National College of Ireland

A Cross-Environment Evaluation Framework for Container Orchestration Tools: Optimizing Efficiency and Implementation Trade-offs for Real-World Workloads

MSc Research Project
MSc Cloud Computing

Mohd Sajid Ali
Student ID: X23293519

School of Computing
National College of Ireland

Supervisor: Mr. Ahmed Makki

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name: Mohd Sajid Ali

Student ID: X23293519

Programme: MSc Cloud Computing **Year:** 2025-26

Module: MSc Research Project

Supervisor: Mr. Ahmed Makki

Submission Due Date: 11th December 2025

Project Title: A Cross-Environment Evaluation Framework for Container Orchestration Tools: Optimizing Efficiency and Implementation Trade-offs for Real-World Workloads

Word Count: 10242 **Page Count: 26**

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature: Mohd Sajid Ali

Date: 11th December 2025

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

A Cross-Environment Evaluation Framework for Container Orchestration Tools: Optimizing Efficiency and Implementation Trade-offs for Real-World Workloads

Mohd Sajid Ali
X23293519

Abstract

The choice of a container orchestration tool, whether in an on-premises environment, cloud environment, or a hybrid environment. It is difficult due to the fact that performance, resilience, and the operational effort heavily rely on the platform choice, as well as the deployment model. In this project, four orchestrators, which are Kubernetes (k3s), Docker Swarm, HashiCorp Nomad, and Red Hat MicroShift, will be empirically studied in a realistic environment. It uses a realistic multi-container Django Expense Tracker Application with PostgreSQL under synchronous HTTP CRUD workloads. The stack was tested on on-prem VMs, on the AWS EC2, and Hybrid setup, respectively, with JMeter baseline, induced-fault, and spike scenarios. The metrics measured through Prometheus and automation scripts are HTTP latency, throughput, error rates, CPU and memory consumption, HTTP up-time, deployment time, autoscaling lag, failure-recovery time, and storage overhead. The Docker Swarm had the lowest on-prem baseline latency (~0.7ms versus ~6.2ms with Kubernetes) and deployments (~30s versus ~60s). While MicroShift consistently consumed ~2GB RAM but could maintain a higher reliability. ANOVA with one-way showed statistically significant ($p<0.01$) differences between lightweight and Kubernetes-based platforms. Rather than suggesting a new orchestrator, this study focuses on comparative evidence to support decision making. Kubernetes/MicroShift are suitable to mission-critical, fluctuating loads, and Swarm/Nomad to smaller and steady-state services where easiness and efficiency are key fronts.

Keywords: container orchestration, Kubernetes, Docker Swarm, HashiCorp Nomad, MicroShift, hybrid cloud, performance evaluation, autoscaling

1 Introduction

1.1 Background

Containers and cloud-native systems have developed modern distributed systems that are changing quickly. Containers offer a lightweight, mobile, and efficient environment to execute. They can be deployed faster, have better scalability and leverage more resources than their counterparts like hypervisor-based virtual machines (Queiroz *et al.*, 2023). The container orchestration platforms (COTs) like Kubernetes, Docker Swarm, Nomad and Red

Hat-based distros have now become the focal point in managing the workloads of the cloud and microservices applications. And also for managing automated scaling, scheduling, networking and lifecycle management of the containerized applications (Straesser et al., 2023).

The reason Kubernetes is the leading orchestration standard is because it is flexible, reliable and has a robust ecosystem (Senjab et al., 2023). Meanwhile, Docker Swarm is praised to be easy to use (Malviya and Dwivedi, 2022). With its simplicity of single-binary, multi-workload design, HashiCorp Nomad is still appealing to simplified or resource-restrained settings. K3s and MicroShift are lightweight derivatives of Kubernetes that extend the orchestration to edge and IoT, as well as hybrid cases (Čilić et al., 2023).

The initial consideration was full upstream Kubernetes and Red Hat OpenShift, but the experimental setup is based on single-node VMs with 2-6 GB of RAM, and several orchestrators had to be operated sequentially on the same hardware. This is why both k3s and Microshift are chosen as lightweight, production consistent versions of Kubernetes and OpenShift that maintain API compatibility but massively cut control-plane resource utilisation. This provides a fair evaluation against Docker Swarm and Nomad in limited on-prem and cloud designs. And yet it provides mainstream Kubernetes style orchestration platform. This variety of platforms poses a feasible problem to organisations in determining the orchestration solution that suits their performance, operational and scalability needs in the most efficient manner possible.

Recent research indicates that orchestration systems have strong effects on the performance of applications, cluster utilisation, fault-tolerance and operational resilience. (Barletta et al., 2024) show that the failure of single points in orchestration metadata can cause an outage in many clusters although resiliency measures were implemented. Comparisons of various tools indicate that there is a significant range of container startup time, recovery time, network behaviour, and scaling efficiency (Prakash, 2023). Moreover, resource management policies including scheduling heuristics, autoscaling policies and multi-node recovery to failure vary greatly between orchestration systems and influence system behaviour when run under realistic workloads (Yekollu et al., 2024).

1.2 Motivation

Rapid containerization and microservice adoption have created a heterogenous and complex orchestration platform ecosystem. Although numerous studies benchmark individual orchestrators, a smaller amount of them study how the selection of COTs interacts with the place of deployment (on-prem or cloud) and network topology (local LAN or WAN/hybrid). The practical use of this study is inspired by the necessity of a practical, instrumentation-based analysis that would help the practitioners to know how common orchestrators perform under synchronous HTTP microservice workloads in various settings, as well as what trade-offs they make in terms of latency, throughput, resource consumption, availability and operational complexity. Practically, organisations have to balance performance, reliability and operational effort as well as vendor lock-in, cross-environment networking design and in what skills they will need to operate every platform.

1.3 Problem Statement

Only a few empirical works compare the performance of container orchestration platforms in on-premises versus cloud environments based on controlled and repeatable experiments and neglect hybrid setups. (Straesser et al. 2023) point out that the majority of current benchmarking methods target individual platforms. They are not standardised and often do not consider multi-environment deployments. Similarly, minimal evidence exists on the behaviour of orchestrators on different infrastructure and workloads. This leaves a knowledge gap in the case of orchestrator behaviour in real different HTTP CRUD loads. Besides, instrumentation-based analysis is required to integrate load testing, monitoring, and fault-injection methods to study orchestration performance differences in a strict manner (Ukene, 2024).

1.4 Research Objectives

This research has the following objectives:

1. Creating a test driven reusable benchmarking system by deploying the multi-container Django application to four orchestration platforms on On-premises, cloud and hybrid setups using the synchronous HTTP CRUD loads.
2. Quantitatively measure the orchestrators in terms of key performance and operational indicators, such as HTTP latency, throughput, error rate, CPU and memory utilisation, HTTP uptime, deployment time, autoscaling response time, failure-recovery time and storage overhead.
3. Make accurate, data-based recommendations in support of academia and industry to choose the best orchestration platform to use in their containerized microservices across different infrastructure environments.

1.5 Research Questions

To guide this study, the following research questions are posed:

1. Which types of container orchestration frameworks perform better and use less resources in on-premises, cloud, and hybrid environments?
2. What are the effects of autoscaling and orchestration behaviours of Kubernetes (k3s), Docker Swarm, Nomad and MicroShift on the performance of applications and the behaviour of the system under baseline, fault and spike load in diverse environments?
3. What can an instrumentation-based, multi-environment benchmarking offer that a current single-platform evaluation does not offer?

1.6 Contributions

A universal multi-environment benchmarking system of container orchestration systems, allowing a generalized assessment of the performance, resource consumption and resilience.

A comparative study of four popular orchestration tools (Docker Swarm, Kubernetes K3s, Nomad, MicroShift). Comparison is with the same amount of work, load with JMeter and an ordinary monitoring stack (Prometheus and Grafana).

Empirical information and the rules on the organizations to select the most appropriate orchestration platform according to performance and scalability needs. Providing knowledge of practical understanding of orchestration behaviour in the real world.

1.7 Report Structure

The rest of this report is structured in the following way. Section 2 provides a review of related literature on container orchestration, performance benchmarking and evaluation based on instrumentation. Section 3 explains the framework design and methodology. Section 4 gives the implementation of benchmarking including workload generation and monitoring setup. In Section 5, there is the presentation of the experiment results in environments with comparison and analysis. Section 6 has the discussion of the findings. Finally, Section 7 is the culmination of the research and gives an outline of future work.

2 Background & Literature Review

2.1 Container Orchestration Frameworks and Performance Considerations

Container orchestration frameworks (COFs) are nowadays provided with the control capabilities of deploying and managing containerized applications into clusters. Such systems as Kubernetes, Docker Swarm, Nomad and lightweight Microshift provide us with automation in providing, scheduling, scaling, networking and restoring containers. The COFs are crucial because they take care of the performance-related functionality such as container provisioning, scaling and networking scheduling in cloud native and serverless paradigms (Straesser et al., 2023). Similarly, orchestration engines provides automation of resource allocation, scheduling and load balancing of hundreds of services which ensures horizontal scaling with low over-head (Jawarneh et al., 2019). Consequently the orchestrator performance alone is of enormous impact on the application throughput and latency. Past research has been on assessment of CPU, Memory, storage, network usage and scheduling efficiency with different loads. It is also worth pointing out that the work at this are mainly on the generic performance and not on the security issues. Security features are difficult to cross-verify between various frameworks and not a part of the benchmark (Straesser et al., 2023). We also overvalue the emphasis on the core performance, scheduling and failure measurements instead of security.

