana *et al.* concluded that Docker Swarm proves to be a useful solution for development teams who want to optimize and performance and availability of distributed web applications, while acknowledging the increased complexity of the overall application architecture.

[8] `https://docs.docker.com/engine/swarm/`

- **Mercl *et al.* [27]** compared different orchestrators such as Kubernetes, Docker Swarm, Fleet[9] and Apache Mesos[10] in a web application consisting out of NGINX, Memcached, MySQL and OwnCloud components. They concluded that Kubernetes and Mesos are best suited for large production-grade deployments, while Docker Swarm is more suited for testing scenarios. At the end, they mentioned that larger companies are releasing their own Kubernetes distributions, such as MicroK8s by Canonical and OpenShift[11] by Red Hat.
- In a feature comparison of open-source CO frameworks, **Truyen *et al.* [55]** aimed to identify common and unique features among Kubernetes, Docker Swarm and Apache Mesos. They found 124 common features and 54 unique features, with Kubernetes supporting the highest number of accumulated features. All three candidates have strengths that make them useful in a specific environment; Kubernetes was considered to be the most generic and Apache Mesos supported the broadest ranges of options to integrate other scheduler frameworks like Hadoop[12] and Kafka[13].
- In a benchmark between container orchestration frameworks, **Straesser *et al.* [35]** compared Kubernetes and Nomad[14] in both a self-hosted and public cloud environment (GCP). They introduced a benchmarking tool called Container Orchestration Frameworks' Full Experimental Evaluation (COFFEE) in order to automate performance tests in a repeatable manner. Straesser *et al.* found that Kubernetes outperforms Nomad in many scenarios, but acknowledged that more experiments with data from production workloads should be conducted.
- **Malviya *et al.* [56]** compared four popular container orchestration tools, Kubernetes, Docker Swarm, Apache Mesos and Red Hat OpenShift in terms of security, ease of deployment, stability and scaleability. They found that Kubernetes and Apache Mesos were best suited for large clusters, while Docker Swarm proved to be relatively simple to set up for smaller scenarios. OpenShift came with predefined policies by default, making it more secure without the need for extra configurations.

3.3.3. Discussion

The majority of these bodies of research approach the comparison of CO frameworks from a more practical point-of-view, which follows a similar approach to this research project. However, the insights gained from these experiments are more from a general technology perspective and are quite limited in terms of practical

[9] `https://github.com/coreos/fleet?tab=readme-ov-file`
[10] `https://mesos.apache.org/`
[11] `https://www.redhat.com/en/technologies/cloud-computing/openshift`
[12] `https://hadoop.apache.org/`
[13] `https://kafka.apache.org/`
[14] `https://www.nomadproject.io/`

recommendations towards developers looking for a solution for their use-case. We believe that we can provide a more in-depth review of the strengths and weaknesses of these frameworks, by submitting them to a use-case implementation with elements found in typical production-grade applications. The feature studies conducted by Truyen *et al.* and Straesser *et al.* give an indication of which features we can analyze in order to make an informed selection of CO candidates for this research project. We can leverage previously accomplished works and further filter down this selection with insights on each CO candidate.

4. Methodology

This chapter details how the research questions were answered. It presents a high-level overview of the approach that was utilized in order to analyze the different components of this research project. The methodology can be split up into four parts, which build on top of each other to come to the final conclusion. First, a review of the state of the art in the domain of container orchestration (CO) was conducted. Then, we defined the use-case which was adopted as an evaluation criterion for the selection of essential container orchestration features. Using this selection and some additional criteria, a selection of viable CO candidates was made, which were untimely subjected to a qualitative analysis. Finally, we drew conclusions from the results of these different analysis stages.

4.1. State of the Art

As a first step, we reviewed the state of the art in the domain of CO, which supported this research by providing an understanding on the current challenges, solutions and technologies. The main sources leveraged for this step were scientific databases such as IEEE Xplore and ACM Digital Library, as well as technical documentations about a given orchestration technology, provided by the company or owner. The focus of this part was on the most popular and widely-used technologies for scenarios related to CO in distributed web applications. We reviewed different conference papers, studies and technical documentations that were relevant to the technology landscape that we were attempting to investigate.

With the knowledge gained from this section, we were able to answer the first research question. This was an essential first step in order to establish a baseline, which was used to outline the use-case and identify crucial features of a CO framework.

4.2. Design of Practical Use-Case

In order to evaluate different CO technologies from a practical perspective, a distributed web application—encompassing multiple elements that are typically part of a production-grade system—was designed. This

application was built from scratch specifically for this research project, with its main purpose being to emulate a containerized workload part of a production environment. Because this thesis is not part of a company or industry project, certain parts of the application simulate the function of their real-world counterparts. However, it is important to note that the use-case was designed with previous industry experience in mind, in such a way that the results of this research would be similar or comparable to an extent if a real-world application—equivalent to the defined application—would be used instead.

We motivated both the technical and non-technical aspects of this use-case in order to be transparent to this thesis' audience about certain architecture or implementation choices. With the use-case clearly defined, we were able to continue this research with the following sections about the selection of container orchestration frameworks and their essential features.

4.3. Evaluation of Container Orchestration Features and Technologies

After defining the use-case, we deducted the specific requirements that were needed to support the application. These requirements were then linked to the features that were essential for a container orchestration technology to be considered a viable candidate. We reused the previous findings from the state of the art to make a further selection of features. It is important to note that these features were selected mainly because of their relevance to this research's use-case specifically and thus should not be considered a general selection of features. We also included optional features that were deemed to be a welcome addition, but were not considered to be essential.

With this specific selection of essential and additional features, we continued by evaluating possible container orchestration technologies. Again, by referencing previous efforts in the state of the art, and by combining these findings with the essential features, we were able to compile a selection of candidates that were suitable for the practical comparison. With both selections made, an answer could be provided to the second and third research questions respectively.

4.4. Qualitative Analysis of Container Orchestration Frameworks

In the final part of this research, each selected candidate was subjected to a practical implementation of the defined use-case. For each candidate, we documented executed steps and reviewed the strengths and weaknesses which were encountered during the implementation process. We leveraged the results of previous

sections to compile a list of criteria, which was used to evaluate each candidate. With this approach, we attempted to produce structured and reproducible results. The observations made during this practical testing were mostly from a developer's perspective, who would have been tasked with setting up the required application with the respective CO technology.

We opted for this point of view because it would yield the most valuable insights for intended audience of this research project, so other developers and other technical profiles could evaluate these experiences and make an informed decision about their own projects. This thesis' primary goal was to contribute to the current technology landscape by providing new perspectives on existing technologies. With the findings from these experiments, we were able to answer the fourth and final research question which encapsulated the results from all previous section into a final verdict, discussed in chapter 7.

4.5. Overview of Methodology

As a conclusion of this chapter, the flowchart in fig. 4.1 visualizes the general methodology and how its different components are connected. The rectangles represent the overall stages as discussed in this chapter, while the ellipses indicate the results from each stage, which also play a role in the subsequent stages.

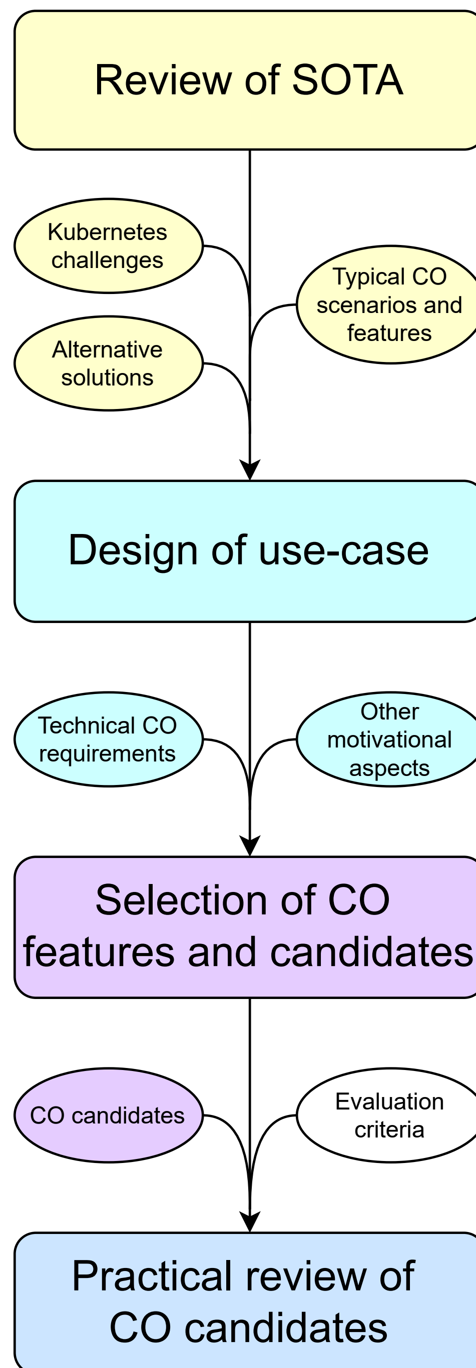

Figure 4.1.: A high-level overview of this research project's methodology

5. Use-Case Definition and Candidate Selection

This chapter presents the use-case around which this research project revolves. It further defines the required features for a possible orchestration candidate. By doing so, an answer to the second and third research questions can be formed, ensuring that all necessary components are in place for the practical testing of the different container orchestration (CO) candidates.

5.1. Use-Case Definition

5.1.1. Motivation

You Build It, You Run It

The phrase "you build it, you run it" can be considered the mantra of the DevOps movement that started more than a decade ago. The initial goal was to bring developers into contact with the day-to-day operations of their software, which would increase the quality of service due to a more direct feedback loop [57]. However, bridging the gap between development and operations proved to be easier said than done, even though both entities are handling the same applications or software.

The domain of information technology (IT) is ever-changing, with legacy technologies becoming obsolete and newer ones replacing them in a seemingly never-ending cycle. This applies to both operational concepts and development technologies, which makes it challenging to stay on top of the latest progressions in both domains. As developers get tasked with usual requirements, such as a new feature implementation or a general application modernization, they have to keep both the application and infrastructure in mind when making changes. With the defined use-case, we attempt to simulate the above situation by designing an application that fits a microservice architecture and that can be supported by common DevOps procedures—such as different deployment strategies with automatic rollbacks.

Limited Resources

In larger organizations, this bridge between development and infrastructure is often facilitated by so-called platform engineering teams. They provide tools and workflows, often with a focus on self-service, to support developers with the deployment of their applications. These platforms are designed within a secure and controlled environment, which should find the right balance between giving developers the freedom they need to operate their application, and implementing secure configurations that apply for the entire infrastructure. This balance between usability and security can be considered a common point of discussion in the field of cybersecurity [58], [59], but it exceeds the scope of this research project.

In small and medium enterprises (SME) however, this challenge needs to be tackled differently, since these organizations often do not possess the same resources to provide secure platforms for their developers and applications. Possible solutions include hiring specialized profiles or consultants to assist the existing development teams with this task, or leveraging cloud services that accommodate the desired deployment strategy. Both of these options require some form of additional monetary investment, which is not always a feasible option. This research's defined use-case is designed to focus on development teams that are facing these issues while attempting to implement modern application architecture such as microservices.

5.1.2. High-Level Overview and Description

High-Level Overview

The use-case implements an address resolution system, similar to the domain name system[1] (DNS), which serves as a highly essential part of the Internet. This system was dubbed ARGO, which stands for Address Resolver written in Go, and it translates between internet protocol (IP) addresses and domain names using a simplified data model with records and servers. Since this research aims to compare between different orchestration solutions, some components of this system were omitted or abstracted in order to accelerate the development process and remove some of the overall complexity of the total architecture. Figure 5.1 visualizes the different components that need to be in place to facilitate this system.

The following clarifications are made for fig. 5.1:

- Every purple rectangle represents an application component which consists out of one or more running container instances.
- Communication flows are indicated with a dotted line between components and segments.

[1] https://www.cloudflare.com/learning/dns/what-is-dns/

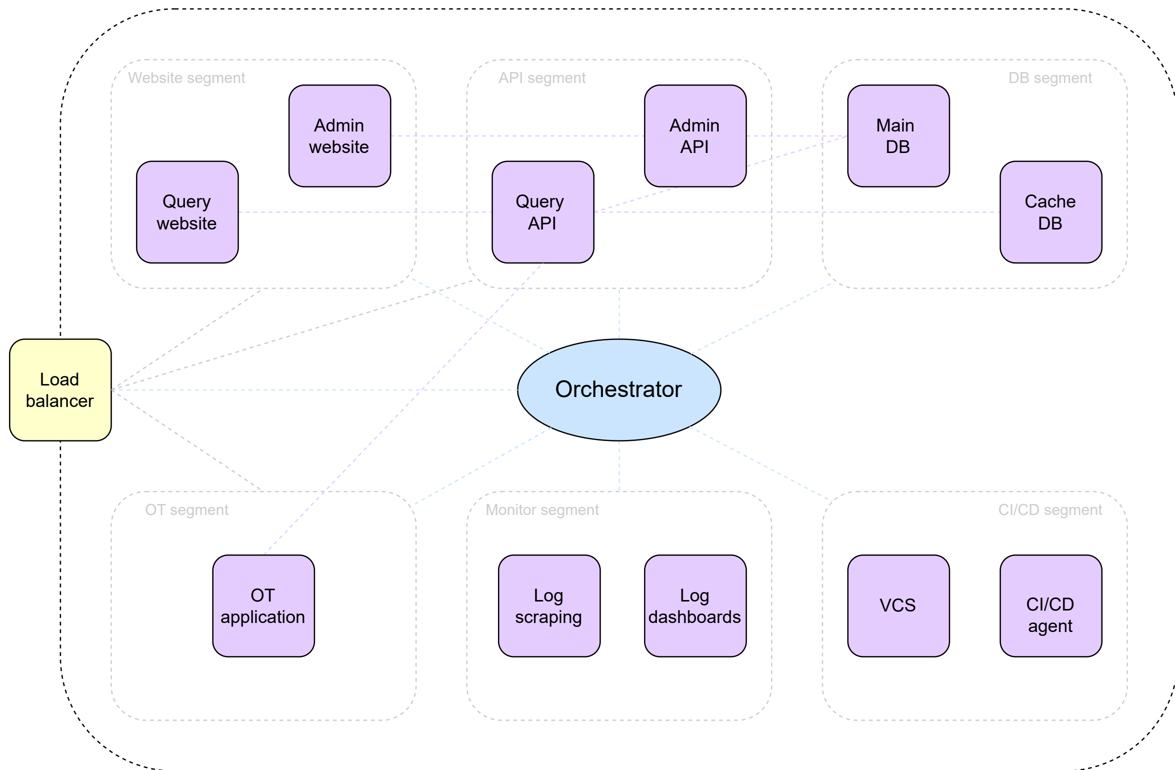

Figure 5.1.: A schematic overview of the microservice architecture for the presented use-case

- Some communication flows are omitted in favor of clarity, such as the interaction between the CI/CD and logging segments and the rest of the components. These communications are required for the functionality of the system and are thus certainly part of the practical implementation.
- The load balancer is visualized in yellow and handles both internal (i.e. requests coming from other components in the orchestration network) and external (i.e. requests coming from other networks) balancing of the incoming requests.
- The central orchestrator represents the different candidates that were subjected to the practical test scenario. Depending on the orchestrator, it was necessary to add or remove additional components within this architecture. For example, a candidate could include a load balancing functionality, eliminating the need for a dedicated solution as shown in the schematic.

Description

The general functionality and workflow of the system could be described with the following steps:

- Users could use the query website to translate between IP addresses and domain names using an intuitive UI.

- Administrators could use the admin website to makes changes to the data objects. This website provided a simple UI for creating, deleting and updating entries in the database.
- Both of these applications leveraged separate APIs to interact with the databases.
- Other applications and systems could use the query API directly to facilitate transformations for non-human users (e.g. operational technology (OT) network or other APIs).
- Aside from the main database that contained the IP address and domain name records, a cache database was present that could be used by the query API to speed up request that happen often.
- The main components—such as the APIs, databases and the underlying infrastructure—were configured with centralized monitoring, so the application developers could have a clear overview of complete system.

The following figures represent two typical user flows of the described system. Figure 5.2 visualizes the creation of a new data record using the admin website. Figure 5.3 shows a query from a non-human user, such as an OT-application or a consuming API.

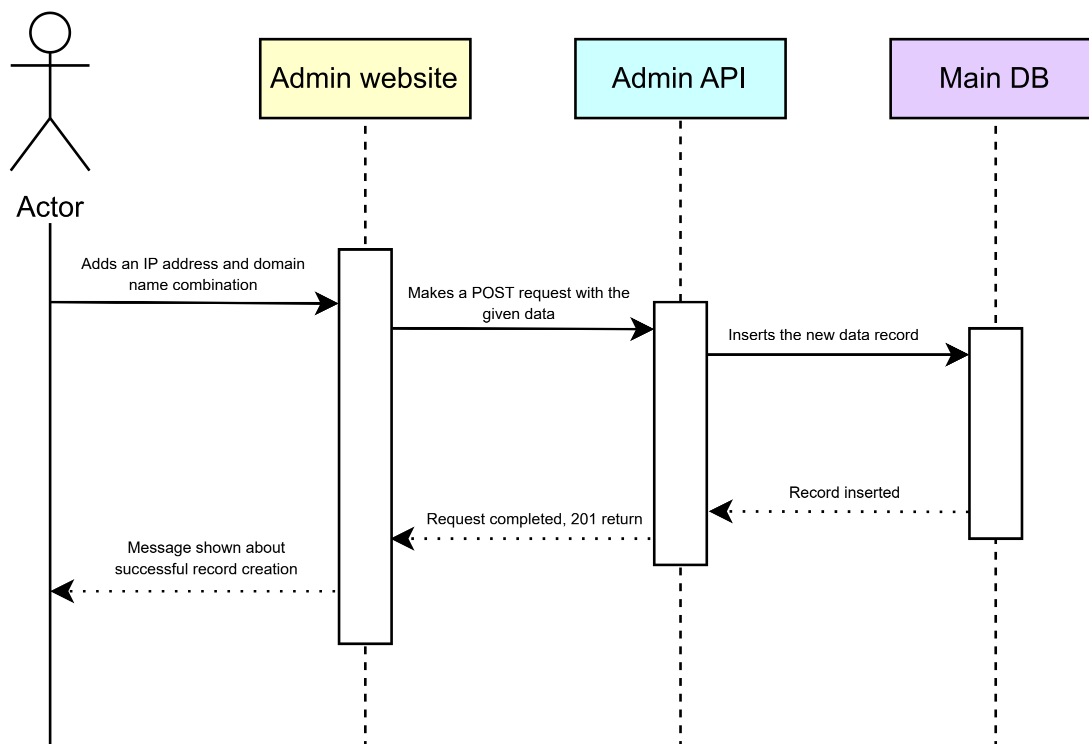

Figure 5.2.: A sequence diagram showing a human user interacting with the admin website

The system was designed in such a way that it served as an ideal candidate for a microservice architecture. All main components in the application were stateless, meaning they did not keep any data or state stored

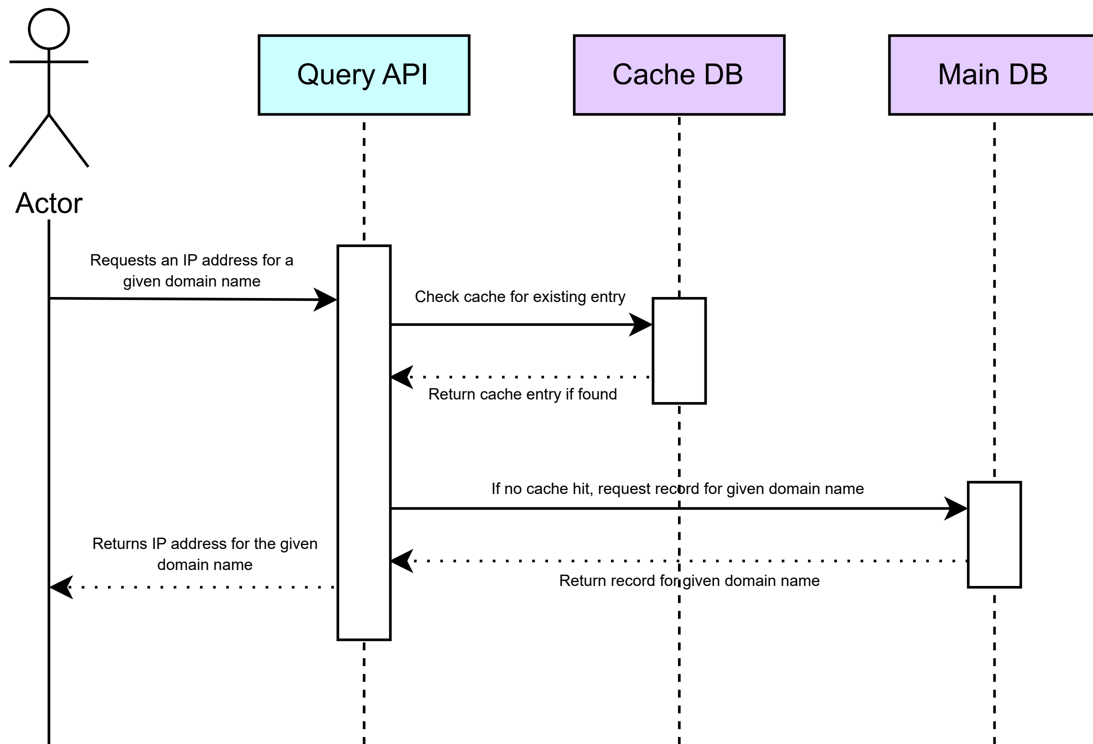

Figure 5.3.: A sequence diagram showing a non-human user interacting with the query API

during their operational lifecycle. All necessary states were kept in the database, which could then be accessed by the APIs in order to interact with the data (e.g. getting a domain name for a given IP or updating a subdomain entry). Since we do not need to handle the state of the websites and APIs, they could easily be containerized and scaled by the orchestration framework.

5.1.3. Components

In this section we highlight the different components of proposed system in more detail. We defined the required components, which absolutely needed to be part of the system, and the optional components, which were considered "nice-to-haves" and were only to be focused on after the required components had been investigated and implemented. The optional components were not stated in a particular order of priority. It depended on the progress with the required components, which optional components and to what degree we were able to implement them.

Required Components

- A **query API** which translates between IP addresses and domain names.

- An **admin API** which allows for changes to the data records.
- A **query website** which consumes the query API for regular human users.
- An **admin website** which consumes the admin API for power users.
- An **OT application** which consumes the query API directly, simulating non-human clients.
- A **main database** which stores the records containing information about the name servers and the domain records.
- A **cache database** which stores the latest requested transformation for performance improvements.
- A **logging and monitoring segment** that centrally stores and visualizes logs and traces from the various components.
- Separate **network segments** that only allow strictly necessary communication between different parts. Components that don't need to be able to interact with each other, should not be able to.
- **Shared storage** in order to achieve high availability (HA) in case of an outage or hardware failure.
- Internal and external **load balancing** and **secure exposure**, in order for users outside the orchestration network to access the query website.

Optional Components

- Secure communication between microservices using **mutual authentication** mechanisms such as mTLS[2].
- A self-hosted **container registry** to store container images.
- A centralized **secret management** system for storing application secrets such as passkeys and certificates.
- A **VCS instance** that stores the source code and provides CI/CD capabilities.

Components Conclusion

We believe that the above components form a stable foundation for a system to be considered "production-grade", since the application contains elements which are often present in real-life applications. We acknowledge that this definition depends on multiple factors that differ between use-cases, but we consider the elements mentioned to be—in some way or another—part of an average production system. As a reference, the following listing includes several controls from the NIST SP800-53 [60] and NIST SP800-204 [61] standards. These publications present industry standard requirements for building secure applications.

[2] https://www.cloudflare.com/learning/access-management/what-is-mutual-tls/

- AC-5 (NIST SP800-53): separating duties of administrators and users to prevent unauthorized access.
- SC-7 (NIST SP800-53): implementing network segmentation between different components.
- SC-8 (NIST SP800-53): using encryption protocols in distributed systems such as a microservices' architecture.
- MS-SS-4 (NIST SP800-204): implementing mutual authentication between microservices using mTLS.
- MS-SS-12 (NIST SP800-204): implementing an API gateway that handles incoming traffic into the cluster, in terms of load balancing and monitoring.
- MS-SS-1 (NIST SP800-204): secret keys or tokens should not be part of the codebase, instead a data vault solution should be used to dynamically use the secret value.

5.2. Feature Selection

In this section, a selection of container orchestration capabilities is made to answer the second research question. We leverage existing research, scoped to the defined use-case, in order to gain a relevant selection of functionalities.

5.2.1. Container Orchestration Features

Earlier in this document, in section 2.4.1, we defined the minimum requirements for a CO framework. In the following paragraphs, we want to elaborate more on each feature and why it was deemed a necessary task that must be fulfilled by the CO framework. Similar to the previous definition of the requirements, we based our selection on the characteristics identified by Straesser *et al.* [35], Khan [36], and Jawarneh *et al.* [50].

Scheduling

The scheduling of changes to be applied within the cluster serves as one of the main responsibilities of a CO framework. Because of this, it becomes possible to abstract the underlying pool of resources as one entity, against which commands or changes can be executed. Scheduling containers is therefore a key component in the efficient usage of the cluster resources [50]. A CO framework should act as the interface through which the infrastructure is managed, allowing the underlying resource pool to be expanded or reduced—for example, when hosts are added to or removed from the cluster.

Scaling

Another key feature of a CO framework is its ability to facilitate horizontal scaling. As discussed earlier in section 2.2.3, it serves as one of the main drivers behind a microservice architecture, so it is naturally expected that a CO framework can fulfill this requirement. Scaling operates in conjunction with scheduling, because in most configuration scenarios, the system administrator defines the desired state for their application. It then becomes the responsibility of the CO framework to schedule and scale the containers in order to achieve or maintain the desired state.

As a practical example to demonstrate this, suppose a system administrator defines a configuration where two containers—a web API container and a database container—are present. The system administrator also specifies there should be at least five instances of each container running at all times, with a maximum of twenty instances per container. With this configuration in place, it becomes the responsibility of the CO framework to keep the cluster in this desired state. This could mean scaling up when additional requests are received, restarting a container if it enters an error state, or scaling down to the minimum number of instances when traffic decreases.

Health Monitoring & Failover Capabilities

Because a CO framework forms the bridge between containers and the underlying resources, it is also responsible for monitoring the health of both layers and scheduling the necessary changes in case of an unhealthy container or node. In a scenario where multiple nodes are running multiple containers, the CO framework should detect if a node becomes unhealthy and move the running workloads to another healthy node. Using this failover mechanism, HA can be guaranteed for the services running in the cluster.

Networking

In terms of networking, a CO framework should facilitate the intercommunication between different containers. It should provide a way of addressing the different nodes and containers, ideally with a form of internal DNS. It should also manage the balancing of the internal load, by delegating incoming requests to the available container instances. This can be done according to a load balancing algorithm, such as "Round Robin", "Least Connections", "Least Response Time" or others [62], [63]. It is important to note that we only specify the load balancing of internal communications, for example, between the aforementioned web API containers and the database containers. The balancing of external request coming into the cluster—for

example from an external client to the web API running in the cluster—is considered as a separate requirement and is discussed in section 5.2.2.

Service Discovery

In order for the cluster to be extensible and flexible, the CO framework needs to employ a sort of service discovery mechanism. This way, the orchestrator is able to reach other running services and communicate with them, regardless on which underlying compute resource the service is running. Additionally, this mechanism supports the intercommunication between different services. This can be accomplished by providing a way of addressing the services, for example by using an internal DNS resolver.

5.2.2. Features Scoped to Use-Case

Shared Storage

In most microservice architectures, there is a need to share some application states between different hosts. Although we stated earlier that microservices ideally are stateless, many containerized systems require a volume to store some form of application or configuration data. This challenge can be tackled by a shared storage solution, which provides a shared volume accessible by the difference nodes, so they are able to run containers with their desired configuration.

We consider this to be a crucial feature for the defined use-case, moreover, we find it necessary to explicitly state this requirement since it is not always assumed to be a “native” CO framework feature. Kubernetes provides a plugin called “PersistentVolume” for this reason [64]. However, the CO framework should provide support for a third party solution that facilitates shared storage.

Internal and External Load Balancing

On top of the internal load balancing mentioned in section 5.2.1, we want to expand this requirement to include the capability for external load balancing as well. This should enable the cluster to handle requests coming from outside the CO environment as well, in a balanced way, so the resources are used efficiently. We acknowledge that this feature may be considered part of a CO framework that providers load balancing in general, but we find it necessary to explicitly declare it because it is crucial for the defined use-case.

Continuous Deployments

As a final addition to the requirements, we want to mention continuous deployments of application updates. The CO framework should be capable of introducing new application versions while maintaining HA. It should support deployment strategies that allow for a rollback mechanism, bleeding of old versions or readiness probes, when updating the containers, without the client noticing a major impact or interruption. This represents one of the key advantages of containerized deployments over typical application deployment methods, as new versions can be introduced seamlessly, eliminating the need for manual configuration and intervention.

5.2.3. Overview of Selected Features

As a conclusion to this section, table 5.1 presents an overview of the previously discussed features. With the last column, we indicate if this feature is required to be included in the CO framework (IN), or if it can be added as an external service (EX). Ideally, a CO framework should provide all the mentioned features, but in order to widen the scope of possible test candidates, we define the absolute minimum requirements, marked with "IN".

Feature	Description	IN/EX
Scheduling	Organizing changes on the cluster	IN
Scaling	Adding or removing instance to achieve the desired state	IN
Monitoring	Detection and response in case of unhealthy components	IN
Networking	Intercommunication between components	IN
Service discovery	Addressing services for simplified communication	EX
Shared storage	Sharing application states and configuration between nodes	EX
Load balancing	Managing the internal and external requests between instances	EX
Continuous deployments	Rolling updates of applications while keeping cluster HA	IN

Table 5.1.: An overview of the selected CO features

5.3. Candidate Selection

This section covers the selection of the CO frameworks subjected to the practical implementation. We make the selection based on the previously mentioned features as well as some additional criteria.

5.3.1. Additional Criteria

The additional criteria are based on the non-technical requirements mentioned in the motivation of the use-case.

Software Licensing

In order for a candidate to be viable, the technology should be available under an open-source license. If it is a commercial product, it should offer a free tier where the software can be used without any additional licensing costs or restricted functionality. Common open-source licenses—which are promoted and approved by the Open Source Initiative (OSI)—include MIT[3], Apache[4], GPL[5] and MPL[6].

Suitability

The CO framework should be suitable for small to medium deployments. We mention this to filter out technologies that might be considered to be "overkill" for the defined use-case. We aim to include solutions which main purpose aligns with the desired implementation goals, and to exclude the ones that are aimed at managing massive workloads, such as thousands of nodes with tens of thousands of containers in large-scale datacenter scenarios, such as the infrastructure of a CSP.

Vanilla Kubernetes

Vanilla Kubernetes is excluded as a viable candidate because we are aiming at finding a suitable alternative for it. However, we include Kubernetes-based solutions—which might rely on Kubernetes as an underlying technology—but which provide extra capabilities which set them apart.

Active Development

With this final criterion, we want to focus on technologies that are under active development and are receiving regular updates. Technologies that are no longer maintained or that are superseded by others are not considered to be relevant for this research project.

[3] `https://opensource.org/license/mit`
[4] `https://opensource.org/license/apache-2-0`
[5] `https://opensource.org/license/gpl-3-0`
[6] `https://opensource.org/license/mpl-2-0`

5.3.2. Evaluation of Candidates

Docker Swarm

Docker Swarm[7] is an advanced cluster management feature built into the Docker Engine. It provides native orchestration features for deployments using Docker containers, such as scaling, service discovery, networking and rolling deployment updates [65]. Swarm is frequently positioned as a simpler alternative to Kubernetes in smaller scale scenarios [27], [53], [54], [56].

An important fact to mention is that Docker Swarm is a feature of the Docker Engine, and thus only available in scenarios where Docker is used. Other OCI-compatible engines—such as Podman or containerd—are not supported. This does not present an issue for this project's use-case, as we leverage Docker as the containerization platform, but it could be a limiting factor in different scenarios. Additionally, Swarm is available under the Apache license and is actively maintained by Docker Inc. as part of their Docker Engine [66], [67].

With regard to this research, Docker Swarm presented orchestration capabilities that are closely integrated with Docker containers, which makes it interesting for SME teams that are already familiar with its concepts and functionality. We believed it could be a possible solution for common CO challenges.

Kubernetes

At this point, Kubernetes should not need an introduction. It stands as the main CO framework and forms the foundation for other frameworks mentioned in this evaluation. Because this research focuses on alternatives for Kubernetes, we chose to not include it as a candidate for the practical use-case implementation.

Lightweight Kubernetes distributions

K3s[8], k0s[9] and MicroK8s[10] are lightweight Kubernetes distributions developed by Rancher (SUSE), Mirantis and Canonical respectively [68]–[70]. They each offer a version of Kubernetes which is lower in resource requirements [49] and that contains a set of feature enhancements aimed at simplifying Kubernetes' configuration and management. These aspects make them interesting potential candidates, since their improvements could be compared more pragmatically to uncover how they benefit developers looking to get

[7] `https://docs.docker.com/engine/swarm/`
[8] `https://k3s.io/`
[9] `https://k0sproject.io/`
[10] `https://microk8s.io/`

started with an understandable orchestration solution. Additionally, all three of the mentioned distributions are certified by the CNCF, meaning that all standard Kubernetes functionalities are supported.

With regard to this research, we chose not to select any of these types of technologies, since they resemble Kubernetes closely and this project's focus lies more on evaluating true alternative approaches. The dedicated time and resources for this research were limited, so the scope needed to be concise, only containing potential alternatives. We acknowledge that k3s and k0s deserve to be evaluated since they seem to address many of Kubernetes' more challenging aspects, which we discuss further in section 8.5 about possible future research efforts in the CO domain.

Mesos

Mesos[11], is a distributed system kernel that can abstract compute resources, enabling efficient resource sharing and distributed scheduling of changes in the cluster [71], [72]. It is mainly aimed at simplifying deployments on a large number computing nodes, on the scale of a datacenter. It can be used for the orchestration of containerized workloads, but it is not its only application [27], [55], [56].

With regard to this research, Mesos did not align well with the defined aspects of this use-case, which focuses more on SME teams tasked with managing their containerized applications. Mesos is primarily targeted toward datacenter-scale of application deployments and scaling requirements, so we chose not to select it as a practical testing candidate.

Nomad

Nomad[12] is an orchestration platform developed by HashiCorp, supporting orchestration features for both containers and VMs. It is designed to be a simple but flexible scheduler, offering an alternative to Kubernetes [35], [73].

Because Nomad is solely a scheduler, it requires an additional service for service discovery, which is an important requirement for the defined use-case. Consul[13], developed by the same company as Nomad, can provide a solution for this scenario. It is also worth mentioning that HashiCorp changed their licensing model in 2023, opting for the BSL over the previous MPL [74]. While this license is not OSI-approved, we

[11] `https://mesos.apache.org/`
[12] `https://nomadproject.io/`
[13] `https://www.hashicorp.com/en/products/consul`

consider it to be an acceptable license for this use-case, because the source code is publicly available, and the product can be used without costs in a self-hosted environment.

With regard to this research, Nomad seemed like an interesting candidate to subject to the practical implementation. It requires an extra service for network discovery, as mentioned earlier, but we believed it could yield valuable insights when compared with other CO candidates.

OpenShift

Red Hat OpenShift Container Platform (OCP) is part of the Red Hat OpenShift[14] product line and offers a cloud-based Kubernetes platform. It aims at offering an enterprise-ready Kubernetes solution with minimal configuration requirements and a strong selection of security and compliance features [56], [75].

OCP leverage Kubernetes as its foundational technology and includes a number of additional features that aim to optimize workflows. It can be considered as a set of different tools and services assisting developers with their application development and delivery. For example, it includes OpenShift Pipelines which allows developers to create CI/CD workflows that integrate with OCP.

With regard to this research, OCP presents itself as an “all-in-one” platform for developing, deploying and managing containerized applications. We chose to not select it for the practical use-case implementation, since OCP’s vast feature set might require us to adapt the use-case, which would make it more susceptible to vendor lock-in.

MicroShift

Red Hat MicroShift is a lightweight Kubernetes platform that is based on the core functionality of OCP, but geared towards deployments on that require a minimal resource footprint, such as IoT or edge computing scenarios.

Similar to OCP, MicroShift did not seem like a good fit for the defined use-case, for similar reasons, and was not selected.

Managed cloud services

Managed cloud services are becoming a popular option of consuming Kubernetes’ benefits without requiring the investment of configuring and managing it [20], [51], [52]. Different services—such as AKS, GKE and

[14] `https://www.redhat.com/en/technologies/cloud-computing/openshift`

EKS—offer various benefits that are especially useful when combined with other cloud services of the respective CSP.

With regard to this research, we chose not to include any cloud-based service as an implementation candidate because they are considered to be paid products and do not satisfy the defined open-source requirement.

5.3.3. Overview of Selected Candidates

As a conclusion to this chapter, table 5.2 presents an overview of the selected CO candidates, which are subjected to the practical use-case implementation in the subsequent chapter.

Candidate	**Company**	**Note**
Swarm	Docker Inc.	Only supports Docker as container runtime
Nomad	HashiCorp	Scheduling capabilities only, requires additional service discovery solution

Table 5.2.: An overview of the selected CO candidates

6. Implementation

This chapter covers the practical implementation of the use-case in combination with the selected CO candidates from the previous chapter. It outlines the testing environment used, along with the different technology choices and specific implementation details for each candidate.

6.1. Testing Environment

The necessary resources for testing the different scenarios were provided by the university. Eight dedicated VMs were allocated for this research project, and they were used to deploy the use-case for each CO candidate in its entirety. We leveraged regular snapshot to enable stable versions, which we could restore when necessary, allowing us to use the given infrastructure efficiently. The VMs each have the following specifications:

- CPU: 8 cores @ 2.194 GHz
- RAM: 32 GiB
- Storage:
  - Main disk: 200 GiB
  - Additional disk: 50 GiB
- OS: Ubuntu 24.04.2 LTS
- Hostname: vm0{1–8}
- IP address: 10.203.96.23{0–3}/24 or 10.203.96.11{5–8}/24

In order to conduct the implementation in a structured and reproducible way, we leveraged Ansible [76], which is a well-known Infrastructure as Code (IaC) tool. It allows for simple management and configuration for scenarios with multiple hosts. As an example, listing 31 presents an Ansible command which shows an overview of the IP address of each host.

It is important to note that “vm01” contains two network interfaces. The first one is in the same subnet as the other hosts, and the second one is part of a different subnet. This is done deliberately to make the host

reachable by other subnets, so we can simulate "external" requests as well.

6.2. Initial Use-Case Implementation

Prior to subjecting the candidates to the use-case, it was necessary to create a base implementation of the specified system which could then be orchestrated by the candidate. The following sections discuss the implementation steps that were taken to create the base scenario that was depicted in fig. 5.1. Each part consists out of one or more Docker containers that package the necessary software components. As a final artifact of this section, the complete Docker Compose file containing all container configurations and image versions is presented. Note that not every implementation step is included in detail, in order to keep this chapter concise and readable. The full source code for the different application components can be found in this GitHub repository[1].

