Overview

This project involved creating a scalable, high-availability Kubernetes (K8s) cluster and deploying essential platform components such as Jenkins, ELK Stack (Elasticsearch, Logstash, and Kibana), Prometheus, and Grafana using Helm and Kubernetes manifests. The primary focus was on ensuring the cluster and the deployed services are production-ready, emphasizing scalability, high availability, and security.

Objectives

1. **Building and Containerizing a Java Application**
2. **Create a Production-Ready Kubernetes Cluster**
3. **Deploy Platform Services (Jenkins, ELK, Prometheus, Grafana)**
4. **Implement CI/CD for Application Build and Deployment**
5. **Document Implementation and Design Decisions**
6. **Next Steps for Further Development**

AWS as the Chosen Platform

AWS was chosen for its robust security features, high availability, scalability, and extensive ecosystem. My personal experience with AWS services and tools further facilitated efficient setup and management of the infrastructure.

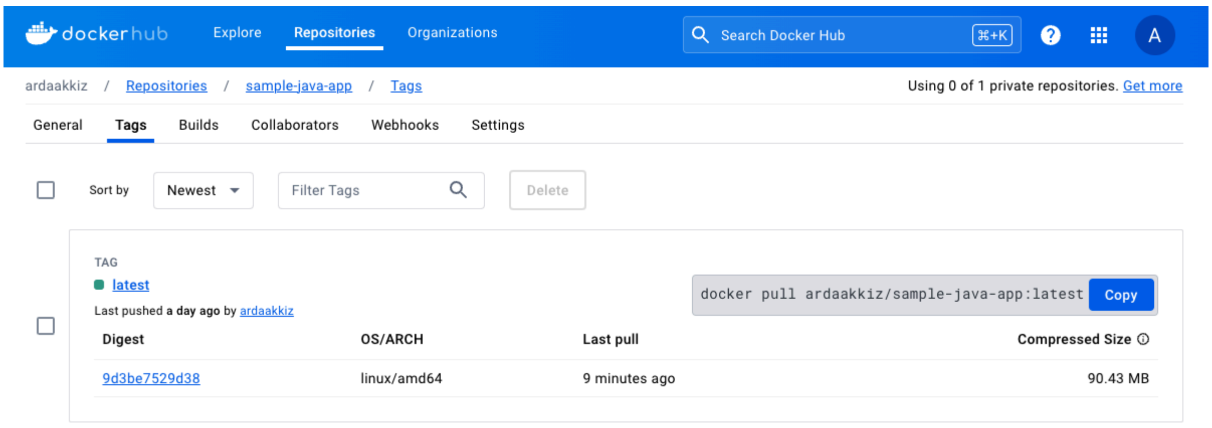
Benefits of AWS:

- **Security:** Advanced security features and compliance certifications.
- **Availability:** Multiple availability zones and regions ensure high availability.
- **Scalability:** Easy to scale resources up or down based on demand.
- **Experience:** My prior experience with AWS services.

1. Building and Containerizing a Java Application

To containerize the Java application, we use a Dockerfile with multi-stage builds.

```
# Stage1: Use an official Maven image to build the application

FROM maven:3.8.4-openjdk-11 AS build

WORKDIR /app

COPY . .

# Unset MAVEN_CONFIG to avoid lifecycle phase issues

ENV MAVEN_CONFIG=

RUN chmod +x ./mvnw && ./mvnw clean package


# Stage2: Use an official OpenJDK image to run the application

FROM openjdk:11-jre-slim

WORKDIR /app

COPY --from=build /app/target/app-0.0.1-SNAPSHOT.jar app.jar

EXPOSE 9001

ENTRYPOINT ["java", "-jar", "app.jar"]
```



ardaakkiz / Repositories / sample-java-app / Tags

Using 0 of 1 private repositories. Get more

TAG
latest
Last pushed **a day ago** by ardaakkiz

```
docker pull ardaakkiz/sample-java-app:latest
```

Digest	OS/ARCH	Last pull	Compressed Size
9d3be7529d38	linux/amd64	9 minutes ago	90.43 MB

2. Creating a Production-Ready Kubernetes Cluster

Cluster Setup

- **Cloud Provider**: AWS EKS (Elastic Kubernetes Service) for managed Kubernetes.
- **Instance Types**: Selected `t3.medium` instances for balanced CPU and memory resources.
- **High Availability**: Deployed nodes across multiple availability zones in the `us-east-1` region.
- **Networking**: Configured VPC, subnets, and security groups to ensure secure and efficient communication between nodes and services.

