Shifting the Orchestration Monoculture: A Practical Evaluation of Alternatives to Kubernetes for Distributed Web Systems

Casper De Keyser
Dept. of Computer Science & Security
St. Pölten University of Applied Sciences
St. Pölten, Austria
cr231502@fhstp.ac.at

***Abstract*—Container orchestration has become a critical 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, its extensive feature set, and intrinsic complexity may not align with the specific requirements of every use-case or the capabilities of every development team, thereby complicating migration and implementation efforts. In such scenarios, alternative technologies that emphasize simplicity and usability over advanced customization may offer more appropriate solutions.**

This study investigates two such alternatives, Docker Swarm and HashiCorp Nomad, through a practical evaluation based on a predefined use-case representative of production-grade environments. We selected these candidates by reviewing current open-source orchestration frameworks that satisfy the use-case's defined requirements, and evaluated against common orchestration features such as scaling, load balancing, service discovery, and networking.

We find that Swarm meets the requirements with minimal additional configuration. Nomad requires greater configuration and domain knowledge, and its advanced feature set makes it more suitable for large-scale deployments. We highlight areas for future research that may enhance the broader orchestration landscape, notably in supporting more advanced application stacks using components such as a dedicated reverse proxy for Swarm or Consul's service mesh for Nomad.

***Index Terms*—microservices, Docker, Kubernetes, Swarm, Nomad**

I. Introduction

Cloud-native technologies have transformed how software is developed and deployed. Containerized workloads are present in diverse environments—from high-performance ML clusters to home labs. 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 (CO) respectively. This shift introduces new configuration and management challenges, particularly with Kubernetes' complexity and knowledge demands. Consequently, development teams and organizations must invest in specific skills to navigate these environments effectively.

In many IT projects, taking a step back and carefully evaluating available tools can assist in selecting the optimal solution. Docker and Kubernetes are both disruptive technologies that have revolutionized the way we operate computing workloads at scale, but they are not "one-size-fits-all" solutions. Kubernetes is often described as "having a steep learning curve", requiring substantial resources and time to achieve the desired setup. Moreover, not all production scenarios require a full-fledged Kubernetes cluster to meet scalability and availability goals. Ultimately, choosing the right tool serves as a crucial step in the design of modern applications. After all, one would prefer a hammer over an axe to drive a nail in a block of wood.

In this paper, we present a practical comparison of container orchestration frameworks using a production-grade use-case, highlighting each framework's distinct characteristics to guide technical profiles in choosing the most appropriate orchestration solution for their specific orchestration needs.

II. Background

A. Monoliths and Microservices

A microservice architecture is based on the idea of dividing an application into smaller programs, each responsible for a specific function. This contrasts with traditional monolithic applications, where functionalities are grouped in layered structures—typically presentation, business logic, and data access layers [1], presented in fig. 1. Frameworks such as Django[1], Ruby on Rails[2] and Laravel[3] enforce this structure using the model–view–controller (MVC) pattern. While this promotes consistency in codebases, the application is still deployed as a single unit containing all functions. This tight coupling can introduce issues during scaling or independent updates.

By contrast, a microservice architecture separates such functions into independent, deployable units [2], [3], visualized in fig. 2. Ideally, each microservice addresses a single business capability, communicates via internal REST APIs, and can be developed, scaled, and deployed independently. Microservices typically exhibit the following core characteristics:

[1]https://www.djangoproject.com/
[2]https://rubyonrails.org/
[3]https://laravel.com/

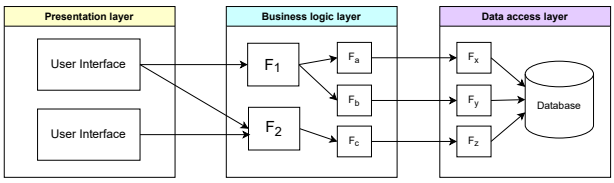

Fig. 1. A three layer architecture, often present in monoliths

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

Microservices can scale horizontally or vertically as needed, and because they are loosely coupled, teams can release updates more frequently without coordination overhead. Moreover, microservices encapsulate their dependencies, increasing portability across environments. Unlike monoliths, microservices also support heterogeneous programming stacks, allowing teams to choose the languages or tools best suited for the task [4]. A key technology that enables the adoption of microservices is containerization.