6.2.1. Databases

As a start, the main database was created and configured with a basic data model. For the relational database management system (RDMS), we chose PostgreSQL[2] as it is a popular open-source relational database technology. Listing 1 shows the SQL statements used to create the data model. Afterward, sample data was added to both tables.

```
1  CREATE TABLE servers (
2      id SERIAL PRIMARY KEY,
3      name VARCHAR(255) NOT NULL,
4      ip VARCHAR(15) NOT NULL,
5      contact VARCHAR(255) NOT NULL,
6      company VARCHAR(255) NOT NULL,
7      timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP
8  );
9  CREATE TABLE records (
10     id SERIAL PRIMARY KEY,
11     timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
12     domain VARCHAR(255) NOT NULL,
13     ip VARCHAR(15) NOT NULL,
14     server INTEGER NOT NULL,
15     FOREIGN KEY (server) REFERENCES servers(id) ON DELETE CASCADE
16 );
```

Listing 1: An SQL snippet containing statements to create data model for the main database

[1] https://github.com/cadeke/argo/tree/main/base

[2] https://www.postgresql.org/

For the cache database, we chose Memcached[3] as it is an in-memory key-value store that fits the use-case perfectly. This database could be used by the web APIs for faster query lookups.

Containerizing databases is a debatable topic, although it is generally considered to be better to host a truly stateful component such as a database in a different way than with containers, because their main purpose is meant for stateless components [77]. This decision can vary depending on the scenario, and for this use-case, we opted for containerization of both databases, because database hosting is not the main scope of this research, and we just required some databases to keep the data states used by the other application components.

6.2.2. Web APIs

Both the admin API and the query API were written in Go[4] and containerized using the Dockerfile shown in listing 2. The created Docker images were pushed to Docker Hub[5], so they could be used for further deployments.

```
1  FROM golang:1.24.1-alpine
2
3  WORKDIR /app
4  COPY go.mod go.sum ./
5  RUN go mod download
6
7  COPY . .
8  RUN go build -o admin-api
9
10 EXPOSE 8080
11 CMD ["./admin-api"]
```

Listing 2: A snippet containing Dockerfile instructions for creating admin API image

6.2.3. Websites

React[6] was chosen as the frontend framework for creating the admin site and the query site. They were both linked up to the relevant routes of the web APIs to facilitate the required actions. For example, a domain name could be resolved as shown in fig. A.1, or a new DNS record could be added as shown in fig. A.2. Listing 3 shows the Dockerfile used for the containerization of both applications.

[3] https://www.memcached.org/
[4] https://go.dev/
[5] https://hub.docker.com/explore
[6] https://react.dev/

```
1  # Stage 1: Build the React app
2  FROM node:20-alpine AS build
3  WORKDIR /app
4  COPY package.json package-lock.json ./
5  RUN npm install
6  COPY . .
7  RUN npm run build
8
9  # Stage 2: Serve the React app with a static server
10 FROM nginx:alpine
11 COPY --from=build /app/dist /usr/share/nginx/html
12 EXPOSE 80
13 CMD ["nginx", "-g", "daemon off;"]
```

Listing 3: A snippet containing Dockerfile instructions for creating admin website image

6.2.4. OT Application

A simple Go program was created to simulate the OT application. The application called the query API with a specified interval to generate traffic and exercise load on the system. Some invalid queries were added as well, which were reflected in the monitoring part of the application stack.

6.2.5. Monitoring

For monitoring the application stack, the following components were introduced:

- Prometheus[7]: scraping and exporting of metrics
- cAdvisor[8]: collecting metrics about running containers
- Node exporter[9]: collecting metrics about Docker hosts
- Postgres exporter[10]: collecting metrics about PostgreSQL database
- Grafana[11]: visualization of logs and metrics through custom or pre-built dashboards

After configuring the different monitoring components, we were able to observe the application stack in terms of performance and metrics. Screenshots of the different resulting dashboards are included in the appendix, visible in figs. A.3 to A.6.

[7] https://prometheus.io/
[8] https://github.com/google/cadvisor
[9] https://github.com/prometheus/node_exporter
[10] https://github.com/prometheus-community/postgres_exporter
[11] https://grafana.com/

6.2.6. CI/CD

For the CI/CD component, GitLab Community Edition[12] was added to the stack. Additionally, a GitLab runner was also added in order to run the build and deploy jobs of the application pipelines. During the practical tests, this component was deemed as redundant since we achieved continuous deployments using different Docker images hosted on Docker Hub. This allowed us to focus more in-depth on the orchestration candidate's features, instead of configuring CI/CD pipelines for automated tests and builds, since that was not the primary target of this research.

6.2.7. Overview

All components mentioned in the previous section were centrally configured with Docker Compose. This configuration file—accessible in this project's accompanying GitHub repository[13]—defined the whole application stack, as well as the specific configurations for each container, such as mapped ports, persistent storage, environment variables, dependencies and networking. With this artifact, the whole application could be deployed to different environments, and it formed the basis of which components were required to be orchestrated by different CO candidates. As an example, listing 4 shows the entire application running locally on a development laptop.

```
1  > docker compose up -d
2  [+] Running 16/16
3    Network argo_argo-backend       Created      0.1s
4    Container postgres              Started      0.4s
5    Container memcached             Started      0.4s
6    Container gitlab                Started      0.5s
7    Container admin-api             Started      0.5s
8    Container query-api             Started      0.5s
9    Container argo-gitlab-runner-2  Started      0.5s
10   Container argo-gitlab-runner-1  Started      0.7s
11   Container admin-site            Started      0.6s
12   Container prometheus            Started      0.7s
13   Container ot-app                Started      0.7s
14   Container query-site            Started      0.7s
15   Container cadvisor              Started      0.9s
16   Container grafana               Started      0.9s
17   Container postgres-exporter     Started      0.8s
18   Container node-exporter         Started      0.9s
```

Listing 4: A snippet of a Docker command showing an overview of ARGO running locally

[12] https://about.gitlab.com/install/?version=ce

[13] https://github.com/cadeke/argo/blob/main/base/docker-compose.yml

6.3. Evaluation Criteria

Prior to starting with the implementation, we reviewed the evaluation criteria once more, so it was clear which aspects deserved more focus during the testing of each candidate. Additionally, we opted for a grading system which allowed us to quantify our experiences in a more structured and comparable way.

6.3.1. List of Criteria

These criteria were deduced from the use-case components mentioned in section 5.1.3, and from the optional and required features from section 5.2. During the implementation, we evaluated how each CO candidate handled a specific requirement, so we could deliver structured and reproducible results. The following list contains some optional criteria as well, which were not the focus of the comparison, but were interesting to investigate nonetheless.

- Scheduling of components (i.e. websites, APIs, databases, monitoring elements)
- Scaling capabilities (e.g. manual scaling, auto-scaling based on metrics)
- Service discovery for internal addressing
- Fault tolerance during container or node failure
- Highly available applications during updates (e.g. rolling updates, rollbacks)
- Network segmentation between the different components, only allowing required communication
- Shared storage between nodes to achieve highly available data volumes
- Load balancing of internal and external requests
- (optional) Encrypted internal communication (e.g. using mTLS)
- (optional) Centralized secret management

6.3.2. Grading Schema

In order to quantify the findings, we introduce a formula which we used while evaluating the container orchestration candidates during the practical testing. The following equation encapsulates the method of calculating a score for each candidate.

$$\sum_{f} \alpha_f \cdot (x_f + y_f) = z$$

x_f Support for a specific feature as presented in a candidate's documentation. Ranges from 1 to 5, detailed in table 6.1.

y_f Experience with a specific feature during the implementation process, measured by the effort required to implement it for the defined use-case. Ranges from 1 to 5, detailed in table 6.1.

α_f Constant that differentiates between required and optional features. Optional features have half of the weight of required features.

z The total score for a candidate, based on documentation support and implementation experience, weighted by feature importance.

f The specific feature that is being analyzed.

Table 6.1 presents the detailed grading schema which was employed to assign a score during the implementation process.

Score	**Documentation support** x_f	**Implementation experience** y_f
1	No mention in documentation, virtually no support	Not possible to implement within given timeframe
2	Minimal mentions in documentation, extra research	Major effort, only subset of implementation possible
3	Some mentions in documentation, extra research	Major effort, but full implementation possible
4	Substantial mentions in documentation, no extra research	Minor effort, some extra research or examples
5	Detailed documentation and examples, first level support	Minimal effort or expected effort

Table 6.1.: An overview of the grading schema

With the evaluation criteria, formula and grading schema defined, the selected candidates could be subjected to the implementation tests and their strengths and limitations could be analyzed in a structured manner. Both candidates' implementation process is covered in the subsequent sections, with the results of this process being discussed in chapter 7.

6.4. Docker Swarm

6.4.1. Initial Configuration

Before setting up Swarm, we updated all packages on all nodes, installed Docker and added the current user to the "docker" group, so it was able to run Docker as a non-root user. This configuration was mainly accomplished with Ansible playbooks, which helped us to execute repetitive steps on the different VMs, as highlighted earlier in this chapter.

With Docker, configuration can be done in an imperative way with the Docker CLI[14], or in a declarative

[14] `https://docs.docker.com/reference/cli/docker/`

way with Docker Compose[15] files. Similar to the application running locally, we opted for the latter option because it produces more repeatable results and is generally considered to be a better practice. As a resulting artifact of this section, the final Docker Compose file can be found in the accompanying GitHub repository[16].

6.4.2. Shared Storage

To achieve a shared storage pool, we investigated Docker's offerings in terms of volumes and storage drivers. According to the documentation, there are several options of sharing data between nodes [78]. Managed cloud storage is mentioned as an option, but since Docker volumes form an abstraction, other types of storage can be mounted to the containers as well. In order to follow the philosophy of the use-case, we chose to investigate block storage devices.

We opted for a distributed file system called GlusterFS[17] in order to create a pool of storage to which each node contributes 50 GiB. With this approach, we can achieve HA and fault-tolerance for the data, since it will be replicated by GlusterFS. We acknowledge that there are some uncertainties about the status of GlusterFS at the time of writing [79], since some projects are dropping support for it [80]–[82]. However, for this use-case, any storage solution can be used, and it is not directly linked to Swarm. For example, alternatives such as Ceph[18], MooseFS[19] or cloud storage solutions are also a viable option.

Swarm supports various volume drivers in order to abstract the storage solution from the application orchestration [83], which can be installed using the "plugin" syntax. Examples of these drivers are technologies such as rclone[20], network file shares (NFS) or Samba shares. However, as far as we could verify, Swarm does not support the container storage interface (CSI) which is a standard for integrations with external storage systems, most notably leveraged by Kubernetes.

After following the documentation steps, we were able to create a shared volume which replicates the data to all nodes of the storage cluster. Listings 32 and 33 show the connected nodes and the replicated volume respectively.

[15] https://docs.docker.com/reference/compose-file/
[16] https://github.com/cadeke/argo/blob/main/swarm/argo-stack.yml
[17] https://www.gluster.org/
[18] https://ceph.io/en/
[19] https://moosefs.com/
[20] https://rclone.org/

6.4.3. Swarm Setup

A Swarm cluster consists out of two types of Docker hosts: managers and workers. Managers are responsible for delegating tasks and workers run the so-called Swarm services [84]. These services can be defined similarly to Docker Compose, so it is possible to use a declarative approach to manage components such as scaling, networking, storage, etc.

In order to achieve HA in the cluster, we followed Docker's recommendation of assigning multiple manager nodes [85]. Since we have eight nodes at our disposal, we decided on a 3/5 split (three manager nodes and five worker nodes), making the cluster resilient with the failure of maximum one manager node.

Listing 5 shows an overview of the Swarm nodes, with three manager nodes and five worker nodes present in the newly-created Swarm cluster.

```
1  student@vm01:~> docker node ls
2  ID                            HOSTNAME   STATUS    AVAILABILITY   MANAGER STATUS   ENGINE VERSION
3  anvacpekzrc35twzg5y3r1gic *   vm01       Ready     Active         Leader           28.0.4
4  i5kgrnmecaqa44rqbbcw52db6     vm02       Ready     Active         Reachable        28.0.4
5  yzrumknlvx6kd383sa7o2jrop     vm03       Ready     Active         Reachable        28.0.4
6  z9uoam07pax0uuwn6asqbf0fp     vm04       Ready     Active                          28.0.4
7  yezg2ojyvca980iehdo0hisqk     vm05       Ready     Active                          28.0.4
8  7dqcqy9pdzirqjuia96phycwp     vm06       Ready     Active                          28.0.4
9  teesrnys35963sbyzp07o20tk     vm07       Ready     Active                          28.0.4
10 h500eow7r8ps171osflrirni9     vm08       Ready     Active                          28.0.4
```

Listing 5: A snippet of a Docker command showing an overview of Swarm nodes

6.4.4. Scheduling

For the deployment of the application to the Swarm cluster, the steps outlined in the documentation were followed in order to make the right adjustments to the deployment file. We configured the cluster to assign the containers to worker nodes as much as possible, to keep the manager nodes free for orchestration tasks. We also added two replicas for the OT application which would generate additional traffic in the cluster.

After deploying this configuration, we could see the different containers running on the nodes. The majority of them were running on the worker nodes, as specified earlier. Only some containers, which need to be present on every node for monitoring purposes, were running on the manager nodes. Figure 6.1 presents an overview of the containers running on each node as part of the deployment.

```
student@vm01:~$ docker stack ps argo-stack
ID             NAME                                             IMAGE                                NODE      DESIRED STATE   CURRENT STATE
qzgkawzjk6iw   argo-stack_a-api.1                               cadeke/argo-a-api:latest             vm06      Running         Running 2 minutes ago
uyj00urfmzs8   argo-stack_a-site.1                              cadeke/argo-a-site:latest            vm06      Running         Running 2 minutes ago
rjj75dkz4sjz   argo-stack_cadvisor.7dqcqy9pdzirqjuia96phycwp    gcr.io/cadvisor/cadvisor:latest      vm06      Running         Running about a minute ago
k8mr4a8zypvv   argo-stack_cadvisor.anvacpekzrc35twzg5y3r1gic    gcr.io/cadvisor/cadvisor:latest      vm01      Running         Running about a minute ago
gpto48euffve   argo-stack_cadvisor.h500eow7r8ps171osflrirni9    gcr.io/cadvisor/cadvisor:latest      vm08      Running         Running about a minute ago
wooxdd7frzny   argo-stack_cadvisor.i5kgrnmecaqa44rqbbcw52db6    gcr.io/cadvisor/cadvisor:latest      vm02      Running         Running about a minute ago
t95dv9j7538x   argo-stack_cadvisor.teesrnys35963sbyzp07o20tk    gcr.io/cadvisor/cadvisor:latest      vm07      Running         Running about a minute ago
p3gc5g4vlnrs   argo-stack_cadvisor.yezg2ojyvca980iehdo0hisqk    gcr.io/cadvisor/cadvisor:latest      vm05      Running         Running about a minute ago
ufpzgoxnhblf   argo-stack_cadvisor.yzrumknlvx6kd383sa7o2jrop    gcr.io/cadvisor/cadvisor:latest      vm03      Running         Running about a minute ago
rbzs1euzmld5   argo-stack_cadvisor.z9uoam07pax0uuwn6asqbf0fp    gcr.io/cadvisor/cadvisor:latest      vm04      Running         Running about a minute ago
0snow1fhsgsf   argo-stack_gitlab-runner.1                       gitlab/gitlab-runner:latest          vm08      Running         Running 2 minutes ago
z3m2ida1aabd   argo-stack_gitlab-runner.2                       gitlab/gitlab-runner:latest          vm07      Running         Running 2 minutes ago
htdt565nhf5p   argo-stack_gitlab.1                              gitlab/gitlab-ce:latest              vm06      Running         Starting 25 seconds ago
lz7qhdhm7r5b   argo-stack_grafana.1                             grafana/grafana:latest               vm05      Running         Running 2 minutes ago
a5sfwqoeozn5   argo-stack_memcached.1                           memcached:latest                     vm08      Running         Running 2 minutes ago
puy5zr52ie8q   argo-stack_node-exporter.7dqcqy9pdzirqjuia96phycwp   prom/node-exporter:latest        vm06      Running         Running 2 minutes ago
fvhd2a7drsv9   argo-stack_node-exporter.anvacpekzrc35twzg5y3r1gic   prom/node-exporter:latest        vm01      Running         Running 2 minutes ago
lg177isoaq1w   argo-stack_node-exporter.h500eow7r8ps171osflrirni9   prom/node-exporter:latest        vm08      Running         Running 2 minutes ago
jkq3w40l4x66   argo-stack_node-exporter.i5kgrnmecaqa44rqbbcw52db6   prom/node-exporter:latest        vm02      Running         Running 2 minutes ago
u7ynmwch07wi   argo-stack_node-exporter.teesrnys35963sbyzp07o20tk   prom/node-exporter:latest        vm07      Running         Running 2 minutes ago
fxrn1z0ykp95   argo-stack_node-exporter.yezg2ojyvca980iehdo0hisqk   prom/node-exporter:latest        vm05      Running         Running 2 minutes ago
i0wu17eioqew   argo-stack_node-exporter.yzrumknlvx6kd383sa7o2jrop   prom/node-exporter:latest        vm03      Running         Running 2 minutes ago
wj7xt9taja5x   argo-stack_node-exporter.z9uoam07pax0uuwn6asqbf0fp   prom/node-exporter:latest        vm04      Running         Running 2 minutes ago
z10boe9kyxve   argo-stack_ot-app.1                              cadeke/argo-ot-app:latest            vm05      Running         Running 2 minutes ago
bbrils2kgx6m   argo-stack_ot-app.2                              cadeke/argo-ot-app:latest            vm04      Running         Running 2 minutes ago
xaswo9w8oz46   argo-stack_postgres-exporter.1                   wrouesnel/postgres_exporter:latest   vm08      Running         Running 2 minutes ago
zza1t06h0yzq   argo-stack_postgres.1                            postgres:latest                      vm05      Running         Running 2 minutes ago
chd4jz49qxjb   argo-stack_prometheus.1                          prom/prometheus:latest               vm04      Running         Running 2 minutes ago
z0exu8k9w8ck   argo-stack_query-api.1                           cadeke/argo-q-api:latest             vm07      Running         Running 2 minutes ago
66nvzjt05g2v   argo-stack_query-site.1                          cadeke/argo-q-site:latest            vm04      Running         Running 2 minutes ago
```

Figure 6.1.: An overview of the ARGO application running on Swarm nodes

Using the Grafana instance, we imported a pre-built dashboard which presents a visual overview of the Swarm cluster, shown in fig. A.7. This dashboard gives us the same information as the CLI output, such as the amount of nodes (8), amount of containers (27) and replicas per service (8 replicas of node-exporter and cAdvisor, 2 replicas of OT application, 2 replicas of the GitLab runner and 1 replica of the other containers).

An important aspect of scheduling is the ability to drain a node of its running containers. This enables administrators to safely reallocated workloads to the different available nodes, when the need for maintenance or troubleshooting on that specific node arises. In Swarm, this can be done by changing a node's availability. Running workloads will be shutdown and restarted on other available nodes. This process is shown in listing 34.

6.4.5. Scaling

As mentioned in the previous section, the OT application was configured with two replicas. Similarly, we could define scaling requirements for each of the defined components, or adjust the scaling of the active deployment via the Docker CLI. As an example, we scaled the query API and the OT application to four replicas each, while the application was deployed on the Swarm cluster. Listing 6 shows the relevant commands and their output. As expected, these changes were also reflected in the monitoring dashboards. Note that all additional containers were also started on worker nodes, as specified by the deployment configuration.

```
1  student@vm01:~/stacks> docker service scale argo-stack_ot-app=4 -d
2  argo-stack_ot-app scaled to 4
3  student@vm01:~/stacks> docker service scale argo-stack_query-api=4 -d
4  argo-stack_query-api scaled to 4
5  student@vm01:~/stacks> docker service ps argo-stack_ot-app
6  ID             NAME                  IMAGE                        NODE      DESIRED STATE   CURRENT STATE
7  rmn28jxr6clr   argo-stack_ot-app.1   cadeke/argo-ot-app:latest   vm08      Running         Running 53 minutes ago
8  gp58bp3k10j6   argo-stack_ot-app.2   cadeke/argo-ot-app:latest   vm07      Running         Running 53 minutes ago
9  goucht9a4tz0   argo-stack_ot-app.3   cadeke/argo-ot-app:latest   vm04      Running         Running 19 seconds ago
10 73wttllgd4cw   argo-stack_ot-app.4   cadeke/argo-ot-app:latest   vm06      Running         Running 19 seconds ago
11 student@vm01:~/stacks> docker service ps argo-stack_query-api
12 ID             NAME                     IMAGE                       NODE      DESIRED STATE   CURRENT STATE
13 vtbxwhsztnsc   argo-stack_query-api.1   cadeke/argo-q-api:latest   vm06      Running         Running 53 minutes ago
14 wqbo7mp7avk5   argo-stack_query-api.2   cadeke/argo-q-api:latest   vm05      Running         Running 17 seconds ago
15 xnydisnlgnds   argo-stack_query-api.3   cadeke/argo-q-api:latest   vm08      Running         Running 17 seconds ago
16 dzuvgzggs4s4   argo-stack_query-api.4   cadeke/argo-q-api:latest   vm07      Running         Running 17 seconds ago
```

Listing 6: A snippet of Docker commands showing the scaling of services in the Swarm cluster

Swarm does not support any form of dynamic scaling. However, it is possible to create custom logic that executes the Docker CLI commands based on metrics, such as CPU load or incoming requests on a given node. As an example, “orbiter” is an implementation of an auto-scaler which works with Swarm [86]. Other implementations with extra components such as load balancers and pipelines to detect changes are also possible, but go beyond the scope of this testing scenario.

6.4.6. Service Discovery

Swarm provides an internal DNS component that makes communication using friendly domain names possible [87], [88]. We already implicitly leveraged this feature when declaring the deployments, because we were able to reference other services by their defined names—such as “postgres”, “query-api”, “admin-site”, “grafana”, etc.

We were able to demonstrate the name resolving by adding a container based on Alpine[21], which contains the “ping” utility. When connected to this container, we were able to reach other services by their defined name, so without knowing their IP address or the node they were running on. Listing 7 shows the result of pinging different services from the “debug” container.

[21] https://www.alpinelinux.org/

```
1  student@vm04:~> docker exec -it argo-stack_debug.1.auw5g70l4n938thf17se48ovz sh
2  / # ping postgres
3  PING postgres (192.168.0.46): 56 data bytes
4  64 bytes from 192.168.0.46: seq=0 ttl=64 time=0.110 ms
5  64 bytes from 192.168.0.46: seq=1 ttl=64 time=0.097 ms
6  64 bytes from 192.168.0.46: seq=2 ttl=64 time=0.106 ms
7  ^C
8  --- postgres ping statistics ---
9  3 packets transmitted, 3 packets received, 0% packet loss
10 round-trip min/avg/max = 0.097/0.104/0.110 ms
11 / # ping grafana
12 PING grafana (192.168.0.51): 56 data bytes
13 64 bytes from 192.168.0.51: seq=0 ttl=64 time=0.176 ms
14 64 bytes from 192.168.0.51: seq=1 ttl=64 time=0.127 ms
15 64 bytes from 192.168.0.51: seq=2 ttl=64 time=0.109 ms
16 ^C
17 --- grafana ping statistics ---
18 3 packets transmitted, 3 packets received, 0% packet loss
19 round-trip min/avg/max = 0.109/0.137/0.176 ms
20 / # ping ot-app
21 PING ot-app (192.168.0.33): 56 data bytes
22 64 bytes from 192.168.0.33: seq=0 ttl=64 time=0.155 ms
23 64 bytes from 192.168.0.33: seq=1 ttl=64 time=0.096 ms
24 64 bytes from 192.168.0.33: seq=2 ttl=64 time=0.094 ms
25 ^C
26 --- ot-app ping statistics ---
27 3 packets transmitted, 3 packets received, 0% packet loss
28 round-trip min/avg/max = 0.094/0.115/0.155 ms
29 / # ping memcached
30 PING memcached (192.168.0.7): 56 data bytes
31 64 bytes from 192.168.0.7: seq=0 ttl=64 time=0.204 ms
32 64 bytes from 192.168.0.7: seq=1 ttl=64 time=0.118 ms
33 64 bytes from 192.168.0.7: seq=2 ttl=64 time=0.099 ms
34 ^C
35 --- memcached ping statistics ---
36 3 packets transmitted, 3 packets received, 0% packet loss
37 round-trip min/avg/max = 0.099/0.140/0.204 ms
```

Listing 7: A snippet of ping commands, testing internal DNS in Swarm cluster

6.4.7. Fault Tolerance

When a worker node becomes unavailable due to a system crash or the Docker service not responding, the Swarm cluster reacts and recreates the replicas of the impacted services on a different available worker node. To test this, we deactivated node “vm08” by stopping the Docker service. This event was logged by the Swarm managers, as visible in the log output from listing 8. As a result of this action, the impacted worker node was considered to be “Down” by the Swarm managers and its workload was moved to a different worker node. This process can be observed in fig. A.8. When we re-enabled the Docker service, worker node “vm08” was marked as “Ready” again and some services were assigned to it, similar to the state before the shutdown.

When a manager node experiences issues, a new leader will be elected from the available manager nodes. Since we configured three manager nodes, we could afford to lose one at most. This process is demonstrated in fig. A.9. First, we deactivated Docker on “vm02”, which was the elected leader at the time. We could then observe the steps taken by the other manager nodes to elect a new leader node and the changes being communicated to the other nodes. At the end, we re-activated “vm02”, so it could join the cluster again as available manager node. Figure A.10 shows the results of this event. All services were recreated, the ones running on “vm02” were shutdown, and after re-joining, “vm02” becomes an available manager node again.

```
1 student@vm01:~> sudo journalctl -fu docker.service
2 # output omitted
3 Mar 28 19:38:15 vm01 dockerd[62385]: time="2025-03-28T19:38:15.152133007Z" level=info msg="Node a158206a14a6 change state
  ↪ NodeActive --> NodeLeft"
4 Mar 28 19:38:15 vm01 dockerd[62385]: time="2025-03-28T19:38:15.152868739Z" level=info msg="vm01(191b72da74cb): Node leave
  ↪ event for a158206a14a6/10.203.96.118"
5 Mar 28 19:38:15 vm01 dockerd[62385]: time="2025-03-28T19:38:15.268658994Z" level=info msg="Node a158206a14a6/10.203.96.118,
  ↪ left gossip cluster"
```

Listing 8: A snippet of Docker logs showing a node leaving the Swarm cluster

6.4.8. Networking Segmentation

Thus far, we had only configured one network, “argo-backend”, over which all traffic was flowing. In order to improve the security of the application, segmentation of networks could be applied in order to minimize the pivoting possibles of a possible attacker. Swarm allows us to define multiple network and which services should be connected to each network. After adjusting the configuration and redeploying to the cluster, we could see the networks were created with the specified subnets, shown in listing 9.

When we repeated the same connectivity tests as in section 6.4.6, we could observe that only the services which resided in the same network as the “debug” container—in this case the “monitoring” network—were reachable. Others, such as “query-api” or “postgres”, were not reachable anymore because we didn’t explicitly define that network connection. Listing 35 shows different “ping” attempts to demonstrate the network segmentation. Note that the DNS entries are not known by the “debug” container, but that also using the IP addresses directly is not allowed by the segmentation.

It is worth noting that during the testing, we encountered a known issue with Swarm’s networking. After several deployments, the containers were not being assigned to worker nodes anymore, even though the nodes were online and reachable. Swarm’s logs indicated an issue with IP addressing and when we investigated further, we found other Swarm users who experienced a similar issue [89]–[91]. In order to resolve

this, we had to specify larger subnet address spaces than a /24 subnet mask (SNM), which only support 254 available addresses. For the segmentation setup, we opted for a SNM of /18, giving us 16,382 useable addresses. This way, Swarm had enough addresses to use for creating containers. Similarly, when testing the ingress routing mesh in section 6.4.10, we encountered the same problem, which required us to delete and recreate the "ingress" network in order to define a larger subnet.

```
1  student@vm01:~/stacks> docker network ls | grep 'argo'
2  356wcggu7kye   argo-stack_apis         overlay   swarm
3  oa9bn7zpxs2b   argo-stack_dbs          overlay   swarm
4  ou8gfxg1wuc0   argo-stack_monitoring   overlay   swarm
5  wcijq30c1wza   argo-stack_sites        overlay   swarm
6  student@vm01:~/stacks> docker network inspect argo-stack_apis argo-stack_dbs argo-stack_monitoring argo-stack_sites | grep
   ↪  "Subnet"
7  "Subnet": "192.168.64.0/18",
8  "Subnet": "192.168.128.0/18",
9  "Subnet": "192.168.192.0/18",
10 "Subnet": "192.168.0.0/18",
```

Listing 9: A snippet of Docker commands showing an overview of networks in Swarm cluster

6.4.9. Continuous Deployments

Swarm enables continuous deployments through rolling updates and automated rollbacks when updating applications [92]. We investigated this mechanism by creating two new versions[22] of the OT application. While version 1.1 just contained a minor implementation update, version 1.2 had a critical implementation flaw which would crash the application at startup.

As a start, we defined the relevant update, rollback and restart policies for OT application to achieve HA. A snippet of the configuration file can be seen in listing 10. Then, we went through the following steps:

1. Deploying the OT application with the current stable version (v1.0)
2. Updating to new working version (v1.1)
3. Updating to new broken version (v1.2)
4. Rolling back to latest stable version (v1.1)

The steps described above were executed, and the results are shown in fig. A.11. Swarm was able to detect a faulty update and rolled back as defined in the configuration in listing 10. Note that as soon as one instance of the new version failed to start correctly, the update was cancelled and the impacted instance was rolled back to the previous stable version. Since we defined the "parallelism" parameter as "1", the application

[22] https://hub.docker.com/repository/docker/cadeke/argo-ot-app/tags

updated one instance at a time. In a larger application landscape, this parameter would probably be increased in order to speed up deployments of new application versions. This balance needs to be evaluated, since a higher amount of parallel updates could incur downtime of the application in case of an unsuccessful update.

```
1      deploy:
2        mode: replicated
3        replicas: 4
4        update_config:
5          parallelism: 1
6          order: start-first
7          failure_action: rollback
8          delay: 10s
9        rollback_config:
10         parallelism: 1
11         order: stop-first
12       restart_policy:
13         condition: any
14         delay: 5s
15         max_attempts: 5
16         window: 120s
```

Listing 10: A YAML snippet showing CD configuration in Swarm cluster

6.4.10. Load Balancing

Swarm provides load balancing features for both internal and external traffic. Internal load balancing is managed by the internal DNS layer that Swarm provides, discussed in section 6.4.6. When an internal service name is used, the requests will be routed by the managers to the workers that are running a replica of the requested service. External requests can be managed through an ingress routing mesh. This enables nodes to accept incoming requests on published ports and forward them to the relevant service, regardless of on which node it is running [93]. Listing 11 shows the result of the services that were published with this mechanism. We chose to publish both websites, both APIs and the Grafana instance, which were then available for clients outside the Swarm cluster, as shown on fig. A.14. Ports 80/tcp, 81/tcp, 8080/tcp, 8081/tcp and 3000/tcp were mapped to internal ports so that requests from outside the cluster to reach the relevant service.

Note that we accessed the services via “fake” domain names, by adding entries for the IP addresses of the Swarm nodes to the “/etc/hosts” file on the client. This was done as an alternative to setting up an internal DNS system, which is beyond the scope of this research. Listing 36 shows the IP addresses of the nodes linked to the domain names, such as “vm01.lab” and “argo.lab”.

```
1  student@vm01:~/stacks> docker service ls
2  ID             NAME                      MODE         REPLICAS   IMAGE                                      PORTS
3  0bkfxgwyaxed   argo_admin-api            replicated   1/1        cadeke/argo-a-api:latest                   *:8081->8080/tcp
4  o43bwx6rf9dt   argo_admin-site           replicated   1/1        cadeke/argo-a-site:latest                  *:81->80/tcp
5  shffjtuqu40f   argo_cadvisor             global       8/8        gcr.io/cadvisor/cadvisor:latest
6  w62isxkxn5q7   argo_grafana              replicated   1/1        grafana/grafana:latest                     *:3000->3000/tcp
7  swlbridujx5e   argo_memcached            replicated   1/1        memcached:latest
8  5jumc3jhs4aw   argo_node-exporter        global       8/8        prom/node-exporter:latest
9  fki27vumojaw   argo_ot-app               replicated   2/2        cadeke/argo-ot-app:v1.1
10 q1af73patsks   argo_postgres             replicated   1/1        postgres:latest
11 ver1jegnvzqe   argo_postgres-exporter    replicated   1/1        wrouesnel/postgres_exporter:latest
12 f8ww90gzdqhu   argo_prometheus           replicated   1/1        prom/prometheus:latest
13 gbe3ibw9pqdx   argo_query-api            replicated   1/1        cadeke/argo-q-api:latest                   *:8080->8080/tcp
14 kpkxqn0bgjbo   argo_query-site           replicated   1/1        cadeke/argo-q-site:latest                  *:80->80/tcp
```

Listing 11: A snippet of a Docker command, showing the published ports in Swarm cluster

Exposing services via the routing mesh is made relatively simple by Swarm. However, this could be improved by introducing a dedicated load balancer. This service could act as the gateway between the internal cluster and the external environment, providing insights about requests and metrics, while enhancing security with TLS-termination and shielding sensitive services like APIs. A suitable candidate for this use-case would be Traefik[23], which is a popular choice for containerized and cloud-native setups using technologies such as Docker or Kubernetes. However, we were not able to successfully achieve this within the given timeframe of this research, since other aspects of Swarm deserved to be investigated instead.

6.4.11. Encrypted Communication

Swarm comes with built-in public key infrastructure (PKI), which enables encrypted communication between nodes with mTLS. When creating the cluster, the manager node creates a root certificate authority (CA) with a key pair. When a new node joins the swarm, the manager issues a new certificate [94]. An example of the root CA certificate and of a node certificate can be found in listing 37. It is possible to specify an externally-generated root CA certificate and pass it through when initializing the cluster. Encrypted communication between nodes is especially useful when running the Swarm cluster in an untrusted environment, for example, a hybrid cloud scenario with servers in multiple different clouds that need to communicate. In this use-case, the traffic resides in an internal network, so there is less risk involved, though we still wanted to investigate and demonstrate this feature.

By default, only the control and management related traffic is encrypted [95]. However, when creating an

[23] https://traefik.io/traefik/

overlay network in a Swarm scenario, we can specify the option to encrypt the communication [96]. This enables IPsec encryption on layer 3, effectively protecting container-to-container communication. As an example, we captured the traffic—using tcpdump[24]—on a worker node before and after enabling encryption on the Swarm networks. When analyzing this traffic with WireShark[25], we could initially see the HTTP requests happening in plain text, visible in fig. A.12. After redeploying the application with encrypted networks, this traffic was no longer visible. Instead, only ESP packets were present, indicating encrypted IPsec traffic, visible in fig. A.13. The complete traffic captures can be found in the accompanying GitHub repository[26].

On top of this, Swarm provides a feature to secure the cluster's TLS key called "auto-locking", which requires an additional key to be provided when restarting the manager nodes [97]. With this feature, only authorized system administrators can restart the manager nodes and possibly extract the TLS encryption key.

6.4.12. Secret Management

Swarm can manage secrets centrally and transmit them to the services that require access to them. The secrets are encrypted at rest and in transit, and are only available to services that are explicitly given access and only when those services are running [98]. Secrets can be updated or rolled back without disrupting running services.

After adjusting the configuration file of the deployment, we created the secret in the Swarm cluster using the Docker CLI, as shown in listing 12. Then, after redeploying the application on the cluster, we were able to log in to Grafana with the updated password. It was also possible to check the mounted secret file in the Grafana container itself, as shown in listing 13, with restrictive file permissions.

```
1  student@vm01:~/stacks> echo "verygoodpassword" | docker secret create grafana_admin_password -
2  afpyy0s7qixro56vui2xsztzn
3  student@vm01:~/stacks> docker secret ls
4  ID                          NAME                     DRIVER    CREATED         UPDATED
5  afpyy0s7qixro56vui2xsztzn   grafana_admin_password             4 seconds ago   4 seconds ago
```

Listing 12: A snippet of Docker commands showing the creation of a secret in Swarm

[24] https://www.tcpdump.org/
[25] https://www.wireshark.org/
[26] https://github.com/cadeke/argo/tree/main/pcap/swarm

```
1 student@vm06:~> docker exec -it argo_grafana.1.bmb2tj8nyo0tujrt7hks3p151 ls -la /run/secrets
2 total 12
3 drwxr-xr-x    2 root     root          4096 Mar 29 16:05 .
4 drwxr-xr-x    1 root     root          4096 Mar 29 16:05 ..
5 -r--r--r--    1 root     root            17 Mar 29 16:05 grafana_admin_password
6 student@vm06:~> docker exec -it argo_grafana.1.bmb2tj8nyo0tujrt7hks3p151 cat /run/secrets/grafana_admin_password
7 verygoodpassword
```

Listing 13: A snippet of various commands, verifying a mounted secret in a Swarm service

6.5. HashiCorp Nomad

6.5.1. Initial Configuration

Since Docker was already installed on the VMs during the previous testing stages, no additional configuration was needed in order to get started with Nomad. We leveraged snapshots in order to roll back to a previous stable state of the VMs, so the upcoming tests were not effected by changes made during the previous implementation of Swarm. As a resulting artifact of this section, the final Nomad files can be found in this project's accompanying GitHub repository[27].

6.5.2. Shared Storage

Nomad supports external storage volumes through storage plugins that adhere to the Container Storage Interface (CSI) [99]. This enables the usage of third party plugins for storing data that can be shared between nodes and jobs in the Nomad cluster. Since we already configured a GlusterFS pool in section 6.4.2, we decided to use this directly as host volumes, without configuring it via CSI, for the shared storage requirement. This choice was made to keep the additional configuration steps to a minimum, however, we acknowledge that there is also a CSI plugin for GlusterFS, which could have been a more optimal way of integrating it with Nomad. As an additional note, support for the CSI plugin for GlusterFS also seems to have dwindled at the time of writing, following the negative trend around GlusterFS that was mentioned in section 6.4.2.

6.5.3. Nomad Setup

A Nomad cluster defines two types of nodes: servers and clients [100]. Servers handle the delegation and synchronization of tasks on available clients, who run the scheduled jobs. Jobs can be defined in HCL or JSON for a declarative approach, or using the Nomad CLI. Each job can have one or more groups, which

[27] https://github.com/cadeke/argo/tree/main/nomad

consist of one or more tasks. Tasks are tightly coupled and will be scheduled on the same node, so splitting up the application components into the necessary amount of groups is crucial, depending on the given scenario [101].

In order to achieve HA in the cluster, HashiCorp's recommendation to use three or five server nodes in a production scenario were followed [102]. This provided us with a failure tolerance of maximum one server node, similar to the cluster setup in Swarm.