Scalability

- **Auto Scaling**: Configured cluster autoscaler to automatically adjust the number of nodes based on the workload.
- **Resource Requests and Limits**: Defined resource requests and limits for each pod to ensure optimal resource allocation and prevent resource contention.

Security

- **IAM Roles and Policies**: Configured IAM roles and policies for Kubernetes nodes to securely access AWS services.
- **RBAC**: Implemented Role-Based Access Control (RBAC) to manage permissions and access control within the cluster.
- **Network Policies**: Defined network policies to control traffic between pods, enhancing security and compliance.

Infrastructure as Code with CloudFormation

Using AWS CloudFormation, we treated infrastructure as code, which offers several benefits:

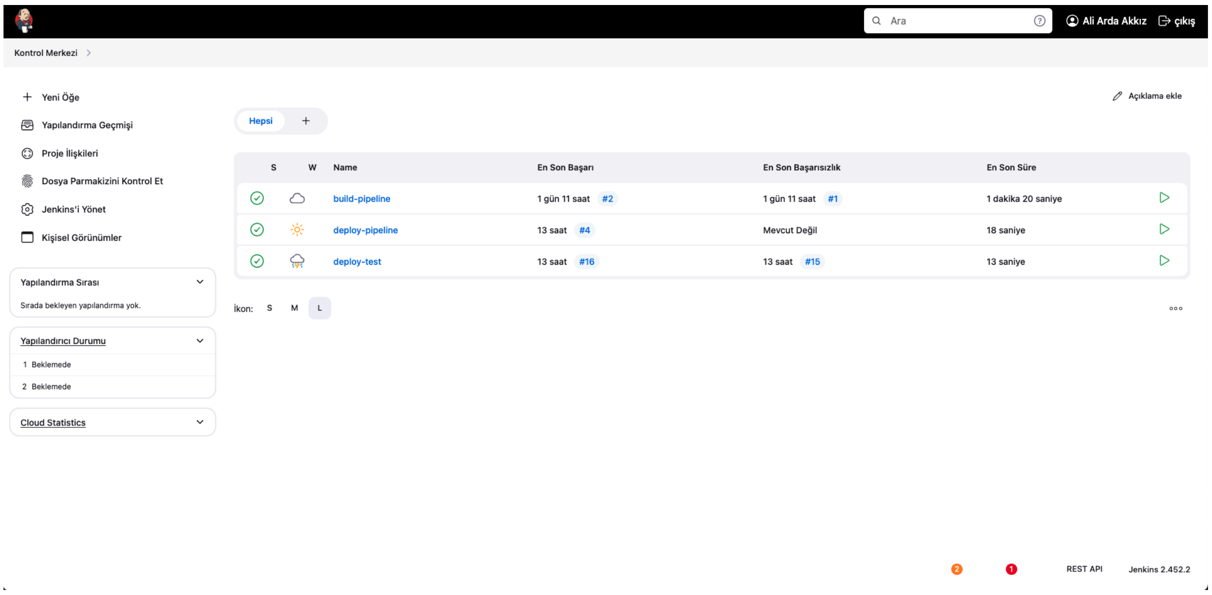
- Version Control: Infrastructure changes can be tracked and managed in version control systems.
- Reusability: Templates can be reused for different environments (development, staging, production).
- Consistency: Ensures consistent setups across different environments.

3. Platform Deployments

Jenkins

- **Deployment Method**: Jenkins was planned to be deployed using Kubernetes manifests. However, I've encountered a critical issue where the Docker daemon was not found within the Jenkins container even though the socket was mounted. To avoid security issues and data corruption risks associated with Docker-in-Docker (DinD), Jenkins was manually installed on the second worker node (an EC2 instance). This approach provided more control over the Jenkins environment and ensured compatibility with our existing infrastructure.
- **Persistence**: Leveraged the persistent storage of the EC2 instance for Jenkins home directory to retain job configurations.
- **Security**: Set up admin credentials, enabled RBAC, and configured secure access.

Jenkins CI/CD pipeline results



Sırada bekleyen yapılandırma yok.