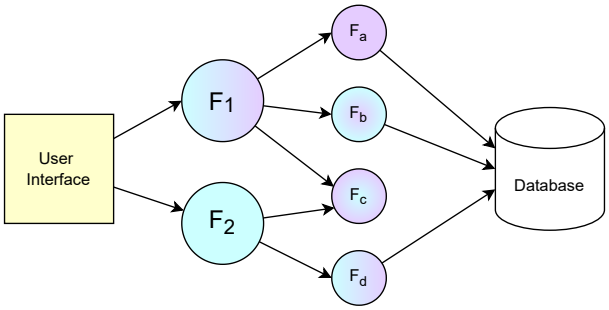

Fig. 2. A microservice architecture

B. Containerization and Docker

Containerization is a more lightweight virtualization method compared to traditional virtual machines (VM) [5], [6]. Figure 3 visualizes the differences between both architectures. Instead of virtualizing on hardware level, containers virtualize on OS level, sharing the kernel, binaries and certain libraries among containers [7]. This results in smaller, faster-starting units that support dynamic horizontal scaling. Because of this design, containers compromise on isolation, potentially increasing the attack surface [8].

Docker stands out as the most popular [9] containerization platform, with other technologies such as Podman[4] and containerd[5] also being prominent options. In the scenario demonstrated in fig. 3, six containers are present. In reality, a much larger amount of containers and underlying hosts is utilized, since the real power of containers comes from implementing distributed systems [7]. With these additional instances, a need for orchestration arises, since manual management is not feasible in these scenarios.

[4]https://podman.io/
[5]https://containerd.io/

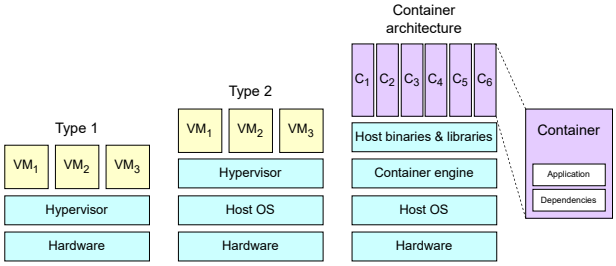

Fig. 3. A high-level comparison between VMs and containers

C. Orchestration and Kubernetes

CO is essential for running microservices, supported by a large amount of containers, at scale. The CO framework forms the abstraction layer between the physical infrastructure and the running applications, visible in fig. 4. An effective orchestrator must address the following core requirements at minimum [10], [11]:

- Scheduling: assigning containers to nodes
- Scaling: adjusting container instance counts
- Monitoring: tracking health of containers and nodes
- High availability: ensuring containers are accessible
- Load balancing: distributing traffic across containers

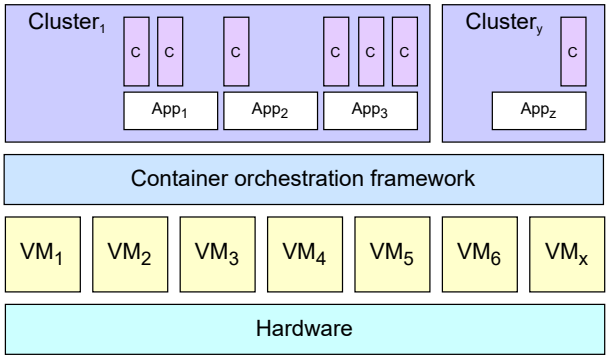

Fig. 4. A high-level overview of a CO architecture

Announced by Google in 2014 as an open-source cluster manager for Docker [7], [12], Kubernetes became the de facto CO standard after joining the CNCF in 2015. Today, over 84% of survey respondents use or evaluate it [13], and it leads CO usage in OpenStack environments [14]–[16]. Kubernetes offers a comprehensive feature set—service discovery, autoscaling, load balancing, storage orchestration, rollback, self-healing, and secrets management [17]. It introduces so-called ”pods”, which provide an abstraction for one or more containers [7]. 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.