Prior to setting up Nomad, we configured Consul on the nodes to assists Nomad in service discovery, which is recommended for enterprise-grade Nomad deployments. Since Consul is also a HashiCorp service, they are highly interoperable and meant to be used together. After following the production deployment guide [103] for Consul, and connecting it together with Nomad [104], we could see the available client nodes and server nodes in the Nomad UI as shown in fig. 6.2. Section 6.5.6 goes into more detail about Consul and its functionalities.

Figure 6.2.: An overview of clients and servers in Nomad cluster

6.5.4. Scheduling

In order to deploy the application on the Nomad cluster, each major application component was defined in a Nomad job. In comparison to Swarm, Nomad does not schedule workloads on server nodes by default, as

they are solely responsible for orchestration tasks. This meant that the application was scheduled to run on the five available client nodes. Similar to Swarm, some containers known as “system” jobs—which were related to monitoring or load balancing—were running on all five client nodes. Others were scheduled by Nomad to run on any available client node, as expected.

After deploying each component on the Nomad cluster, we observed the running application components in the UI provided by Nomad. Figure 6.3 shows an overview of the different jobs and their allocation status, while fig. A.15 provides us with an additional topology view of the Nomad cluster, the running allocations and the available resources. With this built-in UI, Nomad presents a graphical overview of the cluster and the different application components. It allowed us to restart failed jobs, view problematic allocations and dive into logs related to specific containers or Nomad nodes themselves. It also fulfilled the monitoring required, as it provides us with relevant metrics about the application and the underlying nodes running the individual components.

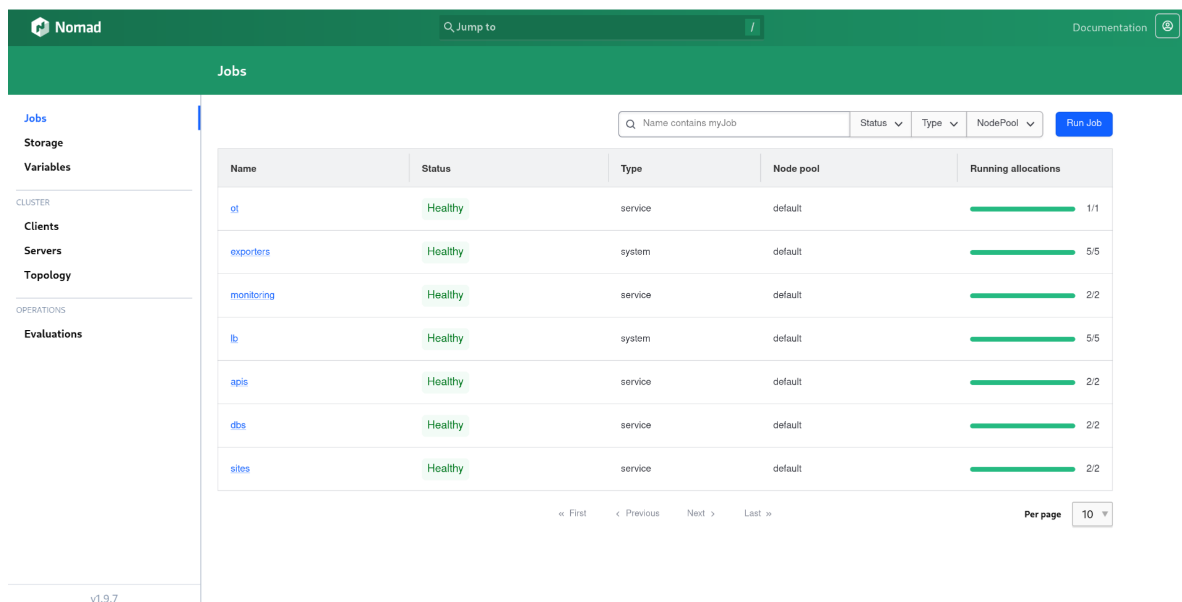

Figure 6.3.: An overview of multiple jobs deploying ARGO components on Nomad cluster

We ran into some issues related to file permissions when scheduling the “postgres” instance. This forced us to add a constraint to always schedule this container on a specific node, “vm04” in this case. Since the database components of the application are not the main focus and could be potentially abstracted, we chose to move forward in favor of investigating the other aspects of Nomad.

In Nomad, we validated the node draining mechanism using the Nomad UI. We marked client node “vm04”

to be drained within a timespan of two minutes, which can be observed in fig. A.16. After a few minutes, all containers were moved and the node was no longer eligible to receive scheduling instructions, shown in fig. A.17.

6.5.5. Scaling

In order to demonstrate the scaling capabilities, the OT application instance count was increased from one to ten, using the Nomad CLI, shown in listing 39. As a result of this, the newly provisioned containers were started on available nodes while the health of all instances was monitored, to ensure no issues occurred during the scaling. The node IDs could be cross-referenced with fig. 6.2 to verify on which nodes the additional container instances were started. Figure 6.4 visualizes this deployment, indicating that the scaling was successful. In this view, we could also check the logs for individual containers, in case an issue occurred. In fig. A.18, we can see the updated topology, showing the OT containers in blue. It provided us with information such as which nodes were running the new instances, how they were spread out and the resource consumption for this specific application.

Since we split up the application into different groups, we were able to scale them individually, giving us granular control over each component's instance count. Using the Nomad CLI, the "count" parameter of a group was adjusted temporarily. If we had redeployed the application, the scaling would not have persisted. We would have needed to adjust the relevant configuration file for this adjustment to be permanent.

Like Swarm, Nomad does not provide any auto-scaling capabilities by default. However, HashiCorp has developed and released an additional application called Nomad Autoscaler to solve this challenge [105]. It can support Nomad by ensuring an appropriate amount of resources is dedicated to a specific application, and can also provide dynamic sizing based on usage data. However, this last feature is only available in the Enterprise version of Nomad Autoscaler. The additional auto-scaling functionality was not evaluated during the implementation tests, since the focus of this research was on other aspects of Nomad.

6.5.6. Service Discovery

As mentioned in section 6.5.3, Nomad leverages Consul for the discovery of services. Both services can be integrated in order to achieve a cluster where running services can be discovered and addressed by other components. Consul requires a separate configuration and setup, and we opted for a similar architecture with three server nodes and five client nodes. The specific server and client configuration can be found in

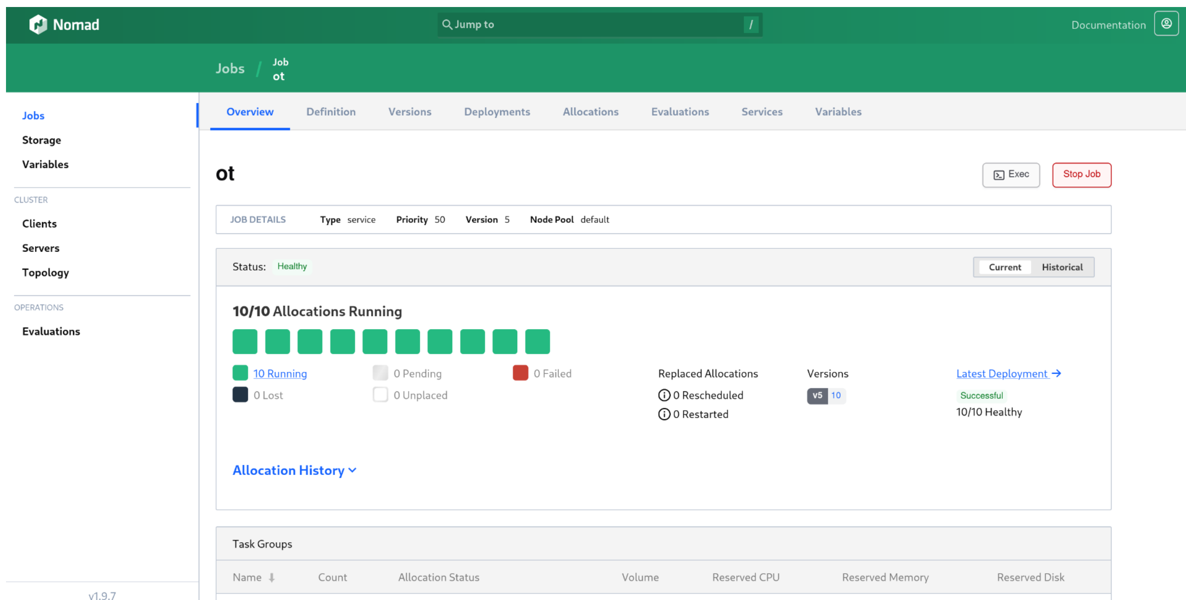

Figure 6.4.: An overview of OT job in Nomad after scaling

this project's accompanying GitHub repository[28]. Figure A.19 shows the eight nodes, with for each node the amount of running services, its internal IP address and the Consul version.

Consul builds up a catalog of registered services that can be queried and updated via Nomad. With this catalog, Nomad can locate on which nodes certain instances of an application component are running. It also provides an overview of the service health checks, which can help to debug a potential issue, with filtering options using tags. Figure 6.5 show the entire catalog, existing out of sixteen services, which are mostly registered via Nomad, as that was the orchestrator in this scenario. As an example, we scaled up the "query-api" to three instances and were able to see the allocated services, their underlying nodes and the dynamically assigned ports in fig. 6.6. Other services were now able to discover this service using the internal Consul DNS, in order to find out on which node and port the service was accessible.

Consul also provides internal DNS addressing, allowing applications to resolve other application names. This allowed us to connect the relevant components (i.e. databases, APIs and websites) to each other, without hard-coding a node IP address and port. However, Consul DNS did not work out of the box. It was required to set up a DNS forwarder in order for the ".service.consul" domain to be resolvable. We opted for dnsmasq[29], as it can function as a lightweight and easy-to-use DNS server. With the configuration shown in listing 14, we assured that DNS queries from applications running in the cluster could resolve the requested

[28] `https://github.com/cadeke/argo/tree/main/nomad/config`
[29] `https://dnsmasq.org/doc.html`

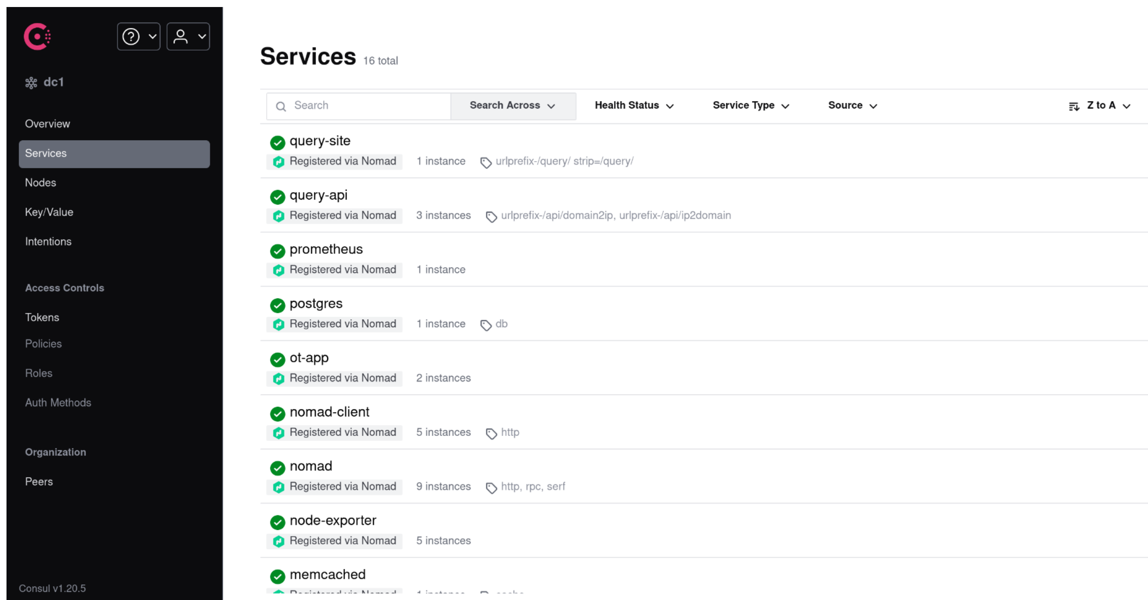

Figure 6.5.: An overview of service catalog in Consul UI

domains, while having fallback options in the form of Google DNS and Cloudflare DNS. In order to follow the desired architecture, we configured the three server nodes as internal DNS servers running dnsmasq. In the Nomad jobs, we could then specify the DNS servers to use, as well as the service names to query, effectively providing internal addressing for the different application components.

```
1  student@vm01:~> cat /etc/dnsmasq.d/consul.conf
2  # Forward .consul queries to local Consul agent
3  server=/consul/127.0.0.1#8600
4
5  # Fallback DNS resolver
6  server=1.1.1.1
7  server=8.8.8.8
8
9  # Listen addresses
10 listen-address=127.0.0.1
11 listen-address=10.203.96.230
```

Listing 14: A snippet of the dnsmasq configuration for server nodes acting as internal DNS servers

As a final part of the service discovery, we integrated a reverse proxy called Fabio[30] which consumes the service catalog from Consul and can provide simple load balancing capabilities. Consul can provide these functionalities as well, through integration with its service mesh known as Consul Connect. This concept is illustrated in more detail in section 6.5.8, however, we found Fabio to be the most straight-forward solution

[30] https://fabiolb.net

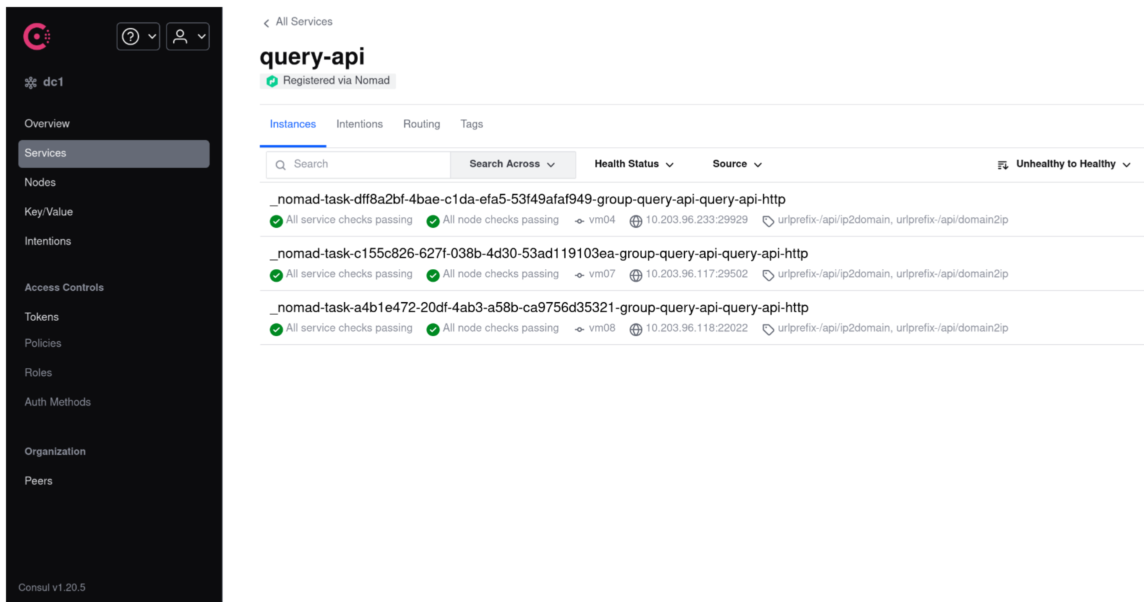

Figure 6.6.: An overview of query API service showing three instances in Consul UI

for the scenario. In section 6.5.9 about load balancing, we dive deeper into Fabio and its configuration. As a final validation, listing 38 includes different pings to other services from a "debug" container using the Consul service names, indicating the internal DNS resolution.

6.5.7. Fault Tolerance

To simulate the failure of a client node, we deactivated "vm08" and observed Nomad take the necessary steps to reschedule the running containers. Listing 15 shows the detection of a heartbeat failure on one of the server nodes, indicating that node "vm08" was down. When verifying in the Nomad UI shown in fig. A.20, we could see the services recovering, which means that they were actively being rescheduled on other available nodes. After a few seconds—in the topology view in fig. A.21—we could see that "vm08" was marked as "down", no running services were present and that the other client nodes were running the impacted services.

```
1  student@vm01:~> sudo journalctl -u nomad -f
2  Apr 14 17:16:22 vm01 nomad[50614]: ==> Newer Nomad version available: 1.10.0 (currently running: 1.9.7)
3  # output omitted
4  Apr 14 18:54:29 vm01 nomad[50614]:     2025-04-14T18:54:29.808Z [WARN]  nomad.heartbeat: node TTL expired:
   ↪  node_id=fd7ca40d-aa92-d77a-7494-e5d8eef21506
```

Listing 15: A snippet of Nomad logs detecting client node failure

In case of a server node failure, depending on whether the impacted node was the current leader, a new leader election would be started between the server nodes. To validate this, we shut the leader node down and observed the logs Nomad produced on another server node. Figure A.22 shows the server is marked a having left the cluster, while fig. A.23 shows the entire output of the logs. With the current division we could afford to lose up to one server node, without the cluster being impacted. However, additional server node failures would cause the entire cluster to experience issues.

6.5.8. Networking Segmentation

As far as we are aware, Nomad does not offer any built-in solution for the segmentation of traffic. Instead, it can leverage other providers that facilitate these mechanisms, with the Consul service mesh being the preferred option [106]. Traffic flows in this mesh can be controlled through so-called "intentions" [107]. Prior to the configuration of these intentions, the Consul service mesh needed to be established.

Enabling and connecting the service mesh required us to configure side-car services for the applications that needed to be reachable through the service mesh. For the sake of simplicity, we only included a selected amount of services in the service mesh, so the concept could be validated. We configured the "query-api", the "ot-app", a "debug" service and both of the databases to communicate over the service mesh. A snippet of this configuration can be seen in listing 16, where the side-car was defined and the upstream services, in this case both the databases, were connected. This side-car—powered by Envoy[31]—would be instantiated next to the "query-api" instance, and it would handle the service-to-service communication via mTLS and abstract this to the actual running service. This can be seen in listing 17, where every registered service has an additional side-car proxy associated with it.

Figure 6.7 displays this in a visual manner, where we used the Consul UI to filter for the services in the service mesh. In a different view of the Consul UI, visible in fig. 6.8, we could observe the connections between the different services, as a result of the upstream configurations mentioned earlier in this paragraph.

With the service mesh connecting the different application components, we were able to leverage its benefits in terms of segmentation, load balancing and encryption. The latter two topics are covered in their respective subsections, sections 6.5.9 and 6.5.11, while segmentation is discussed in the next paragraph.

[31] `https://www.envoyproxy.io/`

```
student@vm01:~/jobs/connect> cat apis.hcl
# output omitted

connect {
    sidecar_service {
        proxy {
            upstreams {
                destination_name = "postgres"
                    local_bind_port = 5432
            }
            upstreams {
                destination_name = "memcached"
                local_bind_port = 11211
            }
        }
    }
}
```

Listing 16: A snippet of a Nomad job showing Consul service mesh configuration for query-api service

```
student@vm01:~> consul catalog services -tags
consul
debug                        debug
debug-sidecar-proxy          debug
memcached                    cache
memcached-sidecar-proxy      cache
nomad                        http,rpc,serf
nomad-client                 http
ot-app
ot-app-sidecar-proxy
postgres                     db
postgres-sidecar-proxy       db
query-api                    api
query-api-sidecar-proxy      api
```

Listing 17: A snippet of a Consul command showing service catalog with an additional sidecar proxy

In order to create the appropriate access flows between the services in the mesh, we used the Consul CLI to create the intentions visible in listing 18. These intentions allowed the consuming clients to access the API and allowed the API to access the databases. All other flows were explicitly denied, in order to follow the principle of least privilege[32]. We verified these access restrictions by attempting connections to different services from the "debug" service, which is only allowed to access the API. Listing 19 shows that the packet filtering only applies to L4 and L7 protocols, such as HTTP for the API and TCP for the Postgres database. Ping requests to the Consul service names still succeed, because they bypass the service mesh layer. The access controls are thus enforced by the side-car proxy. Note that when accessing the "query-api" and "postgres" instances, we were using a local address where the proxy was listening, and not the Consul DNS

[32] https://csrc.nist.gov/glossary/term/least_privilege

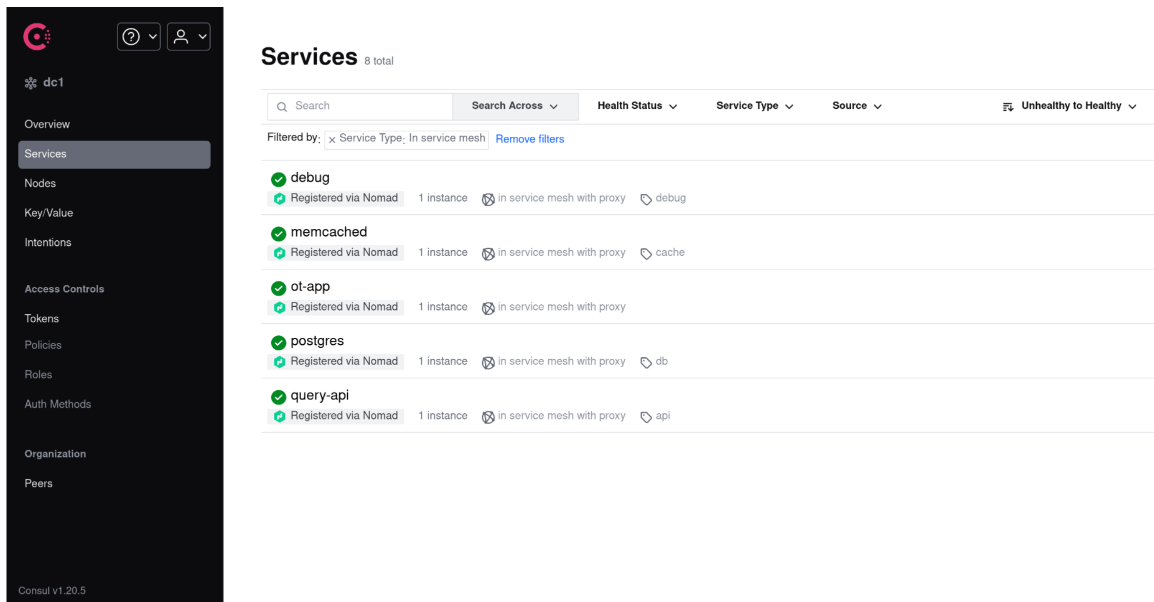

Figure 6.7.: An overview of the Consul catalog of services in service mesh with proxy

service name.

As an extra addition, we verified the intentions in the Consul UI, which gave us more insight into how they could be configured. Figure A.37 shows this interface, providing a visual overview on the left, as well as the configuration menu of a specific intention on the right. There, we were able to define the access rules—from “allow” or “deny” to HTTP specific rules where headers—methods or paths could be matched, which refers back to the L7 awareness of Consul intentions.

Finally, we want to mention Nomad’s CNI support, which is an essential part of Consul Connect. The service mesh leverages a networking mode known as “bridge” networking [108]. This is comparable to the Swarm overlay networks covered earlier. Nomad relies on CNI network plugins to provide this capability to its underlying nodes [109]. Nomad runs allocations in an isolated namespace, which are connected to the bridge network and secured by iptables rules injected by Nomad. For this scenario, the bridge networking combined with the intention provided sufficient segmentation capabilities. However, in more advanced cases, other CNI plugins could also be used to further customize Nomad’s networking configuration.

6.5.9. Load Balancing

As mentioned in section 6.5.6, we leveraged Fabio in conjunction with Consul to achieve load balancing between the different services in the cluster. Naturally, other solutions exist that can fulfil this requirement,

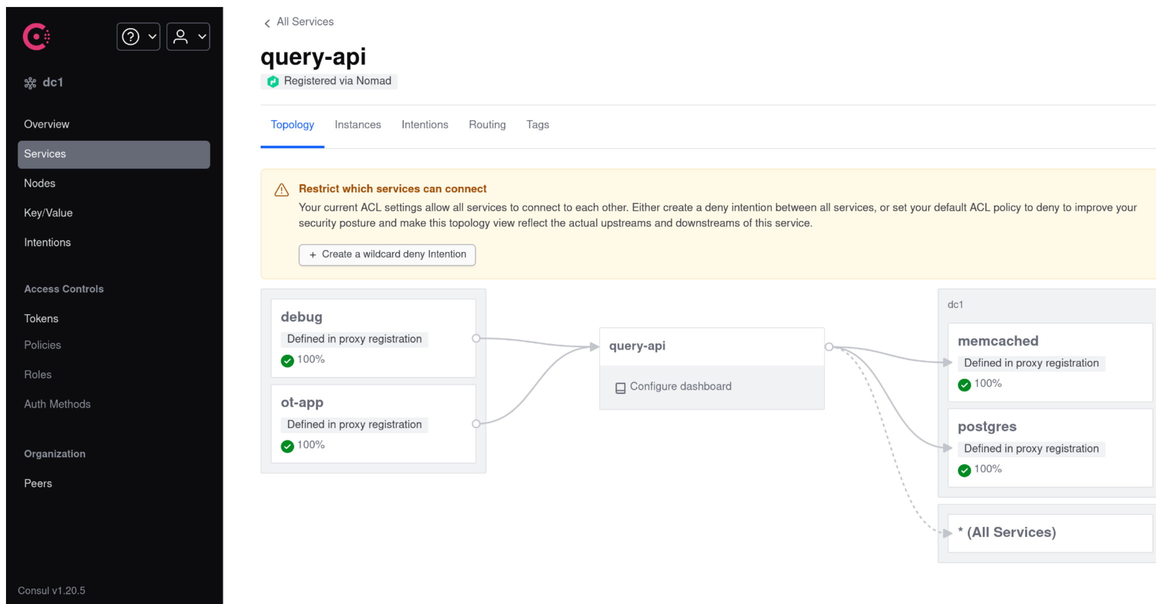

Figure 6.8.: A topology view in Consul of query API, showing upstream and downstream services

```
1  student@vm01:~> consul intention list
2  ID                                    Source     Action  Destination  Precedence
3  e0f17361-e11c-bc62-a22a-e297a2a01743  debug      allow   query-api    9
4  2b02120a-154c-3e08-39c4-e4af9d746580  ot-app     allow   query-api    9
5  a8332444-374e-5e5c-d42b-3b565be34537  query-api  allow   memcached    9
6                                        query-api  allow   postgres     9
7  c42befd0-8b3f-6547-ab85-3a209143922d  *          deny    *            5
```

Listing 18: A snippet of a Consul command showing intentions in service mesh

as long as they can integrate with Consul's service catalog. As a simple demonstration of this, listing 20 shows the DNS request to resolve the "query-api.service.consul" domain, returning the three node IP addresses and accompanying ports on which the application is mapped. In theory, we could also implement client-based load balancing where the consuming application implements this mechanism and then chooses an instance to query. However, this was not desirable in this scenario, since we did not want to make those adjustments to the consuming applications.

Fabio leverages Consul service tags to identify which services should be exposed on which endpoints. These tags were defined on the relevant Nomad tasks, such as the websites and APIs, which were then picked up by Fabio to expose the relevant routes. In this scenario, we exposed the "query-site" and the "admin-site", as well as both the "query-api" and "admin-api". In fig. A.25, we can observe the different routes that Fabio is aware of, as well as the traffic splitting for the three instances of the "query-api". Fabio was deployed

```
student@vm08:~> docker exec -it debug-c021b609-fd3d-8d3e-4de8-d3fe00b4f90e sh
~ # ping query-api.service.consul
PING query-api.service.consul (10.203.96.233) 56(84) bytes of data.
64 bytes from vm04 (10.203.96.233): icmp_seq=1 ttl=63 time=0.191 ms
64 bytes from vm04 (10.203.96.233): icmp_seq=2 ttl=63 time=0.382 ms
^C
--- query-api.service.consul ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1057ms
rtt min/avg/max/mdev = 0.191/0.286/0.382/0.095 ms
~ # ping postgres.service.consul
PING postgres.service.consul (10.203.96.233) 56(84) bytes of data.
64 bytes from vm04 (10.203.96.233): icmp_seq=1 ttl=63 time=0.265 ms
64 bytes from vm04 (10.203.96.233): icmp_seq=2 ttl=63 time=0.496 ms
^C
--- postgres.service.consul ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1011ms
rtt min/avg/max/mdev = 0.265/0.380/0.496/0.115 ms
~ # curl http://127.0.0.1:8080/api/domain2ip?query=photos.google.com
{"ts":"2025-04-19T15:53:27Z","query":"photos.google.com","result":"172.217.9.206","serverIp":"8.8.8.8","serverName":"Google
↪ DNS","id":60}
~ # psql -h 127.0.0.1 -p 5432 -U admin argodb
psql: error: connection to server at "127.0.0.1", port 5432 failed: server closed the connection unexpectedly
        This probably means the server terminated abnormally
        before or while processing the request.
```

Listing 19: A snippet of various commands, verifying intentions in Consul service mesh

as a system job in Nomad, which means it would be scheduled to run on each client node, maximizing the availability, since the load balancer served as the entry point to the cluster's applications. As a result of this, we achieved both internal and external load balancing with the same solution. This can be observed in fig. A.24, where we could access both the "query-site" and "admin-site" via the load balancer, on routes "/query" and "/admin" respectively. Note that the adjustment of the hosts file shown in listing 36 was necessary in order to access the services on "argo.lab". Any client node IP address would also have worked in this case, since the load balancer is running on each of those nodes, as it is configured as a system job. This can be observed in the "Type" column in fig. 6.3.

Since we set up the Consul service mesh in section 6.5.8, we wanted to mention that exposing services and load balancing can also be provided by the service mesh [110]. Consul can enable external connections to the service mesh through an API gateway, where typical mechanisms such as access control, load balancing and TLS-termination can take place. However, this was not investigated practically since the requirement for this test was already met using Fabio.

```
student@vm01:~> dig @127.0.0.1 -p 8600 query-api.service.consul SRV

; «» DiG 9.18.30-0ubuntu0.24.04.2-Ubuntu «» @127.0.0.1 -p 8600 query-api.service.consul SRV
(... output omitted ...)
;; ANSWER SECTION:
query-api.service.consul. 0        IN       SRV       1 1 29929 0acb60e9.addr.dc1.consul.
query-api.service.consul. 0        IN       SRV       1 1 29502 0acb6075.addr.dc1.consul.
query-api.service.consul. 0        IN       SRV       1 1 31679 0acb6073.addr.dc1.consul.

;; ADDITIONAL SECTION:
0acb60e9.addr.dc1.consul. 0        IN       A        10.203.96.233
vm04.node.dc1.consul.        0        IN       TXT       "consul-version=1.20.5"
vm04.node.dc1.consul.        0        IN       TXT       "consul-network-segment="
0acb6075.addr.dc1.consul. 0        IN       A        10.203.96.117
vm07.node.dc1.consul.        0        IN       TXT       "consul-network-segment="
vm07.node.dc1.consul.        0        IN       TXT       "consul-version=1.20.5"
0acb6073.addr.dc1.consul. 0        IN       A        10.203.96.115
vm05.node.dc1.consul.        0        IN       TXT       "consul-network-segment="
vm05.node.dc1.consul.        0        IN       TXT       "consul-version=1.20.5"
```

Listing 20: A snippet of a dig command, querying Consul DNS for query-api instances

6.5.10. Continuous Deployments

Nomad support continuous deployments through different update strategies, such as rolling updates, blue/green deployments and canary deployments [111]. We tested the different strategies similarly as was done for Swarm in section 6.4.9, with new versions[33] of the OT application. The configuration shown in listing 21 was applied on group-level in the Nomad job related to the OT application deployment.

```
update {
    max_parallel = 1
    canary = 0
    min_healthy_time = "10s"
    healthy_deadline = "3m"
    progress_deadline = "10m"
    auto_revert = true
    stagger = "30s"
}
```

Listing 21: A snippet of a Nomad job, showing CD configuration for OT application

Facilitating rolling updates is a first class feature of Nomad, supporting automated rollbacks in case of a failed update. To test this, we prepared four instances running a stable version of the OT application. This can be seen in listing 22. When upgrading to version 1.1, we adjusted the Nomad job and compared the planned changes, shown in listing 23. By planning the changes, we could verify which updates would be

[33] https://hub.docker.com/repository/docker/cadeke/argo-ot-app/tags

scheduled in the cluster. When deploying the job, we could add the job index to make sure that only these changes were deployed, in case the Nomad job was changed again at a later stage. This assures only the expected changes were actually deployed.

```
student@vm01:~/jobs> nomad status ot
# output omitted

Latest Deployment
ID          = b5a9baaf
Status      = successful
Description = Deployment completed successfully

Deployed
Task Group  Auto Revert  Desired  Placed  Healthy  Unhealthy  Progress Deadline
ot-app      true         4        4       4        0          2025-04-15T17:47:22Z

Allocations
ID        Node ID   Task Group  Version  Desired  Status   Created    Modified
04bcd9a2  d56fe237  ot-app      0        run      running  3m47s ago  3m31s ago
278953c6  d216928d  ot-app      0        run      running  3m47s ago  3m30s ago
6637ed92  6f594192  ot-app      0        run      running  3m47s ago  3m22s ago
e9714ff9  3919c96c  ot-app      0        run      running  3m47s ago  3m31s ago
```

Listing 22: A snippet of a Nomad command, showing stable version of OT application

```
student@vm01:~/jobs> nomad job plan argo-ot.hcl
+/- Job: "ot"
+/- Task Group: "ot-app" (1 create/destroy update, 3 ignore)
  +/- Task: "ot-app" (forces create/destroy update)
    +/- Config {
      +/- image: "cadeke/argo-ot-app:v1.0" => "cadeke/argo-ot-app:v1.1"
        }

Scheduler dry-run:
- All tasks successfully allocated.

Job Modify Index: 20564
To submit the job with version verification run:

nomad job run -check-index 20564 argo-ot.hcl

# output omitted
```

Listing 23: A snippet of a Nomad command, planning application update

When applying these changes, we could see the deployment being scheduled across the Nomad cluster, shown in listing 24. The old versions were deactivated one by one, as per our configuration, and the new instances were started and deemed healthy.

```
student@vm01:~/jobs> nomad status ot
# output omitted

Latest Deployment
ID          = c2fe1c53
Status      = successful
Description = Deployment completed successfully

Deployed
Task Group  Auto Revert  Desired  Placed  Healthy  Unhealthy  Progress Deadline
ot-app      true         4        4       4        0          2025-04-15T18:12:01Z

Allocations
ID        Node ID   Task Group  Version  Desired  Status    Created    Modified
0f5fb3ee  6f594192  ot-app      1        run      running   2m11s ago  1m56s ago
6fdb6813  d216928d  ot-app      1        run      running   2m28s ago  2m12s ago
5570832c  3919c96c  ot-app      1        run      running   2m44s ago  2m29s ago
8757c3c5  d56fe237  ot-app      1        run      running   3m2s ago   2m46s ago
04bcd9a2  d56fe237  ot-app      0        stop     complete  27m1s ago  3m2s ago
278953c6  d216928d  ot-app      0        stop     complete  27m1s ago  2m27s ago
6637ed92  6f594192  ot-app      0        stop     complete  27m1s ago  2m11s ago
e9714ff9  3919c96c  ot-app      0        stop     complete  27m1s ago  2m44s ago
```

Listing 24: A snippet of a Nomad command, showing a successful deployment of application update

When repeating the same steps for the upgrade to version 1.2, we could see that Nomad tried to deploy the new version, but was never able to start a new instance because it failed to start up and pass the initial health check. During this time, the application was still available because of the other instances that were kept at the previous stable version. After some time, the stable version was redeployed since the update failed. Listing 25 shows the different versions as a result of this failed deployment. Version 2 indicates the failed deployment attempt for each instance, while version 3 signifies the rollback to the previous stable version.

Nomad also supports canary deployments, allowing for side-by-side deployments so manual validation can be done. If a canary is deemed to be stable through manual evaluation, that version can be promoted as the main deployment. This strategy also enables blue/green deployments using a full or partial mirroring of the stable deployment. To validate this, we configured two canary instances to deploy version 1.0 of the OT application. These instances were started next to the running four instances, enabling optimal availability. Listing 41 shows the result of this deployment, indicating six running instances, with a status message that manual promotion is required. Depending on the manual testing (e.g. dedicated E2E tests, A/B testing or similar test procedures), the deployment could be promoted or failed. In this case, we chose to promote the deployment and thus approve the allocation of two extra instances running version 1.0 of the OT application, consolidating the running instances to the same application version. This is shown in listing 42. A similar mechanism could be used for blue/green deployments as well.

```
student@vm01:~/jobs> nomad status ot
# output omitted

Latest Deployment
ID          = 15534c3f
Status      = successful
Description = Deployment completed successfully

Deployed
Task Group  Auto Revert  Desired  Placed  Healthy  Unhealthy  Progress Deadline
ot-app      true         4        4       4        0          2025-04-15T18:38:44Z

Allocations
ID        Node ID   Task Group  Version  Desired  Status    Created     Modified
63aa90f6  d56fe237  ot-app      3        run      running   6m47s ago   6m36s ago
f4505f4d  d56fe237  ot-app      2        stop     failed    11m2s ago   6m47s ago
2695c2f6  fd7ca40d  ot-app      2        stop     failed    13m46s ago  11m2s ago
35675af4  6f594192  ot-app      2        stop     failed    15m38s ago  13m46s ago
5ea7f5be  3919c96c  ot-app      2        stop     failed    16m47s ago  15m38s ago
f4b35a2e  fd7ca40d  ot-app      3        run      running   18m9s ago   6m37s ago
4885fb67  d216928d  ot-app      3        run      running   18m26s ago  6m36s ago
1736e9f6  d56fe237  ot-app      3        run      running   18m42s ago  6m36s ago
d4ab2533  3919c96c  ot-app      1        stop     complete  18m55s ago  16m47s ago
c5b981ff  3919c96c  ot-app      0        stop     complete  19m31s ago  18m55s ago
e5545db3  d56fe237  ot-app      0        stop     complete  19m31s ago  18m42s ago
b4e1bc2d  fd7ca40d  ot-app      0        stop     complete  19m31s ago  18m9s ago
87308e66  d216928d  ot-app      0        stop     complete  19m31s ago  18m25s ago
```

Listing 25: A snippet of a Nomad command, showing a failed deployment of application update

Continuous deployments in Nomad are also supported in the Nomad UI, where the actions undertaken using the CLI in previous paragraphs were also achievable with a graphical interface. Figure A.26 and fig. A.27 show the version history of the deployments and the canary approval for the OT application respectively, enabling full traceability and control for system administrators who prefer a GUI.

6.5.11. Encrypted Communication

In order to establish secure communication, encryption can be configured on multiple different layers within the current cluster. Using TLS, control plane information related to Nomad, such as CLI commands, server-to-server traffic and other management-related communication, can be secured [112], [113]. Traffic between server nodes related to cluster information, so-called "gossip" traffic, can be encrypted using symmetric encryption through a pre-shared key (PSK) [114]. Similarly, Consul can also be configured with TLS to secure service discovery and agent communication traffic [115]. These two layers ensure that the management layer is fully secured. On the application level, Consul can provide automatic mTLS between applications using the service mesh, in which proxies handle the encryption layer on behalf of the services [116] and has been introduced in section 6.5.8. This enables service-to-service encryption in order to protect the application

data plane, while also enhancing security through increased observability and access control.

