RMIT University Vietnam
School of Science, Engineering and Technology
Department of Information Technology

COSC2767 – System Deployment and Operations

Project Group Report

Semester 3, 2024

Group number: 1
Group name: DevOps-Team-1
Tutorial session: Friday, 8:30am
Mentor/Client: Mr Tom Huynh, PhD Candidate, MSc. in Computer Science and Artificial Intelligence

Group members:

- Vuong Gia An - s3757287
- Le Minh Duc - s4000577
- Tran Minh Nhat - s3926629
- Vu Tien Quang - s3981278
- Le Dinh Ngoc Quynh - s3791159

We declare that in submitting all work for this assessment we have read, understood and agree to the content and expectations of the Assessment declaration.

Ho Chi Minh City, Vietnam
Submission date: January 17th, 2025

Table of Contents

Table of Appendix

List of Figures

1. Pipeline Solution Description

For the detailed implementation of the project, here is the project's GitHub Link:
https://github.com/RMIT-Vietnam-Teaching/cosc2767-assignment-2-group-2024c-devops-team-1
Video Link: https://www.youtube.com/watch?v=vaDWYDjeppE
Other Links:

- Docker Hub for Frontend build: https://hub.docker.com/r/quynhethereal/fe-client-devops-course/tags
- Docker Hub for Backend build: https://hub.docker.com/r/quynhethereal/be-server-devops-course/tags

1.1. Problem statement

The client, managing a monolithic repository of an e-commerce website built with the MERN stack, faces significant challenges due to the absence of a CI/CD pipeline. The manual deployment process is time-consuming, error-prone, and lacks consistency, increasing the risk of website downtime and broken functionality. Without automated testing, bugs can make it into production, compromising the stability and user experience of the platform. Additionally, the lack of visibility into deployment progress and the inability to perform quick rollbacks hinder effective troubleshooting and recovery.

As the website grows and traffic increases, the client feels the struggle to maintain consistent performance and reliability across environments. The absence of Continuous Integration (CI) complicates collaboration, leading to code conflicts and delayed releases. These issues create a high-risk deployment environment, delaying critical updates and making the process stressful and overwhelming for the client. This inefficient workflow detracts from their core development efforts and poses a significant barrier to scalability and long-term growth.

1.2. Technology stack of the pipeline

The pipeline solution leverages a modern technology stack designed to enable seamless development, testing, deployment, and monitoring of a microservices-based application. Below is a brief introduction to each technology and its purpose:

Containerization: Docker, Docker Hub

- **Docker**: A platform that allows developers to package applications and their dependencies into lightweight, portable containers. It ensures consistency across different environments by encapsulating everything the application needs to run.
- **Docker Hub**: A cloud-based registry where container images are stored and shared. It serves as a central repository for the microservices images of the frontend and backend.

Database: MongoDB Atlas

- **MongoDB Atlas**: A cloud-hosted NoSQL database service offering high availability, scalability, and automated management. It is used to store application data, ensuring seamless integration with the backend microservice.

Continuous Integration

- **GitHub PR Rules**: PR rules enforce code quality by running unit tests before allowing a pull request to be submitted. This prevents broken code from being merged into the main branch.
- **Jenkins**: An automation server that integrates with GitHub to automatically build, test, and push container images to Docker Hub whenever changes are made. It handles various types of tests, ensuring code stability throughout the CI process.

Continuous Deployment

- **ArgoCD**: A GitOps-based tool for Kubernetes deployment. It continuously monitors GitHub for changes in Kubernetes configuration files and automatically starts deployments using a **blue/green deployment strategy** to ensure zero downtime and quick rollback in case of issues.

Automation Testing

- **Jest (Unit and Integration Tests)**: Jest is a JavaScript testing framework used to run unit and integration tests on both the frontend and backend, ensuring that individual components and services function correctly.
- **Puppeteer (End-to-End Testing)**: Puppeteer is used to simulate real user behavior and perform end-to-end (E2E) tests. This includes testing flows such as registration, login, and browsing the store to verify the complete user experience in a controlled environment.

Configuration Management

- **Ansible**: A configuration management tool used at the server level to automate setup tasks, such as installing and configuring Jenkins, Docker, and other necessary components on servers.
- **Helm**: A Kubernetes package manager that simplifies the deployment of Kubernetes applications. It is used to manage application configurations and deploy microservices to the cluster on the EKS environment.

Monitoring

- **Prometheus**: A metrics collection tool that gathers real-time data from the application and infrastructure.
- **Grafana**: A visualization and monitoring tool that provides dashboards and alerts based on the metrics collected by Prometheus.
- **Email Notifications**: Alerts are sent via email for critical issues or failures, ensuring that the DevOps team can take prompt action.

Cloud: AWS Services

- **EKS (Elastic Kubernetes Service)**: AWS-managed Kubernetes service used to run and orchestrate the microservices deployed in Docker containers.

- **EC2 (Elastic Compute Cloud)**: Virtual servers used for hosting the Ansible server, Jenkins server, and other components that are not containerized.
- **Load Balancer**: Distributes incoming traffic across different instances or services, ensuring high availability and fault tolerance.

This stack is designed to automate the CI/CD process, ensure high-quality releases, and provide real-time monitoring and alerts for reliable application management in a production environment.

1.3. Features of the pipeline

- Automated Build and Test Process with Jenkins and GitHub PR rules.
- Controlled deployment via manual image tag updates and ArgoCD-based GitOps.
- Blue/Green Deployment Strategy to minimize downtime and enable fast rollbacks.
- Comprehensive Testing at unit, integration, and end-to-end levels.
- Centralized Configuration Management using Ansible and Helm.
- Real-Time Monitoring and Alerting with Prometheus, Grafana, and email notifications.
- Scalability and High Availability through AWS services (EKS, EC2, Load Balancer).
- Autoscaling for Kubernetes resources with Horizontal Pod Autoscaler.
- Easy deployment and manual/automatic rollback mechanism to EKS cluster with argoCD.

2. The Main Requirements

2.1. Cloud integration

2.1.1. Amazon Web Services (AWS)

The cloud infrastructure is built entirely on **AWS**, leveraging key services such as **VPC**, **Subnets**, **EC2 instances**, and an **EKS cluster** to support the CI/CD pipeline and application deployment.

