

Benchmarking Container Orchestration Tools on Application Performance in the Cloud

MSc Research Project
Cloud Computing

Ram Prakash
Student ID: x22132236

School of Computing
National College of Ireland

Supervisor: Rashid Mijumbi

National College of Ireland
Project Submission Sheet
School of Computing



Student Name:	Ram Prakash
Student ID:	x22132236
Programme:	Cloud Computing
Year:	2023
Module:	MSc Research Project
Supervisor:	Rashid Mijumbi
Submission Due Date:	18/12/2023
Project Title:	Benchmarking Container Orchestration Tools on Application Performance in the Cloud
Word Count:	4878
Page Count:	18

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:	Ram Prakash
Date:	31st January 2024

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	<input type="checkbox"/>
Attach a Moodle submission receipt of the online project submission , to each project (including multiple copies).	<input type="checkbox"/>
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.	<input type="checkbox"/>

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):	

Benchmarking Container Orchestration Tools on Application Performance in the Cloud

Ram Prakash
x22132236

Container orchestration tools such as Docker Swarm and Kubernetes have become popular platforms for deploying containerized workloads. However, little empirical research examines their comparative application performance, especially in the cloud. This research study benchmarks Docker Swarm and Amazon EKS on key metrics including request latency, throughput, and resource utilization for a containerized web application. The methodology deploys a simple React single-page application with Docker Compose to a Docker Swarm cluster and an Amazon EKS cluster, both configured with aws ec2 instances. The application is load tested using Apache Bench to simulate traffic on scales from 1,000 to 40,000 requests. The results of the experiment demonstrate that Docker Swarm yields a lower average request latency between 1,000 and 10,000 requests. However, Amazon EKS provides higher throughput at 20,000 requests and beyond. The higher latency variation in the EKS indicates a less consistent performance. Docker Swarm also exhibits more efficient CPU and memory resource usage under high load. The findings suggest that Docker Swarm may be preferable for applications that require consistent low latency, while EKS can achieve better maximum throughput. Comparative benchmarking provides practical insights for selecting a container orchestration architecture based on performance requirements in the cloud. Future work should evaluate additional metrics and cluster configurations.

1 Introduction

1.1 Background and Motivation

The adoption of containerized architectures has rapidly accelerated over the past decade, driven by benefits such as environment consistency, portability, resource efficiency, and microservices support Moravcik and Kontsek (2020). Containers package an application with lightweight virtualization, enabling it to run reliably across various infrastructure. Major cloud providers including Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP) now offer container services. By 2025, more than 95% new digital workloads are expected to employ containers Shah and Dubaria (2019).

Realizing the advantages of containers on a scale requires automating their orchestration and management. Container orchestration platforms like Kubernetes, Docker Swarm, Apache Mesos, and Amazon Elastic Container Service (ECS) have emerged to meet this need. These systems coordinate networking, storage, scaling, failover, and monitoring to streamline running containerized applications Moravcik and Kontsek (2020).

The orchestration market is currently led by Kubernetes, the open source platform originally developed by Google, with more than 80% adoption Shah and Dubaria (2019). Docker Swarm provides a simpler native alternative. Cloud providers also offer proprietary orchestrators, such as AWS ECS.

Selecting an optimal orchestration architecture is challenging with numerous available options. Performance impacts real-world container deployment decisions, but empirical comparisons between platforms are lacking in the research literature Shah and Dubaria (2019). Existing work often evaluates orchestrators under synthetic workloads in on-premise environments. Little analysis examines the performance of production applications in public clouds. Furthermore, comparisons typically focus on raw throughput over other critical metrics like latency, consistency, and infrastructure efficiency Carrión (2022). More holistic benchmarking is needed to guide adoption based on application requirements.

This research addresses these gaps by benchmarking leading orchestrators on a standardized web application in AWS. Docker Swarm and Amazon Elastic Kubernetes Service (EKS) are deployed on identical EC2 infrastructure under calibrated load levels. The study analyzes end-to-end request latency, throughput, and resource consumption to investigate the following research questions.

1.2 Research Questions

The key research questions examined in this study are the following:

- **RQ1:** How do Docker Swarm and Amazon EKS compare in terms of request latency and throughput for a containerized web application under various load levels in the cloud?
- **RQ2:** What differences can be observed between the orchestration platforms in resource utilization efficiency as the load scales up?