Prior to setting up mTLS on Nomad, we captured the traffic on one of the server nodes, "vm01", to make a comparison later on. Figures A.29 to A.31 show the unencrypted gossip, HTTP and RPC traffic respectively. In these WireShark captures, we can spot packets containing plain-text information about the Nomad cluster, such as job names, node ID's, etc. Next, we followed the steps outlined in the documentation to generate a 32-byte key to enable AES-256 and configured the Nomad server nodes to use this key for gossip encryption. This process can be seen in listing 26.

```
1  student@vm01:~> openssl rand -base64 32
2  jAJBQFP8uM0+VnspMh4ge1ICqFyKOBncgqlFzA1BHGA=
3  student@vm01:~> sudo cat /etc/nomad.d/nomad.hcl
4  # output omitted
5
6  server {
7    enabled = true
8    bootstrap_expect = 3
9    encrypt = "jAJBQFP8uM0+VnspMh4ge1ICqFyKOBncgqlFzA1BHGA="
10 }
11
12 # output omitted
```

Listing 26: A snippet of various commands, setting up AES-256 for gossip encryption

To secure the HTTPS and RPC traffic, certificates for each node needed to be rolled out. Nomad provides CLI commands for the generation of these certificates, shown in listing 27, which were executed for each node in the cluster. Every server and client node's configuration file was then adjusted to include the CA file and the relevant agent certificate and accompanying private key, shown in listing 28. As a final step, all nodes needed to be restarted, which was done in a rolling manner to keep the cluster's state stable.

After the cluster restarted, we could no longer access the Nomad UI, see fig. A.28, or execute commands via the Nomad CLI, see listing 40, because client verification was enabled. For ease of use, we configured "verify_https_client" to "false" on "vm01" as an exception, so we were still able to manage the Nomad cluster easily. In a true production scenario, a dedicated client certificate can be generated and presented when accessing the UI or using the CLI.

In order to validate this change, we recaptured the traffic on a server node and inspected the packets with WireShark. The packets related to gossip, HTTP and RPC traffic are visible in figs. A.32 to A.34 respectively. We were no longer able to observe plain-text information about the cluster. Instead, TLSv1.2 packets

```
student@vm01:~/certs/tmp> nomad tls ca create
==> CA certificate saved to: nomad-agent-ca.pem
==> CA certificate key saved to: nomad-agent-ca-key.pem
student@vm01:~/certs/tmp> nomad tls cert create -server -region dc1 -additional-dnsname vm01
==> WARNING: Server Certificates grants authority to become a
    server and access all state in the cluster including root keys
    and all ACL tokens. Do not distribute them to production hosts
    that are not server nodes. Store them as securely as CA keys.
==> Using CA file nomad-agent-ca.pem and CA key nomad-agent-ca-key.pem
==> Server Certificate saved to dc1-server-nomad.pem
==> Server Certificate key saved to dc1-server-nomad-key.pem
student@vm01:~/certs/tmp> nomad tls cert create -client -region dc1 -additional-dnsname vm04
==> Using CA file nomad-agent-ca.pem and CA key nomad-agent-ca-key.pem
==> Client Certificate saved to dc1-client-nomad.pem
==> Client Certificate key saved to dc1-client-nomad-key.pem
```

Listing 27: A snippet of Nomad commands, generating CA, server and client certificates

```
student@vm01:~> sudo cat /etc/nomad.d/nomad.hcl
# output omitted

tls {
  http = true
  rpc  = true

  ca_file   = "/etc/nomad.d/nomad-agent-ca.pem"
  cert_file = "/etc/nomad.d/vm01-dc1-server-nomad.pem"
  key_file  = "/etc/nomad.d/vm01-dc1-server-nomad-key.pem"

  verify_server_hostname = true
  verify_https_client    = true
}
```

Listing 28: A snippet of TLS configuration on a Nomad server node

were present, indicating the management traffic flows were encrypted. The complete traffic captures can be found in the accompanying GitHub repository[34].

As mentioned earlier in this paragraph, when inspecting the traffic on a client node, we are still able to see service-to-service communication. As an example, fig. A.35 shows the WireShark capture of “vm05” which contains plain-text HTTP requests between instances of the “ot-app” and “query-api” applications. In order to fulfill the encryption requirement for this test, we leveraged automatic mTLS for this type of traffic through Consul Connect. However, Consul itself was not configured with mTLS, which follows similar steps as we did earlier for Nomad. Since the focus of this research was on Nomad, we concluded that Consul mTLS configuration was out of scope for this test, although recommended in a production scenario.

[34] https://github.com/cadeke/argo/tree/main/pcap/nomad

After configuring the service mesh, as mentioned in section 6.5.8, we were no longer able to connect to the database or API service directly from an external client or a Nomad node, since only authorized mTLS-enabled services from the mesh were allowed to connect. When inspecting the traffic on “vm08”, we could no longer see the plain-text HTTP requests between the “ot-app” and the “query-api” instances. Instead, TLSv1.2 packets encapsulated that data, which is shown in fig. A.36. In this capture, we can observe solely TLS packets between various nodes. These packets contained data traffic between services, or management traffic between nodes, although we have no way of verifying this since the content is encrypted, which was the main requirement in this test scenario.

Throughout these various confirmation steps, we leveraged Nomad’s and Consul’s built-in certificate generation tooling. Naturally, as was also the case with Swarm, Nomad and Consul both support the usage of an external root CA certificate [113], [117]. We felt it necessary to mention this, since providing an external CA certificate would be the optimal approach for a production scenario where there is already some form of PKI in place. Since the certificate information is stored in the cluster state and replicated to all server nodes, additional security measures need to be in place to protect the PKI [118], [119]. Both Nomad and Consul do not provide any at-rest encryption, so it is up to the cluster administrators to secure the underlying nodes through typical security measures such as restricted file system permissions, disk encryption, OS hardening and access controls.

6.5.12. Secret Management

In terms of secret management, Nomad has a tight integration with Vault[35], a popular open-source secret storage solution developed by HashiCorp. Vault can be leveraged for storing application secrets or certificates that need to be consumed by Nomad and transmitting them securely to consuming services. It can also provide access roles, further enhancing the security posture of the cluster. Consul also provides a key-value store, but this is mainly meant for configuration data, not secret data [120]. For this scenario, however, we wanted to investigate a simpler solution that is native to Nomad itself.

Nomad provides a directory which is private to each task’s runtime environment [121], [122]. This can be used to pass secrets to an application in a simple manner. We acknowledge that in a true production scenario,

[35] `https://developer.hashicorp.com/vault`

a dedicated secret storage solution such as HashiCorp Vault, Azure Key Vault[36] or AWS Secrets Manager[37] is recommended, but this scenario's requirements can be satisfied with a simpler solution.

Listing 29 shows the configuration, which is part of the Nomad job, in order to pass some secrets to an application. We could verify this mechanism by inspecting the container and accessing the secret, which was available as an environment variable in Bash, shown in listing 30.

```
1  template {
2    data = «EOH
3
4  API_KEY="1111-2222-3333-4444"
5  SOME_SECRET="some-value"
6  EOH
7
8    destination = "secrets/argo.env"
9    env = true
10 }
```

Listing 29: A snippet of a Nomad job definition including secret environment variables

```
1  student@vm05:~> docker exec -it ot-app-bfb3ccdb-3c0d-962c-1b3e-6de2cb95546d sh
2  /app # echo $API_KEY
3  1111-2222-3333-4444
4  /app # cd /secrets
5  /secrets # cat argo.env
6  API_KEY="1111-2222-3333-4444"
7  SOME_SECRET="some-value"
```

Listing 30: A snippet of various commands, verifying runtime secrets available in Nomad container

[36] https://azure.microsoft.com/en-us/products/key-vault/
[37] https://docs.aws.amazon.com/secretsmanager/latest/userguide/intro.html

7. Results and Discussion

This chapter covers the results from the analyses and implementations done in previous chapters, as well as a general discussion of the different topics that were covered in this research.

7.1. Kubernetes' Complexity

As discussed in section 3.1.3, the complexity associated with Kubernetes can be attributed to multiple factors. We attempt to group them into two main domains, based on the literature review and the findings gathered during the practical implementation in chapter 6. On the one hand, we found that the responsibilities of a CO framework such as Kubernetes are quite extensive. Its main goal is to abstract the underlying pool of resources to the applications that are running on it. This fact means that the CO framework needs to provide a solution for various topics, such as scheduling, discovery, networking, storage, scaling, access control, etc. Since the main responsibilities of a developer are more related to the application itself, this divide could present an issue. Furthermore, we found that the step between the containerization of an application and the orchestration of that containerized application is rather large, mainly because of these extra requirements that come with it.

On the other hand, we found that because of Kubernetes' popularity as a CO framework, other technologies have emerged that are in some way related to, based on, or inspired by Kubernetes. These tools are often optimized for a specific use-cases, e.g. low resource scenarios such as IOT or edge computing, or target audiences, e.g. consumers of a CSP. This fact makes the decision process for the unfamiliar even more daunting, since the term "Kubernetes" could refer to many different tools within the Kubernetes ecosystem. These technologies are meant to simplify the usage of Kubernetes by abstracting certain parts, however, knowledge of the core concepts is still essential in order to use these services optimally. As a practical example, Azure offers various platforms, such as Azure Kubernetes Service (AKS), Azure Red Hat OpenShift, Azure Container Apps (ACA), Azure Container Instances (ACI) and Azure App Services (AAS), which are all able to host and orchestrate containers in some way or form. This means that the necessary due-diligence

is required prior to implementing one of these services in a given application landscape. Moreover, in order to be able to conduct this research and make a technology decision, the concepts behind container orchestration need to be understood to a certain degree, which refers back to the first point mentioned in this section.

A major advantage of cloud services is the shared responsibility model of most CSPs, which abstracts certain challenges, allowing users to focus more on the development and operation of their applications. We deemed it necessary to mention that many of those managed services could provide an answer to the majority of the orchestration challenges present in an organization. Especially when said organization already maintains a partnership with a given CSP, the step towards a managed Kubernetes services or CaaS should be thoroughly evaluated, since it presents the most accessible gateway towards production-ready orchestrated applications at scale.

As a final point, we want to emphasize that Kubernetes is the most popular CO framework, which would invite a larger range of opinions, both positive and negative, compared to a lesser known CO framework. Additionally, some of Kubernetes' key features are its extensibility, configurability and support for community plugins, such as for CNI and CSI. Naturally, this comes at a cost of increased complexity in the technology landscape, requiring developers to make certain decisions before they can actually start using Kubernetes. In response to this, the aforementioned services built around Kubernetes came into existence, promising a simplified configuration, management, or installation by abstracting some of these decisions towards developers. This neatly ties together both points discussed in the above paragraphs, visualized in fig. 7.1, and thus forming an answer to RQ_1.

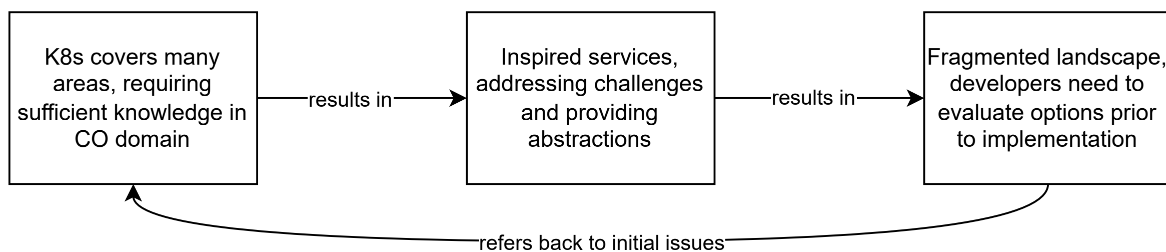

Figure 7.1.: A chart showing interconnected Kubernetes challenges

7.2. Orchestration Features

The selection of features that were crucial for the orchestration of this research's use-case, defined in section 6.3, was assembled through two processes. Firstly, we deduced a list of essential characteristics from the state-of-the-art CO frameworks in section 5.2.1. Features such as scheduling, scaling and failover capabilities were reoccurring topics covered in the related research. Since these works covered CO from a general perspective, the resulting features were quite broad in nature. Secondly, an additional list was compiled in section 5.2.2, containing more specific features that were crucial for this research to investigate. Topics such as shared storage, load balancing and continuous deployments were selected, since we were of the opinion that valuable insights could be gathered by investigate these features. As a final overview, table 5.1 contains all relevant features for the defined use-case, presenting an answer to RQ_2.

7.3. Orchestration Frameworks

Section 5.3.2 evaluated different CO frameworks that were introduced by analyzing the related research in section 3.3. Resulting from this selection were two CO candidates that each addressed common orchestration challenges, while presenting themselves as possible alternatives for Kubernetes. Swarm was mentioned in several of the related studies, while Nomad appeared to be a lesser known yet powerful orchestrator, leveraging strong integrations with other HashiCorp services. Additional candidates, that positioned themselves as "batteries-included" versions of Kubernetes, were certainly interesting possibilities as well and could have been considered for the practical implementation. However, given the limited time and resources that were available for this research, the scope had to remain focused to allow for a detailed analysis of the selected candidates. This presents an opportunity for possible future research in this domain, recreating a similar use-case where other CO candidates could be evaluated against. With this selection made, RQ_3 had been answered, and all necessary components were in place to start the practical implementations.

7.4. Suitability of Orchestration Candidates to Use-Case

In order to answer to RQ_4, observations from both CO candidates during each feature implementation need to be evaluated. The following sections compare the approach both Swarm and Nomad take in order to address each feature requirement. Afterward, we could formulate an overall verdict about the most suitable CO candidate, where the hypotheses and the implementation process in general are reviewed.

7.4.1. Feature Comparison

Setup

Setting up the cluster with Swarm was straight-forward, not requiring any extra installations or configuration since all Swarm functionality is included with the Docker Engine by default. This makes the initial process very approachable for developers already familiar with Docker, only requiring them to choose an appropriate split for their master and worker nodes. After following the instructions to join the nodes to the Swarm cluster, everything was in place to start running workloads.

Accomplishing the same result in Nomad demanded more effort. Firstly, Nomad needed to be installed on each node. Then, configuration files for both server and client nodes need to be written and validated. Additionally, Nomad was registered as a service in systemd for simplified management. Nomad's documentation was complete but concise in this regard, requiring us to consult additional sources to get a working cluster running. Since we opted for Consul, the same process needed to be completed once again. Overall, this took a lot more time and effort, which is expected since two separate solutions needed to be installed, configured and integrated with each other.

Both solutions provide an intuitive CLI for interacting with the cluster. As mentioned during the implementation, Nomad's configuration could be done through HCL or JSON, which meant some time had to be spent understanding the correct syntax and required fields. Overall, we could agree that Swarm's initial setup is more intuitive, mainly because it requires fewer steps to get started.

Shared Storage

Since shared storage was provided externally through a GlusterFS pool and reused for both implementation scenarios, both solutions were able to integrate with it as expected. We faced some permission issues when Nomad tried to access some parts of the shared storage, but this was a minor issue in the total application landscape. Swarm abstract the storage provided to it through so-called volume drivers, which can be extended using a plugin system. Nomad, on the other hand, supports integrations for CSI, which is a standard for developing plugins that integrate with CO frameworks, with the most notable supported being Kubernetes. Because of this fact, we feel that Nomad is more mature, allowing users to leverage the same plugins as typically done in Kubernetes deployments.

Scheduling

After establishing the cluster, both Nomad and Swarm were able to schedule the workloads on available nodes. A notable difference is the fact that Swarm also schedules containers on its master nodes, while Nomad keeps its server nodes free by default, only leveraging the client nodes for running allocations. Both solutions allow the usage of constraints to indicate to the orchestrator where a certain workload should be scheduled, e.g. on a specific node or on master/server nodes only.

Both candidates support node draining, which functioned as expected. Nomad possess a slight advantage, allowing for more options when scheduling a node for draining, such as an estimated time, progress indication and the option to force workloads if required. Combined with Nomad's UI, this feature felt more mature and enterprise-ready than Swarm's more straight-forward approach of marking a node to be drained. Ultimately, both Swarm and Nomad handled this requirement well.

Scaling

Horizontal scaling, another key feature of any orchestration framework, was handled well by both candidates. Both Swarm and Nomad allow for scaling the running services out or in by adding or removing instances of that specific container, either by executing the appropriate CLI command or by configuring it using the "replicas" or the "count" parameter. In Swarm, each defined service can be scaled individually. In Nomad, it depends on more how the deployed is defined, since scaling is applied on group-level. This required slightly more thought when defining the different application components, but overall both solutions handled this flawlessly.

Neither Swarm nor Nomad supports auto-scaling as a native feature. However, Nomad can integrate with a dedicated cluster auto-scaler called Nomad Autoscaler. Automatic scaling based on certain metrics or parameters could thus be establishing with both solutions, with custom scripts or logic leveraging the relevant CLI commands to make this happen. Since auto-scaling was not evaluated practically, we make the assumption that Nomad would handle this requirement better since it can leverage a dedicated service from HashiCorp, while Swarm need to rely on custom logic or community offerings.

Service Discovery

In terms of service discovery, Swarm abstracts this process to the point where users might not realize that are using it. Swarm automatically makes every defined service available through its internal DNS service, so

the defined name can be used to refer to in other services. This process required no additional components or configurations. Nomad relies on Consul as a dedicated service discovery solution, which introduces both advantages and distances. An extra service like Consul requires extra installation and configuration, as discussed in section 7.4.1, which add complexity to the overall application landscape. However, Consul provides an additional UI overview of the cluster, with a useful information in terms of available services.

As an extra note, we want to mention the DNS forwarder that was necessary to solve issues with address resolving within the deployed services. Without this additional component, the Consul DNS could not be reached by nodes, which was problematic for connecting services together. For example, the query API was not able to make a connection to the Postgres database, because the given address “postgres.service.consul” could not be resolved. Luckily, this was solved by adding dnsmasq to the server nodes and configuring those servers as the main DNS servers for every deployment. We acknowledge that this change might not have been necessary and could be the result of a misconfiguration on our part. However, for the sake of transparency, we felt it was important to mention this experience.

Fault Tolerance

Both orchestration scenarios were set up with the same split between master/server nodes and worker/client nodes. This meant that both clusters could afford to lose one master/server node at most. During the tests, both Swarm and Nomad handled the failure of a master/server node as expected. When a heartbeat to leader node fails, the remaining master/server nodes initiate the election process to keep the cluster stable. Similarly, the failure of a worker/client node triggered the rescheduling of the impacted workloads in order to keep the desired state, i.e. the defined services in the deployment.

Networking Segmentation

Swarm’s intuitive networking configuration allowed us to define different networks with the desired subnet ranges in order to segment the traffic in the application landscape. During the testing, we encountered a known Swarm issue with the default subnet ranges, which demanded some extra investigation and configuration, especially since the documentation was not entirely clear. Extra effort in terms of debugging and searching for other users encountered similar issues was reqiured. Once this was sorted, the segmentation worked as expected, blocking traffic between services not only through their DNS name, but also directly through their IP address if communication between them was not explicitly defined.

Nomad’s networking segmentation relied on Consul, more specifically on its service mesh feature. With

this service mesh, Consul can handle access control between services in the mesh with intentions. Again, this component needed an extra time investment in order to provide the necessary capabilities, which was a worthwhile effort since the service mesh also support Nomad's in other topics. The segmentation provided by the mesh proved to be more advanced, allowing for filtering on L7 with specific HTTP paths and headers, on top of regular L4 filtering based on source and destination addresses. However, this segmentation wasn't as strict as Swarm, since requests that bypassed the mesh were still allowed. Custom firewall rules are thus required if this behavior needs to be adapted to the use-case at hand.

Overall, Swarm's approach to segmentation appears to be more accessible and intuitive for users with general IT knowledge about subnetting concepts and was the most straight-forward option. In contrast, Nomad's segmentation accommodated by Consul provides more advanced capabilities, but requires more investment in order to leverage it to its fullest potential.

Load Balancing

Balancing requests, coming from clients both outside or inside the cluster, between healthy instances of a container functions in a straight-forward way. When deploying services in a basic configuration, Swarm automatically routes traffic between healthy containers inside the orchestration network. For requests coming from outside the cluster, Swarm provides a routing mesh. This enables specified ports to be published on all nodes in order for external clients to reach an internal service. Requests to these published ports will then be forwarded to the relevant node that is running a healthy instance of the requested service. With that, the minimum load balancing requirements for both internal and external traffic were stratified. During this implementation, we contemplated introducing Traefik as a dedicated reverse proxy to support more advanced features typically expected from a load balancer, such as TLS-termination. However, this was ultimately deemed to be too complex given the time that was allocated to this research.

Nomad once again leveraged Consul in order to balance requests between healthy instances. However, an additional service called Fabio was introduced in order to consume Consul's catalog of services and provide externally available routes. Fortunately, Fabio's configuration was kept to a minimum, since it only required us to define the routes that in needed to expose, using Consul tags. As an additional mention, we want to state that Consul's service mesh could also be leveraged to provide load balancing and the exposing of services through different types of gateways. This would certainly have been an interesting track to evaluate in more detail, since it is positioned by HashiCorp as one of Consul's main strengths. Though, due to time limitations, these features were not analyzed practically during this implementation test.

Again, the functionality provided by Swarm suffices for this specific scenario, but it would certainly benefit from a dedicated load balancing service. Nomad's offerings aim at solving more advanced use-cases, coming at the cost of increased complexity.

Continuous Deployments

Keeping the application components stable is a crucial feature of any orchestration solution. Both Swarm and Nomad handled this requirement as expected, with smooth updates and automated rollbacks when an issue occurred with a newly deployed version. Swarm allows for slightly more customized functionality during these deployments, relying on parameters controlling the update, rollback and restart behavior, demonstrated in listing 10. Nomad's configuration, visible in listing 21, seemed more focused on time in terms of deadlines by which a change should be deployed.

Nomad sets itself apart with its support for canary deployments. This gives users more flexibility when devising their deployment strategies, allowing updated versions to be deployed in parallel to the stable version and afterward—after manual evaluation of the new version—enabling the promotion of these canary deployments to replace the stable application versions. As a welcome additional feature, Nomad allows the scheduling of a dry run of an update, providing the user with a structured overview of which changes will be propagated in the cluster. This feature enables improved traceability and version history, which could be an essential part of existing change management procedures in an enterprise environment.

Encrypted Communication

Encryption of management communication, i.e. traffic containing scheduling instructions or cluster state updates, is enabled by default in Swarm. Securing the data plane in a similar way only required the addition of a parameter in the network definition. Swarm nodes are automatically configured with mTLS in order to authenticate themselves in the cluster. When validating this change with WireShark, we were able to observe IPsec traffic encapsulating the actual data traffic, effectively masking the requests between the different Swarm services.

Since Nomad's setup consisted out of multiple components, secure the different communication flows with encrypted proved more challenging. By default, the management traffic of both Nomad and Consul happens in plain-text. Enabling encryption for Nomad's server communication consisted out of multiple layers, leveraging symmetric encryption with a PSK and as well as asymmetric encryption with PKI. After these

changes, the gossip traffic between servers was successfully encapsulated with TLS. However, the communication between services in the cluster was still happening over plain HTTP, because this required a separate encryption configuration. We were able to satisfy this requirement through Consul's service mesh, providing automatic mTLS by deploying additional sidecar proxies with every service. Once again, this demanded more investigation into Consul's documentation to make this function as expected. Ultimately, we were able to also secure the data traffic between services in the Nomad cluster with TLS. Consul requires a separate PKI configuration, which was not implementation, since this exceeded the boundaries of this research's defined scope.

As mentioned in the final paragraph of section 6.5.11, the built-in certificate tooling was used for both Nomad and Swarm. However, we want to reiterate that both frameworks support external root CA certificates, allowing users to integrate these tools with existing PKI which is often already present at any given organization. Finally, Swarm offered an advantageous feature with the auto-lockig capability of the Swarm cluster. This feature secures the private key of the root CA certificate when a server nodes restarts, effectively protecting against the extraction of this sensitive information. Nomad does not offer this feature, instead recommending traditional security controls in terms of filesystem permissions, OS hardening and access control.

Secret Management

Both technologies offer an intuitive and simple way to integrate application secrets with service deployments. As this feature was considered to be optional during the implementation process, the practical tests in this domain were limited. Because Nomad is part of the HashiCorp ecosystem, it can benefit from strong integration with other HashiCorp services such as Vault, a dedicated secret management solution. We thus assume Nomad's functionality in the area of secret management could be extended more than Swarm's, although this was not verified practically.

7.4.2. Grading

The grading for both Swarm and Nomad is captured in table 7.1. It includes the rating for each feature per orchestrator, as well as a final score for each orchestrator, which was derived using the formula introduced in section 6.3.2. We can observe that Swarm's total suitability score is slightly higher than Nomad's, mostly due to the increased implementation effort due to Consul. Once again, we want to stress that this is only related to this research's defined use-case, and should not be considered a general fact. The next section

present the results from table 7.1 in a written comparison as the final verdict.

Feature	Verdict	x	y	α	z
Swarm	-	-	-	-	**143**
Scheduling	Worked as expected	5	5	2	-
Scaling	Worked as expected	5	5	2	-
Service discovery	Worked as expected	5	5	2	-
Fault tolerance	Worked as expected	5	5	2	-
Networking segmentation	Significant effort required, confusing documentation	2	3	2	-
Load balancing	Worked as expected, limited to simple exposure	4	4	2	-
Continuous deployments	Worked as expected	5	5	2	-
Encrypted communication	Worked as expected, documentation not entirely clear	3	4	1	-
Secret management	Worked as expected	5	5	1	-
Nomad	-	-	-	-	**124**
Scheduling	Worked as expected	5	5	2	-
Scaling	Worked as expected	5	5	2	-
Service discovery	Significant effort required, Consul integration	3	2	2	-
Fault tolerance	Worked as expected	5	5	2	-
Networking segmentation	Significant effort required, Consul intentions	4	2	2	-
Load balancing	Significant effort required, Consul and Fabio integration	2	1	2	-
Continuous deployments	Worked as expected	5	5	2	-
Encrypted communication	Significant effort required, Nomad and Consul PKI setup	4	2	1	-
Secret management	Worked as expected	5	5	1	-

Table 7.1.: An overview of the feature scores for Swarm and Nomad

7.4.3. Verdict

After carefully evaluating the performance of both candidates across all aforementioned domains, we can determine that Swarm aligns more closely with the requirements of this research's defined use-case.

Swarm's strongest advantage over Nomad is its simple approach to extend the existing Docker functionality with orchestration capabilities. It presents a straight-forward answer to virtually all the defined requirements, from scaling and scheduling to network encryption and segmentation. The fact that Docker is also the most

popular container engine, as established in section 2.3.2, supports Swarm in its adoption by newcomers to the domain of CO. Because the defined use-case heavily focused on simplicity and understandability for developers who often do not possess an in-depth knowledge, Swarm manages to bridge the gap between containerization and the orchestration of containerized applications appropriately.

Despite the facts mentioned in the paragraph above, the insights gained during the implementation of Nomad and its supporting services might be more valuable to some IT profiles. In the majority of the features discussed in previous sections, Nomad required additional configuration. This also comes with a benefit that Nomad is highly customizable for more advanced use-cases, where the administrators often posses some form of previous understanding on common CO topics. Nomad's functionalities also feel more focused on larger deployments, in clusters with several dozen of nodes and hundreds of containers. We are of the opinion that for a specialized scenario that differs slightly from the one defined in this research, Nomad presents a more optimal fit than Swarm. Nomad's integration with other services, such as Consul for its discovery and mesh capabilities, and Vault for secret management, is another key benefit over Swarm. As an added bonus on top of this, the fact that both Nomad and Consul come with a graphical interface, which might be a convincing factor for those who prefer it over CLI-focused tool. Nomad integrates built-in metric visualizations with its UI, giving administrators information about the resource usage of their running nodes and services. Overall, we can state that Nomad is a powerful orchestration that could possibly present a true alternative to Kubernetes for medium to large application deployments.

With this general verdict covering the most essential part of this project, we present an answer to RQ_4, considered to be the main research question of this thesis.

7.4.4. Review of Hypotheses

When reviewing the claims made in section 1.4 with the insights gained during the efforts made in previous chapters, we can confirm that Docker indeed offers a solution to common CO requirements, which was discussed in detail when covering Swarm as an orchestration candidate in section 6.4. Similar, our prediction about Kubernetes' complexity was also partially correct, describing a subdomain of the main issues that arise when leveraging Kubernetes, as covered section 7.1. The assumption regarding Red Hat OpenShift turned out to be inaccurate when taking into account this use-case specific requirements, since other CO technologies, such as Swarm and Nomad, presented a better suitability. Overall, the hypotheses made prior to executing the research were accurate to an acceptable degree.

7.4.5. Review of Implementation Process

As a final section of this discussion, we want to evaluate the implementation process in its entirety. Overall, we are satisfied with the implemented component and the resulting insights. Both Swarm and Nomad were explored in-depth, with the goal of utilizing their capabilities to their full potential. Every component was analyzed in detail and validated through practical testing using the use-case at hand. For Swarm, we would have preferred to integrate Traefik into the deployed application stack, as discussed in section 7.4.1, primarily due to its popularity in cloud-native environments similar to one established in this research. With more dedicated resources in terms of time and personnel, this addition could have been investigated further. In the case of Nomad, we spent comparatively more time in order to the get defined use-case up and running, which is largely due to the complexity that came with the integration with Consul and its service mesh. Nevertheless, we believe that this service mesh could present a solution for many other CO use-cases, and we would have preferred to investigate this particular feature in greater detail.

With the findings discovered throughout this implementation process, we believe that this research contributes to the state of the art in the domain of CO. We have positioned both Swarm and Nomad as potential solutions for specific CO challenges, establishing them as viable alternatives to Kubernetes—although their suitability remains highly dependent on the use-case at hand.

8. Conclusion

This thesis covered various aspects of container orchestration through several layers of research, attempting to address the problem statement established in chapter 1. In chapter 2, we established essential components, such as containerization and orchestration, necessary to build upon in the ensuing chapters. During the review of the state of the art in chapter 3, we considered previous efforts in the domain of orchestration, grouping and discussing the findings in three main categories. Chapter 5 addressed a crucial component, with the definition of the use-case and the selection of the relevant orchestration features and candidates, which were subsequently subjected to a practical investigation in chapter 6. These insights were then bundled and discussed in chapter 7. In doing so, we provided an answer to the research questions established in section 1.3, thus effectively concluding this research project. The remaining sections in this chapter cover the most significant findings from each of the major parts of this thesis.

8.1. Insights with respect to Kubernetes

Kubernetes' status as the primary container orchestrator in today's landscape causes it to be an opinionated topic. It is, to the best of our knowledge, a revolutionary technology that has solved various computing challenges over the years and will continue to do so. Despite this, Kubernetes should not be considered to be a silver bullet for all container orchestration problems, such as the one presented in this research. As discussed in section 7.1, two interconnected main issues were identified that could be part of root cause.

On one side, we acknowledged that orchestration as a domain is often grouped together with containerization, while it encompasses much broader topics such as scheduling, networking, storage, which are often not fully understood by developers since it is not part of their primary responsibilities. This may result in unrealistic expectations or unexpected technical challenges. On the other side, the fragmented landscape of Kubernetes-based technologies came into existence in order to provide solutions to common challenges or provide a specialized experience. Combined with Kubernetes' extensive plugin support, it increases the total friction during initial adoption.

We deem it necessary to explicitly mention that the previously mentioned insights regarding Kubernetes originate from a review of the state of the art done as at the initial phases of this thesis. Kubernetes was not evaluated practically in the same way as the other candidates mentioned, thus limiting our ability to compare the two uniformly.

8.2. Insights with respect to Swarm

Docker Swarm offers powerful and intuitive orchestration capabilities which come packed along with the Docker Engine by default. Because of Docker's widespread popularity as a containerization platform, Swarm serves as an accessible orchestrator for developers looking to benefit from containerized applications, without the need for a significant amount of extra knowledge investment. Production-grade features in terms of networking, fault tolerance and discovery, function out of the box with minimal configuration required. Setting up dedicated subnetworks to isolate application communication channels and masking the traffic through encryption works remarkably well without extra effort. The built-in service discovery and load balancing further enhance its user-friendliness, as does its routing mesh for exposing services externally, although a dedicated reverse proxy solution could be beneficial in certain scenarios. Other aspects such as secret management, horizontal scaling and automated rollbacks in case of failed application updates all functioned as expected, satisfying the defined requirements. These aforementioned characteristics make Swarm a suitable orchestrator for this research's defined use-case, which focused mainly on simplicity and ease-of-use for small to medium-sized development teams.

8.3. Insights with respect to Nomad

HashiCorp Nomad provides a dedicated and robust scheduling solution which benefits greatly from tight integration with other HashiCorp services such as Consul and Vault, although it requires a more significant investment in terms of configuration and background knowledge. In comparison to Swarm, a considerably larger amount of time was spent configuring both Nomad and Consul to achieve comparable results in terms of managing and orchestrating the defined use-case. However, Nomad allows for a more flexible customization overall, providing scheduling features for different deployment scenarios. It accommodates both containerized as non-containerized application runtimes, and supports integrations with industry-standard networking and storage providers through CNI and CSI respectively. Furthermore, Nomad's environment can be managed through a visual interface, which could serve as a decisive factor, along with other unique features such as the service mesh supported by Consul. We can conclude that because of these advanced

configuration options and broader feature set, Nomad appears to be more suited for large-scale environments. It presents itself more as a solution for managing hundreds of active services across a substantial resource pool, aligning more closely with functionality of the scale of a datacenter infrastructure.

8.4. Impact of the Findings

Revisiting this research's defined research questions, we can verify that all main objectives were answered. RQ_1 was answered in section 7.1, stating two interconnected reasons that contribute to the complexity related to Kubernetes. In sections 7.2 and 7.3, we answered RQ_2 and RQ_3 respectively, referring to the relevant sections where the insights were detailed. The main objective of this research, encapsulated in RQ_4, was discussed in section 7.4.3, presenting Swarm as the more suitable container orchestration solution for this research's defined use-case.

We are of the opinion that the aforementioned findings from this chapter can be used by organizations to make an informed decision about their container orchestration needs, which is of course highly dependent on their specific use-case. Developers or other technical profiles can use this work to assess if the technologies that were covered could be sufficient for their requirements, since we evaluated both solutions in-depth and on a technical level. If they deem one of the candidates to be suitable, they can use parts of the demonstrated use-case—source code, application architecture, or other components, available as an open-source application on GitHub[1]—as a model or baseline implementation, which can then be adjusted for their needs. If they don't find the candidates suitable, they can consult the consolidated list of container orchestration technologies from section 5.3.2 as a starting point for alternatives that would suit their requirements more closely. In this way, we believe that we have contributed to the domain of container orchestration, highlight possible alternatives for Kubernetes, depending on the scenario at hand.

As stated multiple times throughout this thesis, the findings on Swarm and Nomad are limited to this research's defined use-case only, and should not be generalized to all environments or workloads. We recognize that the specificity of our employed methodology limits the broader relevance of these results, as results will certainly differ when other use-cases are subjected to the same tests.

[1] `https://github.com/cadeke/argo`

8.5. Future Work

Serving as the final section of this thesis, we want to highlight the possibilities for future efforts in the domain of container orchestration. We acknowledge that some parts of this thesis were limited by a constraint in time and resources, which provides an opportunity for potential continuations in those research areas.

In section 7.4.1, we discussed the attempts done to integrate a dedicated reverse proxy like Traefik into the Swarm stack. Although this effort was ultimately unsuccessful due to his thesis' time constraints, we believe that precisely defining the required use-case components—such as Grafana for dashboards, Prometheus for metrics, and Traefik for load balancing—in advance, could lead to more comparative results. Each orchestration candidate could then be more strictly evaluated based on how they handled each predefined component. In this thesis, the emphasis was on evaluating general functionalities rather than specific technologies used to address particular challenges. We think that this proposed process could result in more comparable results.

As an additional research avenue for Nomad, we think that Consul's service mesh deserves a more in-depth practical evaluation. In this thesis, we leveraged the mesh primarily to achieve automatic mTLS between services through proxies and separating concerts with intentions, thereby only scratching the surface of its capabilities. Further exploration in topics such as load balancing and the secure exposure of internal services through—particularly Consul's various gateway solutions—could yield valuable results for specific use-cases where, based on the research conducted in this thesis, the true potential of Nomad and Consul lies.

Finally, addressing the statements made in section 5.3.2, we want to encourage future research efforts to investigate candidates that were not included in this thesis' practical implementation. Kubernetes-inspired distributions that come with sensible default configurations—such as k3s or k0s—as well as managed cloud services that simplify the initial setup and overall configuration, present promising opportunities for development teams seeking an approachable solution for their orchestration requirements. These technologies were not selected for the practical implementation since the focus on open-source and the given time constraint limited the qualifying candidates for this thesis.