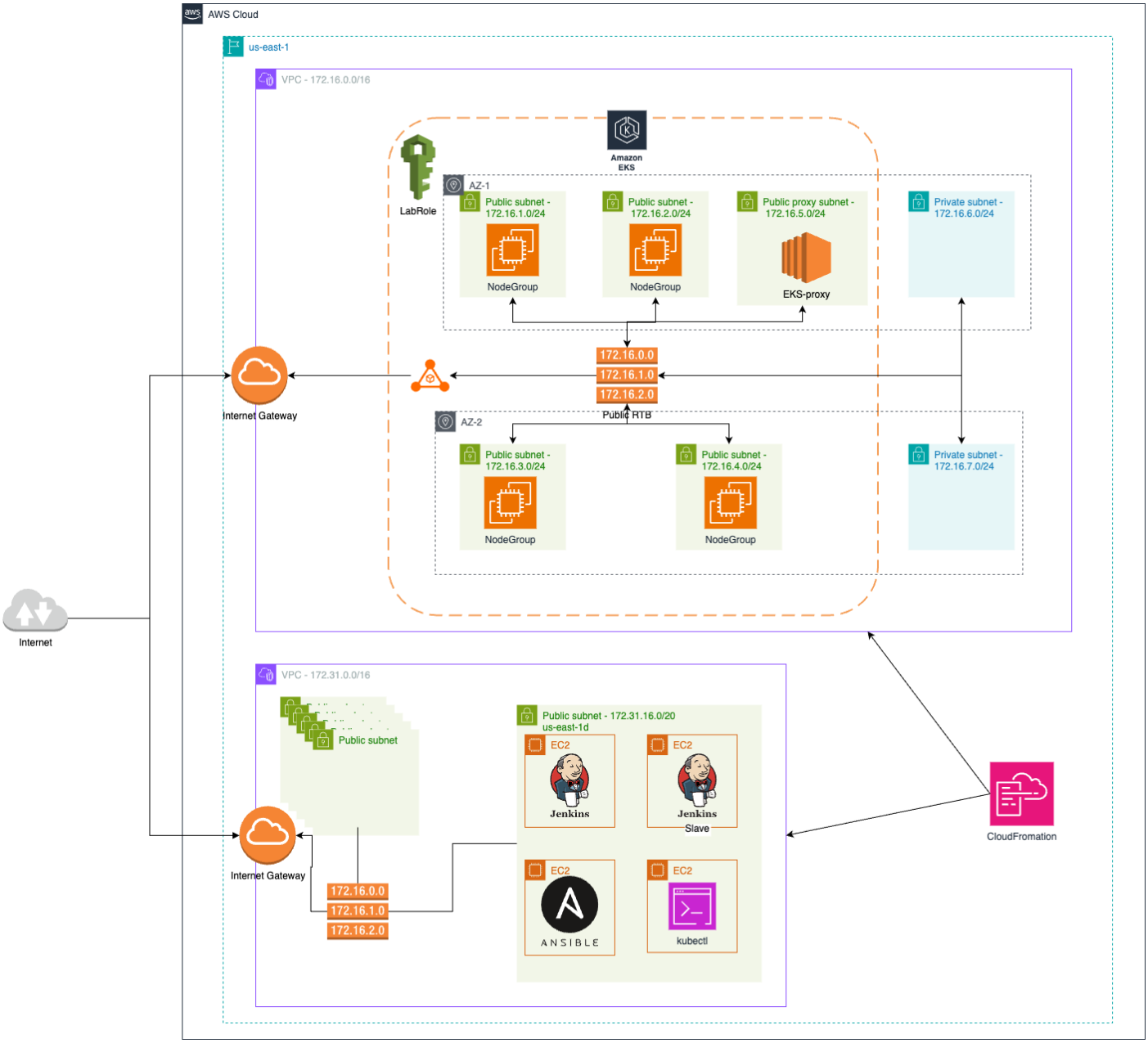


Figure 1. AWS Architecture diagram

VPC (Virtual Private Cloud) acts as an isolated network environment where all resources are deployed. It provides complete control over IP addressing and network access.

Subnets are used to segment the network within the VPC.

- **Public Subnet**: Contains resources requiring internet access, like Jenkins and Ansible servers.

- **Private Subnet**: Contains resources that do not need direct internet access, such as EKS nodes.

The following **EC2 instances** serve different purposes:

Table 1. EC2 instances purposes (see APPENDIX I for specs)

Instance	Purpose
Jenkins Server	Orchestrates the CI/CD pipeline and handles build jobs.
Jenkins Slave	Runs build jobs, tests, and deployments assigned by Jenkins Master.
Ansible Server	Manages configuration automation for other servers and components.
EKS-proxy Server	Installs and configures `kubectl` to manage the **Amazon EKS cluster**.

Each EC2 instance is configured with proper IAM roles and security groups for secure communication.

The **EKS cluster** is deployed within the VPC and uses managed node groups and Application Load Balancer (ALB), where AWS handles scaling and maintenance of worker nodes.

Structure:

- **Control Plane**: Managed by AWS, ensuring high availability without manual intervention.
- **Worker Nodes**: Deployed in private subnets and configured as managed nodes for running containerized microservices.

Purpose:

The EKS cluster orchestrates the containerized frontend and backend microservices, ensuring scalability, load balancing, and high availability. It integrates with ArgoCD for automated deployments and supports blue-green deployment strategies.

We also use an **Application Load Balancer (ALB)** for our deployment to efficiently distribute traffic and enhance the scalability and reliability of our services. For infrastructure provisioning, we rely on **AWS CloudFormation**, enabling automated and consistent deployment of resources across environments.

2.1.2. MongoDB Atlas

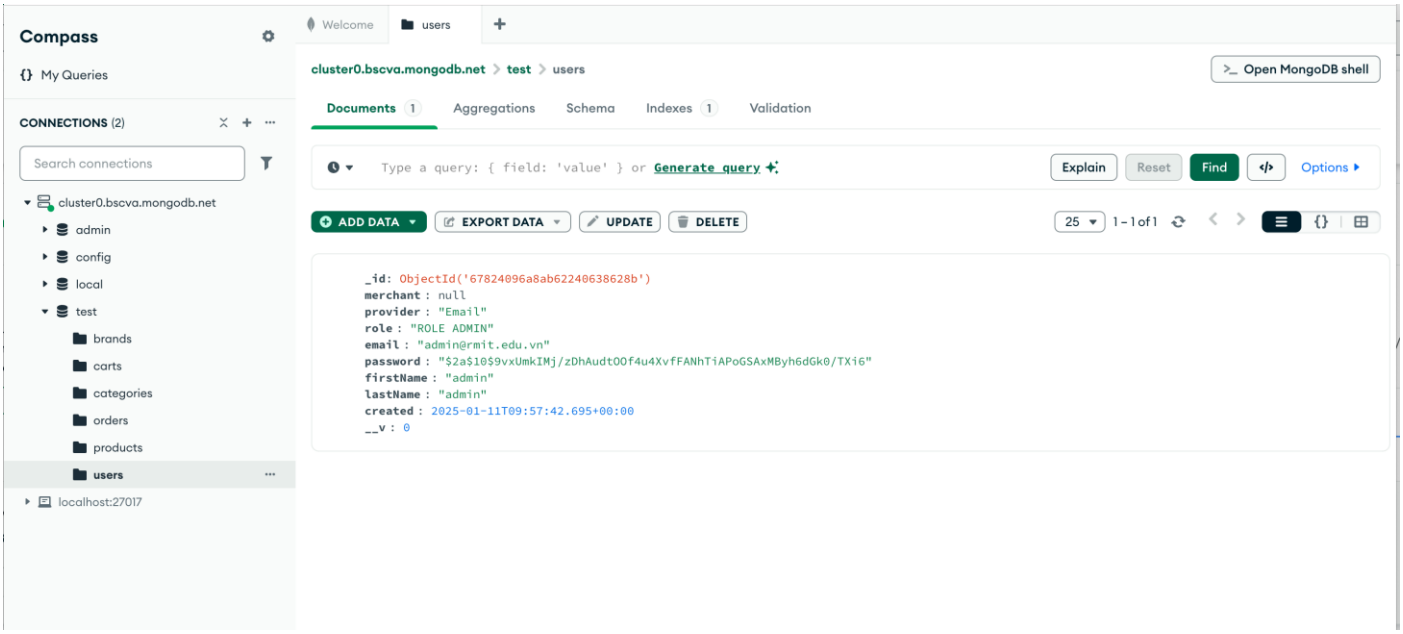


Figure 2. MongoDB Atlas

We chose MongoDB Atlas over self-hosting on EC2 or Kubernetes due to its managed service capabilities, which reduce operational overhead and streamline database management [1]. Atlas simplifies setup, scaling, backups, and security while providing high availability and compliance out of the box. Its autoscaling, performance optimization tools, and pay-as-you-go pricing make it a cost-effective solution compared to the complexities and resource demands of self-hosted options. This allowed us to focus on application development without the burden of managing database infrastructure.

2.2. Containerized Microservices

We containerized the backend and frontend using Docker to ensure consistent runtime environments and simplify deployment and utilizing MongoDB Atlas for the database. Docker provides a consistent runtime environment, eliminating issues related to environment discrepancies, and allows us to build portable and scalable applications [2].

For the frontend, a multi-stage Dockerfile is used to optimize the build and runtime processes. The build stage, based on `node:16`, installs dependencies, builds the app (`npm run build`), and optimizes output for production. The runtime stage uses a lightweight `nginx:alpine` image to serve static files with a custom Nginx configuration, running the app on port 80 for efficient content delivery.

For the backend, a similar multi-stage approach is used. The build stage, based on `node:16-alpine`, installs production dependencies and prepares the app. The runtime stage copies the built app into a clean `node:16-alpine` container, running on port 4000 with `nodemon` for development and `node` for production. This ensures minimal image size and optimized performance.

The Jenkins pipeline automates building and pushing Docker images for both frontend (FE) and backend (BE). It tags images with the current commit hash and timestamp for version tracking and consistency. Using Jenkins' `withCredentials`, Docker Hub credentials are securely managed, enabling authenticated image uploads to their respective repositories. For more details, refer to APPENDIX J.