1 Beklemede
2 Beklemede

S	W	Name	En Son Başarı	En Son Başarısızlık	En Son Süre
✓		build-pipeline	1 gün 11 saat #2	1 gün 11 saat #1	1 dakika 20 saniye
✓		deploy-pipeline	13 saat #4	Mevcut Değil	18 saniye
✓		deploy-test	13 saat #16	13 saat #15	13 saniye

build and deploy pipelines can be viewed at my public github repo: https://github.com/ardaakkiz/app

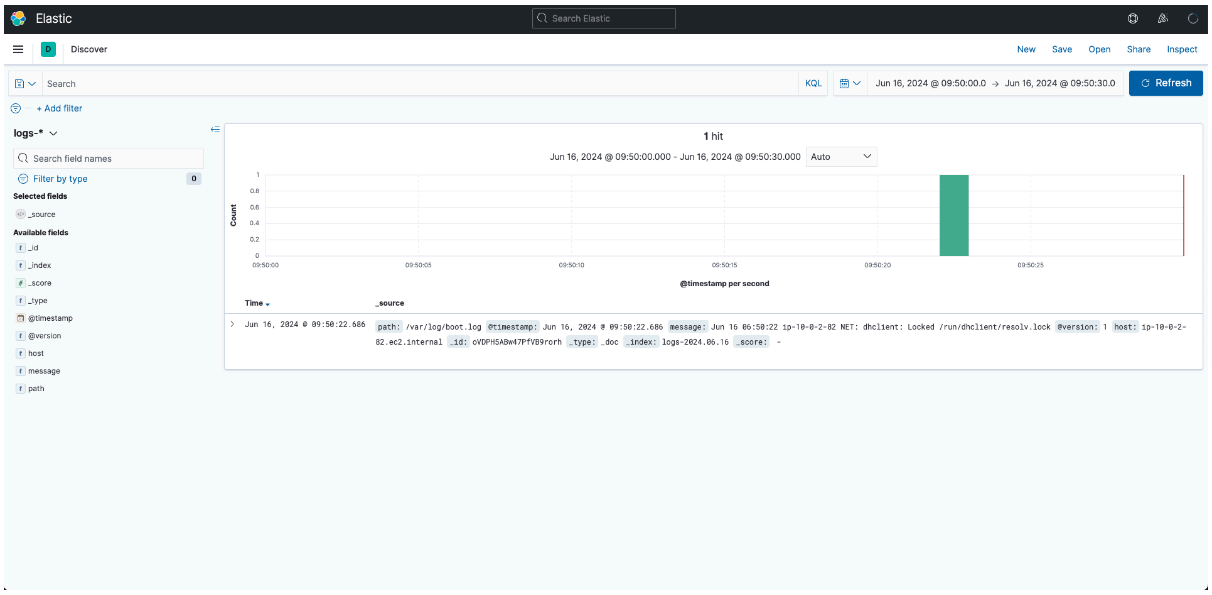
ELK Stack

The ELK stack (Elasticsearch, Logstash, Kibana) was deployed for centralized logging and monitoring.