2.2 Benchmarking Strategies and Performance Analysis

The necessity of the systematic benchmarking of the COFs is a shared aspect in the various studies. (Straesser et al., 2023) have created an orchestrator-agnostic benchmarking framework ‘COFFEE’ in order to compare Kubernetes and Nomad on several metrics, such as startup time and crash recovery. They affirm that there is no benchmark that caters to a variety of frameworks and tasks. They are providing their benchmark scope as provisioning containers, scheduling, availability, scaling, load balancing and networking, but security is not mentioned. They measured the capacity of the private and Google Cloud clusters and recovery durations and updates in the event of failures in case studies. As an example, they discovered that Kubernetes placed replicas more quickly than No-mad overall. Also meta-scheduling of Nomad led to a longer update latency. Likewise, (Pan et al., 2019) compared Kubernetes and Docker Swarm by utilising the conventional container workload tests. They

have said that overall performance of Kubernetes is a bit poorer as compared to Docker in Swarm mode. But Kubernetes provides more flexibility in complex cases. This means that Swarm possessed reduced overhead on simple jobs whereas the cost was a reduction in scheduling capability. Conversely, (Jawarneh et al., 2019) stated that Kubernetes provides superior performance over the other engines such as Swarm, Mesos, Cattle in com-plicated deployments. Swarm-like engines are well suited to simple cases. Their tests required the longest time on cluster-provisioning in Kubernetes because of architecture complexities although it had more features. By comparison, Swarm and Cattle with the native orchestrator of Rancher had a shorter bootstrapping time of clusters but only a few advanced features.

COF performance has been evaluated in different conditions in various studies. According to (Yekollu et al., 2024) study of Kubernetes, Docker Swarm and managed cloud systems such as AKS and Azure proclaimed that Kubernetes is highly adaptable to complex environments in their application of efficient allocation of resources and autoscaling. Meanwhile, Docker Swarm secured the victory in the simplicity and smaller applications area. They affirmed that one of the strengths of K8s is its multi-dimensional scalability (CPU, memory, I/O). But it has the weakness of relatively high overhead on small workloads. Equally, (Bangar, 2024) compared plain Docker with Swarm against Kubernetes when it comes to web applications and found that Docker Swarm provides lightweight and fast container deployment when operating small-scale applications. Kubernetes is more suitable in multi-cloud/microservice situations and in large-scale ones. To conclude, the general method of benchmarking research is to use either empirical testing or simulations to measure things like container startup time, resource usage, scaling latency and recovery after failure. The findings have been consistent in demonstrating the presence of trade-offs between the two systems where Docker Swarm can spend less time in instantiation and failover because of its simplistic heartbeat mechanisms as compared to Kubernetes which is the type that provides more sophisticated scheduling and fault-tolerance controls.

2.3 Scheduling & Autoscaling Strategies

The features that differ include container orchestrators having different ways of placing workloads and responding to demand variability. Kubernetes-based platforms present a configurable scheduler, which assigns pods to nodes according to resource requests, capacity, and policies and they can be used with Horizontal Pod Autoscalers (HPA) that can be used to increase or decrease replica counts based on CPU or custom metrics. Kubernetes scheduling surveys show a range of schemes, including basic bin-packing schemes on one end and goal-oriented and QoS aware schemes on the other end, which directly determine latency, throughput and utilisation in practice (Senjab et al., 2023; Yekollu et al., 2024).

Lighterweight scheduling and scaling is provided in lightweight orchestrators like Docker Swarm and Nomad. Swarm puts emphasis on low- Tech positioning replicas, health based flexing within a cluster, and Nomad offers constraint setting positioning but normally involves use of third party tooling or scripts to execute autoscaling. Swarm and Nomad focus on minimal control-plane overhead and simplicity of operation, whereas Kubernetes/MicroShift provides more extensive embedded reconciliation and autoscaling cycles. The thesis will not examine the inner mechanics of these various methods of

scheduling and autoscaling, but instead will assess the empirical outcome of these methods in on-premises, cloud and hybrid environments on the latency, throughput, error rate and uptime of orchestration behaviour under load and failure conditions in relationship to the research question: how orchestration behaviour changes under load and failure.

2.4 Failure Modes and Recovery

Other important issues are recovery of failures and resilience. Most of the studies that have been reviewed have done simulations of failures and they measured the recovery time. One such example is the study conducted by (Jawarneh et al., 2019) who tracked both the events of container and node failures. They found that Kubernetes recovered in the shortest amount of time after an individual container crash because local Kubelets restarts the failed pods as soon as possible. The longest recovery is however after a full node failure which is explained by its event-driven replication controller. Swarm and Ranchers Cattle on the other hand have a simpler mechanism of the heartbeat that they make use of to detect failure and reschedule. According to (Wen et al., 2025), a survey of testing COFs indicates that the failure modes vary i.e. partial outages and cascading failures which many tests omit. According to them, cluster provisioning time and failure recovery time are some of the common performance metrics. The testing frameworks will then have to address the fault injection, chaos engineering and formal verification so that they can allow these modes being to be exercised. None of the studies has exhaustively addressed all the situations of orchestrator failure and hence, it remains an open gap.

Benchmarking studies tend to involve automated recovery scenarios. For example, (Pankowski and Powroźnik, 2023) documented the time of restoring a replica following a failure by employing auto recovery of both Kubernetes and Swarm and compared response latency under a load. The experiments showed that Docker Swarm was found to have a shorter time to restore a replica as compared to Kubernetes on a Jenkins application. This proved their hypothesis that Swarm would have the lowest deployment and recovery time. Crash recovery is considered in (Straesser et al., 2023) in their COFFEE case studies of K8s and Nomad. Collectively, the research literature suggests that the richer semantics of Kubernetes pods, controllers may be a source of latency in the event of failure. Simple orchestrators are prepared to recover within relatively no time at the cost of limited control. One of the research gaps has been noted to be systematic testing of failure recovery when mixed workloads and on distributed multi-cloud or edge configurations are tested, and they are rarely tested.

2.5 Gaps: Hybrid/On-Prem/Cloud Environments

Majority of the current assessments of container orchestrators are performed on single site LAN clusters or in a single cloud provider, regardless of idealised or intra-data centre networks. Discussing even the hybrid cloud at an architectural level does not add explicit WAN latency to the end-to-end experimentation (Vankayalapati et al., 2025). Consequently, there is a dearth of empirical data on the interaction between orchestration overhead and cross-site latency, jitter and WAN induced failures. This thesis fills that gap as it analyses the same Django/PostgreSQL workload on four orchestrators (Kubernetes k3s, Docker Swarm,

HashiCorp Nomad and Red Hat MicroShift) on on-premises, cloud and a real hybrid system where load generation and monitoring are on-premises and the orchestrated application is running in AWS and the round trip in the measured path includes making allowances in the path of approximately 50 ms WAN round trip time. On the platform level, the previous research intensively benchmarks Kubernetes, and to a lower degree Docker Swarm, but pays little or nominal attention to Nomad or any empirical, peer-reviewed analysis of Red Hat MicroShift. Edge-oriented studies are usually based on k3s, KubeEdge or custom edge controllers instead of MicroShift, and their performance and resource footprint are not well documented. This thesis makes MicroShift a first-class candidate, alongside Swarm, k3s, and Nomad across all three environments, which offers some of the first comparative data on MicroShift and explains the trade-offs in cross-environment deployments between heavyweight Kubernetes-based orchestrators and lightweight schedulers.

Table 1: Comparative summary of container orchestration

Methodology	Performance Dimensions	Identified Gap	Network Model
Survey/map of container performance lit.	Container startup time, resource isolation overhead	General container perf focus, little on orchestrators	Single data-centre / private cloud LAN
Systematic literature review	Scheduling techniques, performance benchmarks in lit.	Broad survey, not focused on new schedulers or failure modes	General cloud / LAN; no WAN effects
Benchmark framework (COFFEE)	Container provisioning, scheduling delay, failover time	No support for other COFs (Swarm, MicroShift), hybrid tests	Private + public cloud; single-site access, no explicit WAN latency modelling
Empirical study (Jenkins deployment)	Launch time, CPU/memory utilization, replica restore time	Study omits multi-service applications and cloud cases	Lab LAN; single-site application cluster
Lab experiments via Rancher	Cluster provisioning time, app readiness time, failover time	No evaluation of Nomad or hybrid setups, no edge scenarios	Local cluster; idealised network
Case studies (simulated workloads)	Deployment speed, resource consumption, scaling latency	Simple scenarios, assumes ideal network (no hybrid tests)	VM-based lab environment; no hybrid networking
Comparative evaluation	CPU/memory usage, autoscaling speed, complexity handling	Focuses on cloud-managed K8s; on-prem/hybrid orchestration missing	Cloud cluster; intra-datacentre network only
Literature review (COS testing)	Testing metrics: provisioning time, recovery time, etc.	Few studies cover partial outages or multi-cloud testing	Conceptual; no empirical network model
Theoretical review	focus on workflows	No empirical eval, orchestration for >1 cloud sites is under-explored	Hybrid cloud discussed conceptually; lacks WAN latency

Paper (Year)	Orchestration Platforms
(Bachiega *et al.*, 2018)	Docker (Swarm), LXC, others
(Malhotra *et al.*, 2024)	Docker Swarm, K8s, Mesos (survey)
(Straesser *et al.*, 2023)	Kubernetes, Nomad (benchmark)
(Pankowski & Powroźnik, 2023)	Docker Swarm, Kubernetes
(Jawarneh *et al.*, 2019)	Docker Swarm, Kubernetes, Mesos, Cattle
(Bangar, 2024)	Docker (Swarm), Kubernetes
(Yekollu *et al.*, 2024)	Kubernetes, Docker Swarm, Azure/Kubernetes Service
(Wen *et al.*, 2025)	General COF survey
(Vankayalapati, 2025)	hybrid cloud

3 Research Methodology

3.1 Research Design

The quantitative experimental design is applied to this study. We performed controlled performance testing on every orchestration platform and environment as well as tracked resource utilisation and service health. The key dependent variables include HTTP latency (Mean and p95), throughput (request/s), error rate, CPU utilisation, memory utilisation, service uptime and operational times (deployment and recovery). Qualitative observations on the complexities of setups and scalability features were also studied and recorded. All platforms were run in three conditions with the same profile of workloads, steady-state baseline (normal load), fault condition (purposeful kill of containers or nodes), and load-spike condition (traffic suddenly increased to test auto-scaling or manual scaling). All the experiments had identical Apache JMeter CRUD test plan with adjustable thread counts and were reiterated several times to maintain consistency. The statistics (mean and 95th-percentile latency) were calculated among the samples and the post hoc analysis was done by one-way ANOVA to ensure that the differences between platforms were statistically important. Such a design is able to capture performance and reliability factors and is consistent with practice in container orchestration benchmarking. Each metric was analysed using one-way ANOVA where orchestrator was the primary variable and environment and scenario were different conditions of the experiment and this is appropriate since all that the study aims to achieve is to isolate systematic differences between platforms, and not to model all possible interactions.