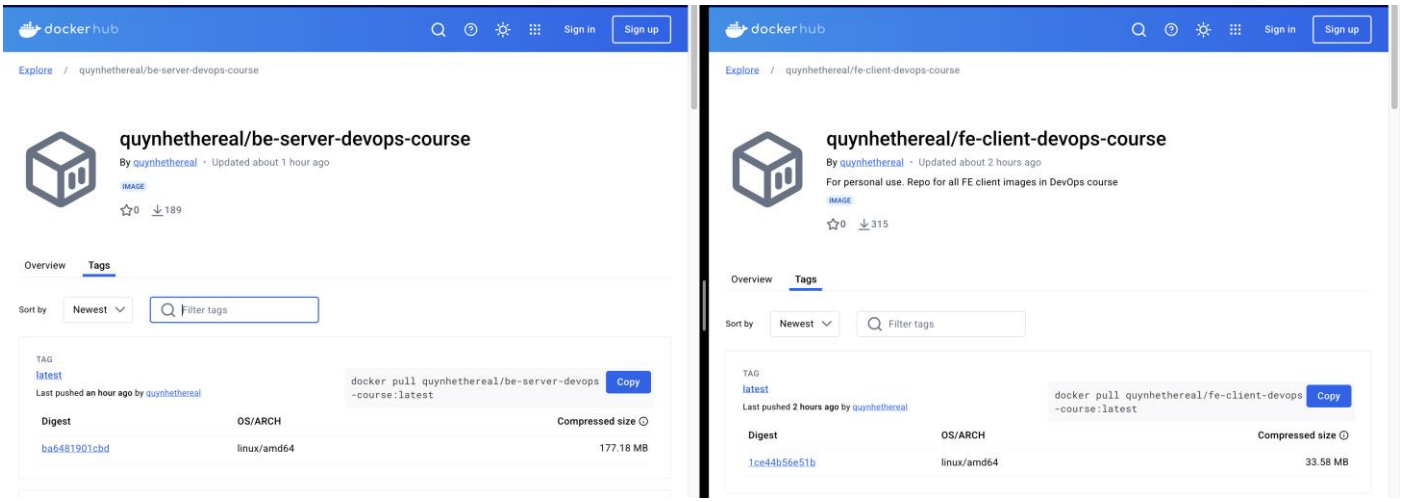


Figure 3. DockerHub Repositories for Backend and Frontend

Figure 3 shows separate repositories for FE and BE to align with **microservices architecture**, allowing independent deployment and scaling of each service. The database is managed externally using **MongoDB Atlas**, ensuring high availability and scalability for data storage.

2.3. Continuous Integration

As we all know, Continuous Integration (CI) is a fundamental practice in modern software development, aimed at enhancing collaboration, ensuring code quality, and accelerating delivery. As development teams often work on different parts of a project simultaneously, integrating their changes frequently becomes essential to prevent integration issues [3]. CI encourages developers to push small, incremental changes to a shared repository regularly, where automated builds and tests are triggered to validate the new code [4]. This approach ensures that defects and conflicts are detected and addressed early, reducing the risk of complex problems during later stages of development.

Our proposed CI pipeline aims to automate key steps in the software development lifecycle, including building, testing, and packaging the application [5]. The pipeline begins with the creation of a Pull Request (PR) by a developer. This PR triggers an automated process that runs unit tests on the Once the PR is approved, it is merged into the main branch, automatically starting the Jenkins pipeline. This pipeline performs a series of tasks to validate, build, and push Docker images for both the frontend and backend services, ensuring that the application is always in a deployable state.

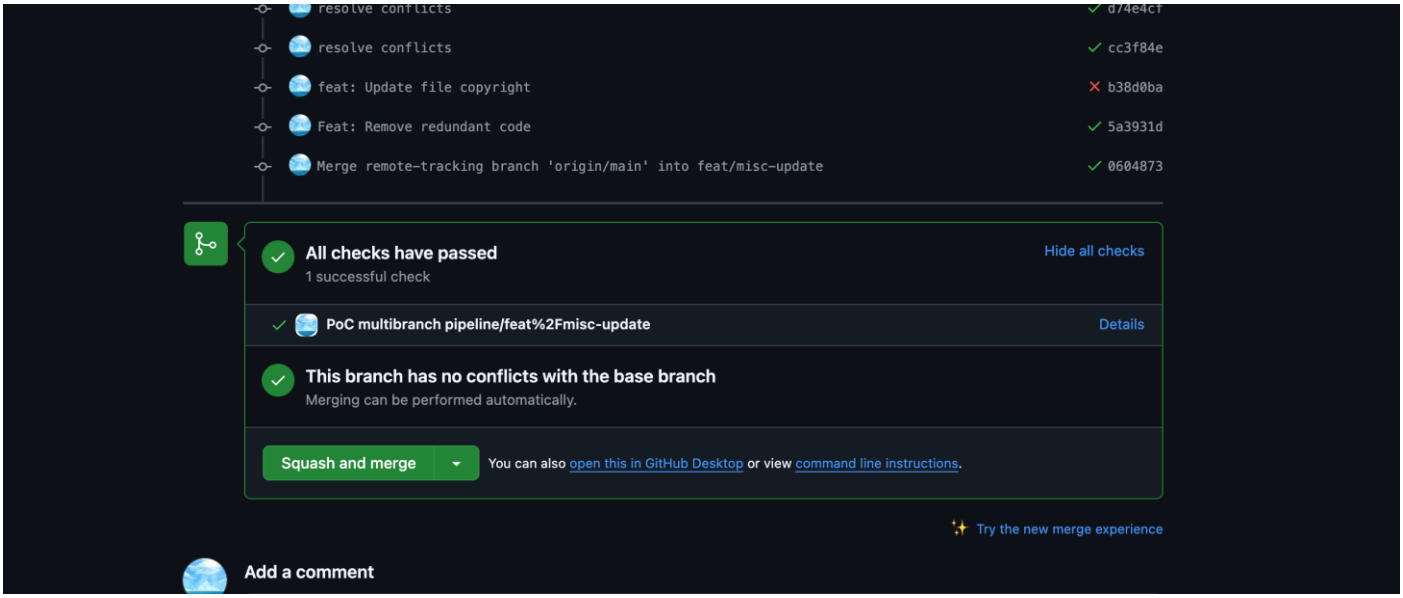


Figure 4. Pull Request Approval for protected main branch

The CI process begins when a developer creates a Pull Request (PR) to the protected main branch. Once the PR is manually reviewed and approved, it can be merged into the main branch, triggering the CI pipeline.

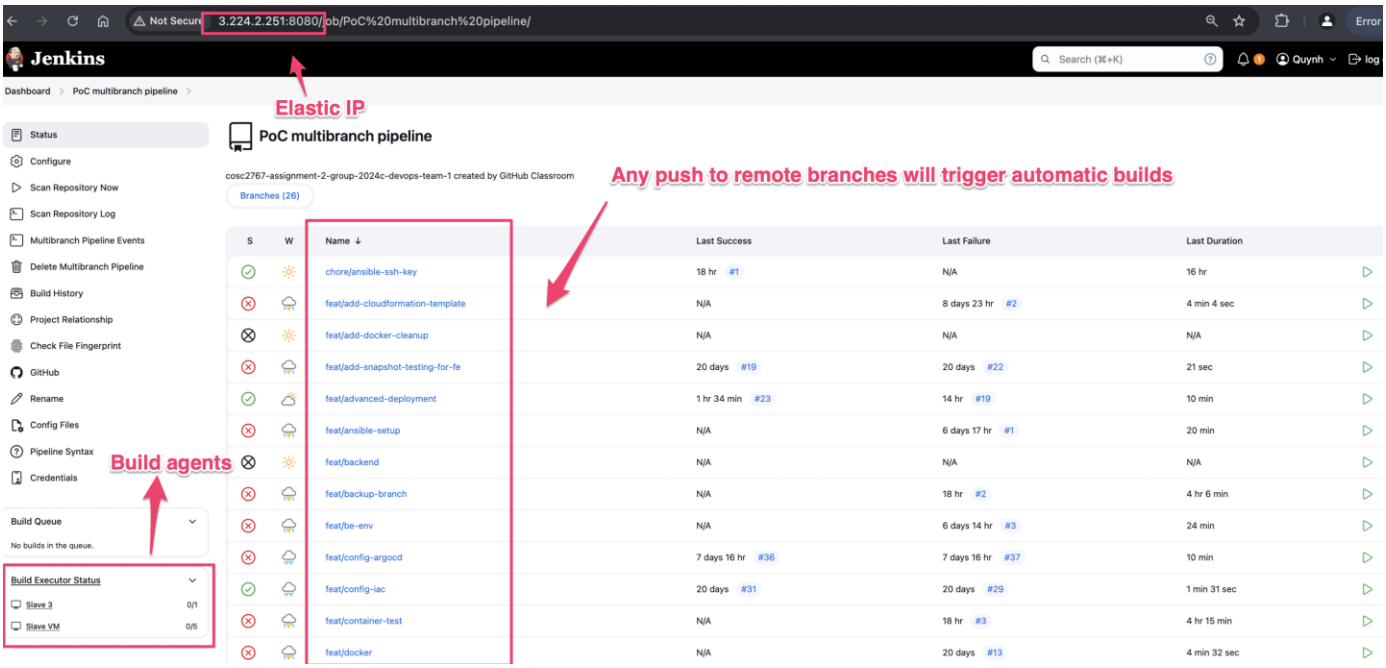


Figure 5. Jenkins pipelines triggered

Once the PR is reviewed and approved by a team member, it is merged into the main branch. This triggers the Jenkins pipeline, which automates the subsequent stages of building, testing, and pushing the updated application, which is explained further in Automation Tool section.