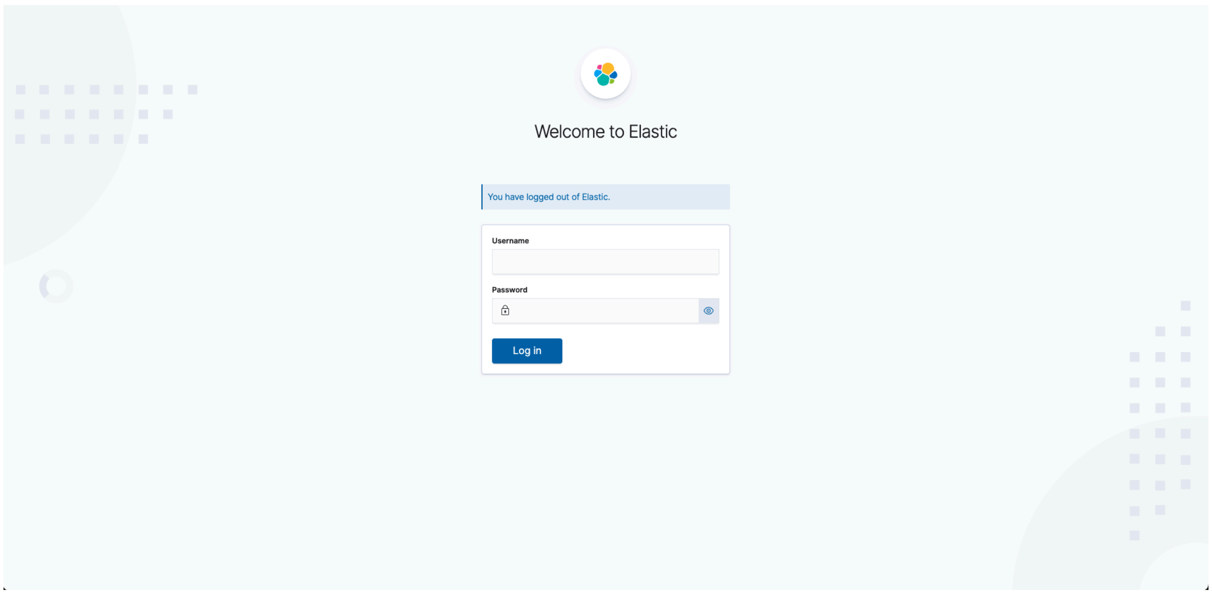
Elasticsearch and Kibana

Elasticsearch and Kibana were deployed using Kubernetes manifests to ensure they could scale with the cluster and leverage Kubernetes' orchestration capabilities. Credentials as kubernetes secrets with required configuration files and security groups, kibana can be accessed from the internet with authentication.

- **Logstash**: Deployed manually on the EC2 instance. Configured input pipelines.