III. Related Work

The related works for this study are grouped into three categories, each addressing a subdomain of this research.

A. Challenges related to Kubernetes

Reference [18] investigated the HA functionalities that Kubernetes offers, finding that their requirements were not automatically satisfied by deploying a microservice with Kubernetes. Reference [19] evaluated his experience running a Kubernetes cluster consisting out of SBCs, identifying a number of reasons why Kubernetes is considered to be "complex". He concludes that this complexity in necessary in order for the total architecture to become more simplified when scaling up the application. In an empirical study of security misconfigurations in Kubernetes manifests, [20] identified 11 categories of issues, underlining the importance of security-focused code reviews and static analysis. Reference [21] and [22] compared different lightweight Kubernetes-compatible platforms, which provide easier deployment and support in environments with limited resources. Reference [23] evaluated container orchestration frameworks such as Kubernetes, Docker Swarm and Apache Mesos in terms of functionality and performance, finding that Kubernetes boast the broadest feature support, thus making it the most complete orchestrator. They mentioned that the associated complexity in Kubernetes' architecture could result in significant overhead hindering its performance.

These studies 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 [20].

B. Managed Kubernetes Cloud Services

In a review of Docker and general container technologies, [4] mentioned the need for container administration and management. They briefly introduced the concept of CaaS, where container engines, orchestrators, and underlying compute resources are provided by CSP. Reference [24] evaluated the performance of major managed Kubernetes offerings—EKS, AKS, and GKE—measuring CPU, RAM, disk, and network metrics. Their results demonstrated that cluster performance is driven primarily by the choice of virtual machine instance types rather than by the orchestration layer's overhead. More recently, [25] compared container runtimes—Docker and containerd—finding that containerd's lighter footprint resulted in better overall performance, and that self-deployed Kubernetes clusters often outperformed managed services due to fewer additional control-plane pods.

These studies indicate that managed Kubernetes services offload much of the configuration and operation—yet they still require informed decisions about cloud provider selection, infrastructure sizing, and runtime configuration. The focus of this study, by contrast, is on open-source, on-premises orchestration options that enable teams to manage their containerized workloads without vendor lock-in. Although managed offerings align with this research's goal of simplifying CO, cloud services may be inaccessible to SME teams, because of budgetary or organizational reasons, which the main target audience for whom this study attempts to provide insights.

C. Comparisons between CO Frameworks

Reference [26] highlighted Docker and Kubernetes as foundational cloud-native technologies, demonstrating Kubernetes' multi-host deployment and management capabilities, and briefly contrasting it with Docker Swarm. Reference [27] presented Docker Swarm as a performance-oriented orchestrator, noting that security mechanisms—such as authentication, authorization, TLS, and network segmentation—are critical to prevent breaches. In another practical scenario, [28] evaluated Kubernetes, Swarm, Fleet, and Mesos, concluding that Kubernetes and Mesos excel in large production deployments while Swarm suits testing scenarios, and noting the emergence of Kubernetes distributions like MicroK8s and OpenShift. In a feature study, [29] identified 124 common and 54 unique features across Kubernetes, Swarm, and Mesos, finding Kubernetes the most feature-rich and Mesos the most extensible. Reference [10] introduced a benchmarking tool to compare Kubernetes and Nomad on self-hosted and GCP clusters, reporting Kubernetes' superior performance but mentioning the need for additional tests using production workloads. Finally, [30] contrasted Kubernetes, Swarm, Mesos, and OpenShift on security, deployment ease, stability, and scalability, finding Kubernetes and Mesos best for large clusters, Swarm simplest for small setups, and OpenShift most secure out of the box.

While these works share a practical approach similar to this study's, their recommendations often remain at a general level, offering limited guidance for teams with specific use-case requirements. By conducting hands-on implementations with CO frameworks against a production-grade use case, this study aim to uncover the strengths and limitations of each framework.