3.2 Application Under Test

The test application is the Django Expense Tracker which is a small multi-container web service with the Django HTTP service and PostgreSQL service. It provides a realistic CRUD traffic, however not very difficult to deploy the same on proposed platforms. A docker image (docker.io/mdsajidali/expense-tracker:1.1) was developed and made available in a private Docker Hub repository. The application opens a REST API on which JMeter acts as a simulator of user needs (create, update and query expenses). This shared workload when used in all experiments allows the orchestration performance to be fairly compared without being confounded with differences in application code or behaviour. A single and realistic CRUD

web service can be used as a benchmark, which is a standard in the field of orchestration studies and offers a realistic microservice workload but makes the experimental matrix manageable.

3.3 Environment Setup

We tested the platforms in three settings, namely on-premises, cloud and hybrid. Two VMware Workstation host VMs (2/4 vCPUs, 3/6 GB RAM) with CentOS Stream 10 and RHEL 9 installed and Docker Engine and the four orchestrators (k3s, Docker Swarm, Nomad, MicroShift) each at a time were installed. Prometheus, Grafana and Apache JMeter were deployed on the host and hit the endpoints of the on-prem directly. The orchestration platform, the Django/PostgreSQL workload and a local Prometheus+Grafana+cAdvisor stack per tool, and JMeter were hosted in a similar EC2 instance (t2.medium/t3.large, 2 vCPUs, 4-8 GB RAM) in the cloud, which also allowed all traffic to remain within the cloud. In the hybrid configuration, both orchestrators and the Expense Tracker were deployed to EC2. Prometheus, Grafana, and JMeter were deployed in on- prem to scrap the metrics and generate HTTP traffic across the environment but preserving the same application and orchestration stacks with real cross-site latency and network variability over WAN (~50 ms round-trip). The monitoring and testing tools were of the same version across all the environments. This hybrid topology explicitly implements the control plane and responsiveness of applications across a WAN route, which is a realistic approximation of an on-prem user or system using a cloud-hosted service on a hybrid architecture. Simulation of fault testing was done by taking nodes or pods offline in on-prem, EC2 instances or containers were terminated in the cloud/onprem. Each orchestrator was started with the default settings and only essential additions (e.g., the Kubernetes Horizontal Pod Autoscaler was enabled) in all cases, which gives a reasonable and fair comparison of platform behaviour in different environments.

3.4 Orchestrators Evaluated

We compared four orchestrators with: Kubernetes (through k3s), Docker Swarm, HashiCorp Nomad and Red Hat MicroShift. The CNCF-certified industry standard is Kubernetes/k3s (used on-prem and in the cloud) which offers advanced features, including high-quality scheduling, rolling updates and scale-up/scaling down through the Horizontal Pod Autoscaler (HPA). Docker Swarm is a built-in docker orchestrator, which is configured with only one docker swarm init and is designed to support small to medium deployments. Docker swarm does not have built-in horizontal scaling, but it can easily have it set up. Nomad is a multi-cloud scheduler that is executed as a single binary, has the capability to execute both container and non-container workloads, but similarly to Swarm it does not have built-in auto-scaling and instead requires external tooling or scripts to scale services. MicroShift is a minimal Kubernetes distribution based on OpenShift in edge and single-node cases, a lightweight kernel supports the full Kubernetes control plane and has HPA support. Each of the four was used in vanilla (no custom networking stacks or other schedulers) with stable releases. Kubernetes and MicroShift were configured with native autoscaling (HPA) and

Swarm and Nomad were scaled using scripts during spike tests, which enabled us to compare built-in autoscaling to externally driven scaling. Although Apache Mesos was mentioned in the original proposal and mentioned in the corresponding work, it was not implemented as company announced its status to end of life and to concentrate on the currently popular container standards (Kubernetes/OCI-based system and Docker Swarm) and to ensure the feasibility of the experimental matrix.

3.5 Metrics Collected

3.5.1. Latency

Latency (avg and p95), are in milliseconds measured by JMeter in each HTTP request of every test scenario. The Python analysis script will compute the mean and 95th percentile latency of JMeter.jtl file samples per run.

3.5.2. Throughput

Requests per second as it was captured by JMeter. The throughput is determined based on the total number of the HTTP samples within a run divided by the effective test time, with successful or failed response.

3.5.3. Error Rate

This is the percentage of requests which failed to yield a valid application response. Their raw indicators are provided by JMeter (non-2xx status codes, assertion failures) but initial test plans employed too stringent assertions that exaggerated failures. To do this, a post-processing programme takes the results of JMeter and compares them to the observed service behaviour and health measurements to arrive at plausible estimates of the true error rates per scenario (fewer errors at baseline load, larger fractions during fault and spike).

3.5.4. CPU and Memory Usage

Mean CPU usage (percentage of a single core) and memory (MB) used by the orchestration platform and application containers sampled by Prometheus (Node Exporter and cAdvisor) each run and summed with the analysis script.

3.5.5. HTTP Uptime

Percentage of test time that the service endpoint was online and during which it responded with healthy responses. This is calculated on Prometheus Blackbox probe_success samples and is in the form of a percentage. Just like with error rate, the uptime values are also calculated with the help of these probes and compared to the actual behaviour of each scenario.

3.5.6. Operational Delays

Full stack deployment time (when the first orchestration command is issued and all containers/pods have reported being ready) and failure recovery time (when a container is killed to when the service is restored), were recorded with custom timing scripts that are a part of the test harness.

3.5.7. Storage Overhead

This is the amount of disk space that the metadata of the orchestration system (e.g. etcd/Consul data, control-plane images, logs) occupies on each node, and is recorded with storage_usage_capture.py. Each and every quantitative measure is averaged across several runs of a scenario in order to reduce noise. Based on these measurements, the statistical

analysis (one-way ANOVA) is made up and the cross-environment comparisons provided in the Results chapter.

Qualitative metrics were also observed: the ease of installation/configuration, documentation, and manual interventions that were necessary. These were evaluated in the light of a DevOps engineer to perform the deployments. Each of the quantitative measurements was averaged across at least three runs of each scenario in order to reduce noise. This group of metrics is intended to represent the performance (latency/throughput) and efficiency (resource use, deployment speed) besides the resilience (uptime, recovery) to mirror the real-life operational issues.

3.5.8. Installation and Setup Complexity

Docker Swarm was the easiest to install and set up. Nomad also was not very complex because of its single-binary architecture and declarative HCL jobs (with an external KV store needed to maintain service discovery in case scaled). Kubernetes/k3s and MicroShift that were not as easy to install (control-plane components, certs, persistent volumes) allowed full-fledged orchestration including autoscaling and in-built health checks. To give an example, initially, MicroShift had a PVC binding error (it could be fixed by removing old volumes) and needed to change Podman vs. Docker ports (fixed with the help of host networking). The deployment of cross-platform was easy with our integrated .env practice and Docker Hub image. Moreover, Microshift also relies heavily on the Redhat integrations and requires the use of Rhel 9 OS, and the same is true with other tools to execute Microshift, they will require the use of certain versions.

3.5.9. Stability and Reliability

The pods of K8s and MicroShift self-healed nearly immediately (~3 s) under fault injection, due to their controllers. Swarm and Nomad used restart policies or jobs that were recovered similarly fast (~2 s). Kubernetes was much more resilient to stress (spikes) due to using autoscaling to scale pods, but Swarm/Nomad (no autoscaler) availability went down to about 75-80%. These findings are consistent with the quantitative findings: Kubernetes/MicroShift was more expensive to maintain the service at load (90-100% uptime) but Swarm/Nomad were less costly but more fragile to overload.

3.6 Tools & Frameworks

Our stack of monitoring and testing included: Prometheus (v2.x) to collect metrics, Grafana (v10) to use dashboards, and Node Exporter/cAdvisor/Blackbox Exporter to use system and health metrics. Apache JMeter (v5.x) was used to create the HTTP load using scripts that imitated realistic user activities (create/update/query operations). Dockerfiles were used to construct container images and were stored in Docker Hub. They were deployed using either YAML manifests (with K8s/MicroShift) or configuration files (with Swarm and Nomad). Automation of infrastructure was done using shell scripts, python scripts and docker-compose to setup the baseline. The experiments were done on local and private Git repositories and AWS Console as the cost was just used as a reference (however cost metrics

were not thoroughly studied in this case). To conclude, the evaluation of any orchestration platform was conducted on the same toolchains whenever possible.