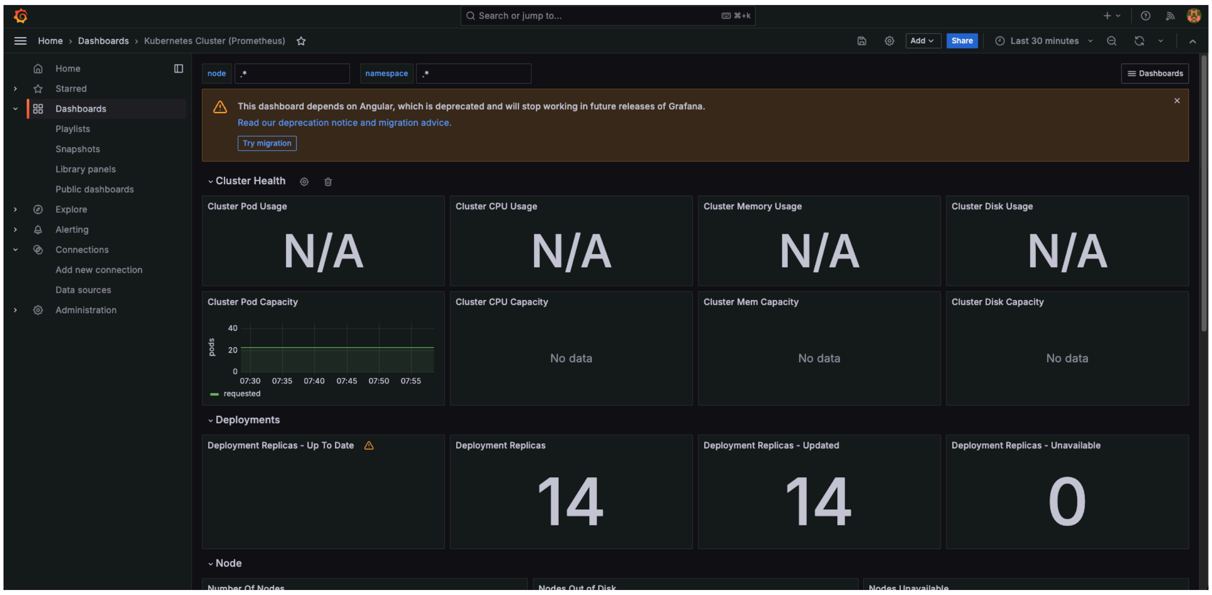


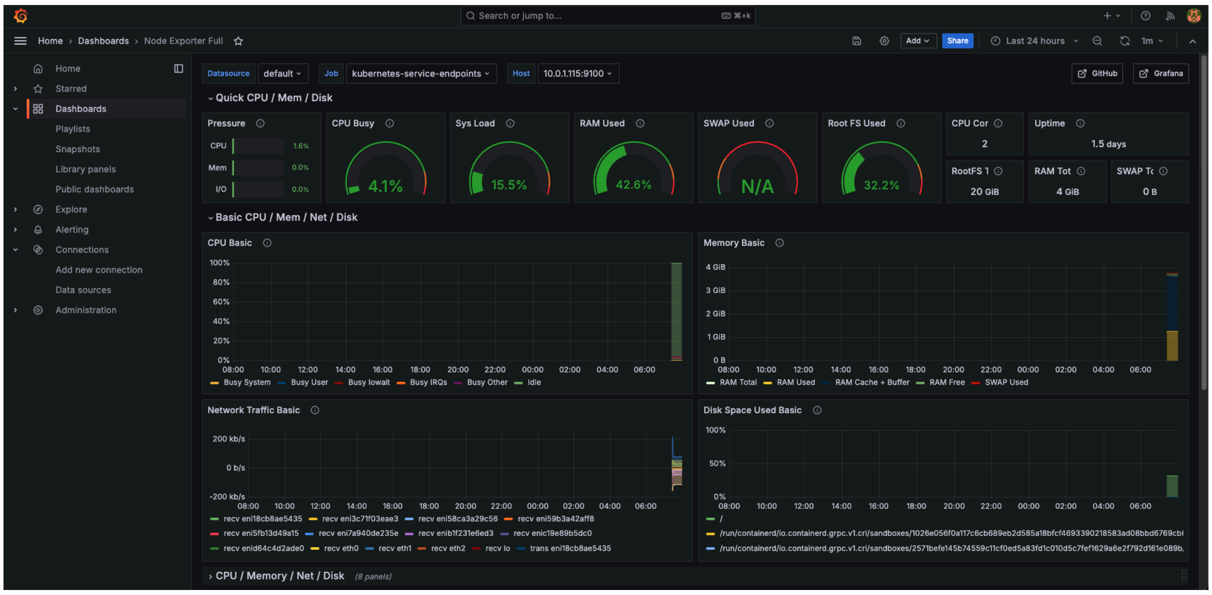
```
Time     _source
Jun 16, 2024 @ 09:50:22.686  path: /var/log/boot.log @timestamp: Jun 16, 2024 @ 09:50:22.686 message: Jun 16 06:50:22 ip-10-0-2-82 NET: dhclient: Locked /run/dhclient/resolv.lock @version: 1 host: ip-10-0-2-82.ec2.internal _id: oVDPH5ABw47PfVB9rorh _type: _doc _index: logs-2024.06.16 _score: -
```



Prometheus and Grafana

- **Prometheus**: Deployed using Helm (`prometheus-community/prometheus`). Configured service discovery for Kubernetes targets.
- **Grafana**: Deployed using Helm (`grafana/grafana`). Configured Prometheus as a data source and imported pre-built dashboards for Kubernetes and application monitoring.


Home › Dashboards › Kubernetes Cluster (Prometheus)
Last 30 minutes
This dashboard depends on Angular, which is deprecated and will stop working in future releases of Grafana.
Read our deprecation notice and migration advice.
Try migration
Cluster Health
Cluster Pod Usage N/A
Cluster CPU Usage N/A
Cluster Memory Usage N/A
Cluster Disk Usage N/A
Cluster Pod Capacity
Cluster CPU Capacity No data
Cluster Mem Capacity No data
Cluster Disk Capacity No data
Deployments
Deployment Replicas - Up To Date
Deployment Replicas 14
Deployment Replicas - Updated 14
Deployment Replicas - Unavailable 0
Node



4. Application Build and Deployment Stages

Build Stages

- **Source Code Management**: Integrated with Git for source control.
- **Build Automation**: Configured Jenkins pipelines to automate the build process, including compiling code, running tests, and creating Docker images.

Deployment Stages

- **Continuous Deployment**: Configured Jenkins pipelines for automated deployment to Kubernetes.
- **Ansible for Configuration Management**: Used Ansible playbooks to manage configuration and deployment tasks across different environments.

Implementing Canary Deployments

Canary deployments release a new application version to a small subset of users to monitor its performance and impact before a full rollout. This helps detect issues early and mitigate risks.