IV. Methodology

This research follows a methodology consisting out of four stages. First, we conducted a review of the state of the art of CO frameworks to identify current challenges, solutions, and technologies. Building on those insights, we then designed a practical use case—a distributed web application with typical production-grade components—to serve as an evaluation benchmark. Third, we distilled the use-case's requirements into a feature set and, by cross-referencing these with the literature findings, selected viable CO candidates for the practical experiments. Finally, each candidate underwent a qualitative implementation of the use-case, where configuration steps, observed behaviors, and strengths and limitations from a developer's perspective were of a particular interest to us.

These stages are highly interconnected. The literature review supported the evaluation criteria, the use-case design, and the selection of CO features and candidates. The practical implementation validated our understanding of each candidate's capabilities with regard to the CO requirements, which are ultimately discussed and presented in the final two chapters of this study.

V. Use-Case and Selection

Prior to the implementation, we define the use-case and make a selection of CO features and candidates.

A. Use-Case Definition

To emulate the real-world challenges related to CO, we present ARGO (Address Resolver written in Go), a simplified DNS-like system that facilitates translations between IP addresses and domain names. This architecture acts as a perfect candidate for a microservices approach, while reflecting typical production-grade components—web APIs, websites, OT application, databases, monitoring, segmentation, etc.—while abstracting non-essential details. Figure 5 illustrates how ARGO's containerized services interact via load balancers and how the orchestrator sits at the core of the architecture, allowing us to change out that part of the application to compare different candidates.

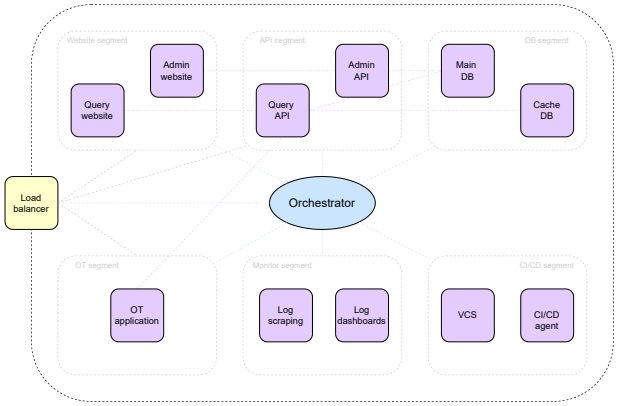

Fig. 5. A schematic overview of the ARGO use-case

The use-case contains the following components, which need to be orchestrated appropriately. These components were introduced with real-world experience in mind, and were partially based security controls part of the NIST SP800-53 [31] and NIST SP800-204 [32] standards.

- **Query API**: translates between IPs and domain names
- **Admin API**: interacts with data model
- **Query site**: consumes query API for regular users
- **Admin site**: consumes admin API for admin users
- **OT app**: consumes query API, simulating non-human clients
- **Main DB**: stores the domain and server records
- **Cache DB**: stores latest queries for faster lookups
- **Monitoring**: centrally exports, scrapes, and visualizes logs
- **Segmentation**: only allows strictly necessary communication between components
- **Shared storage**: to achieve HA among the cluster's nodes
- **Load balancing** and **secure exposure**: access for clients outside orchestration network
- (optional) Secure communication between microservices using **mutual authentication** mechanisms such as mTLS
- (optional) A centralized **secret management** system for storing application secrets
- (optional) A **VCS** for code repositories and providing CI/CD

B. Feature Selection

In section II-C, we defined the minimum requirements for a CO framework to be a viable solution for this study's defined use-case. Similar to this definition of the requirements, we based the selection on the characteristics—presented in table I—as identified by [10], [11], [23].

TABLE I
An Overview of the Selected CO Features

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

C. Candidate Selection

The CO candidates—presented in table II—were selected based on the aforementioned CO features and some additional criteria, which focused on non-technical requirements. Each candidate must be open-source (or provide an unrestricted free-tier), be well-suited to small- and medium-scale deployments, offer true alternatives to vanilla Kubernetes (though Kubernetes-based distributions with unique enhancements remain eligible), and remain under active development with regular updates rather than legacy or deprecated status.