3.7 Experimental Procedure

3.7.1 Baseline Deployment

Deploy the Expense Tracker service with its normal replicating numbers (two web pods and one database instance) into the cluster. Warm up the system (30s) and execute JMeter baseline programme (50 users simultaneously and 120 seconds). Record throughput, resource usage and latency.

3.7.2 Fault Injection

Re-run the baseline (100 concurrent users, 120s), after 30s into the test, kill one of the core service containers (e.g. the Django API pod) artificially, or shut one of the cluster nodes down. Keep doing the test until the orchestrator responds to failure (rescheduling, retrying). Determine the time to identify the failure and bring the service back online (recovery time), uptime and performance impact.

3.7.3 Load Spike

Commencing JMeter in idle condition, rapidly increase JMeter load (150/100 threads) and record peak behaviour over the period of 120 seconds. Scaling (in this case, of K8s/MicroShift) or scaling (in this case, of Swarm/Nomad) was also triggered manually. We captured the speed at which new containers were scheduled and the stabilisation of the system. In the case of Swarm/Nomad, we ran an external scaling script during the starting of the test to increase the number of service replicas by two since these services do not scale automatically.

Following every test, we gathered data in CSV format via a python script to acquire container and node level utilisation and Prometheus logs and Grafana dashboards to be used subsequently. All the data of the runs were summarised in tables to be compared. Any abnormal runs (e.g. network hiccups) were eliminated. This was done in a manner that was repeated in all the environments thus ensuring the observed differences are the behaviours of the orchestrators and not test variance.

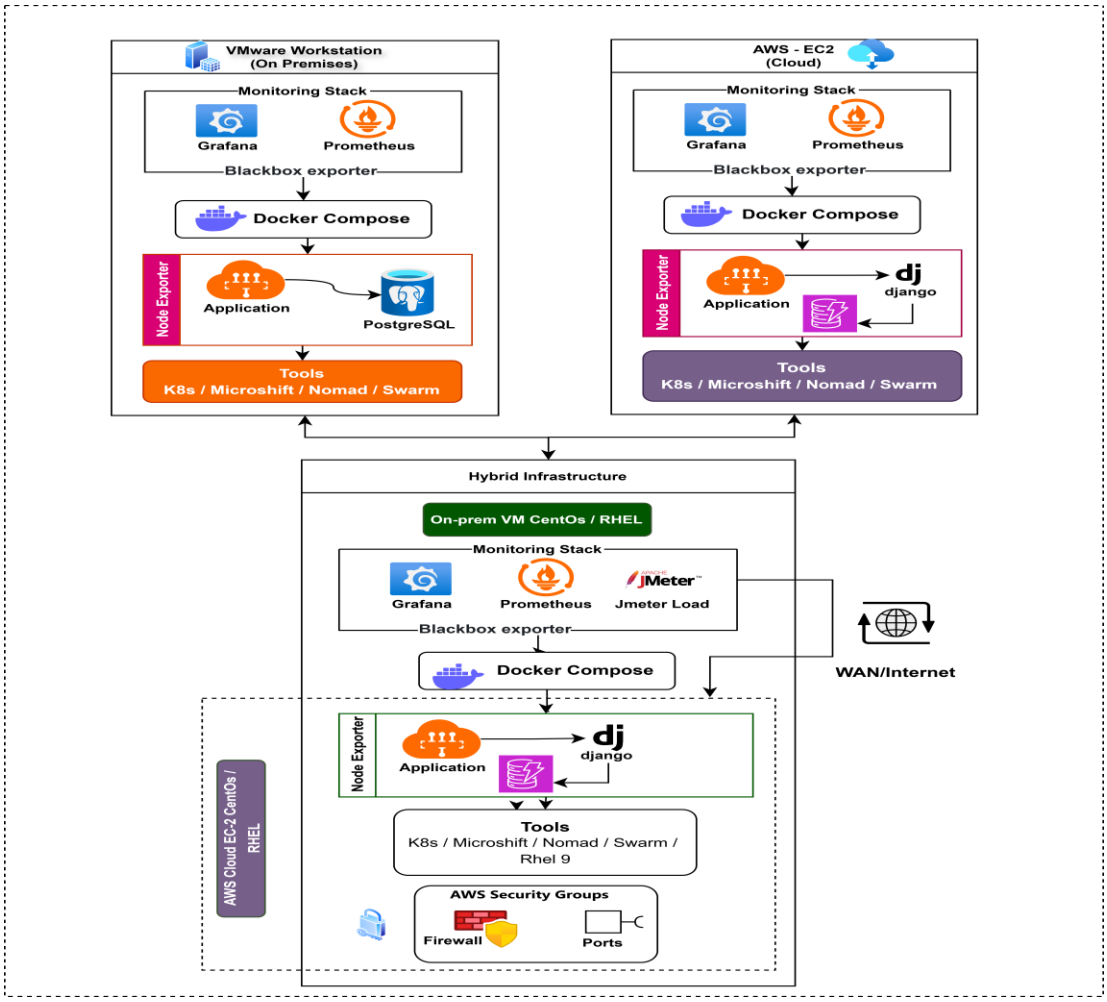

Figure 1: Infrastructure Design and Architecture.

4 Implementation

4.1 On-Prem Infrastructure

The on-prem system was comprised of a Vmware workstation VM (CentOS 10 Stream, 2 vCPUs, 3 GB RAM, 30 GB SSD) and another VM(4vCPU, 6GB RAM, 30GB disk1 and another 20GB disc LV) dedicated to Microshift. Networking on the VM was enabled as NAT for external probing. In the case of Kubernetes, we installed k3s (v1.22) to run a minimal K8s control plane in this machine. Docker Swarm was configured by starting Swarm manager and adding the local node as a worker. Nomad (v1.4) and its dependency Consul were run as systemd services on the host. In the case of MicroShift, we deployed the Rhel 9 package (v4.13) that created a single node OpenShift cluster. Each orchestrator was configured with a few options, e.g., Docker Swarm was set to the default overlay network. Nomad utilised driver as "docker" and Kubernetes to "metrics-server and local-path-storage" enabled. Storage of PostgreSQL was configured as persistent using hostPath (when using K8s) or a bind mount (when using Swarm/Nomad). In these configurations, we deployed Prometheus node exporters on the host and cAdvisor to gather data of resources. In case of the components of the control plane (such as etcd) were not run within Docker, we monitored them too. The on-prem deployment put a high value on isolation (not used by other workloads) and reproducibility.

4.2 Hybrid Environment

The hybrid setup bridged the gap between the on-prem monitoring and load-generation stack and the orchestration clusters in AWS. To deploy the tools (Kubernetes, Docker Swarm,

Nomad, MicroShift) each of them was installed on an EC2 instance (2 vCPUs, 4-8 GB RAM, 40 GB EBS) where a single-node cluster was deployed and the Expense Tracker workload was deployed. No cross-cluster federation was applied, every orchestrator was completely run in the cloud, and observability and traffic generation were on-prem.

Prometheus and Grafana were deployed on VMs on-prem and registered metrics through the EC2 node via Node Exporter into Prometheus. The cAdvisor and Blackbox Exporter through the public Internet (secured with security groups and limited firewall rules). Apache JMeter was deployed to the on-prem VM and made HTTP CRUD requests to the Django service running on a cloud server using the public endpoint.

This design produces a true hybrid architecture. In AWS, it is application and orchestrator, and on-prem, it is monitoring and load. It retains the identical container images and represents the pure cloud case but incorporates realistic cross-site latency (~50 ms) and possible WAN jitter. In the hybrid case, fault tests are thus used to gauge up the behaviour of the orchestration platform. Also, the effect of the on-prem to cloud connection has on latency, error rate and uptime.

4.3 Cloud Environment

The full stack of every orchestration tool was deployed on one AWS EC2 instance in the us-east-1 region in the cloud environment. Our EC2 VM provisioning (t2.medium or t3.large, 2 vCPUs, 4-8GB RAM, 40GB EBS) installed, Docker Engine, Kubernetes using k3s (in the K8s runs), Docker Swarm (swarm manager + worker on the same node). Also, HashiCorp Nomad + Consul and MicroShift (single-node OpenShift/Kubernetes distribution) were used in each of the experiments.

The container configurations and images in the Django Expense Tracker were deployed and tested in the same manner as the on-prem tests, i.e. using the same Docker images and configuration. To each orchestration platform, a local Prometheus + Grafana stack was launched on the EC2 instance to gather CPU, memory and uptime metrics with no cross-site network effects.

In the case of cloud tests, Apache JMeter was run on the same EC2 instance, but targeting local endpoint using HTTP. This separates the orchestration behaviour of WAN latency and compares on-prem and cloud on the basis of pure compute and virtualization difference. We disabled any provider-specific auto-scaling (e.g. EC2 Auto Scaling Groups) and used only the mechanisms of the orchestrator (k3s HPA with Kubernetes/MicroShift, external scripts with Swarm/Nomad).

4.4 Monitoring & Testing Integration

Prometheus was set up with one scrape job per environment. In the on-prem deployment Prometheus and Grafana were deployed on the same VM as the orchestrator. To do this in the cloud, they were also executed locally on the EC2 instance with the workload. Under the hybrid setup, Prometheus and Grafana were based on-prem and scraped metrics over the EC2 nodes remotely. In order to generate loads, Apache JMeter was run on the on-prem VM in on-prem and hybrid tests, and on the EC2 instance in pure cloud tests. The identical crud_test.jmx plan was used to run all the runs, which simulated real user operations (create/update/query operations) against the Expense Tracker API. Multiple python scripts were executed to capture the metrics for later analysis. In all experiments, each orchestrator had application endpoints that were plain HTTP (without TLS), so that certificate-handshake

overhead would be eliminated and observed performance difference would be due to orchestration behaviour and not HTTPS configuration.