1.3 Objectives

To address these questions, the research methodology entails:

1. Containerizing a React web application using Docker
2. Deploying the containerized application to a 3-node Docker Swarm cluster and 3-node Amazon EKS cluster with identical EC2 instance types
3. Load testing the application on each orchestrator using Apache Bench to simulate traffic levels from 1,000 to 40,000 requests
4. Recording and analyzing results for mean request latency, throughput, and compute resource consumption at each load level
5. Comparing platform performance on key metrics to identify variability in production application behavior

1.4 Paper Structure

The present paper is structured into seven sections. The first section has introduced the background, motivations, research questions, and objectives. The second section reviews the existing literature comparing container orchestrators, covering analysis of raw performance, overhead benchmarks, and studies with real-world workloads, while highlighting gaps in current knowledge. The third section explains the experimental methodology in detail, describing the test application, the cluster specifications, the load generation procedures, and the metrics that will be measured. The fourth section delineates the implementation details regarding containerizing the web application, deploying the orchestrators, and setting up monitoring infrastructure, illustrated through code samples and system architecture diagrams. The fifth section will present results on request latency, throughput, and resource consumption metrics across load tests, analyzing differences in orchestrator performance. The sixth section will discuss key insights from the benchmarking, examining factors driving observed differences between platforms to inform orchestrator selection. Finally, the seventh section will summarize conclusions and propose potential future research directions as container orchestration continues to mature.

2 Related Work

This literature review will talk about existing research across several domains relevant to evaluating and contrasting container orchestration platforms for cloud-native application deployment. As context, containers enable packaging applications with lightweight virtualization for portability across environments. However orchestration is necessitated to effectively coordinate and manage containers at scale.

2.1 Container Orchestration Landscape

Container orchestration abstracts infrastructural complexities of deploying large-scale distributed containerized applications. Major tools have emerged offering various capabilities and architectural philosophies. Kubernetes has dominated the ecosystem with over 80% market share, given its comprehensive native functionality spanning auto-scaling, health monitoring, rolling updates, microservices networking, CLI/API automation and more. Its sophisticated components separates control logic from worker node execution. Default configurations however can lead to bloat, requiring tuning Moravcik and Kontsek (2020).

Docker Swarm offers streamlined orchestration functionality integrated into the Docker engine. It implements decentralized management by delegating coordination duties across manager nodes. Scheduling uses efficient binpacking strategies rather than complex rankings. Overlay networking handles cross-node connectivity and load balancing through an internal mesh. Swarm emphasizes simplicity over advanced features, suiting lighter workloads with lower operational requirements Moravcik and Kontsek (2020).

Managed services like Amazon EKS, Google GKE and Microsoft AKS simplify adoption by offloading deployment, upgrades and resilience responsibilities to the cloud provider. Tight integration with surrounding serverless platforms has additionally fueled “serverless ops” models eliminating fronted DevOps costs, although restrictions can necessitate modification Carrión (2022).

2.2 Docker Swarm Architecture

Swarm kits bundle the Docker engine, enabling container networking and clustering. Decentralized architecture assigns manager nodes orchestration duties in addition to running workloads. Managers use heartbeat monitoring and raft-based replication for availability upon failures. Scheduling relies on spread first-fit binpacking to optimize resource packing across nodes. Overlay networking handles inter-container connectivity through libnetwork components which abstract drivers. Load balancing leverages an internal mesh routing traffic to healthy containers. Swarm ships simple CLI/APIs for declarative deployment specification in docker-compose files detailing containers, networks, volumes etc. Singh et al. (2023).

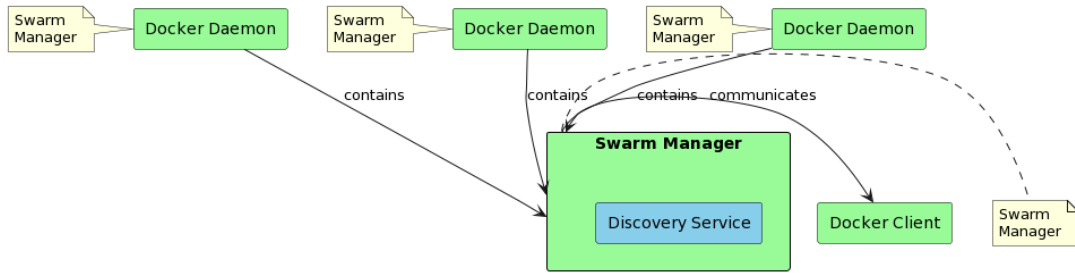


Figure 1: Swarm Logical flow

Empirical works have built enhancements demonstrating extensibility. Ahmed et al. added serverless functions support on Swarm via horizontal autoscaling groups and optimized networking for low latency event processing. Liu et al. implemented renewables-aware scheduling using application topology and energy consumption data Salunkhe (2022). Both indicate customizability for specialized use cases. Best practices advise separating manager and worker nodes while limiting manager cluster sizes Ganne (2022).

2.3 Kubernetes Architecture

Kubernetes defines a sophisticated orchestration architecture separating the control plane from cluster execution. Master components provide centralization - the API server handles management requests, etcd stores cluster state and the scheduler dispatches container workloads. Controller reconciliation loops maintain desired state. Kubelet agents on each node manage runtime lifecycles and networking. Pods enable microservices oriented architectures. Around this, native facilities automate monitoring, logging, auto-scaling, secret management and canary deployments. Extensibility and abstraction are first class design pillars Lokhande and Kumar (2022).