VI. Implementation

For the implementation, we set up the ARGO application as a base scenario in a local development environment. Then, the application was adapted and deployed on a small datacenter consisting out of eight VMs with the characteristics shown in table III. Through the use of snapshots, the same infrastructure could be reused for multiple testing scenarios. All relevant implementation information—source code, configuration

TABLE II
CO CANDIDATE SELECTION OVERVIEW

Candidate	Description	Selected
Swarm	Docker-native orchestration with minimal configuration overhead	✓
Kubernetes	Feature-complete de facto standard—excluded to focus on true alternatives	✗
k3s, k0s, MicroK8s	Lightweight k8s distros aimed at low-resource scenarios, simplifying configuration and management—excluded in favor of true alternatives	✗
Mesos	Datacenter-scale distributed system kernel—excluded, "overkill" for SME-focused use-case	✗
Nomad	Flexible scheduler requiring Consul for service discovery	✓
OpenShift	Enterprise Kubernetes platform with rich features—excluded because of massive feature set	✗
MicroShift	Edge-focused minified OpenShift—excluded for similar reasons as OpenShift	✗
Cloud services	Fully managed cloud Kubernetes services such as EKS, AKS or GKE—excluded due to open-source and on-premise focus	✗

files, Ansible playbooks, traffic dumps—can be found in this project's accompanying GitHub repository[6].

TABLE III
TEST ENVIRONMENT CONFIGURATION

Component	Specification
CPU	8 cores @ 2.194 GHz
RAM	32 GiB
Main disk	200 GiB
Extra 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

Prior to implementing the use-case, we revisited the evaluation criteria and method which would be used to compare features of both CO candidates. 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 IV.
- y_f Experience with a specific feature during the implementation process. Ranges from 1 to 5, detailed in table IV.
- 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.

[6]https://github.com/cadeke/argo

TABLE IV
AN OVERVIEW OF THE GRADING SCHEMA

Score	Documentation support	Implementation experience
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

A. Base

The following technology choices were made for the practical implementing. Table V shows an overview of each component and its chosen technology. Figure 6 shows the entire ARGO application stack running on a local development machine with Docker Compose. This declarative way of defining the application served as a baseline, from which we could adapt in order to align with each specific CO candidate's syntax or deployment approach.

TABLE V
IMPLEMENTATION DETAILS FOR ARGO

Component	Implementation Choice
DBs	Postgres and Memcached
APIs	Dockerized Go web APIs
OT app	Dockerized Go CLI application
Websites	Dockerized React applications
Monitoring	Grafana and Prometheus
Log exporting	cAdvisor, Node exporter and Postgres exporter
CI/CD	GitLab CE and two GitLab runners

```
user@local> docker compose up -d
[+] Running 16/16
  Network argo_argo-backend       Created        0.1s
  Container postgres              Started        0.4s
  Container memcached             Started        0.4s
  Container gitlab                Started        0.5s
  Container admin-api             Started        0.5s
  Container query-api             Started        0.5s
  Container argo-gitlab-runner-2  Started        0.5s
  Container argo-gitlab-runner-1  Started        0.7s
  Container admin-site            Started        0.6s
  Container prometheus            Started        0.7s
  Container ot-app                Started        0.7s
  Container query-site            Started        0.7s
  Container cadvisor              Started        0.9s
  Container grafana               Started        0.9s
  Container postgres-exporter     Started        0.8s
  Container node-exporter         Started        0.9s
```

Fig. 6. An overview of ARGO running with Docker Compose

B. Swarm

Swarm is included in the Docker Engine by default, so no additional configuration is required. A swarm can be started by promoting a Docker host to Swarm leader. Following Docker's recommendations, we configured three managers and five workers [33] in the Swarm cluster, shown in fig. 7. With the nodes configured, we adapted the base Docker Compose file to