List of Figures

2.1 A three layer architecture as derived from Tie *et al.* [15] 7
2.2 A monolithic architecture based on example by Villamizar *et al.* [16] 8
2.3 A microservice architecture based on example by Villamizar *et al.* [16] 9
2.4 A high-level comparison between bare-metal (type 1) and host OS (type 2) hypervisors . . . 11
2.5 A high-level comparison between virtual machine and container architecture 13
2.6 A high-level overview of a container orchestration architecture 15

4.1 A high-level overview of this research project's methodology 26

5.1 A schematic overview of the microservice architecture for the presented use-case 29
5.2 A sequence diagram showing a human user interacting with the admin website 30
5.3 A sequence diagram showing a non-human user interacting with the query API 31

6.1 An overview of the ARGO application running on Swarm nodes 52
6.2 An overview of clients and servers in Nomad cluster 61
6.3 An overview of multiple jobs deploying ARGO components on Nomad cluster 62
6.4 An overview of OT job in Nomad after scaling 64
6.5 An overview of service catalog in Consul UI 65
6.6 An overview of query API service showing three instances in Consul UI 66
6.7 An overview of the Consul catalog of services in service mesh with proxy 69
6.8 A topology view in Consul of query API, showing upstream and downstream services 70

7.1 A chart showing interconnected Kubernetes challenges 82

A.1 A screenshot of the query website, showing a domain query 123
A.2 A screenshot of the admin website, showing the creation of a new record 124
A.3 A Grafana dashboard showing metrics about the query API 124
A.4 A Grafana dashboard showing container metrics 125

A.5 A Grafana dashboard showing node metrics . . . 125
A.6 A Grafana dashboard showing Postgres metrics . . . 126
A.7 A Grafana dashboard showing the ARGO application running on Swarm nodes . . . 126
A.8 A screenshot of Swarm CLI output of deactivated worker node . . . 127
A.9 A screenshot of Swarm logs of deactivated manager node . . . 127
A.10 A screenshot of Swarm service changes due to deactivated manager node . . . 128
A.11 A screenshot showing rolling updates in Swarm cluster . . . 128
A.12 A screenshot of a WireShark capture of plain text traffic on Swarm node . . . 129
A.13 A screenshot of a WireShark capture of encrypted traffic on Swarm node . . . 129
A.14 An overview of exposed services, accessible through Swarm routing mesh . . . 130
A.15 A topology overview of Nomad cluster . . . 130
A.16 A screenshot of Nomad UI showing draining of node vm04 . . . 131
A.17 A screenshot of Nomad UI showing drained node vm04 . . . 131
A.18 A topology overview of Nomad cluster after scaling . . . 132
A.19 An overview of Consul nodes . . . 132
A.20 An overview of deployed jobs, indicating an impacted Nomad cluster due to client failure . . 133
A.21 A topology overview of Nomad after rescheduling services as a result of client failure . . . 133
A.22 An overview of Nomad server nodes after server failure . . . 134
A.23 A screenshot of Nomad logs during server failure . . . 134
A.24 An overview of exposed services, accessible through load balancer on Nomad . . . 135
A.25 An overview of exposed routes via Fabio . . . 135
A.26 A screenshot of Nomad UI showing version history of OT application deployments . . . 136
A.27 A screenshot of Nomad UI showing approval prompt for canary promotion of OT application 136
A.28 A screenshot showing a browser error, indicating client certificate is required . . . 137
A.29 A screenshot of a WireShark capture of plain-text Nomad gossip traffic . . . 137
A.30 A screenshot of a WireShark capture of plain-text Nomad HTTP traffic . . . 138
A.31 A screenshot of a WireShark capture of plain-text Nomad RPC traffic . . . 138
A.32 A screenshot of a WireShark capture of encrypted Nomad gossip traffic . . . 139
A.33 A screenshot of a WireShark capture of encrypted Nomad HTTP traffic . . . 139
A.34 A screenshot of a WireShark capture of encrypted Nomad RPC traffic . . . 140
A.35 A screenshot of a WireShark capture of Nomad node, showing plain-text HTTP requests . . 140
A.36 A screenshot of a WireShark capture of Nomad node, showing encrypted communications . 141

A.37 A screenshot showing both the intentions and the configuration menu in Consul UI 141

List of Tables

2.1 A functionality overview based on example by Villamizar *et al.* [16] 8

5.1 An overview of the selected CO features 36
5.2 An overview of the selected CO candidates 41

6.1 An overview of the grading schema 49

7.1 An overview of the feature scores for Swarm and Nomad 90

List of Listings

1 An SQL snippet containing statements to create data model for the main database 44

2 A snippet containing Dockerfile instructions for creating admin API image 45

3 A snippet containing Dockerfile instructions for creating admin website image 46

4 A snippet of a Docker command showing an overview of ARGO running locally 47

5 A snippet of a Docker command showing an overview of Swarm nodes 51

6 A snippet of Docker commands showing the scaling of services in the Swarm cluster 53

7 A snippet of ping commands, testing internal DNS in Swarm cluster 54

8 A snippet of Docker logs showing a node leaving the Swarm cluster 55

9 A snippet of Docker commands showing an overview of networks in Swarm cluster 56

10 A YAML snippet showing CD configuration in Swarm cluster 57

11 A snippet of a Docker command, showing the published ports in Swarm cluster 58

12 A snippet of Docker commands showing the creation of a secret in Swarm 59

13 A snippet of various commands, verifying a mounted secret in a Swarm service 60

14 A snippet of the dnsmasq configuration for server nodes acting as internal DNS servers . . . 65

15 A snippet of Nomad logs detecting client node failure 66

16 A snippet of a Nomad job showing Consul service mesh configuration for query-api service . 68

17 A snippet of a Consul command showing service catalog with an additional sidecar proxy . . 68

18 A snippet of a Consul command showing intentions in service mesh 70

19 A snippet of various commands, verifying intentions in Consul service mesh 71

20 A snippet of a dig command, querying Consul DNS for query-api instances 72

21 A snippet of a Nomad job, showing CD configuration for OT application 72

22 A snippet of a Nomad command, showing stable version of OT application 73

23 A snippet of a Nomad command, planning application update 73

24 A snippet of a Nomad command, showing a successful deployment of application update . . 74

25 A snippet of a Nomad command, showing a failed deployment of application update 75

26 A snippet of various commands, setting up AES-256 for gossip encryption 76
27 A snippet of Nomad commands, generating CA, server and client certificates 77
28 A snippet of TLS configuration on a Nomad server node 77
29 A snippet of a Nomad job definition including secret environment variables 79
30 A snippet of various commands, verifying runtime secrets available in Nomad container . . . 79

31 A snippet of an Ansible command showing the IP configuration of hosts 142
32 A snippet of a GlusterFS command showing connected nodes 142
33 A snippet of a GlusterFS command showing replicated volume 143
34 A snippet of Docker commands showing the draining a node in Swarm cluster 143
35 A snippet of various commands verifying the network segmentation in Swarm cluster 144
36 A snippet of a "/etc/hosts" file simulating internal DNS 145
37 A snippet of commands showing Swarm root CA and node certificates on "vm01" 145
38 A snippet of ping commands verifying internal DNS in Nomad cluster with Consul 146
39 A snippet of a Nomad command, scaling OT application to ten instances in Nomad cluster . 147
40 A snippet of a Nomad command indicating an unauthenticated attempt after enabling mTLS 147
41 A snippet of a Nomad command showing canary deployment of OT application version . . . 148
42 A snippet of a Nomad command, promoting canary deployment of OT application 148

Acronyms

AAS	Azure App Services
ACA	Azure Container Apps
ACI	Azure Container Instances
ACM	Association for Computing Machinery
ACR	Azure Container Registry
AES	Advanced Encryption Standard
AKS	Azure Kubernetes Service
API	Application Programming Interface
ARGO	Address Resolver written in Go
AWS	Amazon Web Services
CA	Certificate Authority
CaaS	Containers As A Service
CD	Continuous Deployment
CI	Continuous Integration
CLI	Command Line Interface
CNCF	Cloud Native Computing Foundation
CNI	Container Network Interface
CO	Container Orchestration
COE	Container Orchestration Engine
COFFEE	Container Orchestration Frameworks' Full Experimental Evaluation
CPU	Central Processing Unit
CSI	Container Storage Interface

CSP Cloud Service Provider

DB Database
DNS Domain Name System

E2E End-to-End
EC2 Elastic Compute Cloud
ECR Amazon Elastic Container Registry
EKC Amazon Elastic Kubernetes Service
ESP Encapsulating Security Payload

GCP Google Cloud Platform
GKE Google Kubernetes Engine
GUI Graphical User Interface

HA High Availability / Highly Available
HCL HashiCorp Configuration Language
HTTP Hypertext Transfer Protocol
HTTPS Hypertext Transfer Protocol Secure

I/O Input/Output
IaC Infrastructure as Code
IEEE Institute of Electrical and Electronics Engineers
IoT Internet of Things
IP Internet Protocol
IPsec Internet Protocol Security
IT Information Technology

JSON JavaScript Object Notation

K8s	Kubernetes
L4	Layer 4 of OSI Model
L7	Layer 7 of OSI Model
LDAP	Lightweight Direct Access Protocol
LXC	Linux Containers
mTLS	Mutual Transport Layer Security
MVC	Model View Controller
NFS	Network File Share
NIC	Network Interface Card
OCI	Open Container Initiative
OCP	OpenShift Container Platform
OS	Operating System
OSI	Open Source Initiative
OSI Model	Open Systems Interconnection Model
OT	Operational Technology
PKI	Public Key Infrastructure
PSK	Pre-shared Key
RAM	Random Access Memory
RDMS	Relational Database Management System
REST	Representational State Transfer
RPC	Remote Procedure Call
RQ	Research Question
S3	Simple Storage Service

SBC	Single-board Computer
SLA	Service Level Agreement
SLI-KUBE	Security Linter for Kubernetes Manifests
SME	Small and Medium Enterprise
SNM	Subnet Mask
SQL	Structured Query Language
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UI	User Interface
VCS	Version Control System
VM	Virtual Machine
VPS	Virtual Private Server

Bibliography

[1] Amazon Web Services. “Our origins.” [Accessed: Feb. 8, 2025]. (n.d.), [Online]. Available: https://aws.amazon.com/about-aws/our-origins/.

[2] Amazon Web Services. “Announcing amazon elastic compute cloud (amazon ec2) - beta.” [Accessed: Feb. 8, 2025]. (Aug. 2006), [Online]. Available: https://aws.amazon.com/about-aws/whats-new/2006/08/24/announcing-amazon-elastic-compute-cloud-amazon-ec2--beta/.

[3] Paul McDonald. “Introducing google app engine + our new blog.” [Accessed: Feb. 8, 2025]. (Apr. 2008), [Online]. Available: https://cloudplatform.googleblog.com/2008/04/introducing-google-app-engine-our-new.html.

[4] Microsoft. “Windows azure general availability.” [Accessed: Feb. 8, 2025]. (Feb. 2010), [Online]. Available: https://web.archive.org/web/20140511230956/http://blogs.technet.com/b/microsoft_blog/archive/2010/02/01/windows-azure-general-availability.aspx.

[5] John K. Waters. “Oracle launches ’bare metal cloud’ in major iaas play.” [Accessed: Feb. 8, 2025]. (Oct. 2016), [Online]. Available: https://rcpmag.com/articles/2016/10/24/oracle-launches-bare-metal-cloud.aspx.

[6] IBM Developer. “A brief history of red hat openshift.” [Accessed: Feb. 8, 2025]. (Jul. 2019), [Online]. Available: https://developer.ibm.com/blogs/a-brief-history-of-red-hat-openshift/.

[7] Alibaba Cloud. “About alibaba cloud.” [Accessed: Feb. 8, 2025]. (2025), [Online]. Available: https://www.alibabacloud.com/en/about?_p_lc=1.

[8] Mike Loukides. “The cloud in 2021: Adoption continues.” [Accessed: Feb. 8, 2025]. (Dec. 2021), [Online]. Available: https://www.oreilly.com/radar/the-cloud-in-2021-adoption-continues/.

[9] Gartner. “Magic quadrant for strategic cloud platform services.” [Accessed: Feb. 8, 2025]. (Oct. 2024), [Online]. Available: https://www.gartner.com/doc/reprints?id=1-2J4OJHLF&ct=241019&st=sb&trk=249a077f-4dd7-4fe4-933b-3b26936318e6&sc_channel=el&refid=4b8bc24f-122d-4005-bee8-7cc5203c62f1.

[10] Parametrix. “Insurance: Cloud outage and the fortune 500 2023.” [Accessed: Feb. 8, 2025]. (2023), [Online]. Available: https://cdn.prod.website-files.com/66f518fd44a840ad8d3d68cc/673ec7c860545b4d84f4aebf_Parametrix%20Insurance-%20Cloud%20Outage%20and%20the%20Fortune%20500%202023.pdf.

[11] Florin Olariu and Liviu Alboaie, “Challenges in optimizing migration costs from on-premises to microsoft azure,” *Procedia Computer Science*, vol. 225, pp. 3649–3659, 2023. DOI: 10.1016/j.procs.2023.11.407.

[12] Mallesham Thoutam, “Azure cloud migration: Strategies, best practices, and performance optimization in enterprise environments,” *International Journal of Research in Computer Applications & Information Technology*, vol. 7, no. 2, pp. 576–586, 2024. DOI: https://doi.org/10.5281/zenodo.14012087.

[13] Stephen Orban. “6 strategies for migrating applications to the cloud.” [Accessed: Feb. 8, 2025]. (Nov. 2016), [Online]. Available: https://aws.amazon.com/blogs/enterprise-strategy/6-strategies-for-migrating-applications-to-the-cloud/.

[14] Microsoft. “Common web application architectures.” Accessed: 2025-02-10. (2023), [Online]. Available: https://learn.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/common-web-application-architectures.

[15] Jun Tie, Jia Jin, and Xiaorong Wang, “Study on application model of three-tiered architecture,” in *2011 Second International Conference on Mechanic Automation and Control Engineering*, Jul. 2011, pp. 7715–7718. DOI: 10.1109/MACE.2011.5988838. [Online]. Available: https://ieeexplore.ieee.org/document/5988838.

[16] Mario Villamizar, Oscar Garcés, Harold Castro, Mauricio Verano, Lorena Salamanca, Rubby Casallas, and Santiago Gil, “Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud,” in *2015 10th Computing Colombian Conference (10CCC)*, Sep. 2015, pp. 583–590. DOI: 10.1109/ColumbianCC.2015.7333476. [Online]. Available: https://ieeexplore.ieee.org/document/7333476.

[17] Claus Pahl and Pooyan Jamshidi, "Microservices: A Systematic Mapping Study," in *Proceedings of the 6th International Conference on Cloud Computing and Services Science - Volume 1 and 2*, ser. CLOSER 2016, Setubal, PRT: SCITEPRESS - Science and Technology Publications, Lda, Apr. 2016, pp. 137–146, ISBN: 978-989-758-182-3. DOI: 10.5220/0005785501370146. [Online]. Available: https://doi.org/10.5220/0005785501370146.

[18] Wilhelm Hasselbring, "Microservices for scalability: Keynote talk abstract," in *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, ser. ICPE '16, Delft, The Netherlands: Association for Computing Machinery, 2016, pp. 133–134, ISBN: 9781450340809. DOI: 10.1145/2851553.2858659. [Online]. Available: https://doi.org/10.1145/2851553.2858659.

[19] Grzegorz Blinowski, Anna Ojdowska, and Adam Przybyek, "Monolithic vs. microservice architecture: A performance and scalability evaluation," *IEEE Access*, vol. 10, pp. 20 357–20 374, 2022. DOI: 10.1109/ACCESS.2022.3152803.

[20] Sachchidanand Singh and Nirmala Singh, "Containers & docker: Emerging roles & future of cloud technology," in *2016 2nd International Conference on Applied and Theoretical Computing and Communication Technology (iCATccT)*, 2016, pp. 804–807. DOI: 10.1109/ICATCCT.2016.7912109.

[21] The New Stack. "What led amazon to its own microservices architecture." Accessed: 2025-02-12. (2015), [Online]. Available: https://thenewstack.io/led-amazon-microservices-architecture/.

[22] F5 Networks. "Adopting microservices at netflix: Lessons for architectural design." Accessed: 2025-02-12. (2015), [Online]. Available: https://www.f5.com/company/blog/nginx/microservices-at-netflix-architectural-best-practices.

[23] Josh Evans. "Mastering chaos - a netflix guide to microservices." Accessed: 2025-02-12. (2017), [Online]. Available: https://www.youtube.com/watch?v=CZ3wIuvmHeM.

[24] Kevin Goldsmith. "Microservices at spotify." Accessed: 2025-02-12. (2015), [Online]. Available: https://www.youtube.com/embed/7LGPeBgNFuU.

[25] Sahiti Kappagantula. "Microservice architecture learn, build, and deploy applications." Accessed: 2025-02-12. (2018), [Online]. Available: https://dzone.com/articles/microservice-architecture-learn-build-and-deploy-a.

[26] Marek Moravcik and Martin Kontsek, "Overview of docker container orchestration tools," in *2020 18th International Conference on Emerging eLearning Technologies and Applications (ICETA)*, 2020, pp. 475–480. DOI: 10.1109/ICETA51985.2020.9379236.

[27] Lubos Mercl and Jakub Pavlík, "The comparison of container orchestrators," *Advances in Intelligent Systems and Computing*, 2018. DOI: 10.1007/978-981-13-1165-9_62. [Online]. Available: https://api.semanticscholar.org/CorpusID:69844239.

[28] Ann Mary Joy, "Performance comparison between linux containers and virtual machines," in *2015 International Conference on Advances in Computer Engineering and Applications*, 2015, pp. 342–346. DOI: 10.1109/ICACEA.2015.7164727.

[29] Roberto Morabito, Jimmy Kjällman, and Miika Komu, "Hypervisors vs. lightweight virtualization: A performance comparison," in *2015 IEEE International Conference on Cloud Engineering*, 2015, pp. 386–393. DOI: 10.1109/IC2E.2015.74.

[30] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson, "Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors," in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, ser. EuroSys '07, Lisbon, Portugal: Association for Computing Machinery, 2007, pp. 275–287, ISBN: 9781595936363. DOI: 10.1145/1272996.1273025. [Online]. Available: https://doi.org/10.1145/1272996.1273025.

[31] David Bernstein, "Containers and Cloud: From LXC to Docker to Kubernetes," en, *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, Sep. 2014, ISSN: 2325-6095, 2372-2568. DOI: 10.1109/MCC.2014.51. [Online]. Available: https://ieeexplore.ieee.org/document/7036275/.

[32] Theo Combe, Antony Martin, and Roberto Di Pietro, "To docker or not to docker: A security perspective," *IEEE Cloud Computing*, vol. 3, no. 5, pp. 54–62, 2016. DOI: 10.1109/MCC.2016.100.

[33] Luciano Baresi, Giovanni Quattrocchi, and Nicholas Rasi, "A qualitative and quantitative analysis of container engines," *J. Syst. Softw.*, vol. 210, no. C, Apr. 2024, ISSN: 0164-1212. DOI: 10.1016/j.jss.2024.111965. [Online]. Available: https://doi.org/10.1016/j.jss.2024.111965.

[34] Docker Inc; “What is a container?” Accessed: 2025-05-14, Docker Inc. (2025), [Online]. Available: https://docs.docker.com/get-started/docker-concepts/the-basics/what-is-a-container/.

[35] Martin Straesser, Jonas Mathiasch, André Bauer, and Samuel Kounev, “A systematic approach for benchmarking of container orchestration frameworks,” in *Proceedings of the 2023 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE ’23, Coimbra, Portugal: Association for Computing Machinery, 2023, pp. 187–198, ISBN: 9798400700682. DOI: 10.1145/3578244.3583726. [Online]. Available: https://doi.org/10.1145/3578244.3583726.

[36] Asif Khan, “Key characteristics of a container orchestration platform to enable a modern application,” *IEEE Cloud Computing*, vol. 4, no. 5, pp. 42–48, 2017. DOI: 10.1109/MCC.2017.4250933.

[37] Eric Brewer. “An update on container support on google cloud platform.” Accessed: 2025-02-17. (2014), [Online]. Available: https://cloudplatform.googleblog.com/2014/06/an-update-on-container-support-on-google-cloud-platform.html.

[38] Frederic Lardinois. “As kubernetes hits 1.0, google donates technology to newly formed cloud native computing foundation.” Accessed: 2025-02-17. (2015), [Online]. Available: https://techcrunch.com/2015/07/21/as-kubernetes-hits-1-0-google-donates-technology-to-newly-formed-cloud-native-computing-foundation-with-ibm-intel-twitter-and-others/.

[39] David Soldani, Petrit Nahi, Hami Bour, Saber Jafarizadeh, Mohammed F. Soliman, Leonardo Di Giovanna, Francesco Monaco, Giuseppe Ognibene, and Fulvio Risso, “Ebpf: A new approach to cloud-native observability, networking and security for current (5g) and future mobile networks (6g and beyond),” *IEEE Access*, vol. 11, pp. 57 174–57 202, 2023. DOI: 10.1109/ACCESS.2023.3281480.

[40] Cloud Native Computing Foundation. “Cncf annual survey 2023.” Accessed: 2025-02-17. (2023), [Online]. Available: https://www.cncf.io/reports/cncf-annual-survey-2023/.

[41] OpenStack. “October 2016 openstack user survey report.” Accessed: 2025-03-07. (Oct. 2016), [Online]. Available: https://object-storage-ca-ymq-1.vexxhost.net/swift/v1/6e4619c416ff4bd19e1c087f27a43eea/www-assets-prod/survey/October2016SurveyRe pdf.

[42] OpenStack. "November 2017 openstack user survey report." Accessed: 2025-03-07. (Nov. 2016), [Online]. Available: https://object-storage-ca-ymq-1.vexxhost.net/swift/v1/6e4619c416ff4bd19e1c087f27a43eea/www-assets-prod/survey/OpenStack-User-Survey-Nov17.pdf.

[43] OpenStack. "2018 openstack user survey report." Accessed: 2025-03-07. (2018), [Online]. Available: https://www.openstack.org/user-survey/2018-user-survey-report.

[44] Kubernetes. "Kubernetes documentation - concepts - overview." Accessed: 2025-03-11. (2024), [Online]. Available: https://kubernetes.io/docs/concepts/overview/#why-you-need-kubernetes-and-what-can-it-do.

[45] Leila Abdollahi Vayghan, Mohamed Aymen Saied, Maria Toeroe, and Ferhat Khendek, "Deploying microservice based applications with kubernetes: Experiments and lessons learned," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, 2018, pp. 970–973. DOI: 10.1109/CLOUD.2018.00148.

[46] Jeff Geerling. "Kubernetes' complexity." Accessed: 2025-02-18. (2018), [Online]. Available: https://www.jeffgeerling.com/blog/2018/kubernetes-complexity.

[47] Akond Rahman, Shazibul Islam Shamim, Dibyendu Brinto Bose, and Rahul Pandita, "Security misconfigurations in open source kubernetes manifests: An empirical study," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 4, May 2023, ISSN: 1049-331X. DOI: 10.1145/3579639. [Online]. Available: https://doi.org/10.1145/3579639.

[48] Sergii Telenyk, Oleksii Sopov, Eduard Zharikov, and Grzegorz Nowakowski, "A Comparison of Kubernetes and Kubernetes-Compatible Platforms," in *2021 11th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*, ISSN: 2770-4254, vol. 1, Sep. 2021, pp. 313–317. DOI: 10.1109/IDAACS53288.2021.9660392. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9660392.

[49] Heiko Koziolek and Nafise Eskandani, "Lightweight Kubernetes Distributions: A Performance Comparison of MicroK8s, k3s, k0s, and Microshift," in *Proceedings of the 2023 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '23, New York, NY, USA: Association for Computing Machinery, Apr. 2023, pp. 17–29, ISBN: 979-8-4007-0068-2. DOI: 10.1145/3578244.3583737. [Online]. Available: https://doi.org/10.1145/3578244.3583737.

[50] Isam Mashhour Al Jawarneh, Paolo Bellavista, Filippo Bosi, Luca Foschini, Giuseppe Martuscelli, Rebecca Montanari, and Amedeo Palopoli, "Container orchestration engines: A thorough functional and performance comparison," in *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*, 2019, pp. 1–6. DOI: 10.1109/ICC.2019.8762053.

[51] Arnaldo Pereira Ferreira and Richard Sinnott, "A performance evaluation of containers running on managed kubernetes services," in *2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 2019, pp. 199–208. DOI: 10.1109/CloudCom.2019.00038.

[52] Siddhartha Singh, FNU Shivanshi, and Rachit Jain, "A framework for evaluating container performance across diverse kubernetes environments," in *2024 IEEE 13th International Conference on Cloud Networking (CloudNet)*, 2024, pp. 1–6. DOI: 10.1109/CloudNet62863.2024.10815769.

[53] Jay Shah and Dushyant Dubaria, "Building modern clouds: Using docker, kubernetes & google cloud platform," in *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*, 2019, pp. 0184–0189. DOI: 10.1109/CCWC.2019.8666479.

[54] Marian Ileana, Maria Ioana Oproiu, and Constantin Viorel Marian, "Using docker swarm to improve performance in distributed web systems," in *2024 International Conference on Development and Application Systems (DAS)*, 2024, pp. 1–6. DOI: 10.1109/DAS61944.2024.10541234.

[55] Eddy Truyen, Dimitri Van Landuyt, Davy Preuveneers, Bert Lagaisse, and Wouter Joosen, "A Comprehensive Feature Comparison Study of Open-Source Container Orchestration Frameworks," en, *Applied Sciences*, vol. 9, no. 5, p. 931, Mar. 2019, ISSN: 2076-3417. DOI: 10.3390/app9050931. [Online]. Available: https://www.mdpi.com/2076-3417/9/5/931.

[56] Anshita Malviya and Rajendra Kumar Dwivedi, "A Comparative Analysis of Container Orchestration Tools in Cloud Computing," in *2022 9th International Conference on Computing for Sustainable Global Development (INDIACom)*, Mar. 2022, pp. 698–703. DOI: 10.23919/INDIACom54597.2022.9763171. [Online]. Available: https://ieeexplore.ieee.org/document/9763171/?arnumber=9763171.

[57] Charlene O'Hanlon, "A conversation with werner vogels: Learning from the amazon technology platform: Many think of amazon as that hugely successful online bookstore. you would expect amazon cto werner vogels to embrace this distinction, but in fact it causes him some concern.," *Queue*,

vol. 4, no. 4, pp. 14–22, May 2006, ISSN: 1542-7730. DOI: 10.1145/1142055.1142065. [Online]. Available: https://doi.org/10.1145/1142055.1142065.

[58] Noam Ben-Asher, Joachim Meyer, Sebastian Möller, and Roman Englert, "An experimental system for studying the tradeoff between usability and security," in *2009 International Conference on Availability, Reliability and Security*, 2009, pp. 882–887. DOI: 10.1109/ARES.2009.174.

[59] Jay Patty. "Bringing the security vs. usability pendulum to a stop." [Accessed: Feb. 21, 2025]. (Sep. 2024), [Online]. Available: https://www.zscaler.com/cxorevolutionaries/insights/bringing-security-vs-usability-pendulum-stop.

[60] National Institute of Standards and Technology, "Security and Privacy Controls for Information Systems and Organizations," Tech. Rep. Special Publication 800-53 Revision 5, Sep. 2020, Includes updates as of December 10, 2020. DOI: 10.6028/NIST.SP.800-53r5. [Online]. Available: https://csrc.nist.gov/pubs/sp/800/53/r5/upd1/final.

[61] Ramaswamy Chandramouli, "Security Strategies for Microservices-based Application Systems," Tech. Rep. Special Publication 800-204, Aug. 2019. DOI: 10.6028/NIST.SP.800-204. [Online]. Available: https://csrc.nist.gov/pubs/sp/800/204/final.

[62] Cloudflare. "Types of load balancing algorithms." Accessed: 2025-03-21. (2025), [Online]. Available: https://www.cloudflare.com/learning/performance/types-of-load-balancing-algorithms/.

[63] Amazon Web Services. "What is load balancing?" Accessed: 2025-03-21. (2025), [Online]. Available: https://aws.amazon.com/what-is/load-balancing/.

[64] Kubernetes. "Kubernetes documentation - persistent volumes." Accessed: 2025-03-21. (Mar. 2025), [Online]. Available: https://kubernetes.io/docs/concepts/storage/persistent-volumes/.

[65] Docker Inc. "Swarm mode." Accessed: 2025-03-20. (2025), [Online]. Available: https://docs.docker.com/engine/swarm/.

[66] Docker Inc. "Docker engine." Accessed: 2025-04-26, Docker Inc. (2025), [Online]. Available: https://docs.docker.com/engine/#licensing.

[67] moby. "Swarmkit." Accessed: 2025-04-26, moby. (2025), [Online]. Available: https://github.com/moby/swarmkit.

[68] K3s Project Authors. "K3s." Accessed: 2025-03-20. (Mar. 2025), [Online]. Available: https://docs.k3s.io/.

[69] Mirantis. "K0s - the zero friction kubernetes." Accessed: 2025-03-20. (2021), [Online]. Available: https://docs.k0sproject.io/stable/.

[70] Canonical. "Microk8s documentation." Accessed: 2025-03-20. (Jan. 2024), [Online]. Available: https://microk8s.io/docs.

[71] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'11, Boston, MA: USENIX Association, 2011, pp. 295–308.

[72] Apache Software Foundation. "Apache mesos." Accessed: 2025-03-20. (), [Online]. Available: https://mesos.apache.org/.

[73] HashiCorp. "Nomad community." Accessed: 2025-03-20. (), [Online]. Available: https://developer.hashicorp.com/nomad/docs/nomad-vs-kubernetes/alternative.

[74] Armon Dadgar. "Hashicorp adopts business source license." Accessed: 2025-03-20. (Aug. 2023), [Online]. Available: https://www.hashicorp.com/blog/hashicorp-adopts-business-source-license.

[75] Red Hat. "Openshift container platform." Accessed: 2025-03-20. (2024), [Online]. Available: https://docs.redhat.com/en/documentation/openshift_container_platform/4.13/.

[76] Red Hat. "Ansible." Accessed: 2025-03-24. (2025), [Online]. Available: https://www.redhat.com/en/ansible-collaborative?intcmp=7015Y000003t7aWQAQ.

[77] Zheng Li, Nicolás Saldías-Vallejos, Diego Seco, María Andrea Rodríguez, and Rajiv Ranjan, "Long live the image: On enabling resilient production database containers for microservice applications," *IEEE Transactions on Software Engineering*, vol. 50, no. 9, pp. 2363–2378, 2024. DOI: 10.1109/TSE.2024.3436623.

[78] Docker Inc. "Volumes." Accessed: 2025-03-26. (2025), [Online]. Available: https://docs.docker.com/engine/storage/volumes/#share-data-between-machines.

[79] bitchecker (GitHub). "Is the project still alive?" Accessed: 2025-03-27. (Jan. 2024), [Online]. Available: https://github.com/gluster/glusterfs/discussions/4324.

[80] Red Hat. "Red hat gluster storage life cycle." Accessed: 2025-03-27. (2025), [Online]. Available: https://access.redhat.com/support/policy/updates/rhs/.

[81] QEMU. "Changelog/9.2." Accessed: 2025-03-27. (Mar. 2025), [Online]. Available: https://wiki.qemu.org/ChangeLog/9.2.

[82] Micheal Larabel. "Fedora stakeholders have been debating whether to retire glusterfs." Accessed: 2025-03-27. (Jan. 2025), [Online]. Available: https://www.phoronix.com/news/Fedora-Maybe-Retire-GlusterFS.

[83] Docker Inc. "Volumes." Accessed: 2025-04-31. (2025), [Online]. Available: https://docs.docker.com/engine/storage/volumes/.

[84] Docker Inc. "Swarm mode key concepts." Accessed: 2025-03-27. (2025), [Online]. Available: https://docs.docker.com/engine/swarm/key-concepts/.

[85] Docker Inc. "Administer and maintain a swarm of docker engines." Accessed: 2025-03-27. (2025), [Online]. Available: https://docs.docker.com/engine/swarm/admin_guide/#add-manager-nodes-for-fault-tolerance.

[86] gianarb. "Orbiter." Accessed: 2025-03-28. (Jun. 2018), [Online]. Available: https://github.com/gianarb/orbiter.

[87] Docker Inc. "Swarm mode." Accessed: 2025-03-28. (2025), [Online]. Available: https://docs.docker.com/engine/swarm/#service-discovery.

[88] Docker Inc. "Swarm mode key concepts." Accessed: 2025-03-28. (2025), [Online]. Available: https://docs.docker.com/engine/swarm/key-concepts/#load-balancing.

[89] mattlqx. "Swarm tasks inconsistently stuck in new state on service creation." Accessed: 2025-03-28. (Dec. 2017), [Online]. Available: https://github.com/moby/moby/issues/35849.

[90] cima. "Inconsistent overlay network size limits." Accessed: 2025-03-28. (Jan. 2017), [Online]. Available: https://github.com/docker/docs/issues/5853.

[91] Paul Rey. "Docker service stuck in new state (swarm)." Accessed: 2025-03-28. (Jun. 2018), [Online]. Available: https://stackoverflow.com/questions/51078764/docker-service-stuck-in-new-state-swarm.

[92] Docker Inc. "Apply rolling updates to a service." Accessed: 2025-03-29. (2025), [Online]. Available: https://docs.docker.com/engine/swarm/swarm-tutorial/rolling-update/.

[93] Docker Inc. "Use swarm mode routing mesh." Accessed: 2025-03-30. (2025), [Online]. Available: https://docs.docker.com/engine/swarm/ingress/.

[94] Docker Inc. "Manage swarm security with public key infrastructure (pki)." Accessed: 2025-03-29. (2025), [Online]. Available: https://docs.docker.com/engine/swarm/how-swarm-mode-works/pki/.

[95] Docker Inc. "Manage swarm service networks." Accessed: 2025-03-29. (2025), [Online]. Available: https://docs.docker.com/engine/swarm/networking/#swarm-and-types-of-traffic.

[96] Docker Inc. "Overlay network driver." Accessed: 2025-03-29. (2025), [Online]. Available: https://docs.docker.com/engine/network/drivers/overlay/#encrypt-traffic-on-an-overlay-network.

[97] Docker Inc. "Lock your swarm to protect its encryption key." Accessed: 2025-03-29. (2025), [Online]. Available: https://docs.docker.com/engine/swarm/swarm_manager_locking/.

[98] Docker Inc. "Manage sensitive data with docker secrets." Accessed: 2025-03-29. (2025), [Online]. Available: https://docs.docker.com/engine/swarm/secrets/.

[99] HashiCorp. "Container storage interface (csi) plugins." Accessed: 2025-04-19, HashiCorp. (2025), [Online]. Available: https://developer.hashicorp.com/nomad/docs/concepts/plugins/storage/csi.

[100] HashiCorp. "Architecture." Accessed: 2025-04-05. (2025), [Online]. Available: https://developer.hashicorp.com/nomad/docs/concepts/architecture.

[101] HashiCorp. "Glossary." Accessed: 2025-04-19, HashiCorp. (2025), [Online]. Available: https://developer.hashicorp.com/nomad/docs/glossary.

[102] HashiCorp. "Consensus protocol." Accessed: 2025-04-05. (2025), [Online]. Available: https://developer.hashicorp.com/nomad/docs/concepts/consensus#deployment-table.

[103] HashiCorp. "Deployment guide." Accessed: 2025-04-05. (2025), [Online]. Available: https://developer.hashicorp.com/consul/tutorials/production-vms/deployment-guide.

[104] HashiCorp. "Deployment guide." Accessed: 2025-04-05. (2025), [Online]. Available: https://developer.hashicorp.com/nomad/tutorials/enterprise/production-deployment-guide-vm-with-consul.

[105] HashiCorp. "Nomad autoscaler overview." Accessed: 2025-04-14. (2025), [Online]. Available: https://developer.hashicorp.com/nomad/tools/autoscaling.

[106] Nick Ethier and Michael Schurter. "Consul connect integration in hashicorp nomad." Accessed: 2025-04-19, HashiCorp. (Sep. 2019), [Online]. Available: https://www.hashicorp.com/blog/consul-connect-integration-in-hashicorp-nomad.

[107] HashiCorp. "Service intentions overview." Accessed: 2025-04-19, HashiCorp. (2025), [Online]. Available: https://developer.hashicorp.com/consul/docs/connect/intentions.

[108] HashiCorp. "Networking." Accessed: 2025-04-19, HashiCorp. (2025), [Online]. Available: https://developer.hashicorp.com/nomad/docs/networking.

[109] HashiCorp. "Container network interface (cni)." Accessed: 2025-04-19, HashiCorp. (2025), [Online]. Available: https://developer.hashicorp.com/nomad/docs/networking/cni.

[110] HashiCorp. "Gateways overview." Accessed: 2025-04-19, HashiCorp. (2025), [Online]. Available: https://developer.hashicorp.com/consul/docs/connect/gateways.

[111] HashiCorp. "Enable rolling updates." Accessed: 2025-04-15. (2025), [Online]. Available: https://developer.hashicorp.com/nomad/tutorials/job-updates/job-rolling-update.

[112] HashiCorp. "Security model." Accessed: 2025-04-16. (2025), [Online]. Available: https://developer.hashicorp.com/nomad/docs/concepts/security.

[113] HashiCorp. "Enable tls encryption for nomad." Accessed: 2025-04-16. (2025), [Online]. Available: https://developer.hashicorp.com/nomad/tutorials/transport-security/security-enable-tls.

[114] HashiCorp. "Enable gossip encryption for nomad." Accessed: 2025-04-16. (2025), [Online]. Available: https://developer.hashicorp.com/nomad/tutorials/transport-security/security-gossip-encryption.

[115] HashiCorp. "Encrypted communication between consul agents." Accessed: 2025-04-16. (2025), [Online]. Available: https://developer.hashicorp.com/consul/docs/security/encryption.

[116] HashiCorp. "Consul service mesh." Accessed: 2025-04-17. (2025), [Online]. Available: https://developer.hashicorp.com/consul/docs/connect.

[117] HashiCorp. "Built-in certificate authority for service mesh." Accessed: 2025-04-19, HashiCorp. (2025), [Online]. Available: https://developer.hashicorp.com/consul/docs/connect/ca/consul.

[118] HashiCorp. "Consul security model overview." Accessed: 2025-04-20, HashiCorp. (2025), [Online]. Available: https://developer.hashicorp.com/consul/docs/security/security-models/core.

[119] HashiCorp. "Consul security considerations." Accessed: 2025-04-20, HashiCorp. (2025), [Online]. Available: https://developer.hashicorp.com/consul/tutorials/production-vms/security.

[120] HashiCorp. "Key/value (kv) store overview." Accessed: 2025-04-15. (2025), [Online]. Available: https://developer.hashicorp.com/consul/docs/dynamic-app-config/kv.

[121] HashiCorp. "Runtime environment settings." Accessed: 2025-04-15. (2025), [Online]. Available: https://developer.hashicorp.com/nomad/docs/runtime/environment#secrets.

[122] HashiCorp. "Allocation filesystems." Accessed: 2025-04-15. (2025), [Online]. Available: https://developer.hashicorp.com/nomad/docs/concepts/filesystem.