The CI pipeline is configured to start automatically whenever changes are merged into the main branch. This trigger is defined in the project's **Jenkinsfile**, which specifies the stages to be executed during the pipeline

run. By automating the trigger, we ensure that every change to the main branch is immediately validated through the CI process.

2.4. Continuous Delivery

Continuous Delivery (CD) is a practice that automates the deployment of software changes to ensure faster, more reliable releases. By integrating automated processes like testing, artifact creation, and deployment, CD minimizes manual intervention, reduces risks, and ensures that changes can be delivered to production or staging environments seamlessly.

We execute all critical tests on a dedicated testing server, using a docker-compose environment, before building and deploying to our CD pipeline. While this deviates from Tom's original requirements, it ensures that the Docker images are fully tested and functional before deployment. This approach follows the shift-left principle, prioritizing early testing to catch issues sooner and reduce costs [6]. Delaying testing until deployment is often more expensive and disruptive, as issues discovered late in the process require significant rework and can jeopardize release timelines.

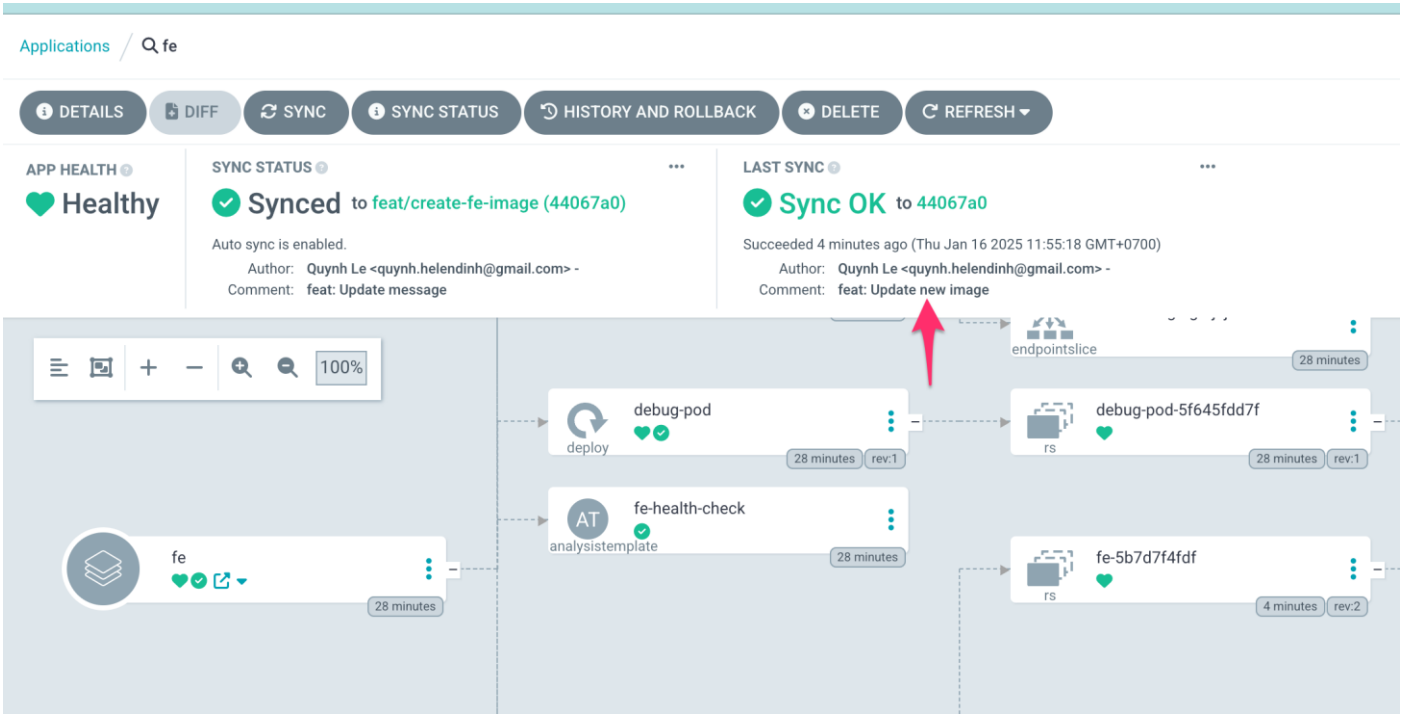


Figure 6. ArgoCD successfully synced

A typical deployment process (assuming the image build step is successful) works as follows:

- The user updates the image tags in the respective frontend and backend manifests with the latest commit-tagged versions.
- ArgoCD automatically syncs periodically, or the user can manually trigger a sync via the ArgoCD UI.

- ArgoCD performs a Blue/Green deployment, running critical tests to ensure the new services with updated images are functional. If successful, the new version of the service goes live.

In the event of failures, ArgoCD automatically alerts the team via email, providing detailed logs and information about the error. This ensures prompt resolution of issues to minimize downtime or delays. While the image tag updates are currently handled manually through pull requests, this limitation can be addressed in the future by integrating GitHub bots to automate the workflow, further streamlining the process. For more details, refer to the Automated Alerting System.

2.5. Testing framework

To ensure the quality of our solution, we integrate comprehensive testing into the CI/CD pipeline for the frontend, backend, and database. Our testing approach includes unit tests for component-level functionality, snapshot tests for UI consistency, integration tests to verify interactions between system components, and end-to-end (E2E) tests to simulate real-world user workflows.

We adopt a **shift-left approach** by incorporating testing as early as possible in the development process. All tests are automatically triggered on every push to feature branches, providing developers with immediate feedback on the quality of their changes. This early detection of issues not only reduces debugging time but also ensures that the system remains stable and reliable throughout the development lifecycle [6].

2.5.1. Unit testing (Jest)

We implement unit tests for both the frontend (FE) and backend (BE) of our solutions, ensuring that individual components and services function correctly in isolation. Our testing framework of choice is **Jest** due to its flexibility, ease of setup, and rich feature set, which includes powerful mocking capabilities, snapshot testing, and an intuitive API [7]. Jest's speed and ability to run tests in parallel also make it ideal for both frontend and backend testing.

We prioritize implementing as many unit tests as possible because they are cost-effective and faster to execute compared to integration tests. Unit tests are easier to write and maintain due to their low coupling with other components, making them an excellent and efficient feedback mechanism for developers. This approach aligns with the **shift-left principle**, as it allows issues to be identified and resolved early in the development cycle [6].

For the frontend, unit tests validate the functionality of individual components under various conditions, ensuring correct behavior and seamless user interactions (e.g., testing callbacks in the SearchBar component). For the backend, unit tests focus on APIs and services, verifying that endpoints handle requests, responses, and edge cases correctly. Service-layer tests ensure proper business logic execution by mocking external dependencies, enabling fast, reliable, and repeatable test results.

Our frontend has 46 test suites with 165 tests (8–10s runtime), and the backend has 27 test suites with 99 tests (52s runtime); see APPENDIX B for details.

2.5.2. Snapshot testing (frontend only)

For the frontend, in addition to functionality testing, we also utilize snapshot testing with the **Jest** framework. Snapshot testing involves capturing a rendered component's output at a specific point in time and comparing it against future outputs to detect unintended changes [8]. These tests are highly effective at identifying subtle changes in components, such as variations in text or color, which might otherwise go unnoticed by developers. For detailed steps and visuals, refer to APPENDIX B.

2.5.3. Integration test (Jest + Puppeteer)

After the unit tests pass, we proceed to build the Docker images for both the backend and frontend individually. Before pushing these images to **Docker Hub** for consumption by the production Kubernetes cluster, we perform integration testing by creating a full-fledged server in the build pipeline using the newly built images. This ensures that the images work seamlessly together in a real-world scenario.

The integration tests are written using **Puppeteer**, a Node.js library that provides a high-level API to control Chrome or Chromium browsers [9]. **Puppeteer** is chosen because it allows for robust testing of user interactions in a headless browser environment, making it ideal for simulating and verifying critical user workflows. The tests are executed on a headless Chrome driver, mimicking real user interactions such as login/logout, adding items to the cart, and checkout (among other interactions). The testing agent interacts with the application as if it were a real user, clicking buttons and navigating the website.