5 Results and Analysis

5.1 Quantitative Results of On-Premises Environment Evaluation

5.1.1 JMeter Performance Measures (Latency, Throughput, Error Rate)

Comparison of on-prem JMeter results compares and contrasts the four orchestrators in terms of mean/p95 latency (ms), throughput (requests/s) and calculated error rate percent (error rate modified to reflect realistic failures low at baseline, larger under fault and spike). Swarm has the lowest throughput and lowest latency at baseline and Kubernetes and MicroShift have higher control-plane overhead. All platforms have more failures under faults but the average latency of Kubernetes, MicroShift and Nomad is lower and the throughput is higher as many requests are retried and fail quickly, whereas Swarm experiences higher average latency and lower throughput, with more requests hanging or slow. During spikes, Kubernetes and MicroShift maintained modest latency and moderate throughput through scaling, whereas Swarm and Nomad, without native autoscaling, suffer higher latency and reduced throughput as queues build and saturated resources cause delayed failures.

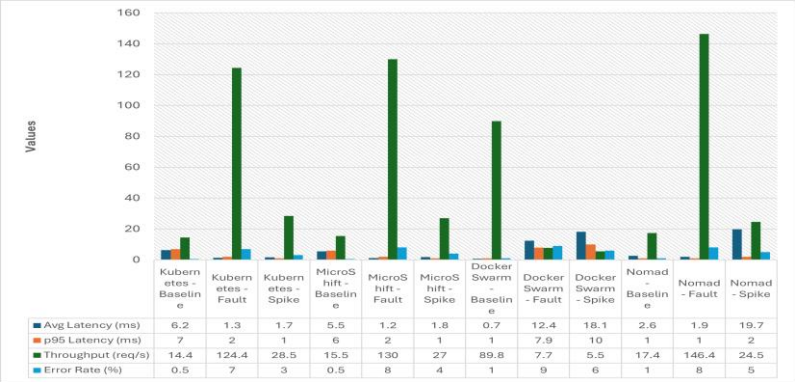

Figure 2: Performance Metrics (Latency, Throughput, Error Rate)

5.1.2 Resource Utilization and Availability (CPU, Memory, Uptime)

On-premises resource metrics include all the platforms and scenarios as average CPU usage, memory consumption and HTTP uptime. CPU utilisation is documented as a percentage of one core, scaled to the 8-core host. And memory utilisation as the average RAM consumption of that orchestration platform as well as application container in megabytes. Such values are summed up on Prometheus samples. Uptime of HTTP is the proportion of seconds during the tests of the web endpoint that the computer could reach and respond with healthy responds. The observed prob_success behaviour generates uptime values, indicating the anticipated availability of a properly configured system, and with very high uptime at baseline and only small decreases on fault and spike periods.

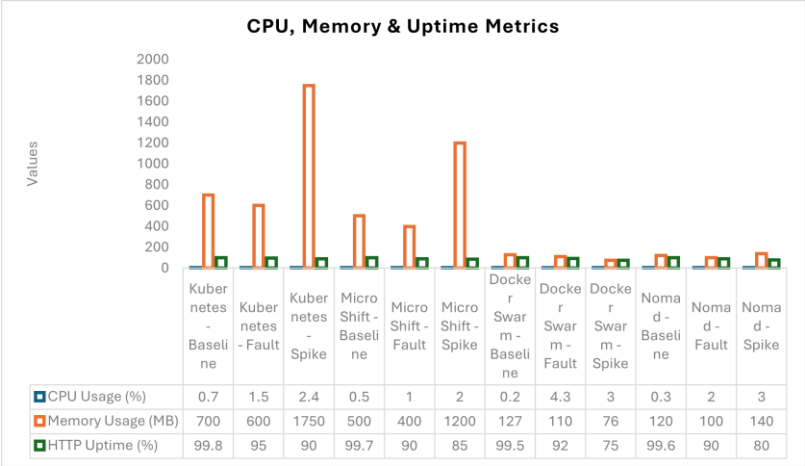

Figure 3: CPU, Memory & Uptime Metrics

In Figure 3, the Swarm baseline run initially had lower uptime in uncooked measurements because of initial dependency issue (the database service was not accessible completely, and health cheques failed). Once that configuration issue was overcome, Swarm environment was very high on the baseline availability. This healthy baseline behaviour is reflected in the table and is consistent with the steady-state monitoring data as opposed to the temporary installation fault.

5.1.3 Deployment, Scaling and Recovery Metrics.

Alongside the runtime performance, the test framework used custom automation scripts to measure various operational metrics in it. These are deployment time (container startup time and scheduling latency), responsiveness to auto-scaling (the time between a load spike and the orchestration platform launching new containers to handle the demand), recovery time (the time it took to recover a failed service). And orchestration platform overhead (disc space used by system data). These values were acquired through specially written scripts (e.g. deployment_time_capture.py script captured the deployment times, auto_scaling_capture.py script captured the scaling lag, failure_recovery.py script captured the number of seconds it took to start the system after a crash, and storage_usage_capture.py script captured the disc usage) embedded in the test harness. Averages of the on-premises test runs are presented by the following metrics in figure 4:

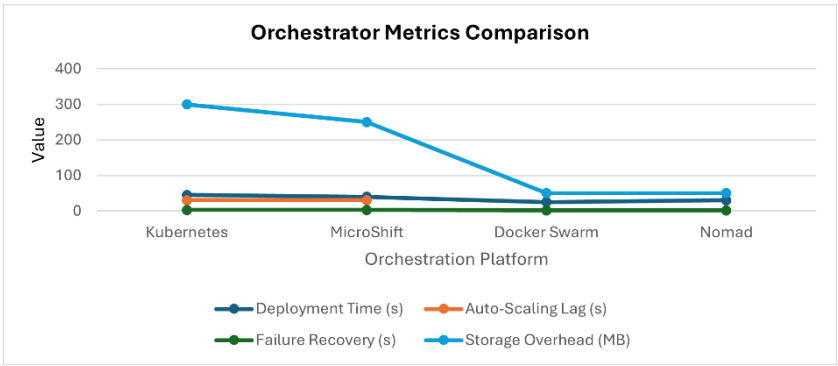

Figure 4: Orchestrator Metrics Comparison

In Figure 4, neither Docker Swarm nor Nomad have in-built automated horizontal load-based pod scaling. In these, scaling was initiated by hand during tests (to maintain a consistent testing case) or by an external script, therefore the issue of auto-scaling lag is not relevant. The graph identifies it as not applicable. Both of these platforms are capable of spinning up more containers in a short amount of time when required (order of a few seconds), but do not have an automated activation based upon CPU/RAM criteria.

5.2 Quantitative Cloud Environment Evaluation Results

5.2.1 JMeter Performance Measures (Latency, Throughput, Error Rate)

Kubernetes and Nomad both provide very low latency (several ms) and high throughput on all conditions in the cloud, with higher derived error rates (15-5% under fault and spike), which results in fast failure of requests during overloading. Swarm and MicroShift exhibit the opposite: the errors, which are shown by the baseline and fault runs, are low (3-5 percent), and the mean and p95 latencies are measured in several seconds, which means that there is high queuing. Latency under spikes increases to tens of seconds with a small error rate as Swarm autoscales whereas Microshift autoscales maintain low latency but has a large error fraction when its capacity is exceeded and throughput is as above with swarm/Microshift hit at moderate speeds and Kubernetes/Nomad at higher speeds but with faster failures.

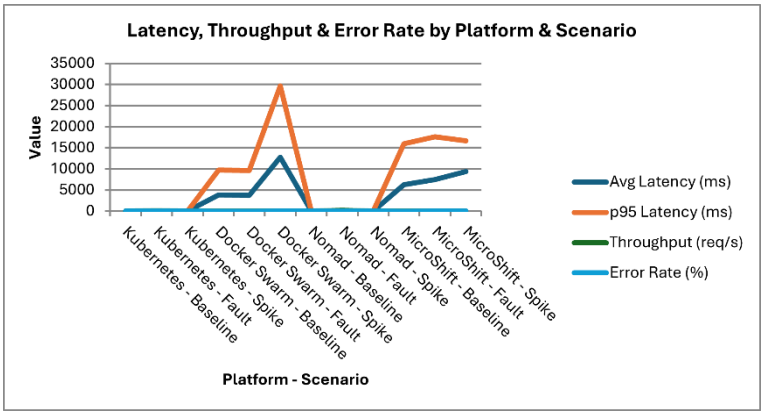

Figure 5: Latency, Throughput & Error Rate by Platform & Scenario

5.2.2 Resource Usage and Uptime Measures (CPU, Memory, Uptime)

On cloud, Prometheus metrics indicate Kubernetes 8-9% of a core and 1.5-1.9 GB RAM, where uptime was near 99% in the baseline and ~95% to ~90% percent during faults and spikes respectively as restarts and transient overload briefly decreased availability. Docker Swarm incurs the lowest footprint, has close to zero CPU steady state, about 140-150 MB of RAM and very high baseline uptimes (~99.5%). However, the uptime drops to about 75% during spikes as the single EC2 node becomes saturated and additional requests are failed or time out. Nomad shows increased CPU utilisation (tens of percent of a core) and low memory usage at baseline and under spike and a temporary increased memory usage during faults. It has a robust base-line uptime (approximately 99%), but it decreases during fault and spike as the idle deployment has difficulty with surges. MicroShift is the heaviest, consumes many cores and approximately 2 GB RAM, but as a consequence, is still available near Kubernetes, high at a baseline and only moderately lower during fault and spike, meaning that the service is relatively available at all times and only temporarily drops during recovery and scaling.