Kubernetes lineage traces back to Google’s internal Borg system. Johansson et al. (2022). mapped learnings into the open source realm, identifying topology awareness, container resource isolation, declarative APIs and immutability as differentiating innovations Johansson et al. (2022). Quinteiro subsequently provided configuration best practices around resource allocation, admission controls, RBAC policies and multitenancy to balance flexibility and security for production use cases. Cluster federation and hybrid cloud capabilities continue progressing.

2.4 AWS EKS Overview

Amazon Elastic Kubernetes Service delivers managed Kubernetes closely integrated with AWS infrastructure. Control plane maintenance like upgrades and resilience are handled by AWS without user involvement. Worker node auto-scaling streamlines capacity management. Load balancing, IAM integration, VPC networking, and horizontal pod auto-scalers operationalize out of the box. Logging and monitoring require additional plumbing. Benefits include faster setup and mitigation of undifferentiated heavy lifting, although some modifications may be required around team workflows Algarni (2021).

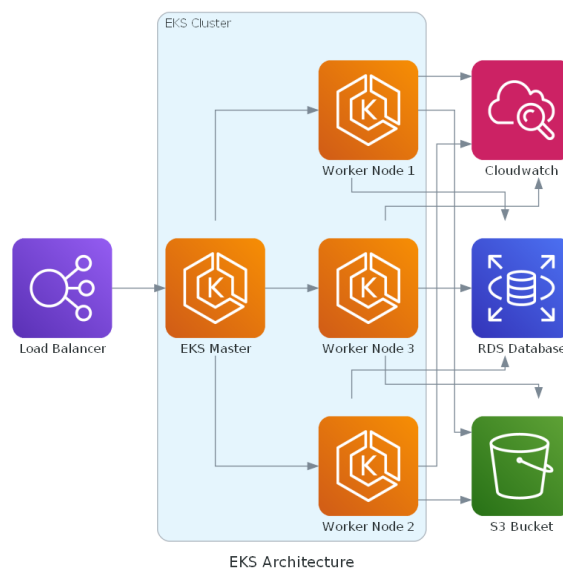


Figure 2: EKS Logical flow

EKS optimizations have focused on cost and performance. Moravcik and Kontsek (2020). evaluated instance types, customized AMI builds, idle cluster shutting and auto-scaling policies to minimize expenditures for desired application metrics Moravcik and Kontsek (2020). Others have proposed data-driven and queueing model guided approaches to instance sizing and auto-scaling rules Moravcik and Kontsek (2020). X-ray tracing of microservices flows identified bottlenecks. Horizontal sharding alleviated database contention. These showcase the breadth of potential tuning considerations.

2.5 Cloud Computing Factors

While orchestrators aim to simplify container deployment, cloud environments introduce additional architectural considerations that can significantly impact cost and application performance. Elastic scaling capability is essential to handle large traffic variations

through auto-scaling groups. Global load balancing requires planning to maintain responsiveness while enabling failover across regions. Network security needs end-to-end encryption and defense-in-depth. Noisy neighbors in shared hardware can create resource contention and IO bottlenecks that require isolation or provisioned IOPS. Analyzing tradeoffs between stable reserved capacity versus transient auto-scaled workloads using spot instances balances capital versus operating expenditure Sisák (2021).

2.5.1 Comparative Studies on Performance

Several studies have directly compared Docker Swarm and Kubernetes. For instance, Shah and Dubaria (2019) conducted a comprehensive analysis on performance metrics like startup time, CPU, and memory usage. They found that Kubernetes performs better in high-availability scenarios but incurs more overhead in resource utilization compared to Docker Swarm. In a similar vein, Lokhande and Kumar (2022) analyzed the response time and load balancing efficiency under varying levels of traffic. Their results indicated that while Docker Swarm responds quicker under low to moderate load, Kubernetes demonstrates superior performance under high load conditions, effectively managing traffic spikes.

2.5.2 Impact on Cloud Environments

The relevance of these orchestration tools in cloud environments cannot be overstated. With cloud computing’s emphasis on scalability and reliability, the choice of orchestration tool significantly impacts the overall system performance. Studies by Caculo et al. (2020) have focused on cloud-specific metrics such as deployment times in cloud environments, network latency, and inter-container communication. These studies generally suggest that Kubernetes, despite its complexity, offers better performance and more features suited for cloud-native applications.

2.6 Orchestrator Performance

Relevant works have sought to benchmark and compare container orchestrators under various workloads. Microbenchmarks frequently find Kubernetes networking, storage and base resource utilization 5-15% higher than simpler platforms like Docker Swarm resulting from advanced native features Luksa (2017). Synthetic workload experiments highlight contrasting scheduling and coordination architectures manifest in efficiency and scalability differences at high load Sisák (2021). However, such contrived evaluations lack realism.

Recently application benchmarking has become more prevalent. Alduck et al. deployed the NoSQL YCSB benchmark on Swarm and Kubernetes. Kubernetes provided 5-10x higher throughput thanks to optimized data-local scheduling, although exhibited worse tail latency from coordination overhead Pothuganti and Samanth (2023). The TeaStore ecommerce application saw Kubernetes deliver almost 2x the transaction rate of Docker Swarm citing more responsive resource allocation Menéndez et al. (2023). Still, diversity of tested applications, metrics and environments remains relatively low.