A. Implementation Details

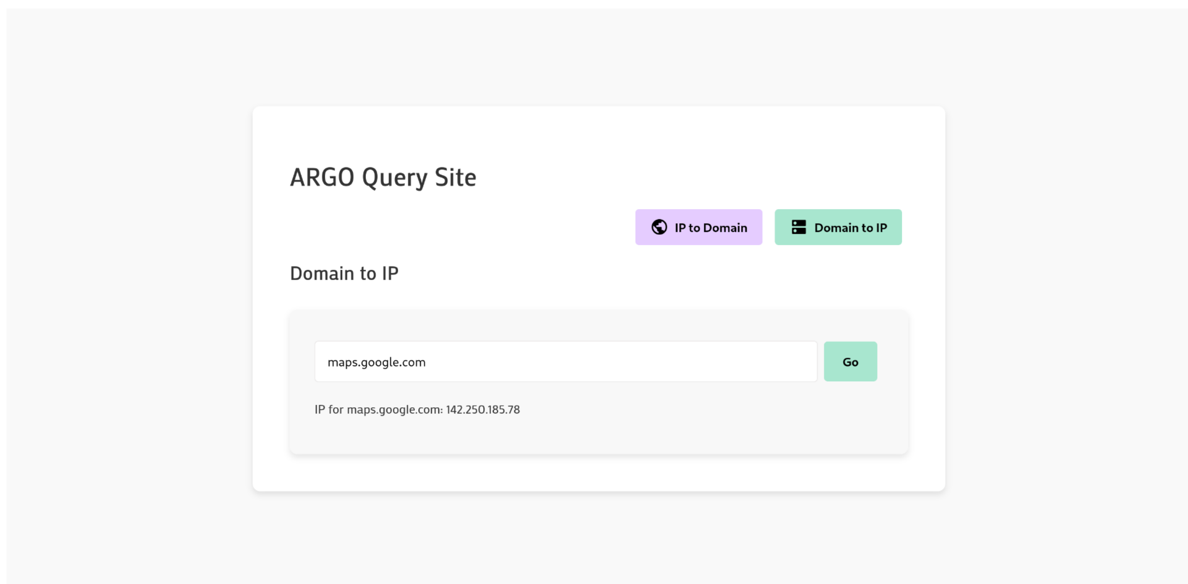

Figure A.1.: A screenshot of the query website, showing a domain query

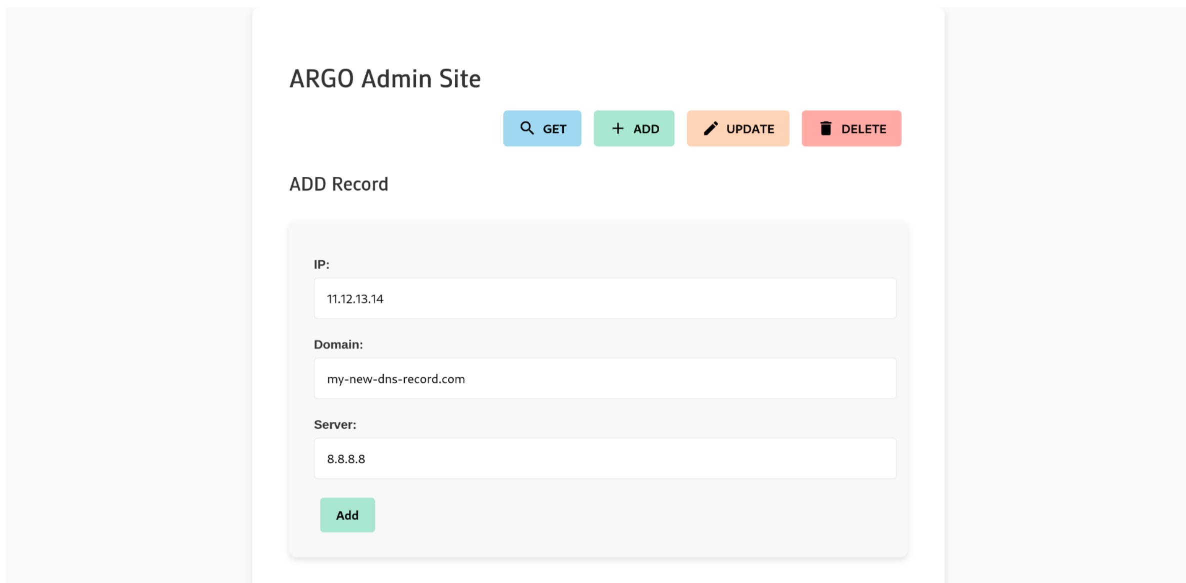

Figure A.2.: A screenshot of the admin website, showing the creation of a new record

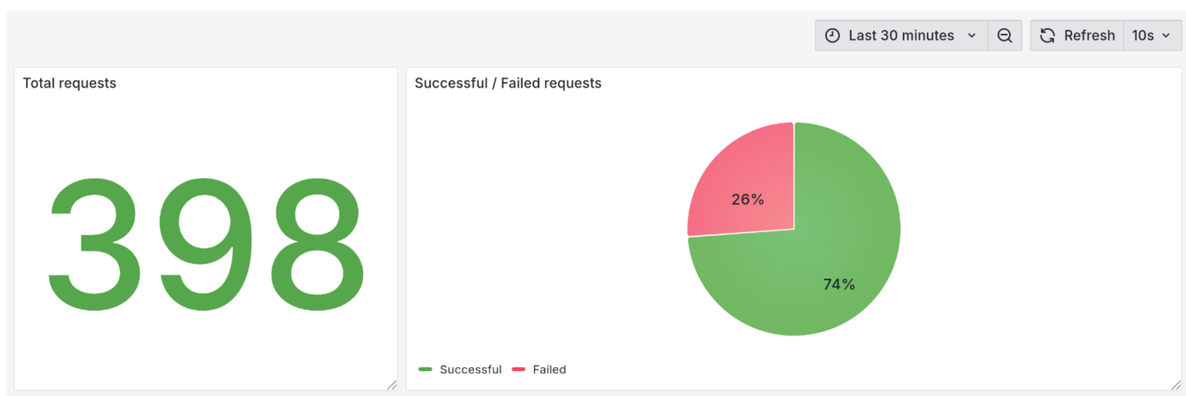

Figure A.3.: A Grafana dashboard showing metrics about the query API

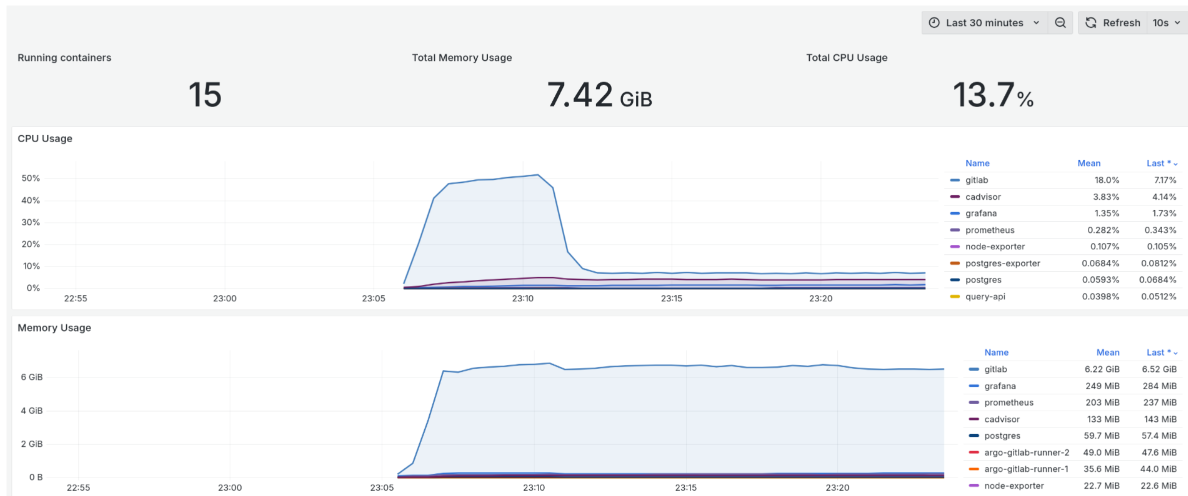

Figure A.4.: A Grafana dashboard showing container metrics

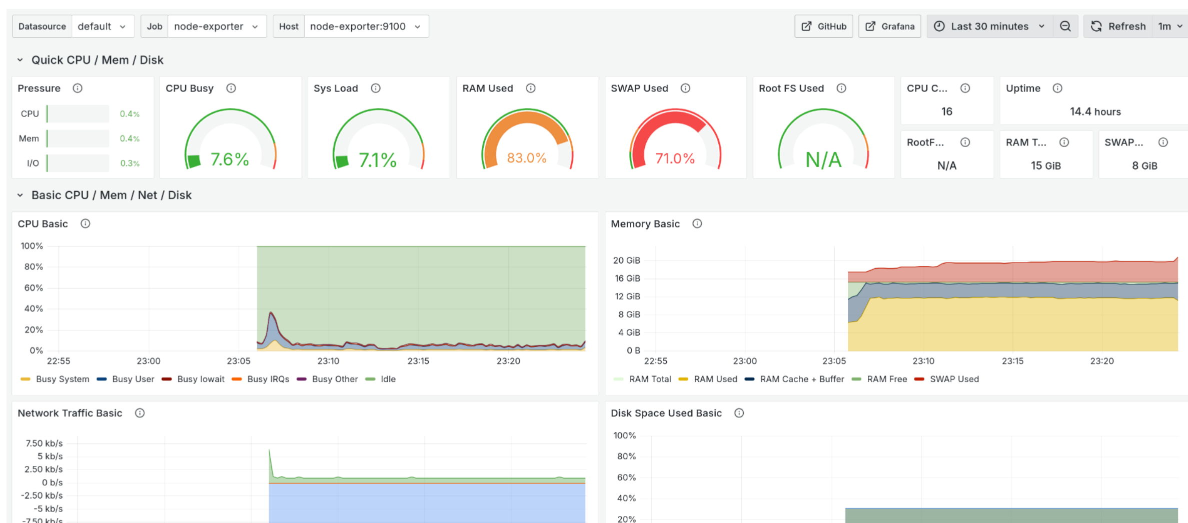

Figure A.5.: A Grafana dashboard showing node metrics

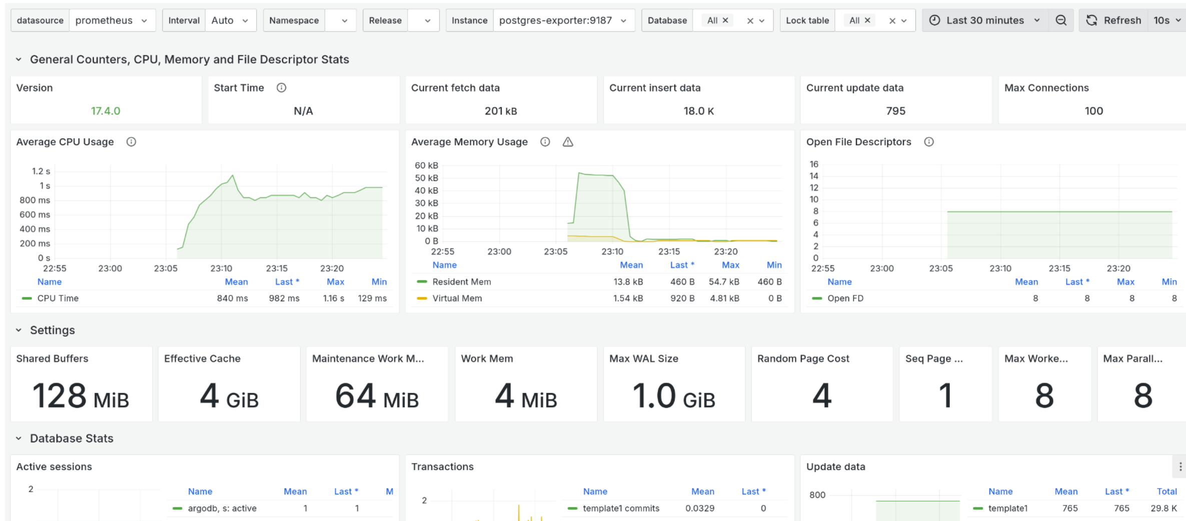

Figure A.6.: A Grafana dashboard showing Postgres metrics

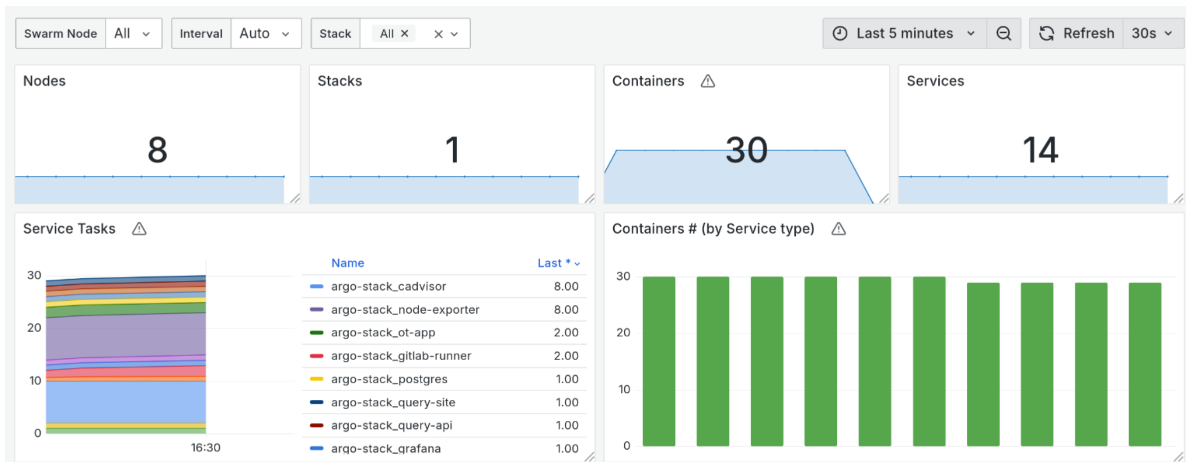

Figure A.7.: A Grafana dashboard showing the ARGO application running on Swarm nodes

```
# NODE VM8 READY AND RUNNING SERVICES
student@vm01:~> docker node ls
ID                            HOSTNAME   STATUS    AVAILABILITY   MANAGER STATUS   ENGINE VERSION
anvacpekzrc35twzg5y3r1gic *   vm01       Ready     Active         Reachable        28.0.4
i5kgrnmecaqa44rqbbcw52db6     vm02       Ready     Active         Leader           28.0.4
yzrumknlvx6kd383sa7o2jrop     vm03       Ready     Active         Reachable        28.0.4
z9uoam07pax0uuwn6asqbf0fp     vm04       Ready     Active                          28.0.4
yezg2ojyvca980iehdo0hisqk     vm05       Ready     Active                          28.0.4
7dqcqy9pdzirqjuia96phycwp     vm06       Ready     Active                          28.0.4
teesrnys35963sbyzp07o20tk     vm07       Ready     Active                          28.0.4
h500eow7r8ps171osflrirni9     vm08       Ready     Active                          28.0.4
student@vm01:~> docker node ps vm08
ID             NAME                                                  IMAGE                            NODE      DESIRED STATE   CURRENT STATE            ERROR     PORTS
6nvyqtu2djk8   argo-stack_cadvisor.h500eow7r8ps171osflrirni9        gcr.io/cadvisor/cadvisor:latest   vm08      Running         Running 5 hours ago
u7lfugcbesw7   argo-stack_node-exporter.h500eow7r8ps171osflrirni9   prom/node-exporter:latest         vm08      Running         Running 5 hours ago
nujkzlqcgxn6   argo-stack_ot-app.1                                   cadeke/argo-ot-app:latest        vm08      Running         Running 5 minutes ago
xcc3l9kcrzln   argo-stack_query-site.1                               cadeke/argo-q-site:latest        vm08      Running         Running 5 hours ago

# NODE VM8 SHUTDOWN
student@vm01:~> docker node ls
ID                            HOSTNAME   STATUS    AVAILABILITY   MANAGER STATUS   ENGINE VERSION
anvacpekzrc35twzg5y3r1gic *   vm01       Ready     Active         Reachable        28.0.4
i5kgrnmecaqa44rqbbcw52db6     vm02       Ready     Active         Leader           28.0.4
yzrumknlvx6kd383sa7o2jrop     vm03       Ready     Active         Reachable        28.0.4
z9uoam07pax0uuwn6asqbf0fp     vm04       Ready     Active                          28.0.4
yezg2ojyvca980iehdo0hisqk     vm05       Ready     Active                          28.0.4
7dqcqy9pdzirqjuia96phycwp     vm06       Ready     Active                          28.0.4
teesrnys35963sbyzp07o20tk     vm07       Ready     Active                          28.0.4
h500eow7r8ps171osflrirni9     vm08       Down      Active                          28.0.4
student@vm01:~> docker node ps vm08
ID             NAME                                                  IMAGE                            NODE      DESIRED STATE   CURRENT STATE            ERROR     PORTS
6nvyqtu2djk8   argo-stack_cadvisor.h500eow7r8ps171osflrirni9        gcr.io/cadvisor/cadvisor:latest   vm08      Shutdown        Running 5 hours ago
u7lfugcbesw7   argo-stack_node-exporter.h500eow7r8ps171osflrirni9   prom/node-exporter:latest         vm08      Shutdown        Running 5 hours ago
nujkzlqcgxn6   argo-stack_ot-app.1                                   cadeke/argo-ot-app:latest        vm08      Shutdown        Running 8 minutes ago
xcc3l9kcrzln   argo-stack_query-site.1                               cadeke/argo-q-site:latest        vm08      Shutdown        Running 5 hours ago
```

Figure A.8.: A screenshot of Swarm CLI output of deactivated worker node

```
# Node vm02 leaves cluster
Mar 28 20:03:59 vm01 dockerd[62385]: time="2025-03-28T20:03:59.286266256Z" level=info msg="Node fe2d3b9aa646 change state NodeActive --> NodeLeft"
Mar 28 20:03:59 vm01 dockerd[62385]: time="2025-03-28T20:03:59.287087707Z" level=info msg="vm01(191b72da74cb): Node leave event for fe2d3b9aa646/10.203.96.231"
Mar 28 20:03:59 vm01 dockerd[62385]: time="2025-03-28T20:03:59.628475426Z" level=info msg="Node fe2d3b9aa646/10.203.96.231, left gossip cluster"
Mar 28 20:04:04 vm01 dockerd[62385]: time="2025-03-28T20:04:04.253831279Z" level=error msg="agent: session failed" backoff=300ms error="session initiation timed out" module=node/agent node.id=anvacpekzrc35twzg5y3r1gic
Mar 28 20:04:04 vm01 dockerd[62385]: time="2025-03-28T20:04:04.253931877Z" level=info msg="manager selected by agent for new session: { }" module=node/agent node.id=anvacpekzrc35twzg5y3r1gic
Mar 28 20:04:04 vm01 dockerd[62385]: time="2025-03-28T20:04:04.253956997Z" level=info msg="waiting 131.206559ms before registering session" module=node/agent node.id=anvacpekzrc35twzg5y3r1gic
Mar 28 20:04:09 vm01 dockerd[62385]: time="2025-03-28T20:04:09.386056098Z" level=error msg="agent: session failed" backoff=700ms error="session initiation timed out" module=node/agent node.id=anvacpekzrc35twzg5y3r1gic
Mar 28 20:04:09 vm01 dockerd[62385]: time="2025-03-28T20:04:09.386176752Z" level=info msg="manager selected by agent for new session: { }" module=node/agent node.id=anvacpekzrc35twzg5y3r1gic
Mar 28 20:04:09 vm01 dockerd[62385]: time="2025-03-28T20:04:09.386221737Z" level=info msg="waiting 335.997954ms before registering session" module=node/agent node.id=anvacpekzrc35twzg5y3r1gic

# Leader election starts
Mar 28 20:04:12 vm01 dockerd[62385]: time="2025-03-28T20:04:12.048180523Z" level=info msg="16c712ae2e74bd2c is starting a new election at term 3" module=raft node.id=anvacpekzrc35twzg5y3r1gic
Mar 28 20:04:12 vm01 dockerd[62385]: time="2025-03-28T20:04:12.048256428Z" level=info msg="16c712ae2e74bd2c became candidate at term 4" module=raft node.id=anvacpekzrc35twzg5y3r1gic
Mar 28 20:04:12 vm01 dockerd[62385]: time="2025-03-28T20:04:12.048340967Z" level=info msg="16c712ae2e74bd2c received MsgVoteResp from 16c712ae2e74bd2c at term 4" module=raft node.id=anvacpekzrc35twzg5y3r1gic
Mar 28 20:04:12 vm01 dockerd[62385]: time="2025-03-28T20:04:12.048383749Z" level=info msg="16c712ae2e74bd2c [logterm: 3, index: 4447] sent MsgVote request to 276742032427b0a5 at term 4" module=raft node.id=anvacpekzrc35twzg5y3r1gic
Mar 28 20:04:12 vm01 dockerd[62385]: time="2025-03-28T20:04:12.048400734Z" level=info msg="16c712ae2e74bd2c [logterm: 3, index: 4447] sent MsgVote request to 35b4c7f664b2a2a5 at term 4" module=raft node.id=anvacpekzrc35twzg5y3r1gic
Mar 28 20:04:12 vm01 dockerd[62385]: time="2025-03-28T20:04:12.048442795Z" level=info msg="raft.node: 16c712ae2e74bd2c lost leader 35b4c7f664b2a2a5 at term 4" module=raft node.id=anvacpekzrc35twzg5y3r1gic
Mar 28 20:04:12 vm01 dockerd[62385]: time="2025-03-28T20:04:12.060908633Z" level=info msg="16c712ae2e74bd2c received MsgVoteResp from 276742032427b0a5 at term 4" module=raft node.id=anvacpekzrc35twzg5y3r1gic
Mar 28 20:04:12 vm01 dockerd[62385]: time="2025-03-28T20:04:12.060958829Z" level=info msg="16c712ae2e74bd2c has received 2 MsgVoteResp votes and 0 vote rejections" module=raft node.id=anvacpekzrc35twzg5y3r1gic
Mar 28 20:04:12 vm01 dockerd[62385]: time="2025-03-28T20:04:12.060992877Z" level=info msg="16c712ae2e74bd2c became leader at term 4" module=raft node.id=anvacpekzrc35twzg5y3r1gic
Mar 28 20:04:12 vm01 dockerd[62385]: time="2025-03-28T20:04:12.061024123Z" level=info msg="raft.node: 16c712ae2e74bd2c elected leader 16c712ae2e74bd2c at term 4" module=raft node.id=anvacpekzrc35twzg5y3r1gic
Mar 28 20:04:12 vm01 dockerd[62385]: time="2025-03-28T20:04:12.063845782Z" level=error msg="error creating cluster object" error="name conflicts with an existing object" module=node node.id=anvacpekzrc35twzg5y3r1gic
Mar 28 20:04:12 vm01 dockerd[62385]: time="2025-03-28T20:04:12.064135115Z" level=info msg="leadership changed from not yet part of a raft cluster to anvacpekzrc35twzg5y3r1gic" module=node node.id=anvacpekzrc35twzg5y3r1gic
Mar 28 20:04:12 vm01 dockerd[62385]: time="2025-03-28T20:04:12.064179315Z" level=info msg="dispatcher starting" module=dispatcher node.id=anvacpekzrc35twzg5y3r1gic

# New leader selected, registering workers
Mar 28 20:04:14 vm01 dockerd[62385]: time="2025-03-28T20:04:14.049896735Z" level=error msg="error sending message to peer" error="rpc error: code = Canceled desc = received context error while waiting for new LB policy update: context canceled"
Mar 28 20:04:14 vm01 dockerd[62385]: time="2025-03-28T20:04:14.509581236Z" level=info msg="worker h500eow7r8ps171osflrirni9 was successfully registered" method="(*Dispatcher).register"
Mar 28 20:04:14 vm01 dockerd[62385]: time="2025-03-28T20:04:14.722966169Z" level=error msg="agent: session failed" backoff=1.5s error="session initiation timed out" module=node/agent node.id=anvacpekzrc35twzg5y3r1gic
Mar 28 20:04:14 vm01 dockerd[62385]: time="2025-03-28T20:04:14.723051577Z" level=info msg="manager selected by agent for new session: { }" module=node/agent node.id=anvacpekzrc35twzg5y3r1gic
Mar 28 20:04:14 vm01 dockerd[62385]: time="2025-03-28T20:04:14.723094256Z" level=info msg="waiting 1.375935061s before registering session" module=node/agent node.id=anvacpekzrc35twzg5y3r1gic
Mar 28 20:04:15 vm01 dockerd[62385]: time="2025-03-28T20:04:15.438963845Z" level=info msg="worker yzrumknlvx6kd383sa7o2jrop was successfully registered" method="(*Dispatcher).register"
Mar 28 20:04:15 vm01 dockerd[62385]: time="2025-03-28T20:04:15.439110266Z" level=info msg="worker teesrnys35963sbyzp07o20tk was successfully registered" method="(*Dispatcher).register"
Mar 28 20:04:15 vm01 dockerd[62385]: time="2025-03-28T20:04:15.554605882Z" level=info msg="worker 7dqcqy9pdzirqjuia96phycwp was successfully registered" method="(*Dispatcher).register"
Mar 28 20:04:16 vm01 dockerd[62385]: time="2025-03-28T20:04:16.178429928Z" level=info msg="worker anvacpekzrc35twzg5y3r1gic was successfully registered" method="(*Dispatcher).register"
Mar 28 20:04:16 vm01 dockerd[62385]: time="2025-03-28T20:04:16.798021048Z" level=info msg="worker yezg2ojyvca980iehdo0hisqk was successfully registered" method="(*Dispatcher).register"
Mar 28 20:04:23 vm01 dockerd[62385]: time="2025-03-28T20:04:23.048142630Z" level=info msg="worker z9uoam07pax0uuwn6asqbf0fp was successfully registered" method="(*Dispatcher).register"
Mar 28 20:04:49 vm01 dockerd[62385]: time="2025-03-28T20:04:49.048163483Z" level=error msg="error sending message to peer" error="rpc error: code = Unavailable desc = connection error: desc = \"transport: Error while dialing: dial tcp 10.203.96.231:2377
: i/o timeout\""
Mar 28 20:04:50 vm01 dockerd[62385]: time="2025-03-28T20:04:50.048824844Z" level=error msg="error sending message to peer" error="rpc error: code = Unavailable desc = connection error: desc = \"transport: Error while dialing: dial tcp 10.203.96.231:2377
: i/o timeout\""
Mar 28 20:04:51 vm01 dockerd[62385]: time="2025-03-28T20:04:51.048103120Z" level=error msg="error sending message to peer" error="rpc error: code = Unavailable desc = connection error: desc = \"transport: Error while dialing: dial tcp 10.203.96.231:2377
: i/o timeout\""
Mar 28 20:05:51 vm01 dockerd[62385]: time="2025-03-28T20:05:51.048178162Z" level=error msg="error sending message to peer" error="rpc error: code = Unavailable desc = connection error: desc = \"transport: Error while dialing: dial tcp 10.203.96.231:2377
: i/o timeout\""
Mar 28 20:05:52 vm01 dockerd[62385]: time="2025-03-28T20:05:52.048894504Z" level=error msg="error sending message to peer" error="rpc error: code = Unavailable desc = connection error: desc = \"transport: Error while dialing: dial tcp 10.203.96.231:2377
: i/o timeout\""
Mar 28 20:05:52 vm01 dockerd[62385]: time="2025-03-28T20:05:52.738483178Z" level=info msg="worker i5kgrnmecaqa44rqbbcw52db6 was successfully registered" method="(*Dispatcher).register"
Mar 28 20:05:53 vm01 dockerd[62385]: time="2025-03-28T20:05:53.048042546Z" level=error msg="error sending message to peer" error="rpc error: code = Unavailable desc = connection error: desc = \"transport: Error while dialing: dial tcp 10.203.96.231:2377
: i/o timeout\""

# Node vm02 rejoins cluster
Mar 28 20:05:53 vm01 dockerd[62385]: time="2025-03-28T20:05:53.220146254Z" level=info msg="Node d0c792a0b564/10.203.96.231, joined gossip cluster"
Mar 28 20:05:53 vm01 dockerd[62385]: time="2025-03-28T20:05:53.220228929Z" level=info msg="Node d0c792a0b564/10.203.96.231, is the new incarnation of the shutdown node fe2d3b9aa646/10.203.96.231"
Mar 28 20:05:53 vm01 dockerd[62385]: time="2025-03-28T20:05:53.220254815Z" level=info msg="Node d0c792a0b564/10.203.96.231, added to nodes list"
```

Figure A.9.: A screenshot of Swarm logs of deactivated manager node

```
student@vm01:~$ # SHUTDOWN MANAGER NODE VM02
student@vm01:~$ docker node ls
ID                            HOSTNAME   STATUS    AVAILABILITY   MANAGER STATUS   ENGINE VERSION
anvacpekzrc35twzg5y3r1gic *   vm01       Ready     Active         Leader           28.0.4
i5kgrnmecaqa44rqbbcw52db6     vm02       Unknown   Active         Unreachable      28.0.4
yzrumknlvx6kd383sa7o2jrop     vm03       Ready     Active         Reachable        28.0.4
z9uoam07pax0uuwn6asqbf0fp     vm04       Ready     Active                          28.0.4
yezg2ojyvca980iehdo0hisqk     vm05       Ready     Active                          28.0.4
7dqcqy9pdzirqjuia96phycwp     vm06       Ready     Active                          28.0.4
teesrnys35963sbyzp07o20tk     vm07       Ready     Active                          28.0.4
h500eow7r8ps171osflrirni9     vm08       Ready     Active                          28.0.4
student@vm01:~$ docker stack ps argo-stack
ID             NAME                                                  IMAGE                                  NODE      DESIRED STATE   CURRENT STATE            ERROR     PORTS
fw81zo450m2h   argo-stack_a-api.1                                    cadeke/argo-a-api:latest               vm04      Running         Running 44 seconds ago
ygzoyk509az7   argo-stack_a-site.1                                   cadeke/argo-a-site:latest              vm05      Running         Running 50 seconds ago
wmyze4e14f2x   argo-stack_cadvisor.7dqcqy9pdzirqjuia96phycwp         gcr.io/cadvisor/cadvisor:latest        vm06      Running         Running 51 seconds ago
xhx5eau21il3   argo-stack_cadvisor.anvacpekzrc35twzg5y3r1gic         gcr.io/cadvisor/cadvisor:latest        vm01      Running         Running 50 seconds ago
n0ibkxdf54wo   argo-stack_cadvisor.h500eow7r8ps171osflrirni9         gcr.io/cadvisor/cadvisor:latest        vm08      Running         Running 52 seconds ago
16a56b3hsqdf   argo-stack_cadvisor.i5kgrnmecaqa44rqbbcw52db6         gcr.io/cadvisor/cadvisor:latest        vm02      Shutdown        Running 5 hours ago
t0ze132zypsv   argo-stack_cadvisor.teesrnys35963sbyzp07o20tk         gcr.io/cadvisor/cadvisor:latest        vm07      Running         Running 51 seconds ago
xbdhr058k141   argo-stack_cadvisor.yezg2ojyvca980iehdo0hisqk         gcr.io/cadvisor/cadvisor:latest        vm05      Running         Running 50 seconds ago
gdya6wev7cng   argo-stack_cadvisor.yzrumknlvx6kd383sa7o2jrop         gcr.io/cadvisor/cadvisor:latest        vm03      Running         Running 51 seconds ago
sn7zcpx3vq00   argo-stack_cadvisor.z9uoam07pax0uuwn6asqbf0fp         gcr.io/cadvisor/cadvisor:latest        vm04      Running         Running 44 seconds ago
auw5g70l4n93   argo-stack_debug.1                                    alpine:latest                          vm04      Running         Running 44 seconds ago
er2uf4wmcv3e   argo-stack_gitlab-runner.1                            gitlab/gitlab-runner:latest            vm04      Running         Running 44 seconds ago
lkpxww8d1o62   argo-stack_gitlab-runner.2                            gitlab/gitlab-runner:latest            vm05      Running         Running 50 seconds ago
wxtpcmcqhr1a   argo-stack_gitlab.1                                   gitlab/gitlab-ce:latest                vm06      Running         Running 51 seconds ago
r4hhw2fa504n   argo-stack_grafana.1                                  grafana/grafana:latest                 vm06      Running         Running 51 seconds ago
n1m95ki6x3pp   argo-stack_memcached.1                                memcached:latest                       vm07      Running         Running 51 seconds ago
vpircq7aan9s   argo-stack_node-exporter.7dqcqy9pdzirqjuia96phycwp    prom/node-exporter:latest              vm06      Running         Running 51 seconds ago
dfwrue8nlthh   argo-stack_node-exporter.anvacpekzrc35twzg5y3r1gic    prom/node-exporter:latest              vm01      Running         Running 50 seconds ago
rqta476s8ad5   argo-stack_node-exporter.h500eow7r8ps171osflrirni9    prom/node-exporter:latest              vm08      Running         Running 52 seconds ago
l318cgnxqz4o   argo-stack_node-exporter.i5kgrnmecaqa44rqbbcw52db6    prom/node-exporter:latest              vm02      Shutdown        Running 5 hours ago
rzmu7rwdox0z   argo-stack_node-exporter.teesrnys35963sbyzp07o20tk    prom/node-exporter:latest              vm07      Running         Running 51 seconds ago
co6p5gjnyly8   argo-stack_node-exporter.yezg2ojyvca980iehdo0hisqk    prom/node-exporter:latest              vm05      Running         Running 50 seconds ago
m7175v07w7vb   argo-stack_node-exporter.yzrumknlvx6kd383sa7o2jrop    prom/node-exporter:latest              vm03      Running         Running 51 seconds ago
uqx3ijjtefz7   argo-stack_node-exporter.z9uoam07pax0uuwn6asqbf0fp    prom/node-exporter:latest              vm04      Running         Running 44 seconds ago
xsv9gbsg1bsv   argo-stack_ot-app.1                                   cadeke/argo-ot-app:latest              vm07      Running         Running 51 seconds ago
xkyqoa5pmr0k   argo-stack_postgres-exporter.1                        wrouesnel/postgres_exporter:latest     vm04      Running         Running 44 seconds ago
dorsp9uem4zg   argo-stack_postgres.1                                 postgres:latest                        vm05      Running         Running 50 seconds ago
xnlo2rp4plrc   argo-stack_prometheus.1                               prom/prometheus:latest                 vm07      Running         Running 51 seconds ago
vtbxwhsztnsc   argo-stack_query-api.1                                cadeke/argo-q-api:latest               vm06      Running         Running 51 seconds ago
h7dyyajca67l   argo-stack_query-site.1                               cadeke/argo-q-site:latest              vm07      Running         Running 51 seconds ago
student@vm01:~$ # START MANAGER NODE VM02 AGAIN
student@vm01:~$ docker node ls
ID                            HOSTNAME   STATUS    AVAILABILITY   MANAGER STATUS   ENGINE VERSION
anvacpekzrc35twzg5y3r1gic *   vm01       Ready     Active         Leader           28.0.4
i5kgrnmecaqa44rqbbcw52db6     vm02       Ready     Active         Reachable        28.0.4
yzrumknlvx6kd383sa7o2jrop     vm03       Ready     Active         Reachable        28.0.4
z9uoam07pax0uuwn6asqbf0fp     vm04       Ready     Active                          28.0.4
yezg2ojyvca980iehdo0hisqk     vm05       Ready     Active                          28.0.4
7dqcqy9pdzirqjuia96phycwp     vm06       Ready     Active                          28.0.4
teesrnys35963sbyzp07o20tk     vm07       Ready     Active                          28.0.4
h500eow7r8ps171osflrirni9     vm08       Ready     Active                          28.0.4
student@vm01:~$
```

Figure A.10.: A screenshot of Swarm service changes due to deactivated manager node

```
# STATE BEFORE UPDATING
student@vm01:~/stacks$ docker stack ps argo | grep "ot-app"
wvw93duwg3hf   argo_ot-app.1                              cadeke/argo-ot-app:v1.0          vm04      Running        Running 6 minutes ago
cvbngwlqz7vd   argo_ot-app.2                              cadeke/argo-ot-app:v1.0          vm08      Running        Running 6 minutes ago
x4wxxafbrxl1   argo_ot-app.3                              cadeke/argo-ot-app:v1.0          vm07      Running        Running 6 minutes ago
znjq6ty1zl5g   argo_ot-app.4                              cadeke/argo-ot-app:v1.0          vm05      Running        Running 6 minutes ago

# SUCCESSFULL UPDATE TO V1.1
student@vm01:~/stacks$ docker service update --image cadeke/argo-ot-app:v1.1 argo_ot-app
argo_ot-app
overall progress: 4 out of 4 tasks
1/4: running   [==================================================>]
2/4: running   [==================================================>]
3/4: running   [==================================================>]
4/4: running   [==================================================>]
verify: Service argo_ot-app converged
student@vm01:~/stacks$ docker stack ps argo | grep "ot-app"
mbcttqn0oo5p   argo_ot-app.1                              cadeke/argo-ot-app:v1.1          vm08      Running        Running 12 seconds ago
wvw93duwg3hf    \_ argo_ot-app.1                          cadeke/argo-ot-app:v1.0          vm04      Shutdown       Shutdown 8 seconds ago
mfa5qabtohww   argo_ot-app.2                              cadeke/argo-ot-app:v1.1          vm05      Running        Running 26 seconds ago
cvbngwlqz7vd    \_ argo_ot-app.2                          cadeke/argo-ot-app:v1.0          vm08      Shutdown       Shutdown 23 seconds ago
7l7lvz8tho4h   argo_ot-app.3                              cadeke/argo-ot-app:v1.1          vm06      Running        Running 56 seconds ago
x4wxxafbrxl1    \_ argo_ot-app.3                          cadeke/argo-ot-app:v1.0          vm07      Shutdown       Shutdown 52 seconds ago
p3yic6egeqby   argo_ot-app.4                              cadeke/argo-ot-app:v1.1          vm07      Running        Running 41 seconds ago
znjq6ty1zl5g    \_ argo_ot-app.4                          cadeke/argo-ot-app:v1.0          vm05      Shutdown       Shutdown 37 seconds ago

# FAILED UPDATE TO V1.2, ROLLBACK TO V1.1
student@vm01:~/stacks$ docker service update --image cadeke/argo-ot-app:v1.2 argo_ot-app
argo_ot-app
overall progress: rolling back update: 4 out of 4 tasks
1/4: running   [==================================================>]
2/4: running   [==================================================>]
3/4: running   [==================================================>]
4/4: running   [==================================================>]
rollback: update rolled back due to failure or early termination of task o809e01r2orxincsatq9w0237
verify: Service argo_ot-app converged
student@vm01:~/stacks$ docker stack ps argo | grep "ot-app"
mbcttqn0oo5p   argo_ot-app.1                              cadeke/argo-ot-app:v1.1          vm08      Running        Running about a minute ago
wvw93duwg3hf    \_ argo_ot-app.1                          cadeke/argo-ot-app:v1.0          vm04      Shutdown       Shutdown about a minute ago
mfa5qabtohww   argo_ot-app.2                              cadeke/argo-ot-app:v1.1          vm05      Running        Running 2 minutes ago
cvbngwlqz7vd    \_ argo_ot-app.2                          cadeke/argo-ot-app:v1.0          vm08      Shutdown       Shutdown 2 minutes ago
noned99kric4   argo_ot-app.3                              cadeke/argo-ot-app:v1.1          vm04      Running        Running 8 seconds ago
jmmim9b90qxe    \_ argo_ot-app.3                          cadeke/argo-ot-app:v1.2          vm04      Shutdown       Shutdown 10 seconds ago
o809e01r2orx    \_ argo_ot-app.3                          cadeke/argo-ot-app:v1.2          vm04      Shutdown       Failed 12 seconds ago         "task: non-zero exit (1)"
7l7lvz8tho4h    \_ argo_ot-app.3                          cadeke/argo-ot-app:v1.1          vm06      Shutdown       Shutdown 10 seconds ago
x4wxxafbrxl1    \_ argo_ot-app.3                          cadeke/argo-ot-app:v1.0          vm07      Shutdown       Shutdown 2 minutes ago
p3yic6egeqby   argo_ot-app.4                              cadeke/argo-ot-app:v1.1          vm07      Running        Running 2 minutes ago
znjq6ty1zl5g    \_ argo_ot-app.4                          cadeke/argo-ot-app:v1.0          vm05      Shutdown       Shutdown 2 minutes ago
```