To run this on the Jenkins pipeline, our approach involves using **Docker Compose** to build and orchestrate the images into a running server. Once the server is up, we execute the integration tests by running **yarn test:integration**, which internally triggers the **Puppeteer** scripts. This end-to-end process ensures the integrity and functionality of the application before the images are deployed to production.

We currently have **4 test suites** comprising a total of **7 tests** running in about **25 seconds**, which cover essential application operations such as login, sign-up, page navigation, and checkout. For more details including docker-compose content, test coverage, and test case sign-up flow, refer to APPENDIX B.

2.6. Automation Tool

We use Jenkins to automate our artifact-building process due to its flexibility, extensive plugin ecosystem, and strong community support, which make it highly adaptable to our pipeline needs. Our setup follows a **Jenkins controller-agent architecture**, distributing the builds across multiple virtual machines while keeping the Jenkins master server instance size relatively small [10]. This architecture offloads builds to

Jenkins agents, reducing the load on the master server and minimizing downtime risk. It also enables easy horizontal scaling by adding more agents as needed, with cost efficiency achieved by using t3.medium instances for agents and a t2.micro for the master server to leverage AWS's free tier.

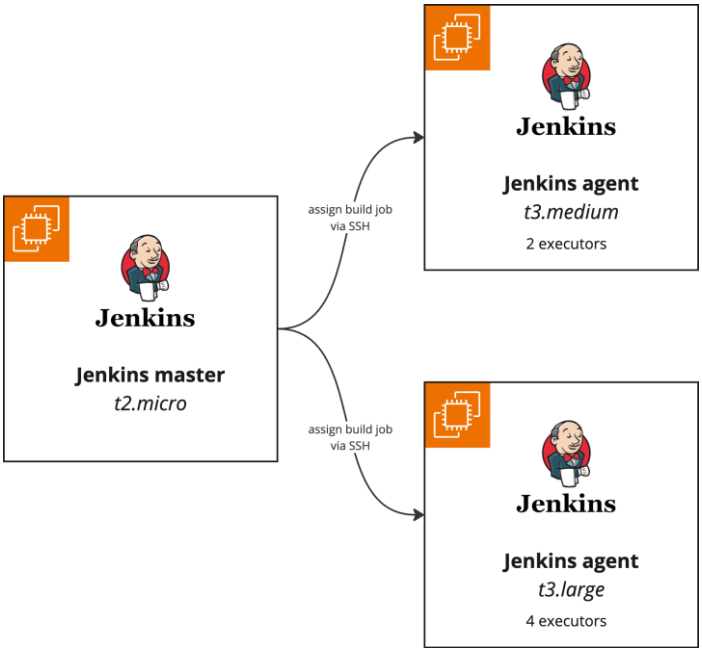


Figure 7. Jenkins Master-Agent Architecture

For the **agent** setup, we begin by installing the necessary dependencies on the agent virtual machines, such as **Docker**, **Jenkins**, and **Java**, using an **Ansible** playbook for automation and consistency.

Once the setup is complete, we configure the Jenkins agents by launching them via SSH. This involves inputting the private key and the agent's public IP address, allowing the Jenkins controller to securely connect to these agents and assign build jobs to them.

For **controller** setup, we follow the standard installation in the lab instructions and configure the necessary plugins like **Git** or **Docker** for image builder tasks. We also attach **an Elastic IP** to the controller server, ensuring easy access for the team and simplifying navigation to the Jenkins dashboard.

For the pipeline, we opted to use a **multibranch pipeline** instead of a **Freestyle** project, declaring the pipeline configuration in a Jenkinsfile located in the root folder of the repository. This approach provides better version control, as the pipeline definition resides alongside the source code, enabling developers to easily modify and review the pipeline configuration through Git [11]. Additionally, this ensures that each branch can have its own tailored pipeline logic if needed.

The Jenkinsfile plays a central role in orchestrating the pipeline, defining each stage from code checkout to image creation and pushing to Docker Hub. This automation reduces manual effort and ensures that each build follows a consistent process. With automated triggers and well-defined stages, the pipeline facilitates a

streamlined and reliable development workflow. For Jenkins automation process and screenshots, refer to APPENDIX C.

2.7. Configuration Management

Ansible is an open-source automation tool designed for configuration management, application deployment, and infrastructure provisioning. It simplifies complex IT tasks by using human-readable YAML playbooks, enabling efficient and repeatable processes across multiple environments. [12]

In our project, we leverage Ansible to automate the deployment and configuration of key components across our **Amazon EKS cluster** and supporting infrastructure. This includes installing and configuring **ArgoCD** for GitOps-based continuous deployment, ensuring a seamless and automated process for managing application deployments and updates directly from version control. Additionally, we set up applications within the cluster, integrating critical tools like the **AWS Load Balancer Controller** to handle ingress traffic and load balancing for services.

Beyond the cluster, we use **Ansible** to streamline the setup of supporting components, such as **Jenkins** controllers and agents, ensuring consistent configuration and scalability for our CI/CD pipelines. Our tasks are structured as Ansible playbooks, which clearly define the commands and processes required to provision and configure these components. This approach enables modularity, making it easier to reuse and update configurations as our requirements evolve.

Ansible's idempotent nature ensures that playbooks can be safely re-executed without causing inconsistencies, making it a reliable tool for managing both initial setups and ongoing updates [13]. We also leverage variables and templates within playbooks to dynamically configure components based on environment-specific parameters, ensuring compatibility across development, testing, and production environments.

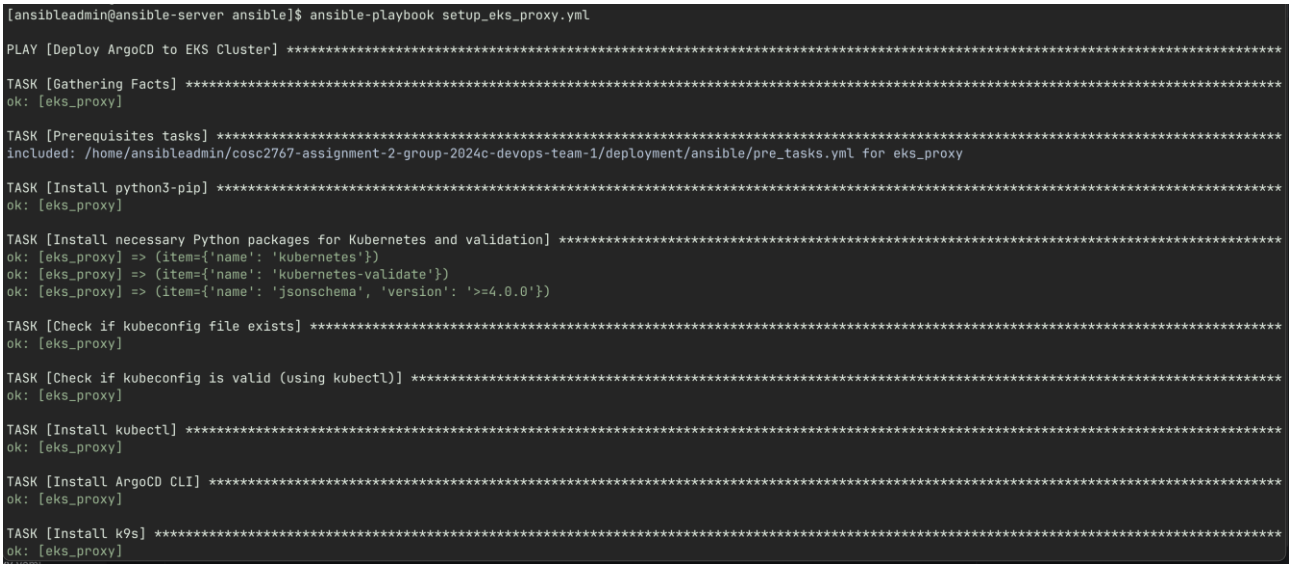


Figure 8 - Ansible Playbook to install custom CRDs

2.8. Automated Alerting System

We implemented automated alerting systems for CI (managed by Jenkins) and CD (managed by ArgoCD) to ensure timely notifications and rapid issue resolution across the deployment pipeline. Jenkins sends email alerts for build failures, providing detailed logs to help the team address issues promptly. Similarly, ArgoCD generates notifications for deployment failures or sync issues, ensuring the team can quickly identify and resolve problems before they impact the live environment. For details including examples, refer to APPENDIX D.