deploy ARGO with Swarm. This process included configuring the scaling and rollback configurations, defining networks and optional encryption with IPsec, exposing services through the routing mesh, verifying internal DNS addressing, and investigating central secret management. After executing the necessary steps, we could observe ARGO running in Swarm, shown with in fig. 8. Swarm's final configuration file can be found in this project's accompanying GitHub repository[7].

```
student@vm01> docker node ls
ID                            HOSTNAME   STATUS    AVAILABILITY   MANAGER
anvacpekzrc35twzg5y3rlgic *   vm01       Ready     Active         Leader
i5kgrnmecaqa44rqbbcw52db6     vm02       Ready     Active         Reachable
yzrumknlvx6kd383sa7o2jrop     vm03       Ready     Active         Reachable
z9uoam07pax0uuwn6asqbf0fp     vm04       Ready     Active
yezg2ojyvca980iehdo0hisqk     vm05       Ready     Active
7dqcqy9pdzirqjuia96phycwp     vm06       Ready     Active
teesrnys35963sbyzp07o20tk     vm07       Ready     Active
h500eow7r8ps17losflrirni9     vm08       Ready     Active
```

Fig. 7. An overview of Swarm nodes

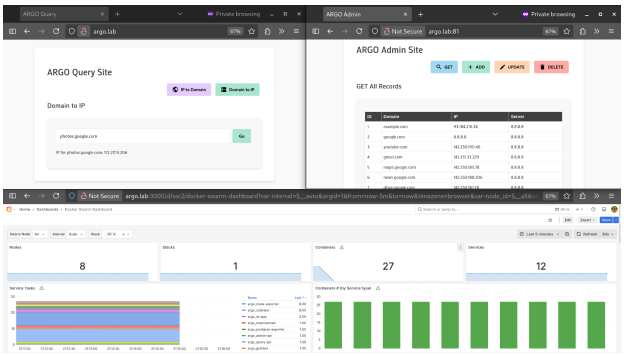

Fig. 8. Screenshots of ARGO, showing admin-site, query-site and Grafana

C. Nomad

Repeating similar steps, we configured both Nomad and Consul in our cluster with three servers and five clients, shown in fig. 9. This required significantly more effort than with Swarm, since two additional services needed to be installed and configured. Nomad requires deployments to be configured in so-called "jobs", which could then be individually managed and scaled. Each ARGO component was captured in a separate job file[8], written in HCL or JSON. Ultimately, we were able to run ARGO with Nomad and to test the different use-case requirements, gaining insights in Nomad's characteristics. Figure 10 shows the different Nomad jobs that made up the ARGO deployment.

VII. Results and Discussion

A. Kubernetes' Complexity

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

[7]https://github.com/cadeke/argo/blob/main/swarm/argo-stack.yml
[8]https://github.com/cadeke/argo/tree/main/nomad/jobs

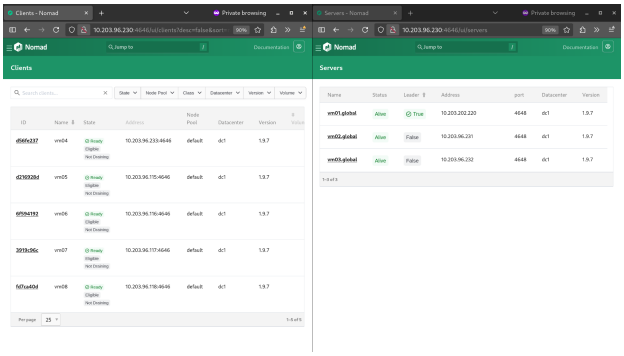

Fig. 9. An overview of Nomad client and server nodes

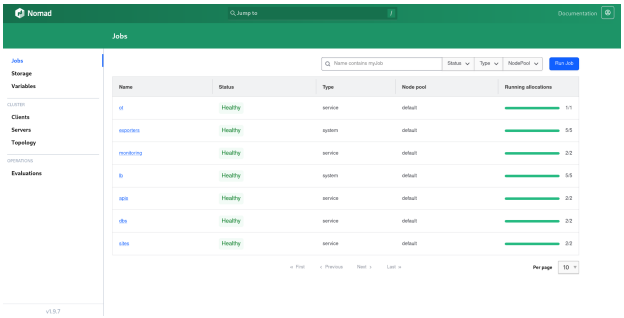

Fig. 10. An overview of ARGO's jobs deployed on Nomad

findings gathered during the practical implementation. 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. 11.