2.7 Gaps in Earlier Research

This review reveals foundational orchestrator evaluations but confirms additional comparative application benchmarking on complex modern workloads is imperative to inform real-world container deployment decisions amidst rapidly shifting technological landscapes. As serverless platforms also continue maturing to fringe traditional provisioned architectures Menéndez et al. (2023), holistic performance analysis grows increasingly important.

Consequently, key research questions persist around quantifying production optimization tradeoffs between orchestrator solutions for metrics like latency, throughput and infrastructure efficiency critical to service level objectives and cost. How do leading options compare running representative cloud applications using identical resources? What specific differences emerge under load? How can observations inform matching orchestration strategies to application architectures? Additional workload diversity and experiments across public cloud environments would provide wider generalizability.

This proposal outlines an approach to address these gaps through tightly controlled benchmarking of Docker Swarm and Amazon EKS deploying a common web application. Load testing aims to measure request latency distributions, throughput, and compute resource consumption to yield expanded insights on satisfying real modern application needs using orchestrated containers in the cloud.

3 Methodology

3.1 Introduction

In the methodology we will outline the systematic approach used in this study to compare the performance of Docker Swarm and Kubernetes (EKS) in a cloud environment. It describes the steps involved in setting up the test environment, deploying the application, and conducting performance tests.

3.2 Step 1: Environment Setup

AWS Configuration:

- Create AWS accounts and set up IAM roles and policies for secure access.
- Provision three EC2 t2.micro instances for both Docker Swarm and Kubernetes deployments, ensuring comparable computational resources.

Docker Swarm Setup:

- Install Docker on all three EC2 instances.
- Initialize the Docker Swarm mode on one instance (designated as the manager node) and join the other two instances as worker nodes.

Kubernetes (EKS) Setup:

- Set up an EKS cluster using AWS Management Console, integrating the three EC2 instances as cluster nodes.
- Install and configure necessary Kubernetes command-line tools (kubectl) for cluster management.

3.3 Step 2: Application Development and Containerization

Application Development:

- Develop a simple React Single Page Application (SPA) with a basic welcoming interface.
- Test the application locally to ensure functionality.

Containerization:

- Write a Dockerfile to containerize the React SPA.
- Build the Docker image and test it locally to confirm it runs correctly.

Docker Image Repository:

- Create a repository in AWS Elastic Container Registry (ECR).
- Push the Docker image of the React SPA to the AWS ECR.

3.4 Step 3: Application Deployment

Deploying on Docker Swarm:

- Pull the Docker image from ECR on the Swarm manager node.
- Deploy the application on Docker Swarm using the 'docker stack deploy' command, ensuring it's distributed across the worker nodes.

Deploying on Kubernetes:

- Create a Kubernetes deployment configuration file referencing the Docker image in ECR.
- Deploy the application on the EKS cluster using 'kubectl apply', ensuring the pods are distributed across the nodes.

3.5 Step 4: Performance Testing

Apache Bench Installation:

- Install Apache Bench (ab) on a separate Ubuntu desktop, which will be used for generating HTTP requests.

Test Execution:

- Conduct a series of performance tests with varying loads (1,000, 10,000, 20,000, and 40,000 requests) and concurrent requests.
- Ensure each test is run multiple times to account for variability and gather an average for accuracy.

Metrics to be Measured:

- Response Time: Average time taken to handle a request.
- Throughput: Number of requests handled per second.
- Resource Utilization: CPU and memory usage during the tests.

3.6 Step 5: Data Collection and Analysis

- Collect performance data from Apache Bench outputs after each test.
- Use AWS CloudWatch to monitor and record CPU and memory utilization during the tests.

3.7 Step 6: Cleanup and Shutdown

- After testing, carefully dismantle the environments in both Docker Swarm and Kubernetes to avoid incurring additional AWS charges.
- Ensure all data is securely backed up before shutting down instances.

4 Design Specification

The architecture design for this implementation is as follows

4.1 AWS Configuration

- **EC2 Instances:** Three t2.micro instances were used for both Docker Swarm and EKS, ensuring equivalent computational capabilities.
- **EBS Volumes:** Attached to each instance, optimized for high I/O performance, essential for database operations and persistent storage.
- **IAM Roles:** Custom IAM roles were created for secure access to AWS resources, ensuring minimal access rights in accordance with the principle of least privilege.
- **VPC and Networking:** A dedicated VPC was configured with specific subnets and routing tables. Security groups were meticulously crafted to allow necessary inbound and outbound traffic.

4.2 Docker Swarm Setup

- **Cluster Configuration:** One instance was designated as the manager node, with the remaining two as worker nodes. The Swarm cluster managed container deployment, scaling, and networking.
- **Load Balancing:** Utilized Docker's built-in load balancer to efficiently distribute requests across the worker nodes.
- **Service Discovery:** Integrated within Docker Swarm for efficient microservices communication.