2.9. Orchestration

For orchestration, including autoscaling and managing containerized applications, we leverage AWS Elastic Kubernetes Service (EKS) due to its seamless integration with the AWS ecosystem and its ability to reduce the operational overhead of managing Kubernetes clusters. AWS provides a fully managed control plane, which eliminates the need to manually manage the Kubernetes master nodes, ensuring high availability, scalability, and security [14].

Kubernetes is chosen over Docker Swarm as it offers greater flexibility for advanced architectural setups, such as GitOps-based continuous delivery with ArgoCD, enabling automated and declarative deployments. Additionally, Kubernetes provides better support for scaling complex workloads, making it a more robust choice for modern, production-grade environments. These capabilities allow us to efficiently deploy, scale, and manage our applications while maintaining high reliability and performance. [15]

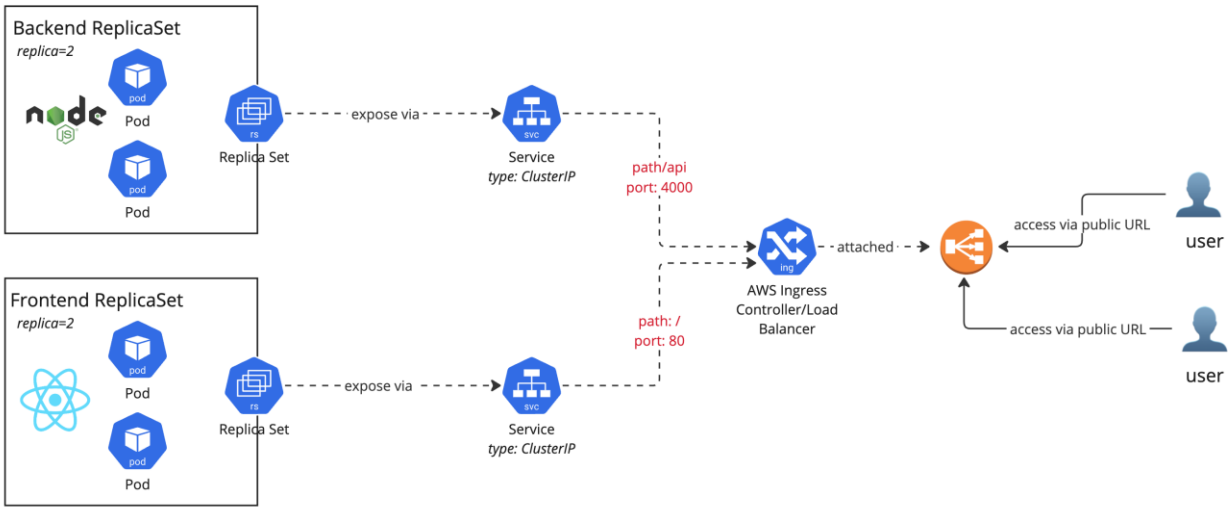


Figure 9. EKS Microservices with Ingress Controller and AWS Load Balancer

We deploy the frontend and backend containerized applications to Kubernetes with a **ReplicaSet** configured to maintain **two replicas** for each application. This setup ensures high availability and fault tolerance by distributing traffic across multiple pods. If one pod fails, another is always available to handle requests, minimizing downtime and ensuring uninterrupted service.

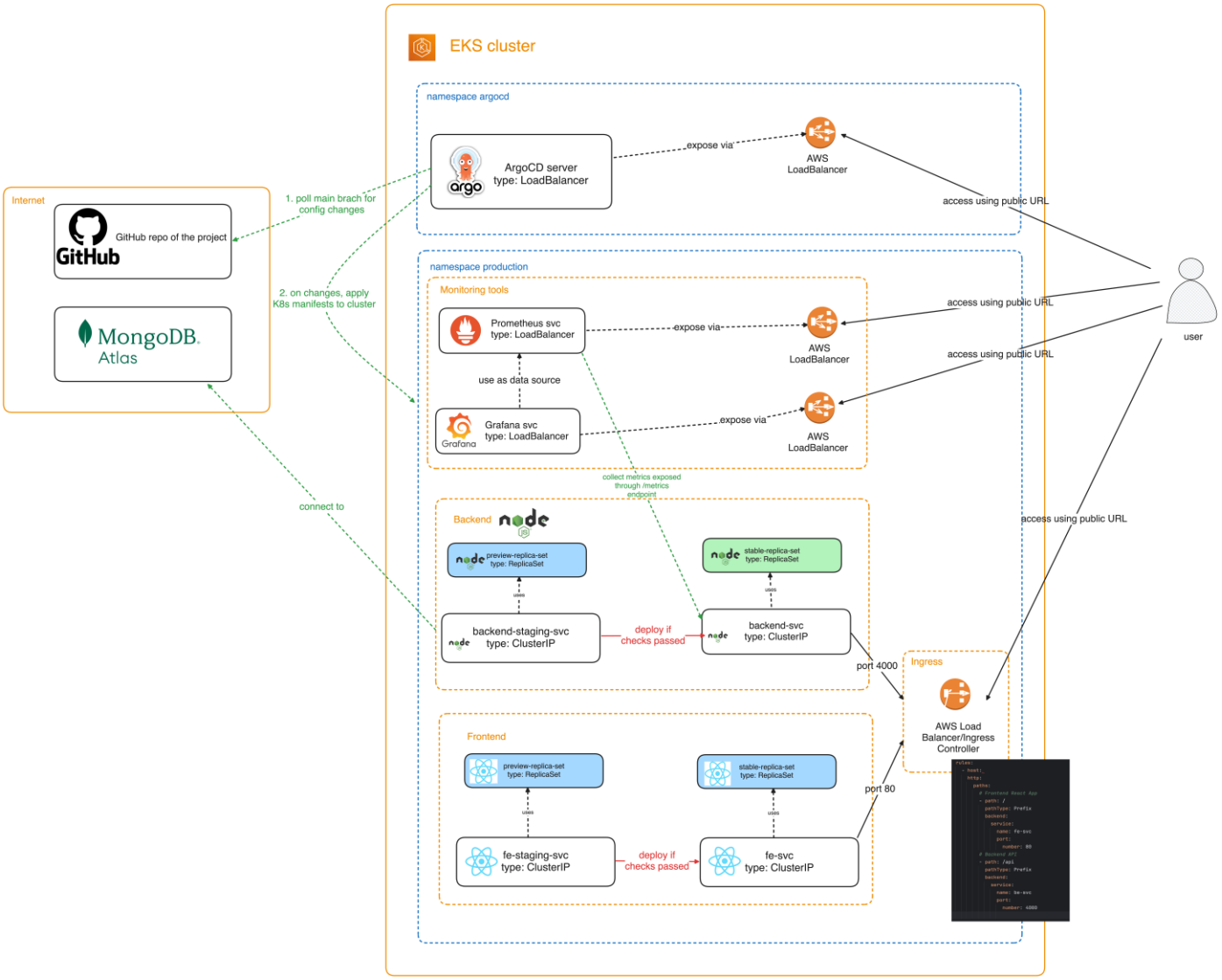


Figure 10. EKS Cluster Diagram

The pods are exposed internally within the cluster using **ClusterIP** services, which provide stable internal IPs for seamless communication between services. To expose the application to external users, we configure an AWS Ingress Controller with an **Application Load Balancer** (ALB). The ALB manages incoming HTTP traffic, directing it to the appropriate services based on defined routing rules. This setup not only ensures secure and efficient traffic handling but also allows for advanced features like SSL termination, path-based routing that we can iterate upon in the future.

Additionally, we have installed **Prometheus** and **Grafana** within the cluster to enable robust monitoring and visualization capabilities. The configuration details for these tools will be covered in the **Monitoring and Alerting** section and for the dashboards, refer to APPENDIX D.

For continuous deployment to the cluster, we utilize **ArgoCD** to ensure seamless and automated application updates. Additionally, for new image updates, we implement Argo Rollouts with a Blue/Green deployment strategy, enabling safe and controlled rollouts by keeping both the old and new versions active until the new

version is verified. These aspects will be discussed in greater detail in the Automatic deployment with ArgoCD.

3. Advanced Requirements

3.1. Infrastructure as Code

For Infrastructure as Code (IaC), we chose **CloudFormation** due to its native integration with AWS. Additionally, we are unable to use **Terraform** in the pipeline because the sandbox environment restricts role creation and non-interactive logins. As a result, we write the configuration files and deploy them manually via the AWS UI.