Figure A.11.: A screenshot showing rolling updates in Swarm cluster

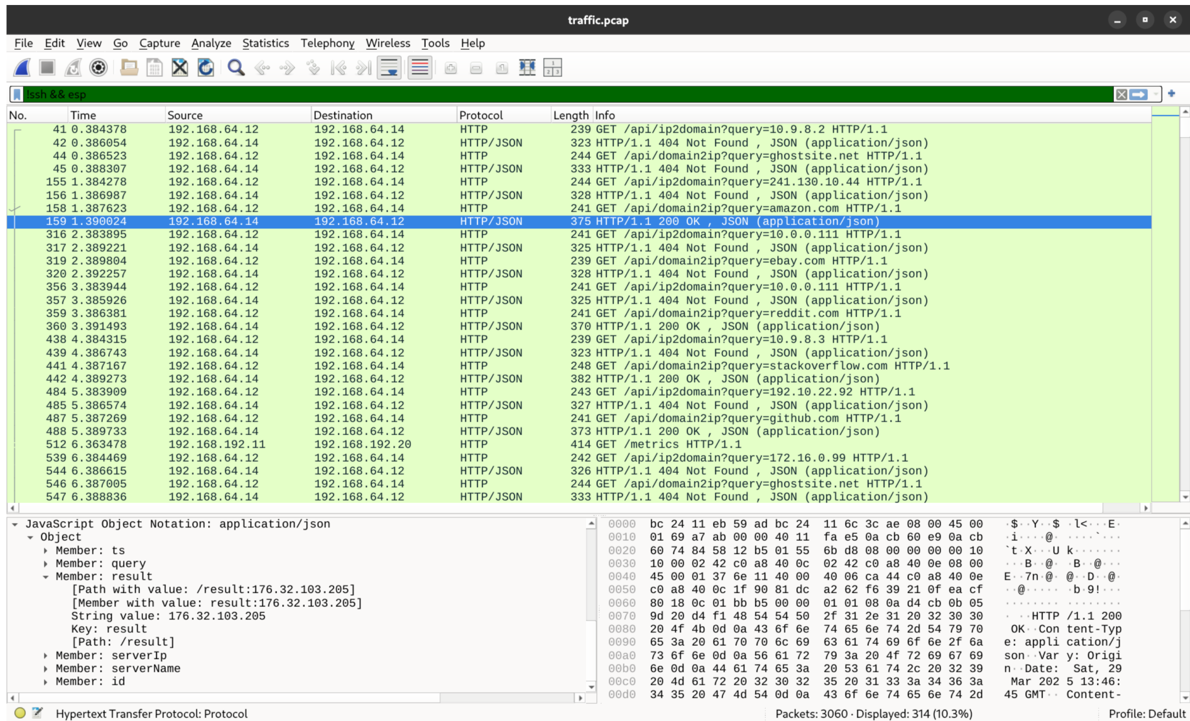

Figure A.12.: A screenshot of a WireShark capture of plain text traffic on Swarm node

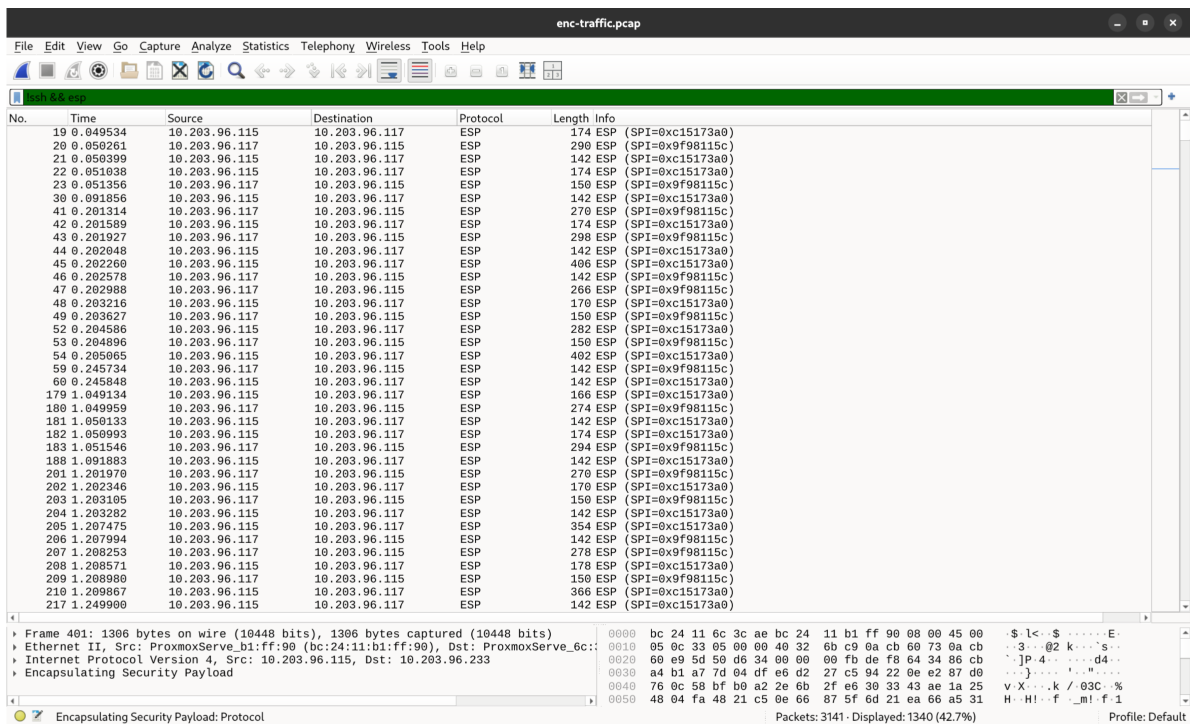

Figure A.13.: A screenshot of a WireShark capture of encrypted traffic on Swarm node

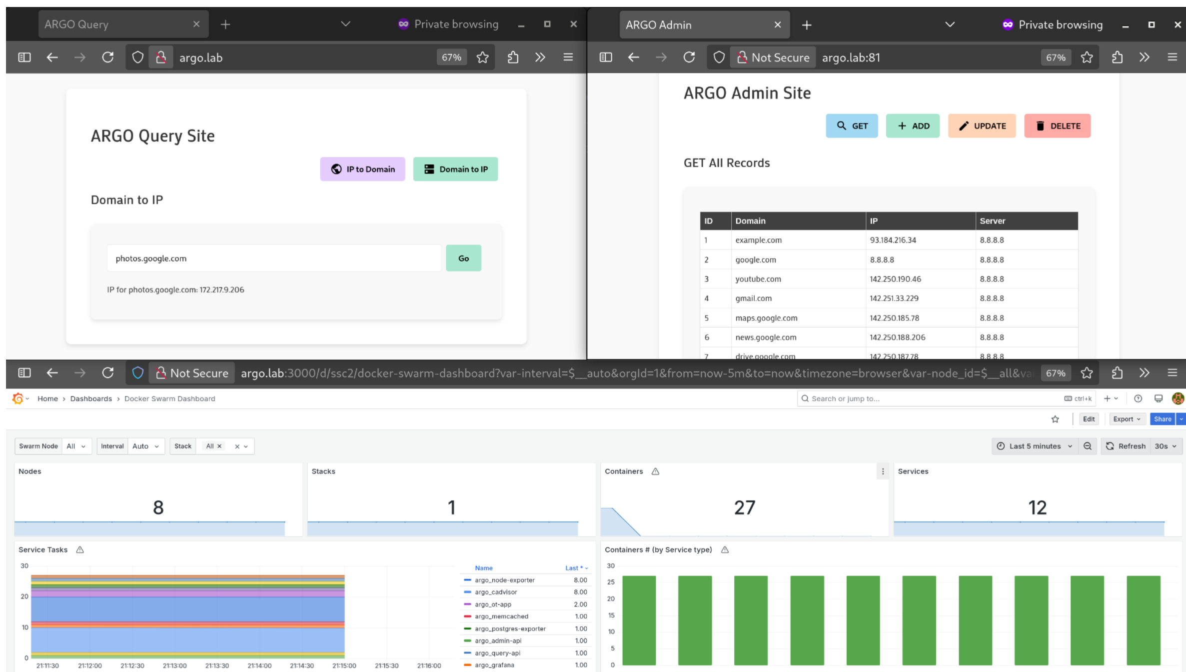

Figure A.14.: An overview of exposed services, accessible through Swarm routing mesh

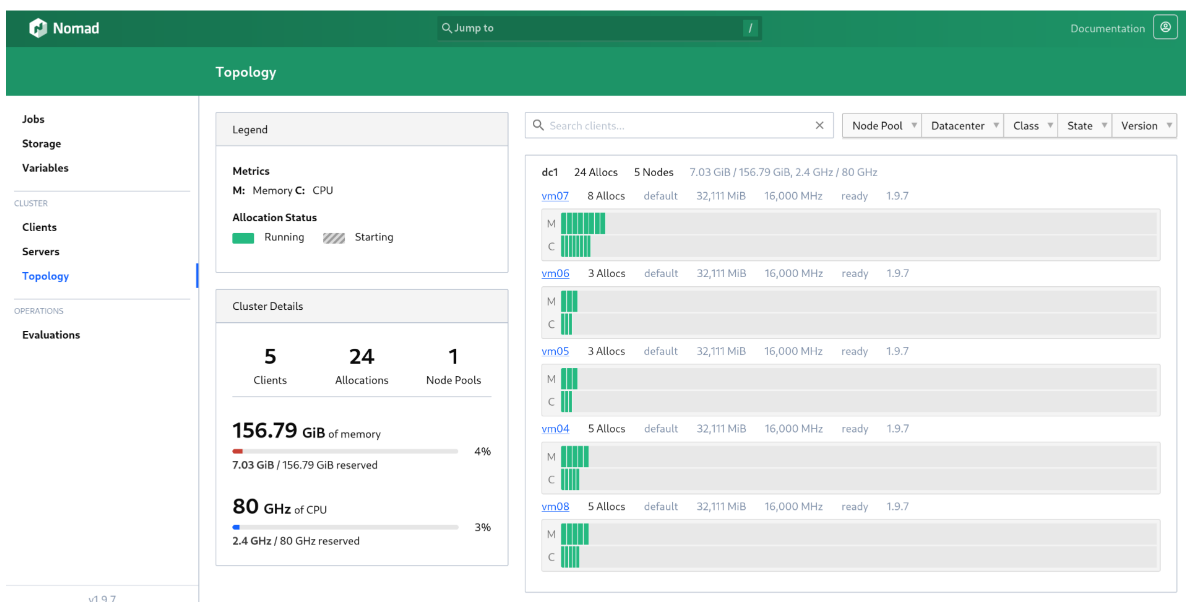

Figure A.15.: A topology overview of Nomad cluster

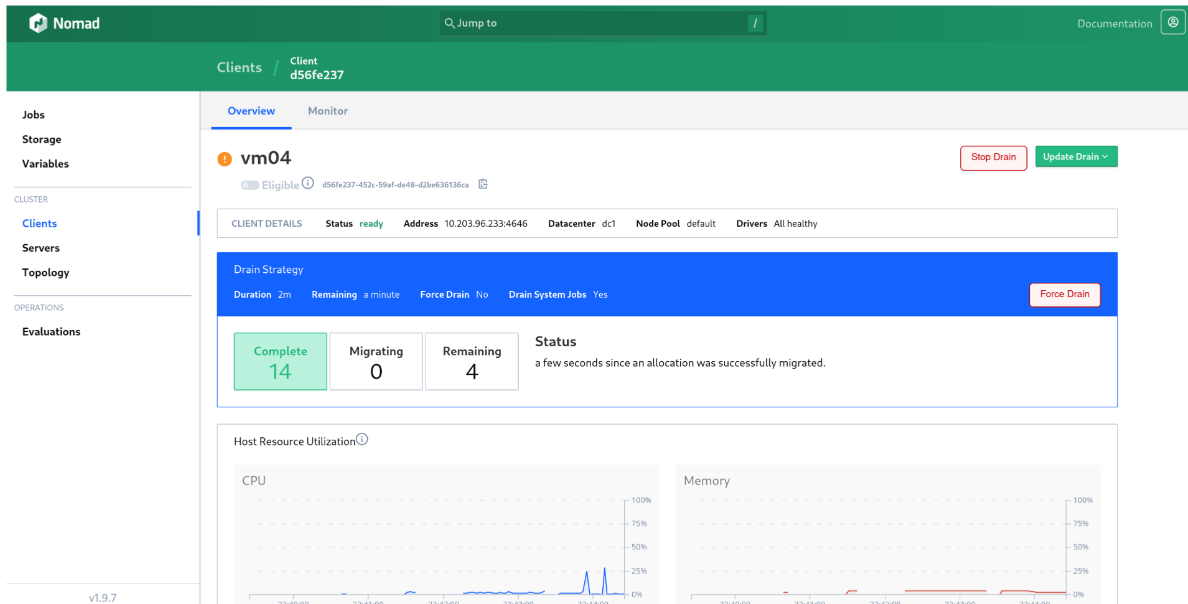

Figure A.16.: A screenshot of Nomad UI showing draining of node vm04

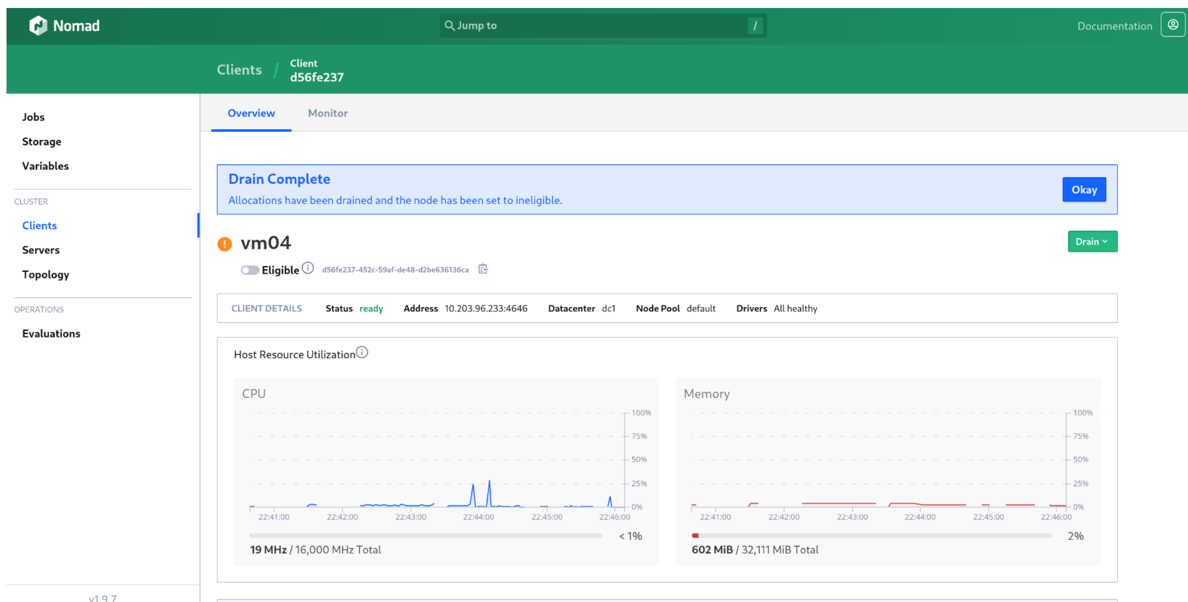

Figure A.17.: A screenshot of Nomad UI showing drained node vm04

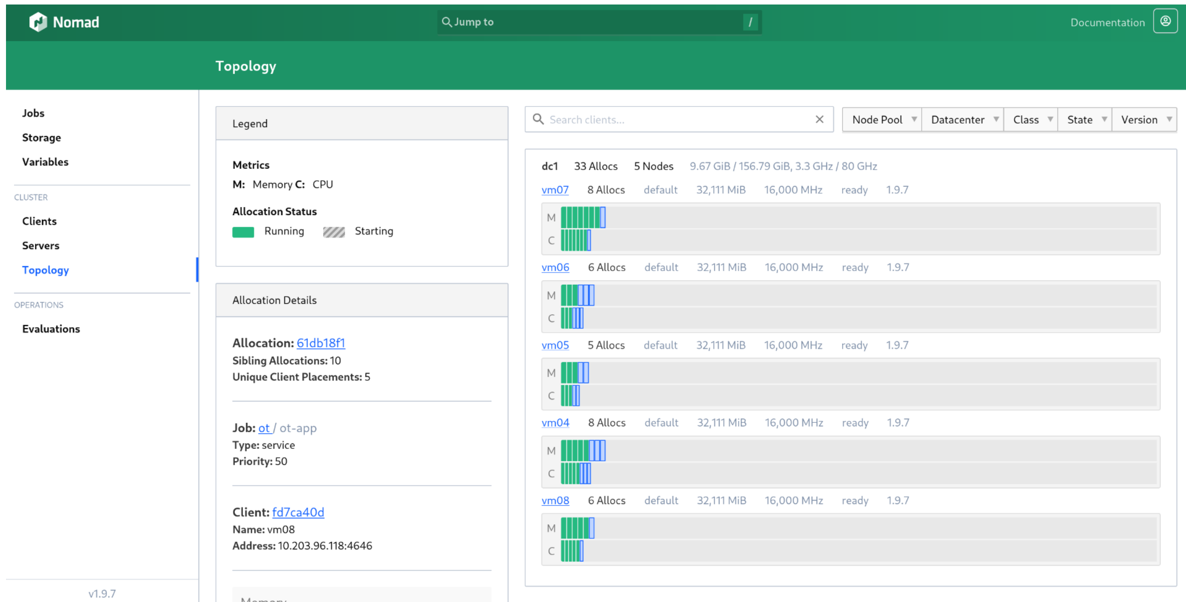

Figure A.18.: A topology overview of Nomad cluster after scaling

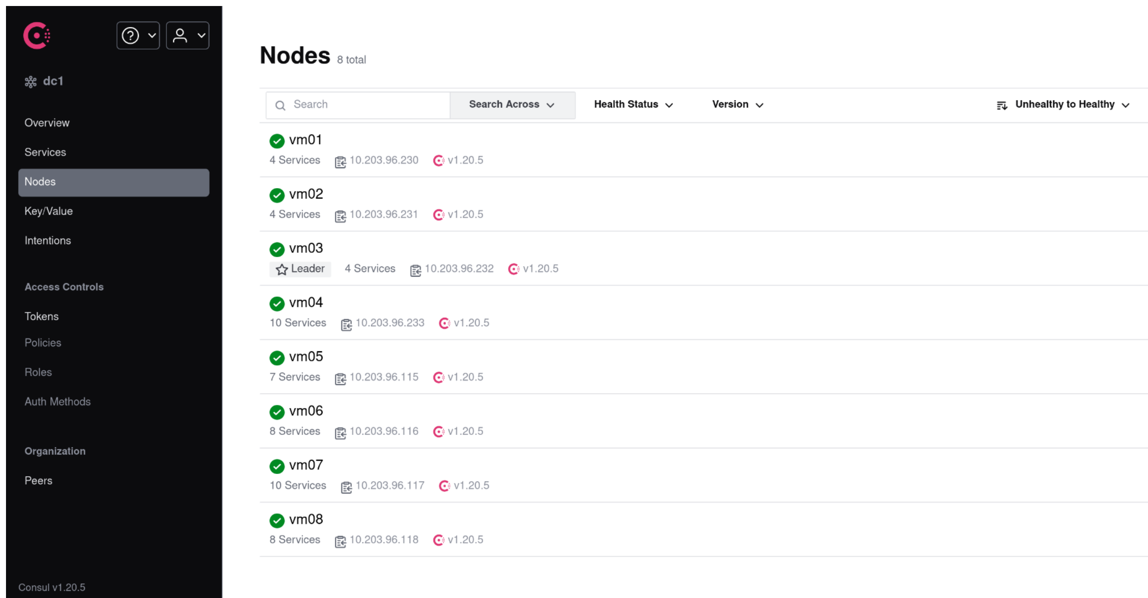

Figure A.19.: An overview of Consul nodes

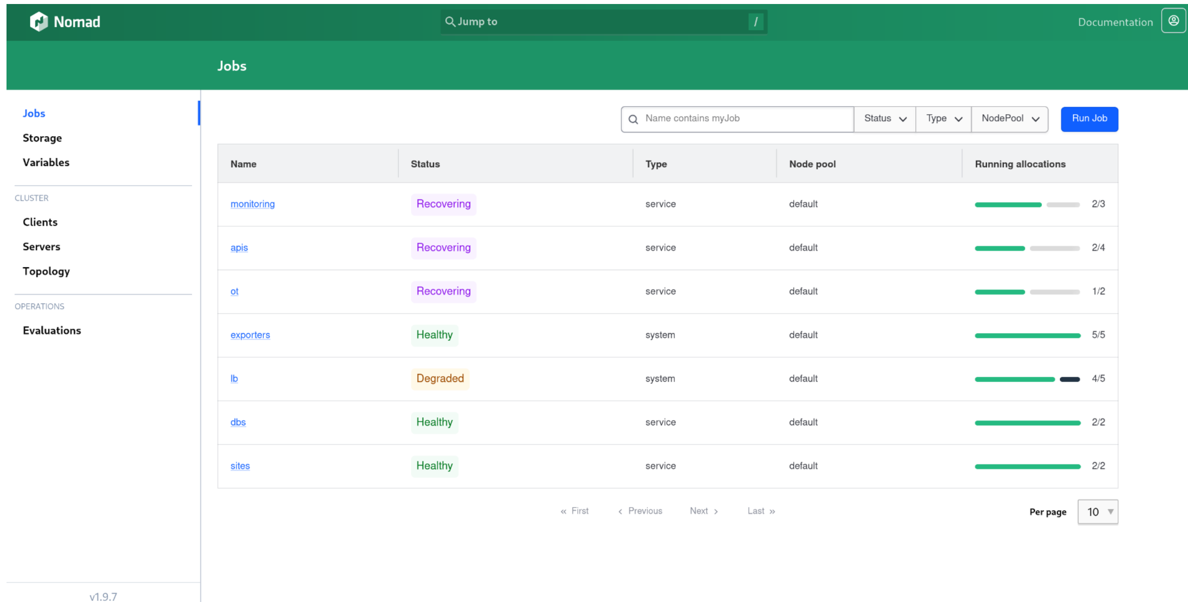

Figure A.20.: An overview of deployed jobs, indicating an impacted Nomad cluster due to client failure

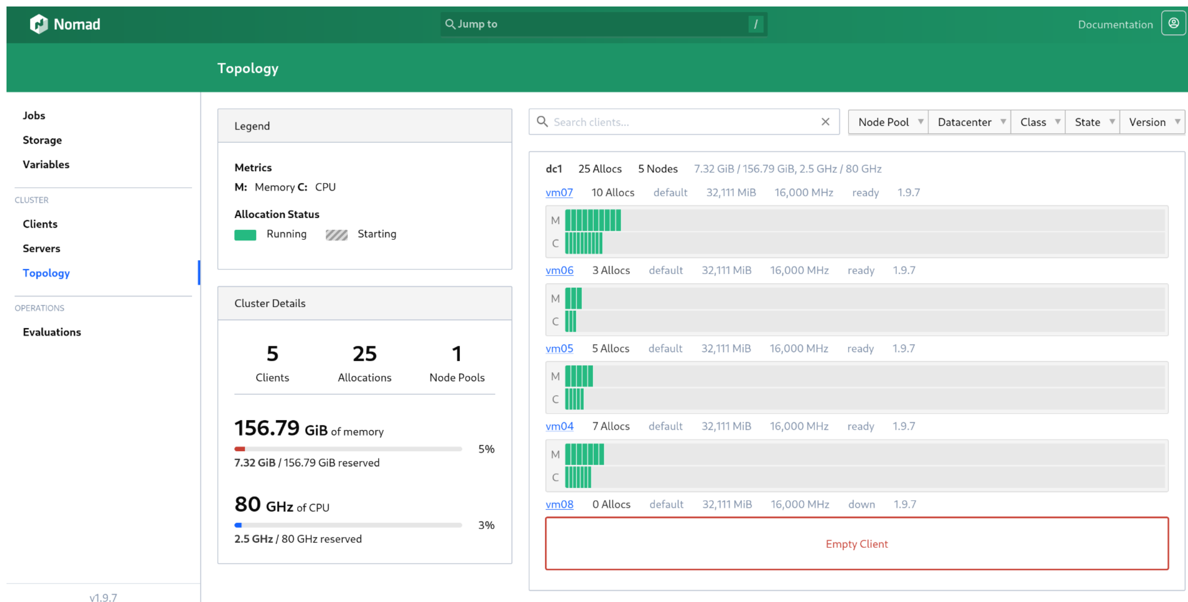

Figure A.21.: A topology overview of Nomad after rescheduling services as a result of client failure

Nomad | Jump to | Documentation

Servers

Jobs
Storage
Variables

CLUSTER
Clients
Servers
Topology

OPERATIONS
Evaluations

Name	Status	Leader	Address	port	Datacenter	Version
vm02.global	Alive	True	10.203.96.231	4648	dc1	1.9.7
vm01.global	Left	False	10.203.202.220	4648	dc1	1.9.7
vm03.global	Alive	False	10.203.96.232	4648	dc1	1.9.7

1–3 of 3

v1.9.7

Figure A.22.: An overview of Nomad server nodes after server failure

```
# Server vm01 leaves the cluster
Apr 14 19:15:42 vm03 nomad[48108]:    2025-04-14T19:15:42.268Z [ERROR] worker: failed to dequeue evaluation: worker_id=34e4f0f3-4c43-5778-5734-7ef7b2be22c0 error="rpc error: eval broker disabled"
Apr 14 19:15:42 vm03 nomad[48108]:    2025-04-14T19:15:42.268Z [ERROR] worker: failed to dequeue evaluation: worker_id=ff4fcbd1-2f62-177b-b66c-235afd5f2614 error="rpc error: eval broker disabled"
Apr 14 19:15:43 vm03 nomad[48108]:    2025-04-14T19:15:43.734Z [WARN]  nomad.raft: heartbeat timeout reached, starting election: last-leader-addr=10.203.202.220:4647 last-leader-id=97cc9d0d-1ad5-41b8-4bdb-45fc729d71e3

# Leader election starts
Apr 14 19:15:43 vm03 nomad[48108]:    2025-04-14T19:15:43.734Z [INFO]  nomad.raft: entering candidate state: node="Node at 10.203.96.232:4647 [Candidate]" term=11
Apr 14 19:15:43 vm03 nomad[48108]:    2025-04-14T19:15:43.734Z [ERROR] nomad.raft: failed to make requestVote RPC: target="{Voter 97cc9d0d-1ad5-41b8-4bdb-45fc729d71e3 10.203.202.220:4647}" error=EOF term=11
Apr 14 19:15:43 vm03 nomad[48108]:    2025-04-14T19:15:43.735Z [INFO]  nomad.raft: pre-vote campaign failed, waiting for election timeout: term=10 tally=1 refused=2 votesNeeded=2
Apr 14 19:15:43 vm03 nomad[48108]:    2025-04-14T19:15:43.769Z [INFO]  nomad.raft: entering follower state: follower="Node at 10.203.96.232:4647 [Follower]" leader-address= leader-id=
Apr 14 19:15:52 vm03 nomad[48108]:    2025-04-14T19:15:52.331Z [ERROR] worker: failed to dequeue evaluation: worker_id=7775bf98-df48-4ef6-29de-d846e380cb2a error="rpc error: failed to get conn: dial tcp 10.203.202.220:4647: i/o timeout"
Apr 14 19:15:52 vm03 nomad[48108]:    2025-04-14T19:15:52.331Z [ERROR] worker: failed to dequeue evaluation: worker_id=34e4f0f3-4c43-5778-5734-7ef7b2be22c0 error="rpc error: failed to get conn: rpc error: lead thread didn't get connection"

# Server vm01 marked as failed
Apr 14 19:15:53 vm03 nomad[48108]:    2025-04-14T19:15:53.413Z [INFO]  nomad: memberlist: Suspect vm01.global has failed, no acks received
Apr 14 19:16:03 vm03 nomad[48108]:    2025-04-14T19:16:03.412Z [INFO]  nomad: memberlist: Suspect vm01.global has failed, no acks received
Apr 14 19:16:13 vm03 nomad[48108]:    2025-04-14T19:16:13.412Z [INFO]  nomad: memberlist: Suspect vm01.global has failed, no acks received
Apr 14 19:16:23 vm03 nomad[48108]:    2025-04-14T19:16:23.414Z [INFO]  nomad: memberlist: Marking vm01.global as failed, suspect timeout reached (0 peer confirmations)
Apr 14 19:16:23 vm03 nomad[48108]:    2025-04-14T19:16:23.414Z [INFO]  nomad: serf: EventMemberFailed: vm01.global 10.203.202.220
Apr 14 19:16:23 vm03 nomad[48108]:    2025-04-14T19:16:23.414Z [INFO]  nomad: removing server: server="vm01.global (Addr: 10.203.202.220:4647) (DC: dc1)"
Apr 14 19:16:24 vm03 nomad[48108]:    2025-04-14T19:16:24.811Z [INFO]  nomad: serf: EventMemberLeave (forced): vm01.global 10.203.202.220
Apr 14 19:16:24 vm03 nomad[48108]:    2025-04-14T19:16:24.811Z [INFO]  nomad: removing server: server="vm01.global (Addr: 10.203.202.220:4647) (DC: dc1)"
Apr 14 19:16:28 vm03 nomad[48108]:    2025-04-14T19:16:28.412Z [INFO]  nomad: memberlist: Suspect vm01.global has failed, no acks received

# Server vm01 rejoins cluster
Apr 14 19:17:34 vm03 nomad[48108]:    2025-04-14T19:17:34.681Z [INFO]  nomad: serf: EventMemberJoin: vm01.global 10.203.202.220
Apr 14 19:17:34 vm03 nomad[48108]:    2025-04-14T19:17:34.681Z [INFO]  nomad: adding server: server="vm01.global (Addr: 10.203.202.220:4647) (DC: dc1)"
```

Figure A.23.: A screenshot of Nomad logs during server failure

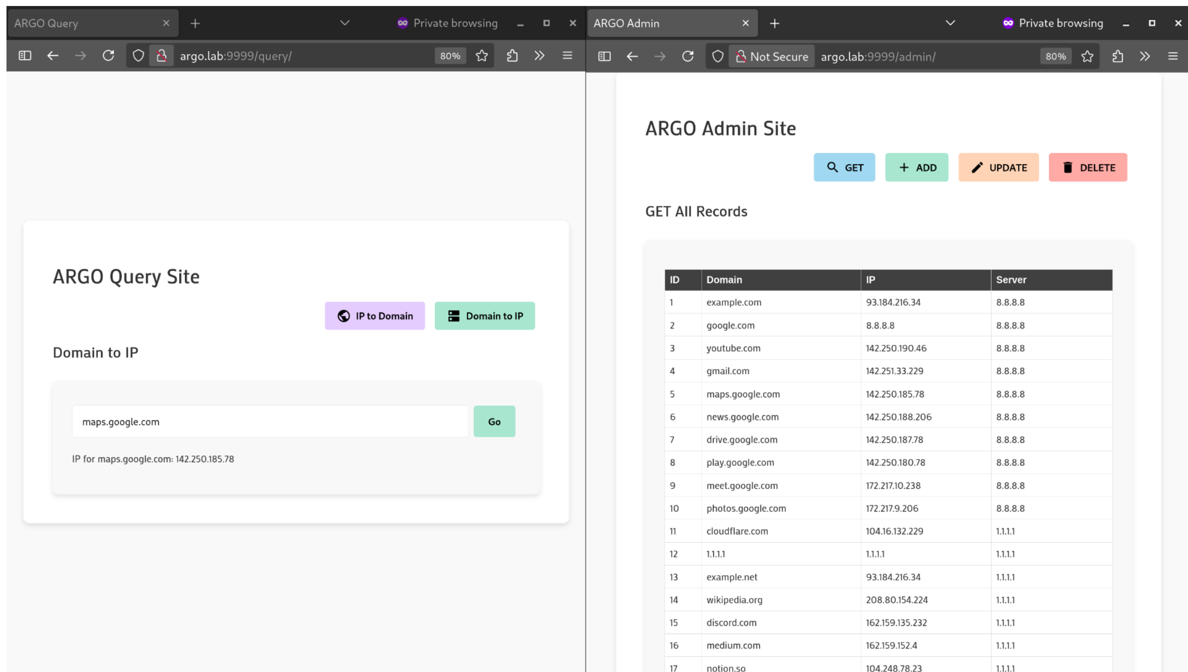

Figure A.24.: An overview of exposed services, accessible through load balancer on Nomad

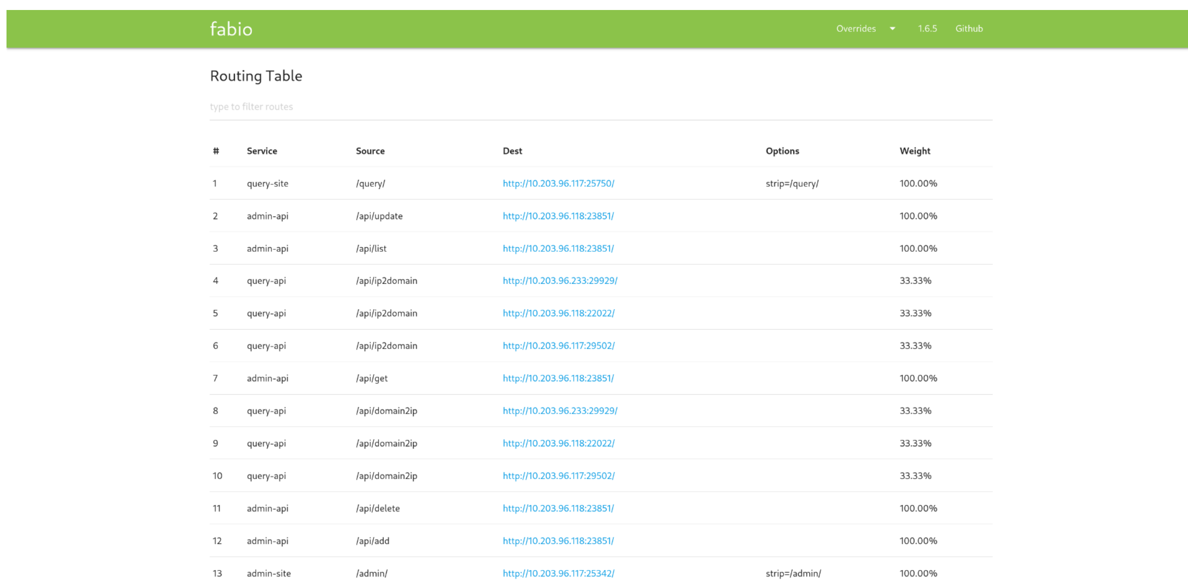

Figure A.25.: An overview of exposed routes via Fabio

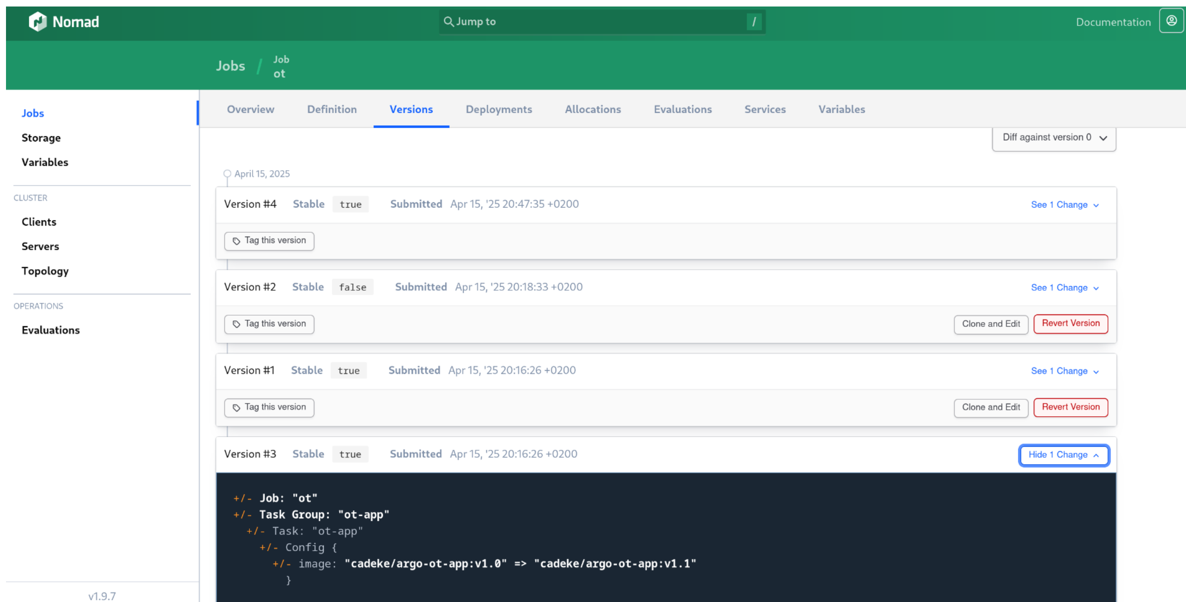

Figure A.26.: A screenshot of Nomad UI showing version history of OT application deployments

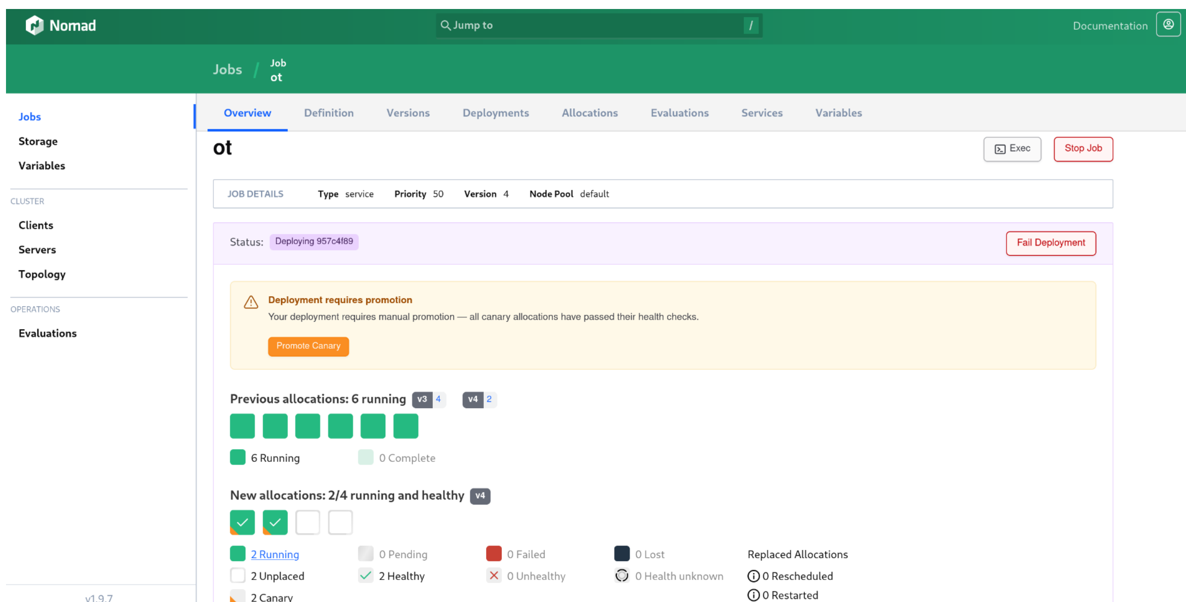

Figure A.27.: A screenshot of Nomad UI showing approval prompt for canary promotion of OT application