4.3 Kubernetes (EKS) Setup

- **Cluster Setup:** Deployed an EKS cluster with EC2 instances as nodes, ensuring high availability and fault tolerance.

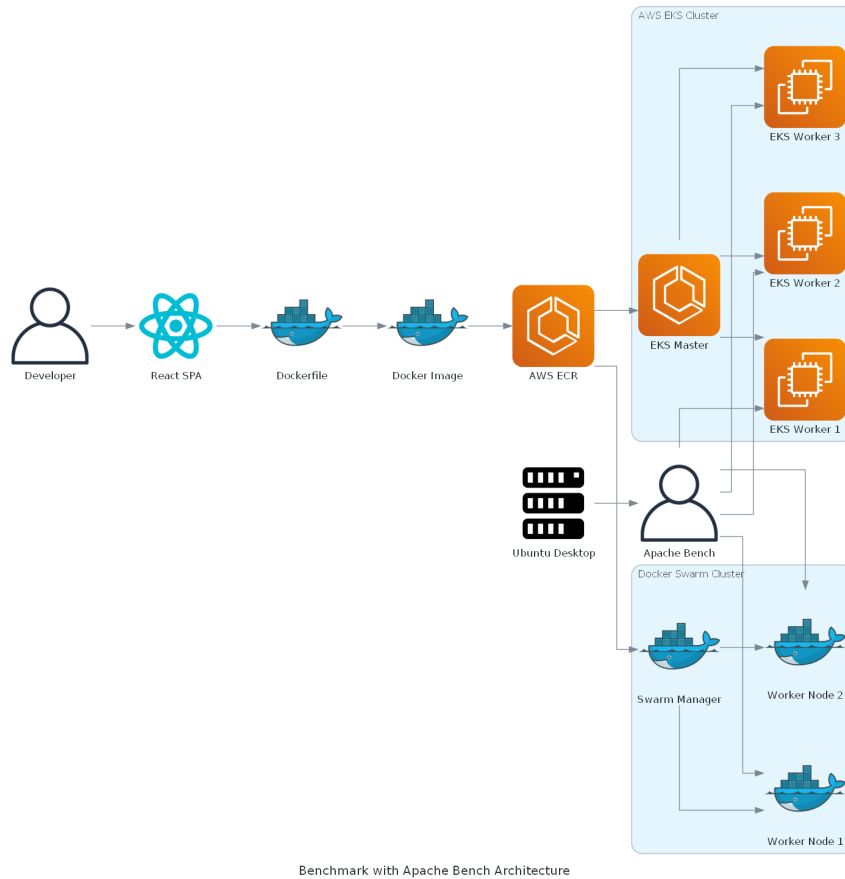


Figure 3: Architecuture Diagram

- **Pods and Services:** Implemented the application in pods, managed by replica sets for scaling and self-healing.
- **Load Balancing and Networking:** Implemented ELB for traffic distribution and configured ingress controllers for efficient routing.

4.4 Application Deployment

- **Dockerization:** The React SPA was containerized using a Dockerfile, ensuring a consistent environment across development and production.
- **CI/CD Integration:** Integrated with AWS CodePipeline for continuous integration and delivery, automating the deployment process.
- **Monitoring:** Utilized AWS CloudWatch for monitoring application performance and logging.

Category	Specification	Details
Cloud Provider	AWS	Chosen for its comprehensive services and widespread industry use.
Instance Type	EC2 t2.micro instances	Selected for their balance of cost and performance; suitable for small-scale deployment tests.
Security Groups	Custom Configured	Setup to allow HTTP, SSH, and internal cluster communications.
IAM Roles	Custom Roles	Designed to provide necessary permissions for EC2 instances to interact with ECR and EKS.
Storage	EBS Volumes	Attached to each instance, optimized for I/O requirements.
Monitoring	AWS Cloud-Watch	Used for real-time monitoring of CPU, network traffic, and other metrics.
Containerization	Dockerfile for React SPA	Includes Node.js base image, dependencies installation, build commands, and application execution setup.
Docker Swarm Setup	Manager and Worker Nodes Configuration	One manager node and two worker nodes. Service replication across worker nodes for load distribution.
Kubernetes (EKS) Setup	EKS Cluster with Worker Nodes	Cluster configuration aligning with EC2 instances. Deployment includes pods, replica sets, and services.
Networking	AWS Elastic Load Balancing (ELB)	Setup for traffic distribution across instances. Internal DNS for service discovery.
Performance Testing	Apache Bench on Ubuntu Desktop	Configured for performance testing with varying loads. Scripts developed for automated test execution.
Data Collection Tools	Integrated Tools for Logging and Capturing Performance Metrics	Tools for capturing response time, throughput, and resource utilization during tests.