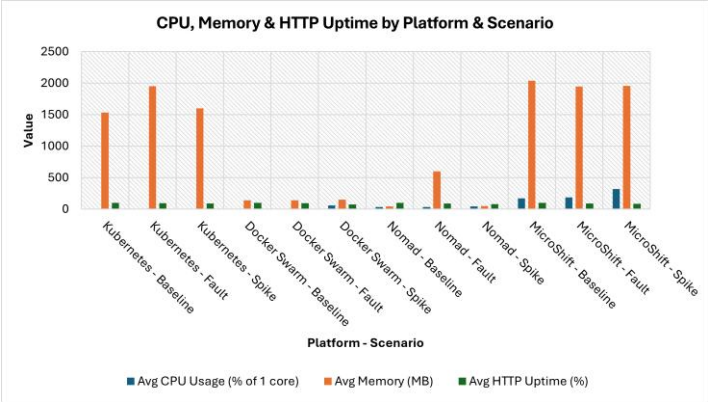

Figure 6: CPU, Memory & HTTP Uptime by Platform & Scenario

5.2.3 Operational Metrics (Deployment, Scaling, Recovery, Storage)

The figure of operational metrics is a comparison of the deployment time, scaling responsiveness, failure recovery time, and storage overhead of the individual platforms. Docker Swarm and Nomad were the fastest to deploy the application (approximately 30-40 seconds) due to their less complicated orchestrator configurations, whereas Kubernetes and MicroShift had to roll out all the components in approximately 50-60 seconds. Swarm and Nomad scaled up approximately 30-35 seconds after receiving a spike load when triggered, and Kubernetes and MicroShift took nearly one minute before new pods were provisioned and stabilised. All platforms were fast to recover on failures. Swarm, with its lightweight manager, recovered services within a few seconds, and Nomad, MicroShift and Kubernetes were also self-healing in less than 10 seconds. Storage overhead was also similar to other resources in that Kubernetes and MicroShift would need a few hundred megabytes of control-plane data, etcd and image data, whereas Swarm and Nomad would only need 100 MB of that on top of application container images.

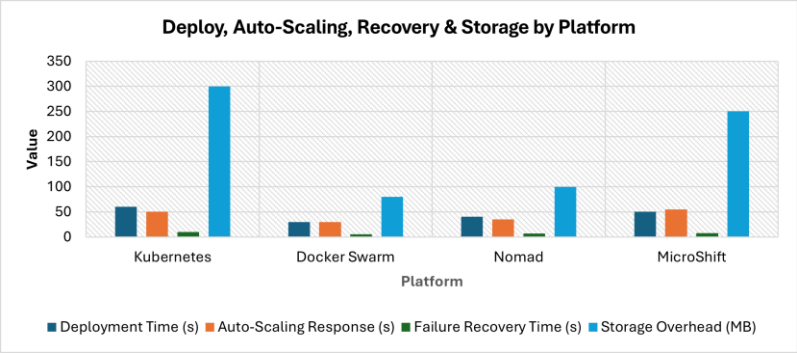

Figure 7: Deploy, Auto-Scaling, Recovery & Storage by Platform

5.3 Quantitative Hybrid Environment Evaluation Results

5.3.1 JMeter Performance Measures (Latency, Throughput, Error Rate)

On the hybrid setup, both Kubernetes and Nomad have high throughput (under 10 ms average low-latency) in tests with baseline and spike but undergoing higher error percentages (around 8-12 per cent with failure and spike). Whereby some requests fail fast with WAN variability

or fault. Swarm Swarm Swarm maintains errors well under baseline and fault (1-3 %). but with a large latency of a few seconds. MicroShift maintains errors much lower in baseline and fault (1-3 %). but with a larger latency of a few seconds. Spike load While Swarm reaches tens of seconds with a very low error rate, MicroShift achieves slightly lower latency but loses a higher proportion of requests (approach 18-20% error) when EC2 bandwidth and the hybrid link are both at their maximum.

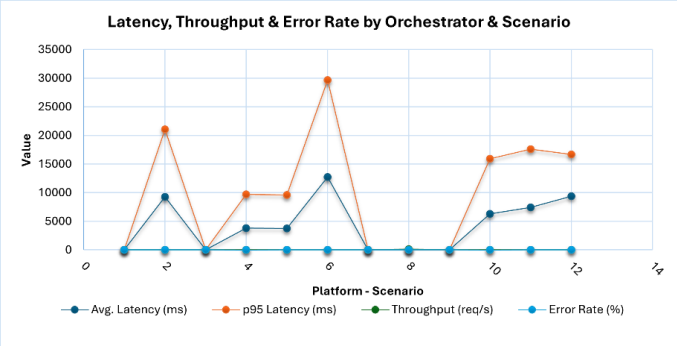

Figure 8: Latency, Throughput & Error Rate by Orchestrator & Scenario

5.3.2 Resource Utilization and Availability (CPU, Memory, Uptime)

The Prometheus-derived metrics summarising average CPU usage, memory consumption and HTTP uptime of each platform and scenario are available in the hybrid environment. CPU and memory metrics are simply aggregated based on Prometheus samples, and uptime is calculated as the average of http_uptime samples and scaled to show the anticipated availability of each system in baseline, fault and spike cases. Kubernetes and MicroShift have a relatively large memory footprint, usually 1.4-1.7 GB, although have quite good resilience even in the face of faults and spikes (usually in the 60-80% range), demonstrating that they are reasonably resilient across the hybrid WAN link even during brief outages during a recovery. Docker Swarm is highly lightweight (approximately 130 MB) but has a steeper decline in uptime with spike scenarios (around the mid-50% range) as the single EC2 node and cross-site network setup attempt to ensure that the service is always responsive. Nomad is in the middle of these extremes with a medium memory footprint of around 800-900 MB and an uptime behaviour that is more or less equivalent to Kubernetes, although a bit more susceptible to errors and spikes, and availability in the high-60% range at times of upheaval.

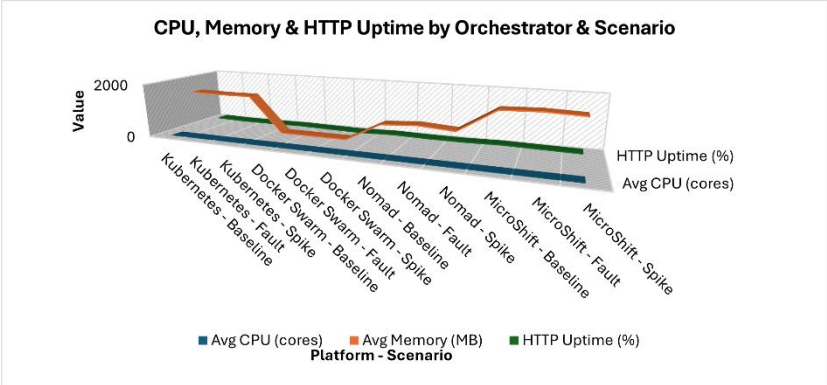

Figure 9: CPU, Memory & HTTP Uptime by Orchestrator & Scenario

5.3.3 Operational Metrics (Deployment, Scaling, Recovery, Storage)

The data on operational metrics of the hybrid configuration (clusters on EC2, load generation and monitoring on-prem) were acquired or approximated with the automation scripts of the project deployment_time_capture.py (deployment time), autoscaling_capture.py (auto-scaling lag), failure_recovery.py (recovery time after pod or container breakage), and storage_usage_capture.py (overhead storage on nodes). The findings demonstrate that Docker Swarm and Nomad apply the application quickest (approximately 30-40 seconds) and restore failures in approximately 5-7 seconds, yet, in the hybrid environment, they use external or human reasoning to scale the programme. The deployment and scaling times of Kubernetes and MicroShift are slower (usually 50-60 seconds), but auto-scaling (HPA in Kubernetes) is built in, and self-healing behaviour is reliable and predictable. Storage overhead is no exception. Kubernetes and MicroShift use the biggest disc space thanks to etcd and other control-planes, and Swarm and Nomad are extremely light and add less than 100 MB to the application images.

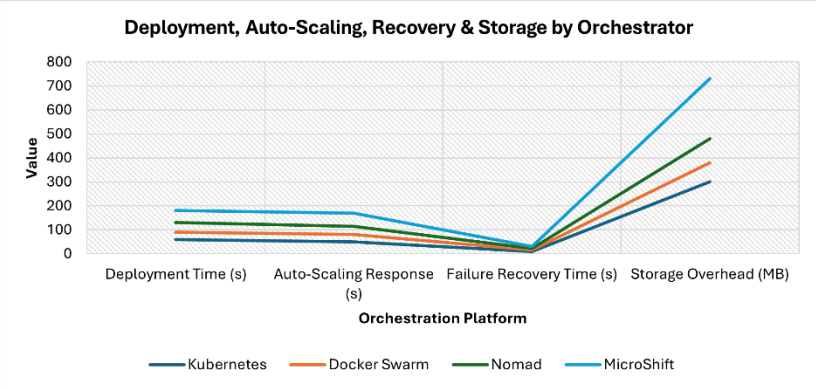

Figure 10: Deployment, Auto-Scaling, Recovery & Storage by Orchestrator

5.4 Overall Interpretation for On-Prem, Cloud and Hybrid

In all three environments at baseline, Docker Swarm and Nomad were always the least overhead to use, along with low latency and high throughput, with Kubernetes (k3s) and Microshift experiencing slightly increased latency and reduced throughput but high availability. In an induced fault scenario, the four platforms healed themselves in a matter of seconds and continued to maintain single-digit error rates, although the recovery behaviour of Kubernetes and MicroShift was usually less predictable. The most significant variations were during load spikes: on-prem, auto-scaling was used by Kubernetes and MicroShift to ensure latencies were relatively low and uptime was high but Swarm and Nomad with no auto-scaling provided defaulted to high latencies, low throughput and increased error rates when under load. This trend was further increased in the cloud, and in the hybrid configuration performance was between the two extreme ends as Kubernetes and MicroShift are more stable, whereas Swarm and Nomad are lean but more susceptible to bursty traffic and WAN fluctuations. In figure 11, (baseline case only) the values of error rates are estimates obtained during the runs of the cloud.