IaC benefits us in several ways. The most significant advantage is that, given our frequent migrations through the learner's sandbox, configuring through the UI is both tedious and error-prone. IaC streamlines the setup process and ensures consistency across environments. Additionally, it allows for version-controlled infrastructure changes, simplifies collaboration, and improves reproducibility, reducing the risk of configuration drift over time. [16]

We have prepared several **CloudFormation** YAML files for various resources in our architecture, including the EKS cluster and associated networking components such as VPC, network security groups, and administrative infrastructure. This also includes access management, EKS EC2 instance proxy, and EC2 instances to host services like Jenkins master, and slave instances.

To provision the necessary infrastructure, we navigate to **CloudFormation** in the AWS portal and upload the CloudFormation template. After filling in key parameters such as **RoleArn** (LabRole ARN) and **KeyName**, we submit the stack for deployment. This stack provisions essential resources, including the **VPC**, **subnets**, **security groups**, and the **EKS cluster**, granting access for specified roles to interact with the cluster via **kubectl** and the AWS portal. For detailed steps and visuals on how to setup an EKS cluster, refer to APPENDIX H.

3.2. Automatic deployment with ArgoCD

3.2.1. Introduction

To continuously deploy to our Kubernetes cluster, we have chosen ArgoCD because of its GitOps-based workflow, which simplifies application deployment and management through declarative configuration and seamless integration with our CI/CD pipelines.

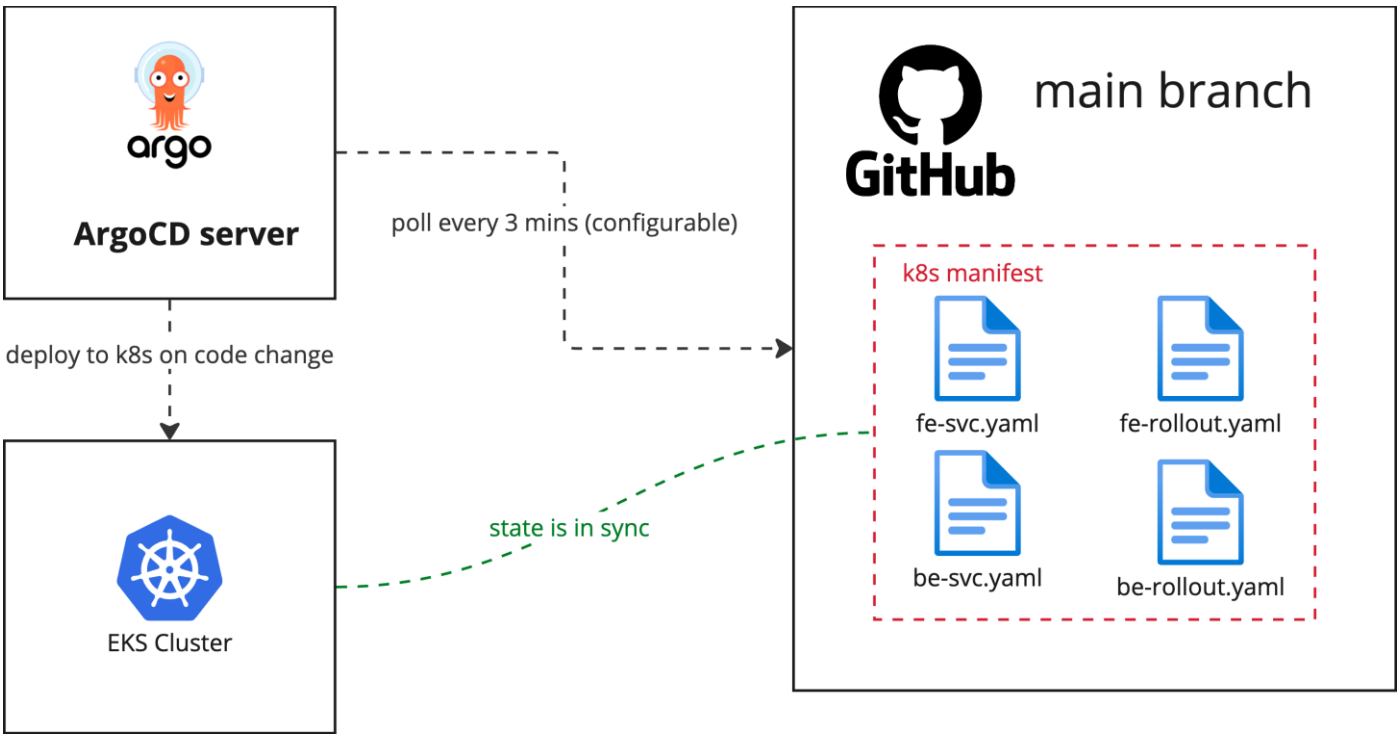


Figure 11. ArgoCD syncing mechanism diagram

ArgoCD watches for Kubernetes manifests defined in the main branch of our Git repository. It periodically polls these files, and whenever changes are detected, it automatically deploys the updated Kubernetes manifests, ensuring the cluster's state matches exactly what's defined in the Git repository. This approach embodies the principles of GitOps, where Git serves as the single source of truth for infrastructure and application configuration. [17]

There are several benefits: it ensures consistency by keeping the cluster state in sync with the repository, provides auditability by tracking changes in Git, and automates deployments, reducing the need for manual intervention and the risk of errors. Additionally, ArgoCD integrates seamlessly with existing CI/CD pipelines, enabling faster deployments and offering scalability for managing multiple clusters. Compared to manually running kubectl commands after each pipeline, ArgoCD streamlines the process, making it more automated and reliable. Lastly, ArgoCD provides continuous self-healing for applications by continuously syncing with Git, ensuring that the desired state defined in the repository is maintained in the cluster. Therefore, even if there is an accidental deletion or disruption within AWS, as long as the environment is operational, ArgoCD will restore the application to its desired state.

Because ArgoCD uses Git as the source of truth, we strongly recommend Tom to protect his main branch from unauthorized changes and only allow well-tested and reviewed code to be merged to it. This is compliant with the GitOps development flow.

3.2.1. Configuration

After initializing the EKS cluster, we installed ArgoCD and ArgoCD Rollouts using Helm by running an Ansible playbook on the EKS Proxy. This instance has the necessary permissions to execute kubectl commands on the EKS cluster. ArgoCD is then patched to expose the application via a load balancer for easy access. The

playbook also configures the connection to the EKS cluster via a kubeconfig file and installs K9s, a terminal-based UI that simplifies the management of Kubernetes clusters by providing an intuitive interface for inspecting cluster resources and monitoring deployments. Finally, the playbook sets up a user-friendly admin password for easy login. For more information on the Ansible setup, refer to **Ansible** section.

For ArgoCD setup, we leverage the ArgoCD "App of Apps" pattern, where a master ArgoCD application manages sub-applications such as Prometheus, Grafana, and the backend/frontend application [18]. This setup offers several benefits, including streamlined management of multiple applications, improved scalability, and simplified monitoring and upgrades, all from a single point of control.