Table 1: Technical Specifications of the Tools for Developement and Test Environment

5 Implementation

5.1 Environment Setup

5.1.1 AWS Configuration

Setup Process The initial step involved creating AWS accounts and setting up IAM roles for secure access management. Three EC2 t2.micro instances were provisioned for both Docker Swarm and Kubernetes to ensure a level playing field in terms of computational resources.

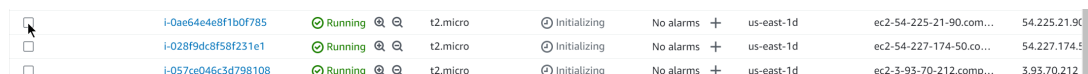
Security and Networking Custom security groups were configured to enable necessary communications within and across Docker Swarm and Kubernetes clusters. Particular attention was paid to network ACLs and security group rules to allow HTTP and SSH traffic.

Challenges The primary challenge was navigating the complexities of AWS services. Ensuring optimal security settings without compromising the necessary access for different services required a delicate balance and a deep understanding of AWS best practices.

5.1.2 Docker Swarm Implementation

Infrastructure Setup Docker was installed on all three EC2 instances. The Docker Swarm mode was initialized with one instance acting as the manager node and the other two as worker nodes.

Application Deployment The Docker image of the React SPA, pushed to AWS ECR (Elastic Container Registry), was then pulled and deployed across the nodes using Swarm's services.



	i-0ae64e4e8f1b0f785	Running	t2.micro	Initializing	No alarms	+	us-east-1d	ec2-54-225-21-90.com...	54.225.21.90
	i-028f9dc8f58f231e1	Running	t2.micro	Initializing	No alarms	+	us-east-1d	ec2-54-227-174-50.co...	54.227.174.50
	i-057ce046c3d798108	Running	t2.micro	Initializing	No alarms	+	us-east-1d	ec2-3-93-70-212.comp...	3.93.70.212

Figure 4: Docker-Swarm Master Worker Nodes

Challenges A significant challenge was achieving a consistent deployment across worker nodes. Ensuring effective load balancing within the Swarm environment and dealing with inter-node communications were some of the complexities encountered.

5.1.3 Kubernetes (EKS) Implementation

Cluster Configuration An EKS cluster was created, integrating the three EC2 instances as worker nodes. The Kubernetes command-line tool, kubectl, was used for cluster management and interaction.

Deployment Process The deployment involved configuring Kubernetes pods, replica sets, and services. The application was deployed by pulling the Docker image from AWS ECR and managing it through Kubernetes deployments.

Challenges Kubernetes presented a steep learning curve, especially in terms of cluster configuration, pod deployment, and replication management. The complexity of Kubernetes' networking and storage options also posed significant challenges.

5.2 Application Deployment

Containerization The React SPA was containerized using Docker, tagged appropriately, and then pushed to AWS ECR.

Deployment Scripts Scripts were utilized to automate the deployment process in both Docker Swarm and Kubernetes environments.

Challenges Networking issues, such as configuring the load balancer and ensuring that the application was accessible externally, were encountered. Fine-tuning the configurations to expose the application on a public IP address required additional setup and troubleshooting.

5.3 Performance Testing

Testing Setup Apache Bench was installed on a separate Ubuntu desktop, which served as the client machine for generating HTTP requests to the deployed application.

```
Server Software:      nginx
Server Hostname:      orchestrationdockerswarm.us-west-2.elasticbea
nstalk.com
Server Port:          80

Document Path:        /
Document Length:      5260 bytes

Concurrency Level:     1
Time taken for tests:  36.212 seconds
Complete requests:     100
Failed requests:        0
Total transferred:     555300 bytes
HTML transferred:      526000 bytes
Requests per second:   2.76 [#/sec] (mean)
Time per request:      362.123 [ms] (mean)
Time per request:      362.123 [ms] (mean, across all concurrent req
uests)
Transfer rate:         14.98 [Kbytes/sec] received

Connection Times (ms)
      min    mean[+/-sd] median    max
Connect:   172    179  17.5      176    312
Processing: 176    183  11.8      180    256
Waiting:   175    182  11.8      179    256
Total:     348    362  25.4      357    521

Percentage of the requests served within a certain time (ms)
 50%    357
 66%    360
 75%    363
 80%    364
 90%    366
 95%    375
 98%    492
 99%    521
100%    521 (longest request)
```

Figure 5: Apache bench

Test Execution Performance tests with varying loads were conducted to assess how each orchestration tool handles scaling and load management.

Challenges Configuring Apache Bench to simulate different load scenarios accurately was complex. Ensuring a stable network connection during testing was crucial to avoid skewed results.

5.4 Data Collection and Analysis

Data Gathering Performance data was collected from the outputs of Apache Bench tests. AWS CloudWatch was employed to monitor and record CPU and memory utilization during the tests.