Tools for Canary Deployments

1. **Kubernetes**: Offers built-in capabilities for canary deployments via Deployment and Service objects, managed using `kubectl` and Helm.
2. **Istio**: A service mesh that manages traffic routing and monitoring, allowing fine-grained control over traffic distribution between different versions.
3. **Argo Rollouts**: Enhances deployment strategies including canary, integrating well with CI/CD pipelines for progressive delivery.

Traffic Management

- **Istio or Similar Tools**: Create a VirtualService to route a small percentage of traffic to the canary version.

Monitoring and Rollback

- **Prometheus and Grafana**: Monitor metrics and logs. Rollback can be automated or manual if issues are detected.

By integrating these tools and strategies into our CI/CD pipeline, we ensure a smooth and reliable deployment process, minimizing risks and improving stability.

5. Documentation of Implementation and Design Decisions

Kubernetes Cluster

- **Cluster Configuration**: Documented the cluster configuration, including instance types, networking setup, and security policies.
- **Scaling Policies**: Detailed the auto scaling configuration and resource management strategies.

Platform Deployments

- **Helm Charts & K8S Manifests**: Documented the Helm charts used for deploying Jenkins, ELK Stack, Prometheus, and Grafana.
- **Configuration Files**: Included configuration files and values used for customizing the deployments.
- **Persistence Strategy**: Explained the persistent storage setup and its importance for data durability.

CI/CD Pipeline

- **Jenkins Pipelines**: Provided the Jenkins pipeline scripts used for building and deploying applications.
- **Ansible Playbooks**: Included Ansible playbooks for configuration management and deployment tasks.
- **Deployment Strategy**: Detailed the deployment strategy.

6. Next Steps for Further Development

- **Automated Rollback**: Implement automated rollback mechanisms in Jenkins pipelines to revert to previous stable versions in case of deployment failures.
- **Prometheus and Grafana Dashboards**: Enhance monitoring by creating more detailed and application-specific dashboards in Grafana.
- **Alerting Rules**: Define more granular alerting rules in Prometheus to detect and notify on potential issues earlier.
- **Log Aggregation**: Improve log aggregation and search capabilities by tuning Elasticsearch and Kibana configurations.
- **Blue-Green Deployments**: Introduce blue-green deployment strategies to minimize downtime and reduce risk during deployments.
- **Progressive Delivery**: Utilize Argo Rollouts for more advanced deployment strategies like progressive delivery, where traffic is gradually shifted from the old to the new version.
- **Vulnerability Scanning**: Integrate security vulnerability scanning tools into the CI/CD pipeline to identify and mitigate potential security risks in dependencies and container images.
- **Network Policies**: Strengthen network policies within Kubernetes to limit communication between services to only what is necessary.
- **IAM Role Refinement**: Refine IAM roles and policies for least privilege access to AWS services.
- **Terraform Integration**: Move infrastructure provisioning from CloudFormation to Terraform for better modularity and reusability.
- **Configuration Management**: Use tools like Ansible and Puppet more extensively for managing and maintaining configurations across environments.
- **Horizontal Pod Autoscaling (HPA)**: Implement Horizontal Pod Autoscaling to automatically adjust the number of pods based on CPU/memory usage or custom metrics.
- **Cluster Autoscaler Tuning**: Fine-tune the cluster autoscaler settings to improve scaling efficiency and response times.
- **Structured Logging**: Standardize on structured logging across all services to improve log parsing and analysis.
- **Custom Metrics**: Develop and expose custom application metrics to Prometheus to gain deeper insights into application performance and behavior.
- **Detailed Documentation**: Expand documentation to cover all aspects of the infrastructure, deployments, and CI/CD processes.
- **Resource Optimization**: Continuously monitor and optimize resource usage to reduce costs without compromising performance.
- **Reserved Instances and Savings Plans**: Leverage AWS Reserved Instances and Savings Plans for cost savings on compute resources.
- **Service Mesh**: Evaluate and possibly integrate a service mesh like Istio for better traffic management, security, and observability.
- **Serverless Functions**: Investigate the use of AWS Lambda for certain workloads to improve scalability and reduce operational overhead.

Conclusion

The project successfully established a production-ready Kubernetes cluster and deployed essential platform services using Helm and Kubernetes manifests. The implementation focused on scalability, high availability, and security to ensure a robust and resilient environment for running applications. The documented implementation and design decisions provide a comprehensive guide for maintaining and scaling the setup as needed.