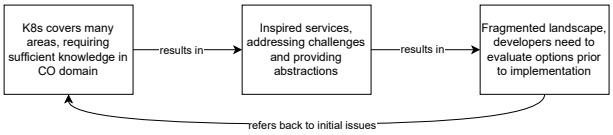

Fig. 11. A chart showing interconnected Kubernetes challenges

B. CO Alternatives

During the implementation, both Swarm and Nomad were analyzed based on the requirements shown in table VI, which were based on the findings from section V-B. Overall, Swarm aligned more closely with the requirements of the defined use-case, as demonstrated in the grading scores in section VII-C. Swarm's strongest advantage over Nomad lies in its simple approach to extend Docker's existing 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, 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.

TABLE VI
EVALUATION CRITERIA FOR CO CANDIDATES

#	Criterion
1	Scheduling of components (i.e. websites, APIs, databases, monitoring elements)
2	Service discovery for internal addressing
3	Fault tolerance during container or node failure
4	Highly available applications during updates (e.g. rolling updates, rollbacks)
5	Network segmentation between the different components, only allowing required communication
6	Shared storage between nodes to achieve highly available data volumes
7	Load balancing of internal and external requests
8	(optional) Encrypted internal communication (e.g. using mTLS)
9	(optional) Centralized secret management

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 investigated, 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 seem 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 study, Nomad could present 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.

C. Grading

The grading for both Swarm and Nomad is captured in table VII. 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 VI. 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.

TABLE VII
AN OVERVIEW OF THE FEATURE SCORES FOR SWARM AND NOMAD

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	-

D. Review of Implementation

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[9] into the deployed application stack, 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.

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

VIII. CONCLUSION

A. 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 a silver bullet for all container orchestration problems, such as the one presented in this research. As discussed in section VII-A, 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 study. Kubernetes was not evaluated practically in the same way as the other candidates mentioned, thus limiting our ability to compare the two uniformly.

B. 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 SME development teams.

C. 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.

D. Future Work

In section VII-D, we mentioned the attempts done to integrate a dedicated reverse proxy like Traefik into the Swarm stack. Although this effort was ultimately unsuccessful due to his project's 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 study, the emphasis was on evaluating general functionalities rather than specific technologies used to address particular challenges.

As an additional research avenue for Nomad, we think that Consul's service mesh deserves a more in-depth practical evaluation. In this study, 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 project, the true potential of Nomad and Consul lies.

Finally, we want to encourage future research efforts to investigate candidates that were not included in this project's 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 study.

ACKNOWLEDGMENT

This paper summarizes the key findings of the accompanying master's thesis [34]. Readers are referred to the full thesis for additional implementation details.

REFERENCES

[1] J. Tie, J. Jin, and X. 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. [Online]. Available: https://ieeexplore.ieee.org/document/5988838

[2] C. Pahl and P. 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. [Online]. Available: https://doi.org/10.5220/0005785501370146

[3] W. Hasselbring, "Microservices for scalability: Keynote talk abstract," in *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, ser. ICPE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 133–134. [Online]. Available: https://doi.org/10.1145/2851553.2858659

[4] S. Singh and N. 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.

[5] A. M. Joy, "Performance comparison between linux containers and virtual machines," in *2015 International Conference on Advances in Computer Engineering and Applications*, 2015, pp. 342–346.

[6] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. 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. New York, NY, USA: Association for Computing Machinery, 2007, p. 275–287. [Online]. Available: https://doi.org/10.1145/1272996.1273025

[7] D. Bernstein, "Containers and Cloud: From LXC to Docker to Kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, Sep. 2014. [Online]. Available: https://ieeexplore.ieee.org/document/7036275/

[8] T. Combe, A. Martin, and R. Di Pietro, "To docker or not to docker: A security perspective," *IEEE Cloud Computing*, vol. 3, no. 5, pp. 54–62, 2016.