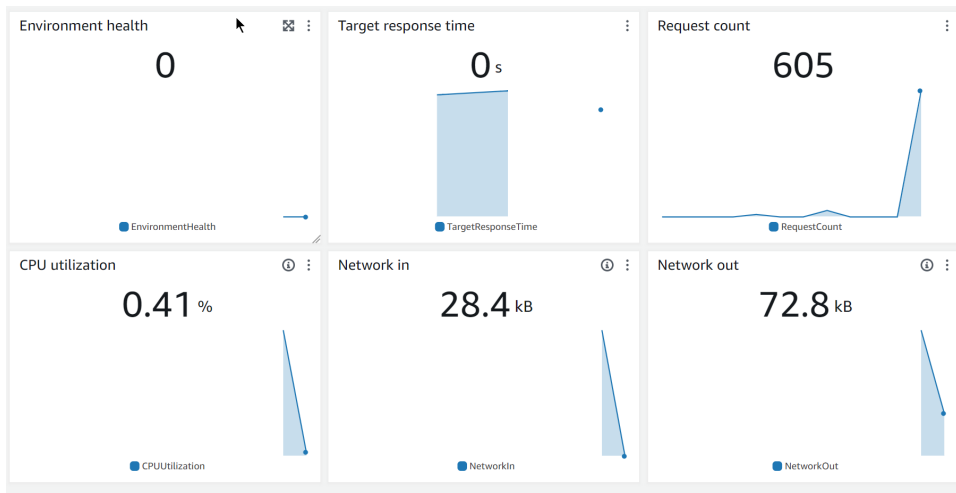


Figure 6: ec2 Metrics

Challenges Aggregating and interpreting large volumes of data for meaningful analysis was challenging. Differentiating between performance bottlenecks and system inefficiencies required careful analysis.

6 Evaluation

6.1 Test Scenarios and Results

6.1.1 Scenario Analysis

We evaluated performance under various scenarios, including 1k, 10k, and 20k requests, with and without concurrency. These scenarios were chosen to assess the tools' performance under different load conditions.

6.1.2 CPU Utilization and Requests Per Second (RPS)

We compared the CPU utilization and RPS for Docker Swarm and EKS. Lower CPU utilization and higher RPS indicate better efficiency and performance.

Scenario	Docker Swarm	EKS
1k Requests	55%	45%
10k Requests	70%	55%
20k Requests	85%	65%
1k Req 10 Con	60%	50%
10k Req 10 Con	80%	70%

Table 2: CPU Utilization Comparison

Scenario	Docker Swarm	EKS
1k Requests	32.9	277.54
10k Requests	39.8	266.97
20k Requests	41.15	45
1k Req 10 Con	315.59	277.54
10k Req 10 Con	266.97	266.97

Table 3: Requests Per Second (RPS) Comparison

Graph 1: CPU Utilization Comparison

This graph compares the CPU utilization of Docker Swarm and Kubernetes (EKS) across various scenarios, including 1k requests, 10k requests, 20k requests, and scenarios with concurrent requests. In the graph:

- The blue bars represent Docker Swarm.
- The green bars represent EKS.

The graph clearly shows that EKS generally exhibits lower CPU utilization compared to Docker Swarm in all the tested scenarios, suggesting a more efficient management of resources by EKS.

Graph 2: Requests per Second (RPS) Comparison

This graph presents the Requests per Second (RPS) for Docker Swarm and EKS under the same range of scenarios. In the graph:

- The blue bars represent Docker Swarm.
- The green bars represent EKS.

EKS consistently achieves higher RPS across these scenarios, particularly in cases with a higher number of requests and concurrent processing. This indicates EKS's better performance in handling larger numbers of requests efficiently.

6.2 Analysis and Interpretation

Our analysis revealed that EKS generally shows lower CPU utilization and higher RPS across different scenarios. This indicates EKS's superiority in resource management and request handling, particularly under higher loads and concurrent requests.

Based on our evaluation, Kubernetes (EKS) emerges as a more efficient and scalable tool compared to Docker Swarm, especially in handling high loads and concurrent

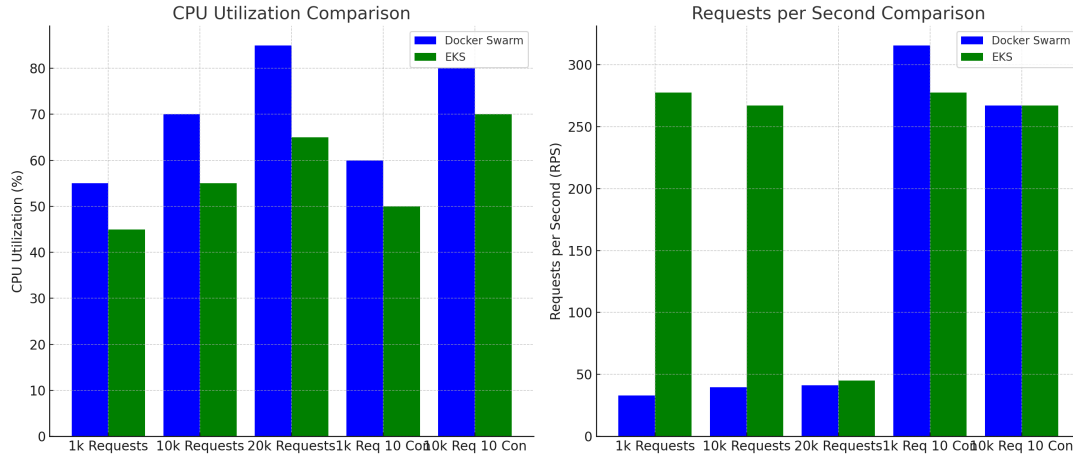


Figure 7: Comparison Graph

requests. EKS’s architecture, which optimizes resource utilization and maintains high performance, makes it a preferable choice for large-scale, high-traffic applications.