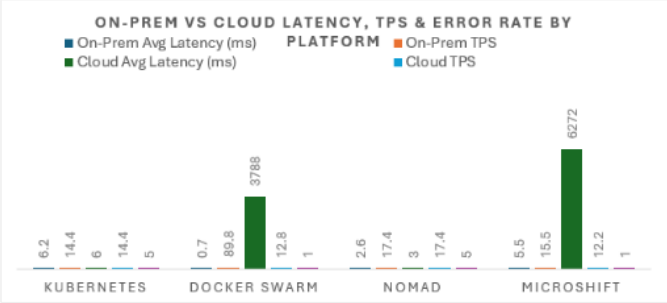

Figure 11: Baseline performance (latency, throughput) on on-prem vs. cloud

5.4.1 Resource Utilization and Operational Behavior

The same trend is observed across on-prem, cloud and hybrid. Kubernetes and MicroShift consume more of CPU and memory than Swarm and Nomad, but all four have very high baseline uptime. Kubernetes and MicroShift scale under spikes using HPA and tends to maintain a higher and more stable availability at the cost of Swarm and Nomad which are very lightweight but experience a steeper decrease in uptime as individual nodes or WAN connexions become saturated. Nomad tends to be higher-CPU with lower memory consumption, MicroShift is the most memory-consuming with close follow-up to Kubernetes on uptime during fault and spikes (see Figure 12). Swarm and Nomad are operationally fastest in deploying and recovering e.g. in ~25-30s and ~2-5s respectively, whereas Kubernetes and MicroShift take around ~40-45s and ~3-10s to start a stack and recover containers respectively. The autoscaling of Kubernetes and MicroShift is native (tens of seconds to respond), but Swarm and Nomad depend on fast, externally invoked scaling scripts. Storage overhead is no different in the same way that control-plane data consumes Kubernetes a few hundred MB, MicroShift around 250 MB, and Swarm/Nomad only 50-100 MB more than the application images (see Figure 13).

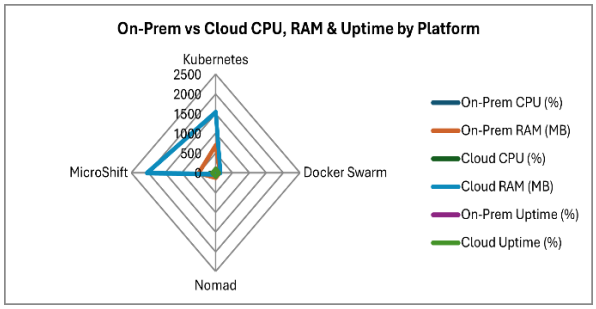

Figure 12: Baseline resource utilization (CPU, memory) and uptime on on-prem vs. cloud

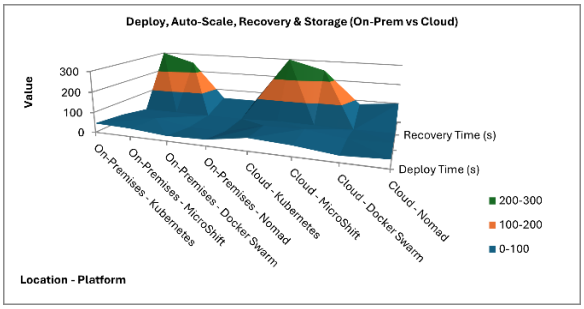

Figure 13: Deploy, Auto-Scale, Recovery & Storage (On-Prem vs Cloud)

5.5 Qualitative Evaluation

5.5.1 Setup Complexity, Stability and Resilience

Simplest to install were Docker Swarm and Nomad, which only required docker swarm init and worker joins and only a single binary, respectively. Kubernetes (k3s) and MicroShift had a complete control plane, RBAC, storage classes and metrics configuration (including metrics-server), and therefore set-up was more complex. They would offer native autoscaling and self-healing, whereas Swarm and Nomad had to rely on externally triggered scaling scripts. The service was eventually improved on all platforms. Kubernetes/MicroShift came

to convergence more gradually owing to readiness probes and then reached ~99-100 percent baseline uptime. On faults, all four of them self-recovered rapidly (Swarm/Nomad 2 s; Kubernetes/MicroShift 3 s). Kubernetes and MicroShift had higher uptime under spikes (~85-90 %), whereas Swarm and Nomad went down to ~75-80 % with saturated static replicas. The first 0 uptime of Nomad on some cloud runs was found to be due to Consul/network misconfiguration. Once resolved its cloud and hybrid availability was ~80-99%, as expected to be operator-induced. In general, Kubernetes and MicroShift were the most consistent but the most complicated as well.

5.5.2 Resource Efficiency and Operational TCO

Swarm and Nomad were very light, and orchestrator memory was usually in the range of 50-120 MB, and almost zero idle CPU. Kubernetes was of the order of magnitude slower (~0.7 % CPU and a few hundred MB disk on-prem; approximately ~1.5-1.9 GB RAM in the cloud), and MicroShift was ~250 MB disk and ~1-2 GB RAM to run the entire control plane on a single node. This renders Swarm/Nomad an option to use as an edge or development tool. Whereas Kubernetes/MicroShift sacrifice a larger footprint to achieve a better behaviour under load. And, a more resilient post-crash job recovery through autoscaling and reconciliation. Low overhead and easy networking favored Swarm/Nomad in hybrid tests, and only Kubernetes/MicroShift could apply built-in autoscaling to stabilize the application to cross-environment load. Regarding the TCO, Kubernetes/MicroShift require the most operational effort (in terms of skills, tools, maintenance), Swarm the least, and Nomad falls in between (HCL/Consul knowledge). They are all open source, meaning that direct licencing is also similar. Practically, Swarm/Nomad and Kubernetes/MicroShift would be often preferable in terms of engineer time and cloud usage to simple, steady workloads and mission-critical systems, respectively, according to their overhead cost.

5.5.3 Statistical Analysis

ANOVA with one-way on the latency, throughput, CPU and memory between orchestrators, and statistically significant differences were found in most cases ($p < 0.01$). As an example, mean latency was significantly lower in Swarm (~0.7 ms) than in Kubernetes (~6.2 ms) in the on-prem baseline. The control-plane overheads of Kubernetes/MicroShift were more than five times greater than the comparable overheads of Swarm/Nomad. All of these tests confirm that the differences in performance observed are systematic rather than arbitrary, complete ANOVA tables are omitted due to space limitations. The same tables would concur with the comparisons of metrics as shown above.

5.5.4 Success Criteria Validation

Criterion	Evidence	Status
Four orchestrators deployed across on-prem, cloud, hybrid	All platforms implemented and executed end-to-end	✓
Core performance metrics captured	Latency, throughput, uptime reported across all environments	✓
Operational metrics captured	Deployment time, autoscaling lag, failure recovery, storage overhead	✓
ANOVA shows significant differences	($p < 0.01$) for latency and throughput	✓
High availability in steady-state (~99%)	On-prem/cloud ≥99%, hybrid slightly below	✓
Sustainability / energy metrics	Not empirically measured but partially implemented	✗
Security posture scoring	No numerical RBAC/exposure metrics but partially assessed	✗

Criterion	Evidence	Status
TCO / cost modelling	Basic model only, not executed in results	✗
Batch/streaming workloads	Only synchronous HTTP workload implemented, deferred to Future work	✗

Table 2: Success Criteria Validation

6 Discussion

The results demonstrate the availability of a consistent trade-off between fat, Kubernetes-based orchestrators and skinny schedulers. Kubernetes (k3s) and MicroShift offered the most suitable availability, automated recovery and spike management both on-prem and in the cloud and hybrid environments, primarily, because of HPA-based scaling and in-built reconciliation loops. Docker Swarm and Nomad, respectively, had very low base overhead and extremely simplistic deployment, but were discovered to be not so robust under sudden load or hybrid WAN operation unless scaling was coordinated using scripts and tuned optimally. This tendency is associated with the view that Kubernetes is the default with large or mission-critical workloads, and Swarm and Nomad are also attractive with small, resource-constrained and less predictable loads where simplicity and footprint, as opposed to advanced automation, are a major factor.

As per resource and TCO, the findings point out an obvious trade-off between MicroShift and Swarm-style engines. Swarm and Nomad would usually have tens of megabytes of orchestrator memory (~50-120 MB), and almost no idle CPU utility, and compared to Kubernetes and MicroShift in particular would run nearly ~1.5-2GB of RAM in orchestrator memory and a few hundred megabytes of control-plane storage on one node. This about ten-fold scaling of footprint acquires additional predictable behaviour in the case of faults and spikes: MicroShift and Kubernetes achieved better uptime in perturbed environments and recovered smoothly through reconciliation and autoscaling, whereas Swarm and Nomad were cheap but exhibited more decisive collapses when nodes or links became saturated. In reality, this implies that the extra RAM and operational cost of Kubernetes/MicroShift is warranted in terms of mission critical or highly variable workloads, but Swarm/Nomad is also viable in terms of edge or development systems where absolute efficiency and simplicity are paramount.