[9] L. Baresi, G. Quattrocchi, and N. Rasi, "A qualitative and quantitative analysis of container engines," *J. Syst. Softw.*, vol. 210, no. C, Apr. 2024. [Online]. Available: https://doi.org/10.1016/j.jss.2024.111965

[10] M. Straesser, J. Mathiasch, A. Bauer, and S. 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. New York, NY, USA: Association for Computing Machinery, 2023, p. 187–198. [Online]. Available: https://doi.org/10.1145/3578244.3583726

[11] A. Khan, "Key characteristics of a container orchestration platform to enable a modern application," *IEEE Cloud Computing*, vol. 4, no. 5, pp. 42–48, 2017.

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

[13] C. N. C. Foundation. (2023) Cncf annual survey 2023. Accessed: 2025-02-17. [Online]. Available: https://www.cncf.io/reports/cncf-annual-survey-2023/

[14] OpenStack. (2016, Oct.) October 2016 openstack user survey report. Accessed: 2025-03-07. [Online]. Available: https://object-storage-ca-ymq-1.vexxhost.net/swift/v1/6e4619c416ff4bd19e1c087f27a43eea/www-assets-prod/survey/October2016SurveyReport.pdf

[15] ——. (2016, Nov.) November 2017 openstack user survey report. Accessed: 2025-03-07. [Online]. Available: https://object-storage-ca-ymq-1.vexxhost.net/swift/v1/6e4619c416ff4bd19e1c087f27a43eea/www-assets-prod/survey/OpenStack-User-Survey-Nov17.pdf

[16] ——. (2018) 2018 openstack user survey report. Accessed: 2025-03-07. [Online]. Available: https://www.openstack.org/user-survey/2018-user-survey-report

[17] Kubernetes. (2024) Kubernetes documentation - concepts - overview. Accessed: 2025-03-11. [Online]. Available: https://kubernetes.io/docs/concepts/overview/

[18] L. Abdollahi Vayghan, M. A. Saied, M. Toeroe, and F. 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.

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

[20] A. Rahman, S. I. Shamim, D. B. Bose, and R. Pandita, "Security misconfigurations in open source kubernetes manifests: An empirical study," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 4, May 2023. [Online]. Available: https://doi.org/10.1145/3579639

[21] S. Telenyk, O. Sopov, E. Zharikov, and G. 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)*, vol. 1, Sep. 2021, pp. 313–317, iSSN: 2770-4254. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9660392

[22] H. Koziolek and N. 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. [Online]. Available: https://doi.org/10.1145/3578244.3583737

[23] I. M. A. Jawarneh, P. Bellavista, F. Bosi, L. Foschini, G. Martuscelli, R. Montanari, and A. Palopoli, "Container orchestration engines: A thorough functional and performance comparison," in *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*, 2019, pp. 1–6.

[24] A. Pereira Ferreira and R. 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.

[25] S. Singh, F. Shivanshi, and R. 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.

[26] J. Shah and D. 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.

[27] M. Ileana, M. I. Oproiu, and C. 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.

[28] L. Mercl and J. Pavlík, "The comparison of container orchestrators," *Advances in Intelligent Systems and Computing*, 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:69844239

[29] E. Truyen, D. Van Landuyt, D. Preuveneers, B. Lagaisse, and W. Joosen, "A Comprehensive Feature Comparison Study of Open-Source Container Orchestration Frameworks," *Applied Sciences*, vol. 9, no. 5, p. 931, Mar. 2019. [Online]. Available: https://www.mdpi.com/2076-3417/9/5/931

[30] A. Malviya and R. K. 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. [Online]. Available: https://ieeexplore.ieee.org/document/9763171/?arnumber=9763171

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

[32] R. Chandramouli, "Security Strategies for Microservices-based Application Systems," Tech. Rep. Special Publication 800-204, August 2019. [Online]. Available: https://csrc.nist.gov/pubs/sp/800/204/final

[33] D. Inc. (2025) Administer and maintain a swarm of docker engines. Accessed: 2025-03-27. [Online]. Available: https://docs.docker.com/engine/swarm/admin_guide

[34] C. D. Keyser, "Shifting the orchestration monoculture: A practical evaluation of alternatives to kubernetes as container orchestration framework for distributed web systems," 2025, unpublished.