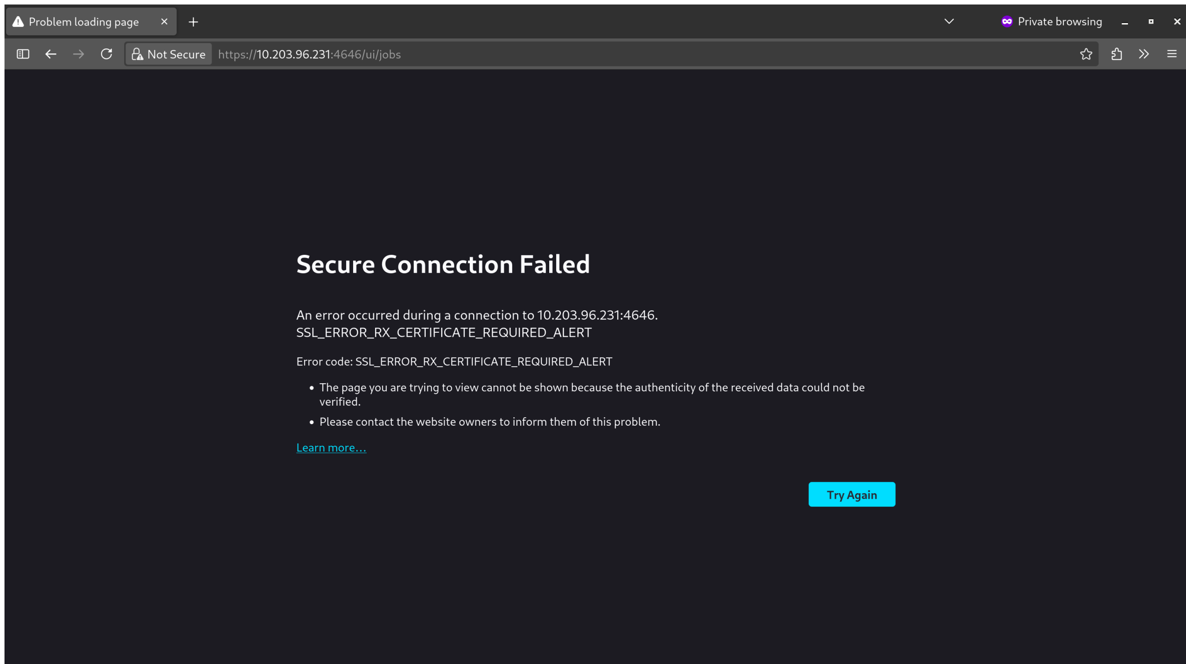

Figure A.28.: A screenshot showing a browser error, indicating client certificate is required

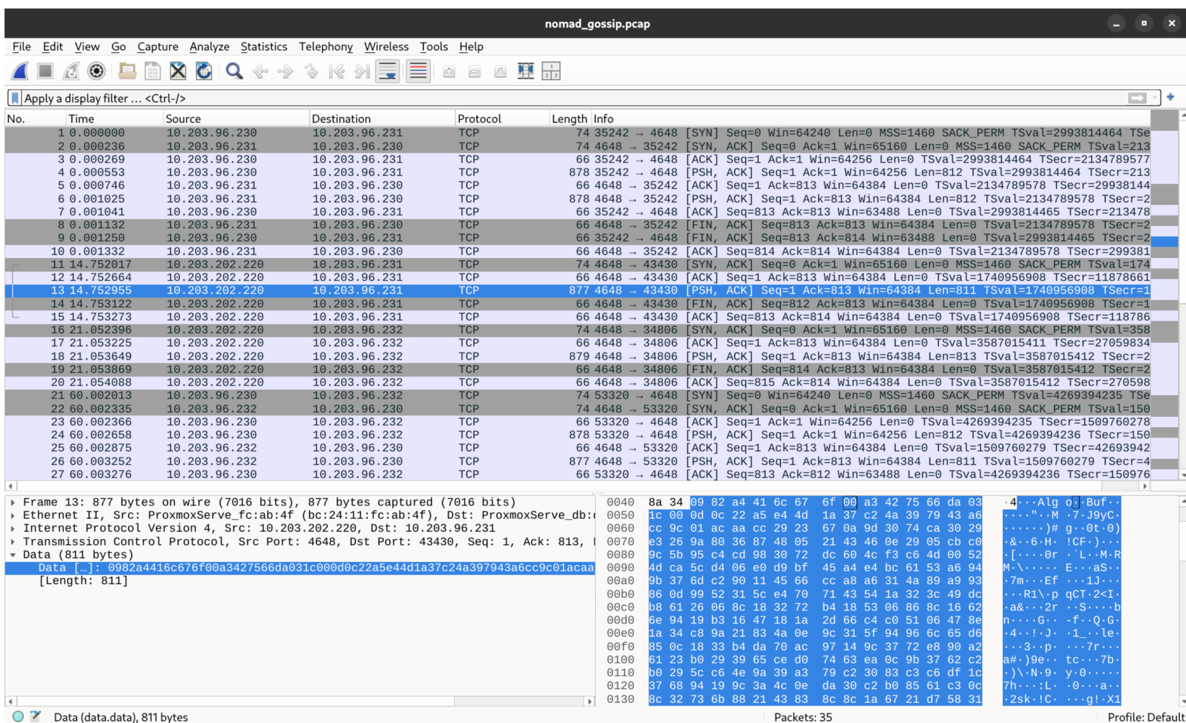

Figure A.29.: A screenshot of a WireShark capture of plain-text Nomad gossip traffic

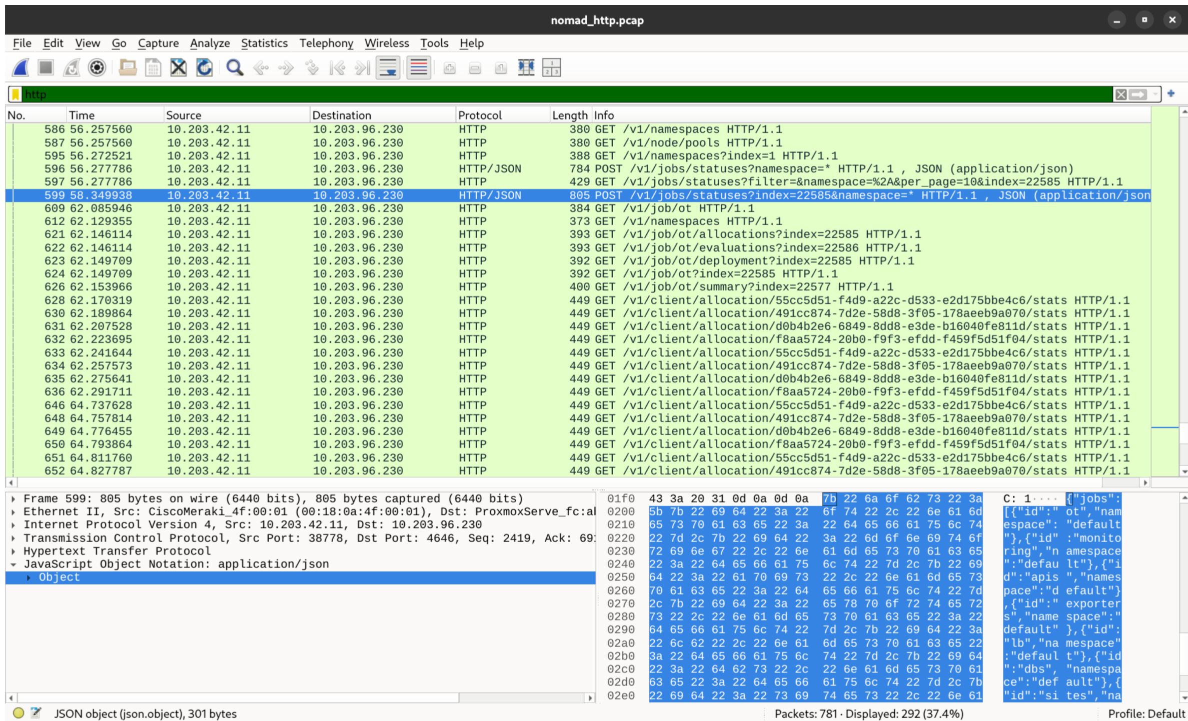

Figure A.30.: A screenshot of a WireShark capture of plain-text Nomad HTTP traffic

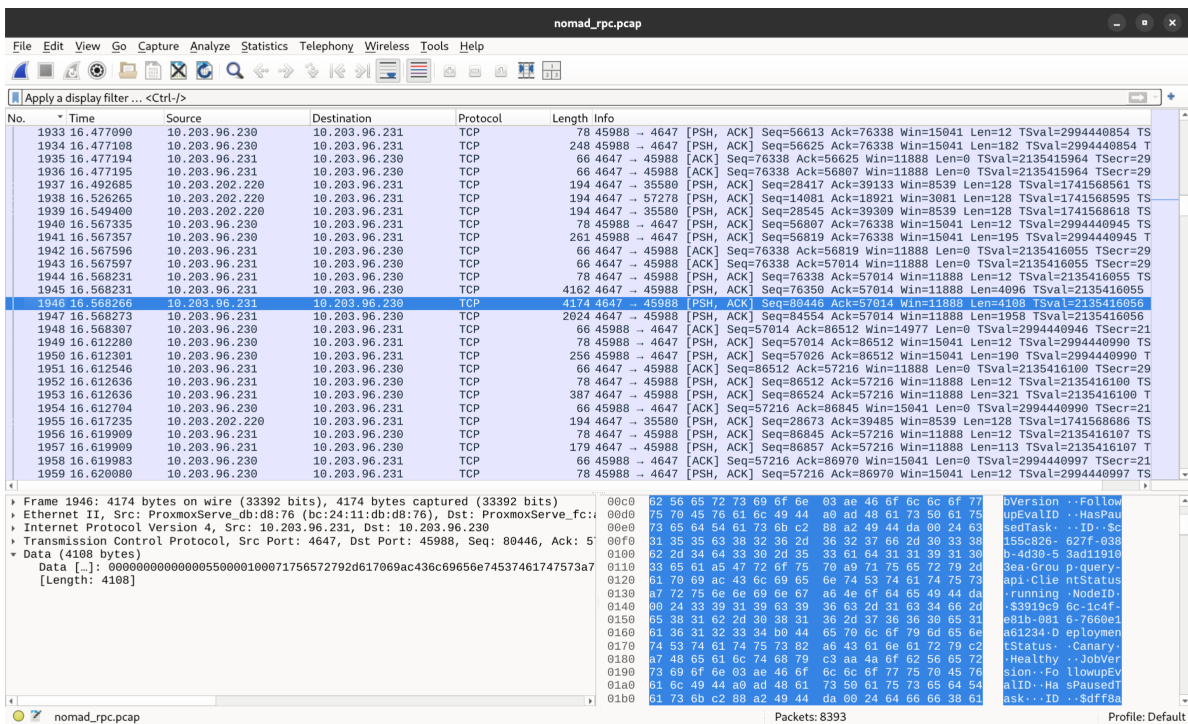

Figure A.31.: A screenshot of a WireShark capture of plain-text Nomad RPC traffic

Figure A.32.: A screenshot of a WireShark capture of encrypted Nomad gossip traffic

Figure A.33.: A screenshot of a WireShark capture of encrypted Nomad HTTP traffic

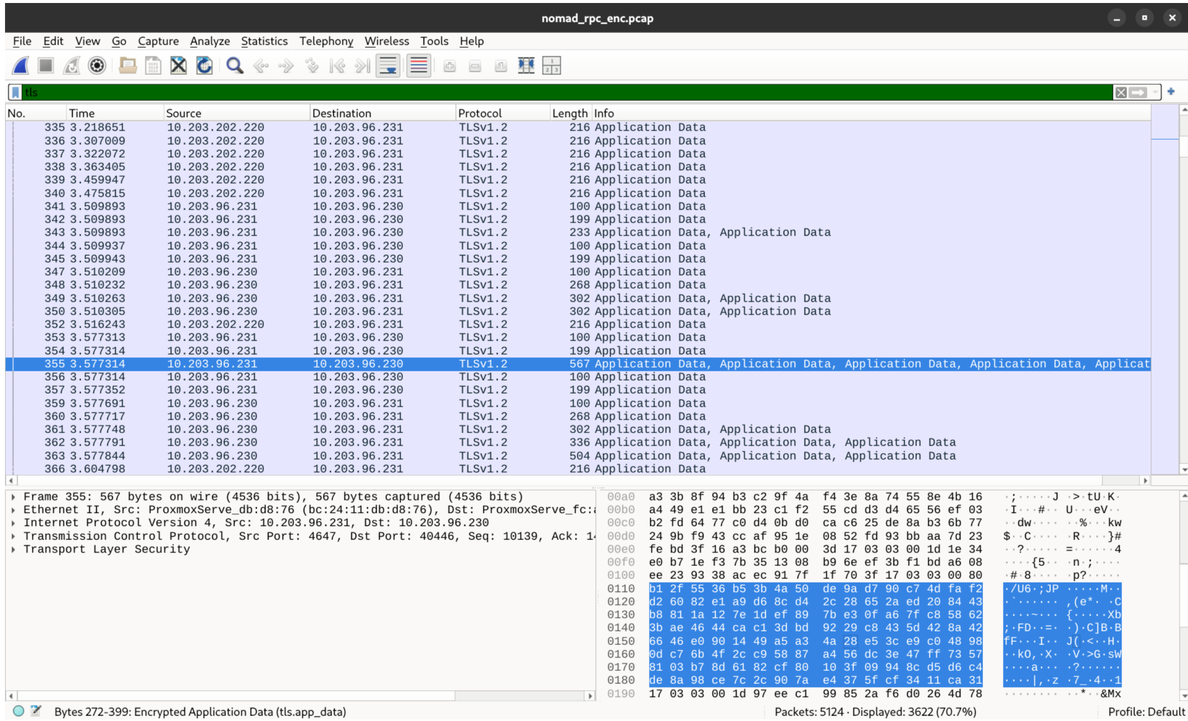

Figure A.34.: A screenshot of a WireShark capture of encrypted Nomad RPC traffic

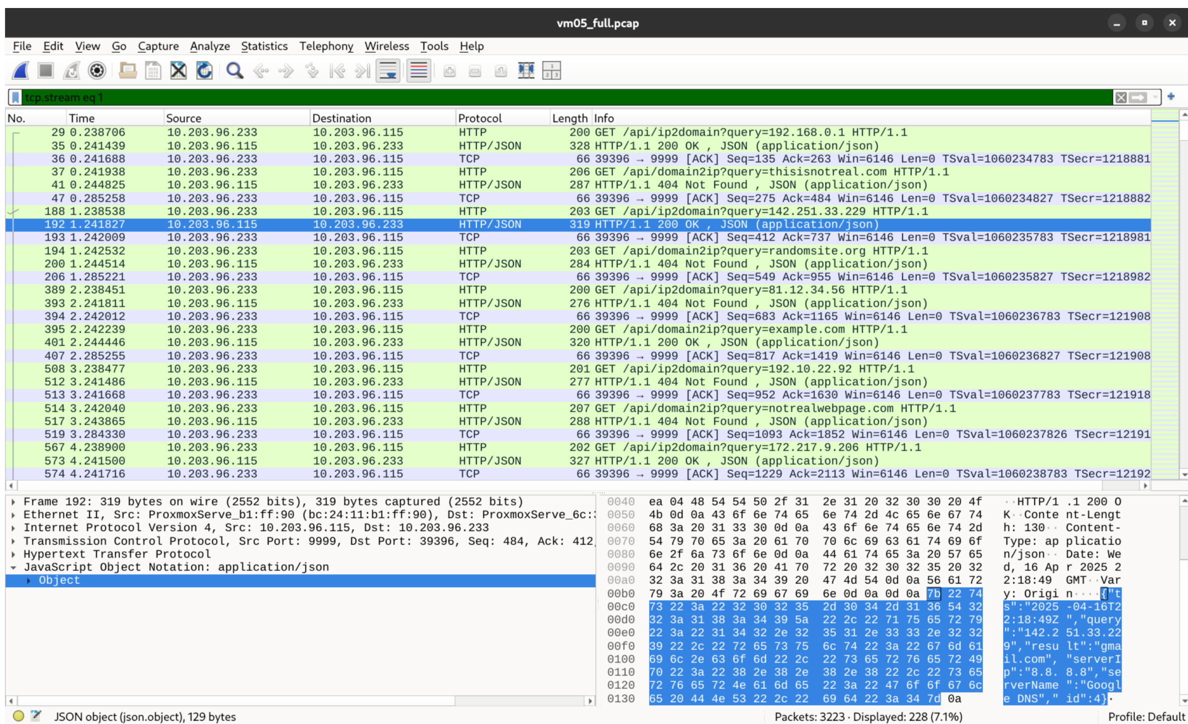

Figure A.35.: A screenshot of a WireShark capture of Nomad node, showing plain-text HTTP requests

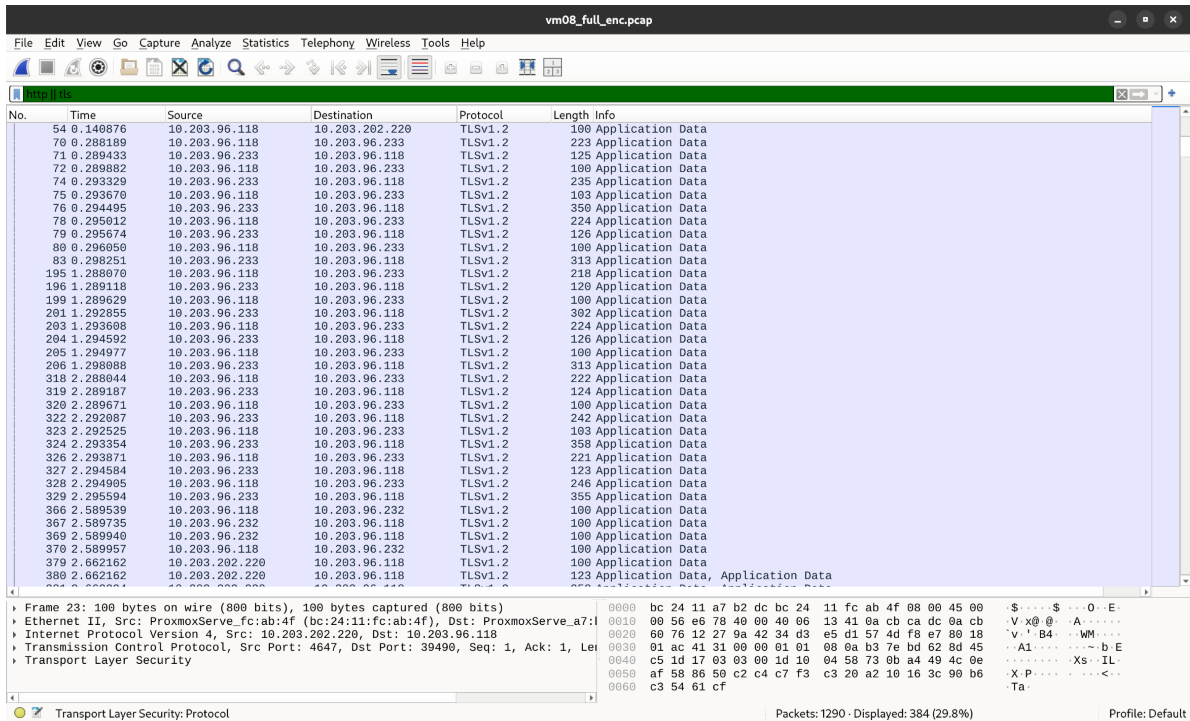

Figure A.36.: A screenshot of a WireShark capture of Nomad node, showing encrypted communications

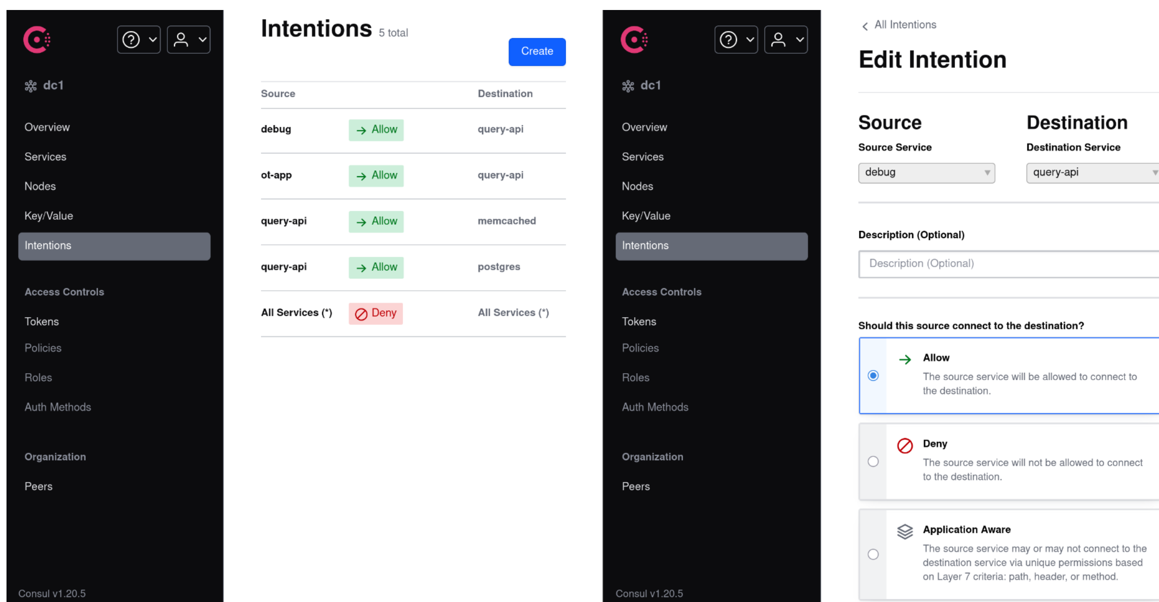

Figure A.37.: A screenshot showing both the intentions and the configuration menu in Consul UI

```
1  > ansible all -i inventory/hosts.yml -m shell -a "ip a | grep 'inet 10'"
2  vm01 | CHANGED | rc=0 »
3      inet 10.203.96.230/24 brd 10.203.96.255 scope global ens18
4      inet 10.203.202.220/24 brd 10.203.202.255 scope global ens19
5  vm02 | CHANGED | rc=0 »
6      inet 10.203.96.231/24 brd 10.203.96.255 scope global ens18
7  vm03 | CHANGED | rc=0 »
8      inet 10.203.96.232/24 brd 10.203.96.255 scope global ens18
9  vm04 | CHANGED | rc=0 »
10     inet 10.203.96.233/24 brd 10.203.96.255 scope global ens18
11 vm05 | CHANGED | rc=0 »
12     inet 10.203.96.115/24 brd 10.203.96.255 scope global ens18
13 vm06 | CHANGED | rc=0 »
14     inet 10.203.96.116/24 brd 10.203.96.255 scope global ens18
15 vm07 | CHANGED | rc=0 »
16     inet 10.203.96.117/24 brd 10.203.96.255 scope global ens18
17 vm08 | CHANGED | rc=0 »
18     inet 10.203.96.118/24 brd 10.203.96.255 scope global ens18
```

Listing 31: A snippet of an Ansible command showing the IP configuration of hosts

```
1  student@vm01:~> sudo gluster pool list
2  UUID                                    Hostname        State
3  a60ec945-120d-4782-b88e-195ebab64044    vm08            Connected
4  a44fe56e-63e8-4f61-920f-279da9bf151e    vm07            Connected
5  789ebbc8-a25e-49a7-990a-485e23b3f1d4    vm06            Connected
6  e5d72fad-ff55-4a25-a826-1b8962700796    vm05            Connected
7  88f17aaf-1cee-43c7-9dce-465550187502    vm04            Connected
8  d27a0286-9bdc-45e4-bdc8-d86378a780ba    vm03            Connected
9  9401f2da-03bf-47d9-a830-73b7f2ab5a6c    vm02            Connected
10 50e70289-5d89-42ad-9c05-8b604e1a0c21    localhost       Connected
```

Listing 32: A snippet of a GlusterFS command showing connected nodes

```
student@vm01:~> sudo gluster volume info

Volume Name: gv1
Type: Replicate
Volume ID: e48e5cac-5194-40dc-9e27-730e3439c0eb
Status: Started
Snapshot Count: 0
Number of Bricks: 1 x 8 = 8
Transport-type: tcp
Bricks:
Brick1: vm01:/data/brick1/gv1
Brick2: vm02:/data/brick1/gv1
Brick3: vm03:/data/brick1/gv1
Brick4: vm04:/data/brick1/gv1
Brick5: vm05:/data/brick1/gv1
Brick6: vm06:/data/brick1/gv1
Brick7: vm07:/data/brick1/gv1
Brick8: vm08:/data/brick1/gv1
Options Reconfigured:
cluster.granular-entry-heal: on
storage.fips-mode-rchecksum: on
transport.address-family: inet
nfs.disable: on
performance.client-io-threads: off
```

Listing 33: A snippet of a GlusterFS command showing replicated volume

```
student@vm01:~/stacks> docker stack ps argo | grep vm04
v5xw1lrm90fz   argo_cadvisor.z9uoam07pax0uuwn6asqbf0fp       gcr.io/cadvisor/cadvisor:latest      vm04      Running
↪  Running 4 minutes ago
prdgqd5esudk   argo_node-exporter.z9uoam07pax0uuwn6asqbf0fp   prom/node-exporter:latest           vm04      Running
↪  Running 5 minutes ago
0wrqztr9mfdf   argo_ot-app.1                                  cadeke/argo-ot-app:v1.0             vm04      Running
↪  Running 5 minutes ago
student@vm01:~/stacks> docker node inspect vm04 | jq '.[].Spec'
{
  "Labels": {},
  "Role": "worker",
  "Availability": "active"
}
student@vm01:~/stacks> docker stack ps argo | grep vm04
v5xw1lrm90fz   argo_cadvisor.z9uoam07pax0uuwn6asqbf0fp       gcr.io/cadvisor/cadvisor:latest      vm04      Shutdown
↪  Shutdown about a minute ago
prdgqd5esudk   argo_node-exporter.z9uoam07pax0uuwn6asqbf0fp   prom/node-exporter:latest           vm04      Shutdown
↪  Shutdown about a minute ago
0wrqztr9mfdf    \_ argo_ot-app.1                              cadeke/argo-ot-app:v1.0             vm04      Shutdown
↪  Shutdown about a minute ago
student@vm01:~/stacks> docker node inspect vm04 | jq '.[].Spec'
{
  "Labels": {},
  "Role": "worker",
  "Availability": "drain"
}
```

Listing 34: A snippet of Docker commands showing the draining a node in Swarm cluster

```
student@vm06:~> docker network inspect argo_dbs | jq ".[].Containers"
  # output omitted

  "7f1da95c8a6241e021a1757a7c82460142055413bcb4a20b5f9b9a0892ff1282": {
    "Name": "argo_postgres.1.fxs70kak4416yuu4ho2zyofdk",
    "EndpointID": "0fde3a0749c9df6692faa6983296a271d7fca289679cee2b52449bb63dd7bd51",
    "MacAddress": "02:42:c0:a8:80:03",
    "IPv4Address": "192.168.128.3/18",
    "IPv6Address": ""
  },
student@vm06:~> docker network inspect argo_monitoring | jq ".[].Containers"
  # output omitted

  "79dbe66f7ad81205f57db09c79b2873a4b8e0b51868d27ef48a62697d75ef9f8": {
    "Name": "argo_debug.1.ji8qqd31vi0ig4gmd75x9q6xm",
    "EndpointID": "44b42b97ef43345d8e6f9468658905704df69c491be876a1f855547688fc13e3",
    "MacAddress": "02:42:c0:a8:c0:18",
    "IPv4Address": "192.168.192.24/18",
    "IPv6Address": ""
  },
student@vm06:~> docker exec -it argo_debug.1.ji8qqd31vi0ig4gmd75x9q6xm sh
~ # ping grafana
PING grafana (192.168.192.25) 56(84) bytes of data.
64 bytes from 192.168.192.25: icmp_seq=1 ttl=64 time=0.080 ms
64 bytes from 192.168.192.25: icmp_seq=2 ttl=64 time=0.106 ms
^C
--- grafana ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1000ms
rtt min/avg/max/mdev = 0.080/0.093/0.106/0.013 ms
~ # ping prometheus
PING prometheus (192.168.192.5) 56(84) bytes of data.
64 bytes from 192.168.192.5: icmp_seq=1 ttl=64 time=0.068 ms
64 bytes from 192.168.192.5: icmp_seq=2 ttl=64 time=0.123 ms
^C
--- prometheus ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 0.068/0.095/0.123/0.027 ms
~ # ping postgres
ping: postgres: Name does not resolve
~ # nslookup grafana
Server:                 127.0.0.11
Address:        127.0.0.11#53

Non-authoritative answer:
Name:       grafana
Address: 192.168.192.25

~ # nslookup postgres
Server:                 127.0.0.11
Address:        127.0.0.11#53

** server can't find postgres: NXDOMAIN
~ # ping 192.168.128.3
PING 192.168.128.3 (192.168.128.3) 56(84) bytes of data.
^C
--- 192.168.128.3 ping statistics ---
5 packets transmitted, 0 received, 100% packet loss, time 4127ms
```

Listing 35: A snippet of various commands verifying the network segmentation in Swarm cluster

```
1     ~ cat /etc/hosts
2   # Loopback entries; do not change.
3   # For historical reasons, localhost precedes localhost.localdomain:
4   127.0.0.1   localhost localhost.localdomain localhost4 localhost4.localdomain4
5   ::1         localhost localhost.localdomain localhost6 localhost6.localdomain6
6   # See hosts(5) for proper format and other examples:
7   # 192.168.1.10 foo.example.org foo
8   # 192.168.1.13 bar.example.org bar
9   # ARGO THESIS LAB
10  10.203.96.230 vm01.lab argo.lab
11  10.203.96.231 vm02.lab argo.lab
12  10.203.96.232 vm03.lab argo.lab
13  10.203.96.233 vm04.lab argo.lab
14  10.203.96.115 vm05.lab argo.lab
15  10.203.96.116 vm06.lab argo.lab
16  10.203.96.117 vm07.lab argo.lab
17  10.203.96.118 vm08.lab argo.lab
```

Listing 36: A snippet of a “/etc/hosts” file simulating internal DNS

```
1   root@vm01:/var/lib/docker/swarm/certificates# openssl x509 -in /var/lib/docker/swarm/certificates/swarm-root-ca.crt -text
    ↪  -noout
2   (...snippet...)
3   Certificate:
4       Data:
5           Version: 3 (0x2)
6           Serial Number:
7               78:d9:b6:c0:47:0e:47:61:50:15:63:91:cc:1a:88:1d:69:79:61:5d
8           Signature Algorithm: ecdsa-with-SHA256
9           Issuer: CN = swarm-ca
10          Validity
11              Not Before: Mar 27 18:09:00 2025 GMT
12              Not After : Mar 22 18:09:00 2045 GMT
13          Subject: CN = swarm-ca
14
15  root@vm01:/var/lib/docker/swarm/certificates# openssl x509 -in /var/lib/docker/swarm/certificates/swarm-node.crt -text
    ↪  -noout
16  (...snippet...)
17  Certificate:
18      Data:
19          Version: 3 (0x2)
20          Serial Number:
21              0f:d5:eb:9b:48:f0:0e:94:10:d5:c5:b1:da:67:a7:af:e5:26:7c:26
22          Signature Algorithm: ecdsa-with-SHA256
23          Issuer: CN = swarm-ca
24          Validity
25              Not Before: Mar 27 17:14:00 2025 GMT
26              Not After : Jun 25 18:14:00 2025 GMT
```

Listing 37: A snippet of commands showing Swarm root CA and node certificates on “vm01”

```
~ # ping postgres.service.consul
PING postgres.service.consul (10.203.96.233) 56(84) bytes of data.
64 bytes from vm04 (10.203.96.233): icmp_seq=1 ttl=63 time=0.290 ms
64 bytes from vm04 (10.203.96.233): icmp_seq=2 ttl=63 time=0.263 ms
64 bytes from vm04 (10.203.96.233): icmp_seq=3 ttl=63 time=0.470 ms
^C
--- postgres.service.consul ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2077ms
rtt min/avg/max/mdev = 0.263/0.341/0.470/0.091 ms
~ # ping grafana.service.consul
PING grafana.service.consul (10.203.96.118) 56(84) bytes of data.
64 bytes from vm08 (10.203.96.118): icmp_seq=1 ttl=63 time=0.289 ms
64 bytes from vm08 (10.203.96.118): icmp_seq=2 ttl=63 time=0.330 ms
64 bytes from vm08 (10.203.96.118): icmp_seq=3 ttl=63 time=0.292 ms
^C
--- grafana.service.consul ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2076ms
rtt min/avg/max/mdev = 0.289/0.303/0.330/0.018 ms
~ # ping ot-app.service.consul
PING ot-app.service.consul (10.203.96.118) 56(84) bytes of data.
64 bytes from vm08 (10.203.96.118): icmp_seq=1 ttl=63 time=0.217 ms
64 bytes from vm08 (10.203.96.118): icmp_seq=2 ttl=63 time=0.338 ms
64 bytes from vm08 (10.203.96.118): icmp_seq=3 ttl=63 time=0.479 ms
^C
--- ot-app.service.consul ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2037ms
rtt min/avg/max/mdev = 0.217/0.344/0.479/0.107 ms
~ # ping memcached.service.consul
PING memcached.service.consul (10.203.96.117) 56(84) bytes of data.
64 bytes from vm07 (10.203.96.117): icmp_seq=1 ttl=64 time=0.038 ms
64 bytes from vm07 (10.203.96.117): icmp_seq=2 ttl=64 time=0.052 ms
64 bytes from vm07 (10.203.96.117): icmp_seq=3 ttl=64 time=0.079 ms
^C
--- memcached.service.consul ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2030ms
rtt min/avg/max/mdev = 0.038/0.056/0.079/0.017 ms
```

Listing 38: A snippet of ping commands verifying internal DNS in Nomad cluster with Consul

```
1  student@vm01:~/jobs> nomad job scale ot ot-app 10
2  ==> 2025-04-13T22:05:27Z: Monitoring evaluation "4a8d6914"
3      2025-04-13T22:05:27Z: Evaluation triggered by job "ot"
4      2025-04-13T22:05:28Z: Evaluation within deployment: "4a41bb6a"
5      2025-04-13T22:05:28Z: Allocation "62a38cdc" created: node "6f594192", group "ot-app"
6      2025-04-13T22:05:28Z: Allocation "825a7f7c" created: node "d216928d", group "ot-app"
7      2025-04-13T22:05:28Z: Allocation "b6361c4c" created: node "d216928d", group "ot-app"
8      2025-04-13T22:05:28Z: Allocation "c92c10eb" created: node "6f594192", group "ot-app"
9      2025-04-13T22:05:28Z: Allocation "5feb8f8a" modified: node "3919c96c", group "ot-app"
10     2025-04-13T22:05:28Z: Allocation "61ddee31" created: node "d56fe237", group "ot-app"
11     2025-04-13T22:05:28Z: Allocation "69bd797a" created: node "d56fe237", group "ot-app"
12     2025-04-13T22:05:28Z: Allocation "c6da8591" created: node "6f594192", group "ot-app"
13     2025-04-13T22:05:28Z: Allocation "385f29b4" created: node "d56fe237", group "ot-app"
14     2025-04-13T22:05:28Z: Allocation "61db18f1" created: node "fd7ca40d", group "ot-app"
15     2025-04-13T22:05:28Z: Evaluation status changed: "pending" -> "complete"
16 ==> 2025-04-13T22:05:28Z: Evaluation "4a8d6914" finished with status "complete"
17 ==> 2025-04-13T22:05:28Z: Monitoring deployment "4a41bb6a"
18    Deployment "4a41bb6a" successful
19
20     2025-04-13T22:05:54Z
21     ID          = 4a41bb6a
22     Job ID      = ot
23     Job Version = 5
24     Status      = successful
25     Description = Deployment completed successfully
26
27     Deployed
28     Task Group  Desired  Placed  Healthy  Unhealthy  Progress Deadline
29     ot-app      10       10      10       0          2025-04-13T22:15:52Z
```

Listing 39: A snippet of a Nomad command, scaling OT application to ten instances in Nomad cluster

```
1  student@vm02:~> nomad server members
2  Error querying servers: Unexpected response code: 400 (Client sent an HTTP request to an HTTPS server.)
```

Listing 40: A snippet of a Nomad command indicating an unauthenticated attempt after enabling mTLS

```
student@vm01:~/jobs> nomad status ot
# output omitted

Latest Deployment
ID          = 957c4f89
Status      = running
Description = Deployment is running but requires manual promotion

Deployed
Task Group  Auto Revert  Promoted  Desired  Canaries  Placed  Healthy  Unhealthy  Progress Deadline
ot-app      true         false     4        2         2       2        0          2025-04-15T18:57:59Z

Allocations
ID        Node ID   Task Group  Version  Desired  Status   Created    Modified
e4fbd356  6f594192  ot-app      4        run      running  4m50s ago  4m25s ago
9a460b51  3919c96c  ot-app      4        run      running  4m50s ago  4m34s ago
63aa90f6  d56fe237  ot-app      3        run      running  23m51s ago 23m40s ago
f4505f4d  d56fe237  ot-app      2        stop     failed   28m6s ago  23m51s ago
2695c2f6  fd7ca40d  ot-app      2        stop     failed   30m50s ago 28m6s ago
35675af4  6f594192  ot-app      2        stop     failed   32m42s ago 30m50s ago
5ea7f5be  3919c96c  ot-app      2        stop     failed   33m51s ago 32m42s ago
f4b35a2e  fd7ca40d  ot-app      3        run      running  35m13s ago 23m41s ago
4885fb67  d216928d  ot-app      3        run      running  35m30s ago 23m40s ago
1736e9f6  d56fe237  ot-app      3        run      running  35m46s ago 23m40s ago
# output omitted
```

Listing 41: A snippet of a Nomad command showing canary deployment of OT application version

```
student@vm01:~/jobs> nomad deployment promote 957c4f89
==> 2025-04-15T18:55:03Z: Monitoring evaluation "2a00806d"
    2025-04-15T18:55:03Z: Evaluation triggered by job "ot"
    2025-04-15T18:55:03Z: Evaluation within deployment: "957c4f89"
    2025-04-15T18:55:04Z: Allocation "6d30a70e" created: node "d56fe237", group "ot-app"
    2025-04-15T18:55:04Z: Evaluation status changed: "pending" -> "complete"
==> 2025-04-15T18:55:04Z: Evaluation "2a00806d" finished with status "complete"
==> 2025-04-15T18:55:04Z: Monitoring deployment "957c4f89"
  Deployment "957c4f89" successful

    2025-04-15T18:55:37Z
    ID          = 957c4f89
    Job ID      = ot
    Job Version = 4
    Status      = successful
    Description = Deployment completed successfully

    Deployed
    Task Group  Auto Revert  Promoted  Desired  Canaries  Placed  Healthy  Unhealthy  Progress Deadline
    ot-app      true         true      4        2         4       4        0          2025-04-15T19:05:36Z
```

Listing 42: A snippet of a Nomad command, promoting canary deployment of OT application