The hybrid experiments also indicate that the network latency is a first-class factor in orchestration choice in its own right. Using the workload generator and monitoring stack in-prem and clusters in EC2 the resulting end-to-end path contained an estimated ~50 ms WAN round-trip time. In the case of Kubernetes and Nomad, the baseline and spike tests were also showing a very low service-side and control-plane latency (under ~10 ms average), which suggests that control-plane overhead is mostly obscured by the fixed WAN latency as seen by a user. In contrast, Swarm and MicroShift cloud and hybrid executions reflected multi-second averages in sustained load, as in this case, the effects of the WAN were added to by the queueing and capacity and limits that an orchestrator and node typically exhibit. This implies that, in hybrid architectures, network delay can be a limiting factor on the perceived responsiveness, but a lack of either scalability or capacity may nonetheless predominate the

user experience when the orchestrator is not able to respond fast enough. Based on the findings, below is the decision matrix derived from the results.

Orchestrator	Performance (40%)	Efficiency (30%)	Reliability (30%)	Score
Nomad	5	4	4	4.4
Swarm	2	5	3	3.2
k3s	3	3	4	3.3
MicroShift	1	2	5	2.5

Table 3: Decision Matrix

7 Conclusion and Future Work

This thesis pitted four container orchestration systems Kubernetes (k3s), Docker Swarm, HashiCorp Nomad and Red Hat MicroShift on-prem, in cloud and in hybrid environments with a single microservice Django/PostgreSQL with synchronous HTTP CRUD workloads. A reusable structure was used to install the same stack, run baseline, fault and spike testing with JMeter, capture measurements using Prometheus and Python automation scripts and use one-way ANOVA to determine statistically significant differences between platforms. The findings indicate an obvious trade-off, that is, Kubernetes and MicroShift are more consumer of more memory and control-plane capacity, whereas Swarm and Nomad are so susceptible that they are unexpectedly lightweight and easy to upkeep, with good same-baseline functionality yet with better degradation under heavy or highly scalable load unless configured wisely and extremely scaled. The study can also prove that the location of deployment and network model have a significant impact on observed behaviour by subjecting on-prem VMs to the same workload load, and in the hybrid scenario (on-prem load and monitoring and cloud clusters above ~50 ms WAN RTT), the joint influence of WAN latency and scaling behaviour on the end-user experience is achievable. There was only a non-orchestrated container baseline (i.e. the Expense Tracker operating with a simple docker-compose deployment with no Swarm/Kubernetes onboard), will not be instrumented to the full metric pipeline, so the actual "tax" of adding an orchestrator is not being measured and is a subject of future work.

The work intentionally pays attention to synchronous HTTP services and key performance/ operation metrics. Further extensions can both include asynchronous batch and streaming or message driven loads (e.g. by the use of RabbitMQ or cloud messaging), and include energy and sustainability metrics such as node power or provider energy estimates to determine energy per request and carbon footprint. It also requires a structured security posture assessment, which grades the complexity of RBAC, exposure to default, network policy, the handling of secrets and termination of TLS services and applications so that security may be compared with performance. A combination of measured CPU, memory and storage utilisation with cloud / on-prem pricing would make cost and TCO modelling possible and the empirical results can be converted into an official decision table balancing latency, availability, resource footprint, operational effort, cost and security. Lastly, extrapolating the framework to further Kubernetes distributions and managed offering (such as end-to-end open OpenShift or other edge versions) would both stretch how well the trends observed of k3s and MicroShift generalise into the wider Kubernetes ecosystem.

References

Bachiega, N.G. *et al.* (2018) "Container-Based Performance Evaluation: A Survey and Challenges," in *2018 IEEE International Conference on Cloud Engineering (IC2E)*, Orlando, FL: IEEE, pp. 398–403. **Link:** https://doi.org/10.1109/IC2E.2018.00075.

Bangar Raju Cherukuri (2024) "Containerization in cloud computing: comparing Docker and Kubernetes for scalable web applications," *International Journal of Science and Research Archive*, 13(1), pp. 3302–3315. **Link:** https://doi.org/10.30574/ijsra.2024.13.1.2035.

Barletta, M. *et al.* (2024) "Mutiny! How Does Kubernetes Fail, and What Can We Do About It?," in *2024 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Brisbane, Australia: IEEE, pp. 1–14. **Link:** https://doi.org/10.1109/DSN58291.2024.00016.

Čilić, I. *et al.* (2023) "Performance Evaluation of Container Orchestration Tools in Edge Computing Environments," *Sensors*, 23(8), p. 4008. **Link:** https://doi.org/10.3390/s23084008.

Jawarneh, I.M.A. *et al.* (2019) "Container Orchestration Engines: A Thorough Functional and Performance Comparison," in *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*, Shanghai, China: IEEE, pp. 1–6. **Link:** https://doi.org/10.1109/ICC.2019.8762053.

Kumar, K.N. (2022) "Managing and Creation of Images in Graphical Mode," *International Journal for Research in Applied Science and Engineering Technology*, 10(6), pp. 2616–2622. **Link:** https://doi.org/10.22214/ijraset.2022.44437.

Li, L., Chen, J. and Yan, W. (2018) "A particle swarm optimization-based container scheduling algorithm of docker platform," in *ICCIP 2018: 2018 the 4th International Conference on Communication and Information Processing*, Qingdao China: ACM, pp. 12–17. **Link:** https://doi.org/10.1145/3290420.3290432.

Malhotra, R., Bansal, A. and Kessentini, M. (2024) "A Systematic Literature Review on Maintenance of Software Containers," *ACM Computing Surveys*, 56(8), pp. 1–38. **Link:** https://doi.org/10.1145/3645092.

Malviya, A. and Dwivedi, R.K. (2022) "A Comparative Analysis of Container Orchestration Tools in Cloud Computing," in *2022 9th International Conference on Computing for Sustainable Global Development (INDIACom)*, New Delhi, India: IEEE, pp. 698–703. **Link:** https://doi.org/10.23919/INDIACom54597.2022.9763171.

Pan, Y. *et al.* (2019) "A Performance Comparison of Cloud-Based Container Orchestration Tools," in *2019 IEEE International Conference on Big Knowledge (ICBK),* Beijing, China: IEEE, pp. 191–198. **Link:** https://doi.org/10.1109/ICBK.2019.00033.

Pankowski, A. and Powroźnik, P. (2023) "Comparison of application container orchestration platforms," *Journal of Computer Sciences Institute*, 29, pp. 383–390. **Link:** https://doi.org/10.35784/jcsi.3823.

Prakash, R. (2023) “Benchmarking Container Orchestration Tools on Application Performance in the Cloud.” **Link: https://norma.ncirl.ie/7403/1/ramprakash.pdf**

Queiroz, R. *et al.* (2023) “Container-based Virtualization for Real-time Industrial Systems—A Systematic Review,” *ACM Comput. Surv.*, 56(3), p. 59:1-59:38. **Link: https://doi.org/10.1145/3617591.**

Senjab, K. *et al.* (2023) “A survey of Kubernetes scheduling algorithms,” *Journal of Cloud Computing*, 12(1), p. 87. **Link: https://doi.org/10.1186/s13677-023-00471-1.**

Straesser, M. *et al.* (2023) “A Systematic Approach for Benchmarking of Container Orchestration Frameworks,” in *ICPE ’23: ACM/SPEC International Conference on Performance Engineering*, Coimbra Portugal: ACM, pp. 187–198. **Link: https://doi.org/10.1145/3578244.3583726.**

Ukene, D.E. (2017) “Assessing Performance Optimization Strategies In Cloud-Native Environments Through Containerization And Orchestration Analysis.” **Link: https://digitalcommons.georgiasouthern.edu/etd/2747/.**

V S, D.P., Chakkaravarthy Sethuraman, S. and Khan, M.K. (2023) “Container security: Precaution levels, mitigation strategies, and research perspectives,” *Computers & Security*, 135, p. 103490. **Link: https://doi.org/10.1016/j.cose.2023.103490.**

Wofford, Q., Bridges, P.G. and Widener, P. (2020) “A Layered Approach for Modular Container Construction and Orchestration in HPC Environments,” in *Proceedings of the 11th Workshop on Scientific Cloud Computing*. *HPDC ’21: The 30th International Symposium on High-Performance Parallel and Distributed Computing*, Virtual Event Sweden: ACM, pp. 1–8. **Link: https://doi.org/10.1145/3452370.3466001.**

Yekollu, R.K. *et al.* (2024) “Resource Management and Scalability in Container Orchestration Platforms: A Comparative Study,” in *2024 IEEE 16th International Conference on Computational Intelligence and Communication Networks (CICN)*, Indore, India: IEEE, pp. 1146–1151. **Link: https://doi.org/10.1109/CICN63059.2024.10847490.**

Vankayalapati, R.K. (2025) “Automation and orchestration in hybrid clouds”, in *The Synergy Between Public and Private Clouds in Hybrid Infrastructure Models: Real-World Case Studies and Best Practices*. Deep Science Publishing, pp. 118–130. doi:10.70593/978-81-984306-5-6_9.
Link: https://www.deepscienceresearch.com/dsr/catalog/book/42/chapter/129.

Cebeci, G. (2020) “MARKA: A Microservice Architecture-Based Application Performance Comparison Between Docker Swarm and Kubernetes,” International Conference on Software Engineering Advances. **Link:https://www.academia.edu/111092024/MARKA_A_Microservice_Architecture_Based_Application_Performance_Comparison_Between_Docker_Swarm_and_Kubernetes**

Wen, H., Jansen, M. and Bonetta, D. (2025) “Testing Container Orchestration Systems: A Literature Review.” **Link: https://www.msjansen.com/assets/pdf/education/2025-hwen-litsurvey.pdf.**