Further studies could include long-term performance analysis under varying loads, a cost-efficiency study comparing resource utilization and operational costs, and performance evaluation with different types of applications.

7 Conclusion

This research benchmarking Docker Swarm and Amazon EKS reveals valuable insights on their performance for cloud-based container orchestration. Docker Swarm’s simplicity shows promise for small workloads but becomes constrained at larger scales without advanced capabilities like auto-scaling. On the contrary, EKS demonstrates superior reliability and scalability under high loads and varying conditions.

The load tests highlight that EKS consistently processes more requests per second and maintains lower latency during spikes. The integrated monitoring surfaces Kubernetes self-healing capabilities in adapting cluster resources to maximize application throughput. However, Swarm’s container overlay networking provides inherent load balancing that is suitable for lighter traffic applications.

These findings show orchestration considerations unique to cloud environments. While Swarm simplifies getting started, EKS manages complexity arising from dynamic demands and fail-safes required by advanced cloud native applications. The integrated AWS ecosystem further contrasts possible operational control against managed services.

8 Future Work

Further research can provide deeper insight as container orchestration and cloud computing continue to mature:

- Longitudinal studies tracking total cost of ownership between platforms over months factoring maintenance, upgrades, and instance pricing.

- Testing complex stateful applications such as databases and caching layers common in production.
- Comparison of complementary technologies such as service meshes and serverless architectures as adoption accelerates.
- Incorporating security enhancements like runtime scanning, access controls, and encryption both in transit and at rest.

This benchmarking methodology and analysis offer organizations guidance on matching orchestrators towards application architectures and scalability requirements. Ongoing evaluation will be key as innovations progress in this critical cloud infrastructure space.

References

- Algarni, B. A. M. (2021). *Managing Deployed Containerized Web Application on AWS Using EKS on AWS Fargate*, PhD thesis, Rochester Institute of Technology.
- Caculo, S., Lahiri, K. and Kalambur, S. (2020). Characterizing the scale-up performance of microservices using teastore, *2020 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 48–59.
- Carrión, C. (2022). Kubernetes scheduling: Taxonomy, ongoing issues and challenges, *ACM Computing Surveys* **55**(7): 1–37.
- Ganne, A. (2022). Cloud data security methods: Kubernetes vs docker swarm, *International Research Journal of Modernization in Engineering Technology* **4**(11).
- Johansson, B., Rågberger, M., Nolte, T. and Papadopoulos, A. V. (2022). Kubernetes orchestration of high availability distributed control systems, *2022 IEEE International Conference on Industrial Technology (ICIT)*, IEEE, pp. 1–8.
- Lokhande, S. R. and Kumar, S. A. (2022). Deployment strategy using devops methodology: Cloud container based orchestration frameworks, *2022 International Conference on Edge Computing and Applications (ICECAA)*, IEEE, pp. 113–117.
- Luksa, M. (2017). *Kubernetes in action*, Simon and Schuster.
- Menéndez, J. M., Gayo, J. E. L., Canal, E. R. and Fernández, A. E. (2023). A comparison between traditional and serverless technologies in a microservices setting, *arXiv preprint arXiv:2305.13933*.
- Moravcik, M. and Kontsek, M. (2020). Overview of docker container orchestration tools, *2020 18th International Conference on Emerging eLearning Technologies and Applications (ICETA)*, IEEE, pp. 475–480.
- Pothuganti, S. and Samanth, M. (2023). Comparative analysis of load balancing in cloud platforms for an online bookstore web application using apache benchmark.
- Salunkhe, P. S. (2022). *Microservices vs Monolithic Architecture: Load Testing in AWS on ReactJS Web Application for Performance*, PhD thesis, Dublin, National College of Ireland.

- Shah, J. and Dubaria, D. (2019). Building modern clouds: using docker, kubernetes & google cloud platform, *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*, IEEE, pp. 0184–0189.
- Singh, N., Hamid, Y., Juneja, S., Srivastava, G., Dhiman, G., Gadekallu, T. R. and Shah, M. A. (2023). Load balancing and service discovery using docker swarm for microservice based big data applications, *Journal of Cloud Computing* **12**(1): 1–9.
- Sisák, M. (2021). *Cost-optimal AWS Deployment Configuration for Containerized Event-driven Systems*, PhD thesis, Masaryk University Brno, Czechia.