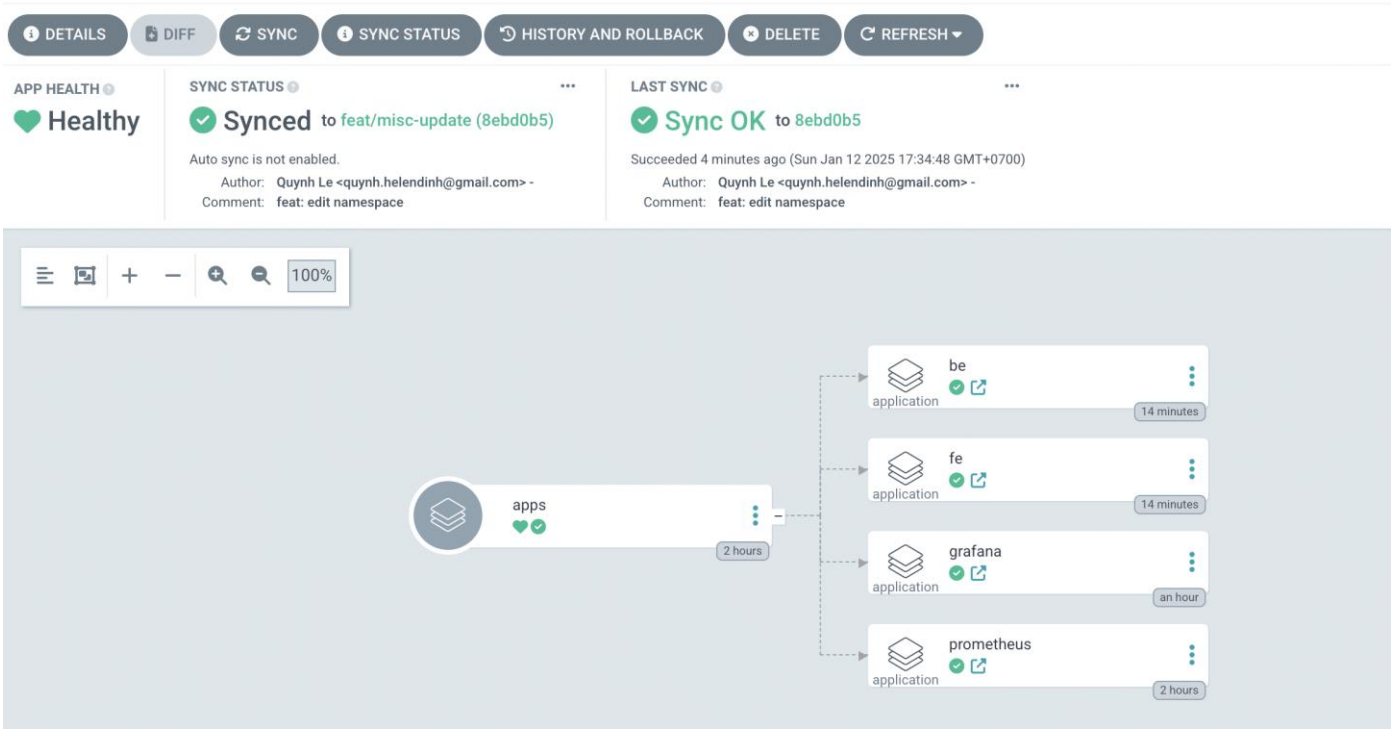


Figure 12. ArgoCD GUI with its App of Apps pattern

With this approach, we can easily bootstrap complex configurations and deploy applications efficiently. This is particularly important for us, as we rely on multiple sandboxes due to budget constraints and need a quick, reliable way to spin up everything. Additionally, it significantly improves debugging, as Git serves as the single source of truth. Any unauthorized changes—often made during testing—are eliminated, ensuring consistency and reducing troubleshooting efforts.

3.3. Monitoring and Alerting

For monitoring and alerting, we have selected **Prometheus** and **Grafana** due to several reasons. These tools are open-source, cost-effective, and widely adopted in the industry for observability. Prometheus provides robust metrics collection and alerting capabilities, while Grafana excels in data visualization, enabling us to create insightful dashboards. Additionally, both tools integrate seamlessly, making them a powerful combination for our observability stack. Their active community support and extensibility further enhance their value as reliable solutions for monitoring and alerting. [19] [20]

Prometheus and **Grafana** are deployed using a declarative **GitOps** approach, with configurations stored in our Git repository and deployed by **ArgoCD**. This setup ensures that changes are automatically applied to the cluster, reducing manual effort and human error. Unlike using a Kubectl proxy on EC2 instances, this method is more cost-effective, especially given the high frequency of testing in our clusters. For a detailed configuration process of the monitoring setup, refer to APPENDIX G.

We collect metrics from the default **Prometheus** client library for Node.js to monitor system performance, including CPU usage, memory consumption, and process lifecycle. Key metrics include CPU time (*process_cpu_seconds_total*), process start time (*process_start_time_seconds*), memory usage, Node event loop.

Additionally, custom metrics are implemented to track application-specific performance, such as HTTP request counts (*http_request_count*), request durations (*http_request_duration_seconds*), active connections (*nodejs_active_connections*), failed requests, and memory usage. These metrics provide deeper insights into the application's API health and resource usage. These metrics are used as input for Grafana dashboards (see APPENDIX D), allowing for real-time visualization and analysis of application performance. This setup helps identify performance bottlenecks, monitor trends, and ensure optimal application health.

3.4. Advanced Deployment Strategy (Blue/Green)

Due to time constraints and considering Tom's scale, we opted for Blue/Green (B/G) deployment over Canary, as it is simpler and more suitable for our current needs [21]. Blue/Green deployment involves maintaining two environments, Blue (current) and Green (new), allowing for smooth rollouts and quick rollback in case of issues.

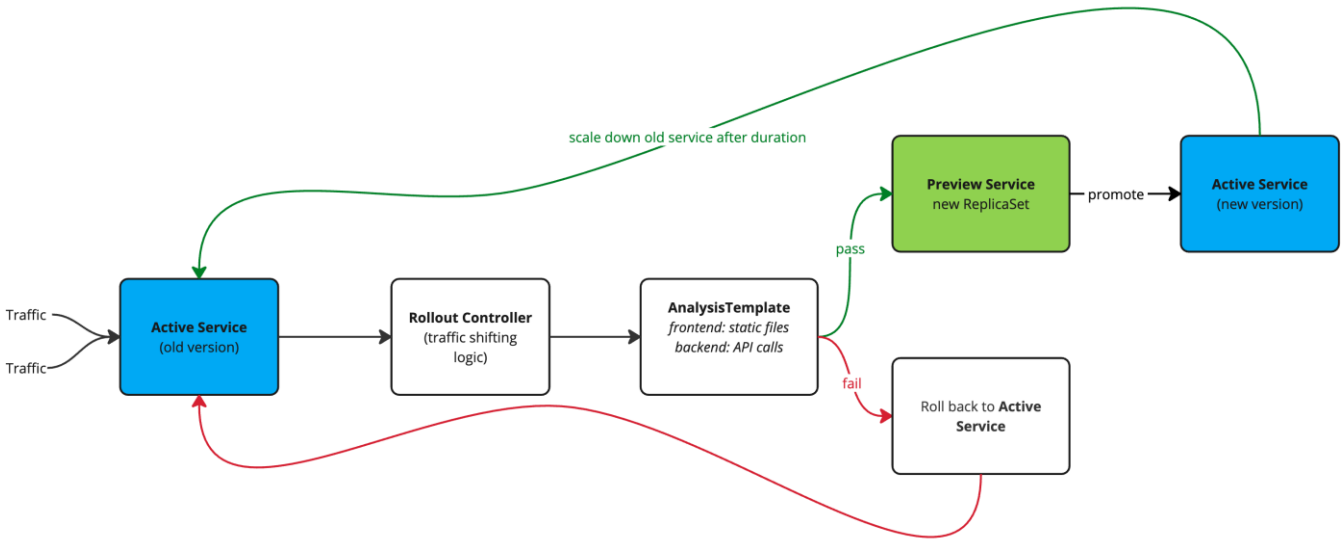


Figure 13. ArgoCD Rollout process diagram

ArgoCD Rollouts works by directing traffic through a unique hash for each ReplicaSet, with the Active Service routing traffic to the old version and the Preview Service to the new version. When the *.spec.template* field changes, a new ReplicaSet is created, and traffic is routed accordingly. Once the new ReplicaSet is available, the Active Service points to it, and after a delay defined by blueGreen.scaleDownDelaySeconds, the old ReplicaSet is scaled down [22].

To implement Blue/Green, we use ArgoCD Rollouts. We replaced the standard Deployment manifest with a Rollouts manifest that defines the deployment strategy, specifying the Active (current) and Review (new) services (see APPENDIX F).

In our configuration, we included an **Analysis Template CRD** to perform custom checks on the preview (blue) service before finalizing deployment. For the frontend, we verify that the preview service serves static files correctly, including necessary containers (e.g., root). For the backend, we execute typical API calls to ensure the responses return successful status codes (2xx) and that the results contain valid data, confirming that the database connection is functioning as expected.

If the analysis is successful, traffic is gradually shifted from the active service to the preview service, promoting it to production. However, if the analysis fails, the rollout process is halted, and the active service continues handling all traffic, preventing any potential disruption to users. This ensures a safe and controlled deployment process.

3.5. Autoscaling

For **autoscaling**, we are using **Horizontal Pod Autoscaler** (HPA), which is a Kubernetes feature that automatically adjusts the number of pods in a deployment, replication controller, or stateful set based on observed CPU, memory, or custom metrics.

HPA ensures that applications can scale dynamically to handle variable workloads efficiently, preventing over-provisioning and under-utilization of resources. We configure resource requests and limits for both the front-end (FE) and back-end (BE) services, specifically for CPU and memory usage. This ensures that the HPA has clear thresholds to determine when to scale up or down. For screenshot of HPA configuration, refer to APPENDIX K.

To verify this functionality, we used hey, a command-line tool that simulates load traffic. We ran it against the ingress endpoint for 5 minutes and observed that the Horizontal Pod Autoscaler (HPA) effectively triggered scaling. As a result, from our load testing (see Load Testing with Hey in APPENDIX B), the number of replicas increased from 2 to 6. This demonstrated that the system is capable of elastic scaling, dynamically adjusting resources based on the user request load to maintain performance and stability.

4. Application Flow (Diagram)

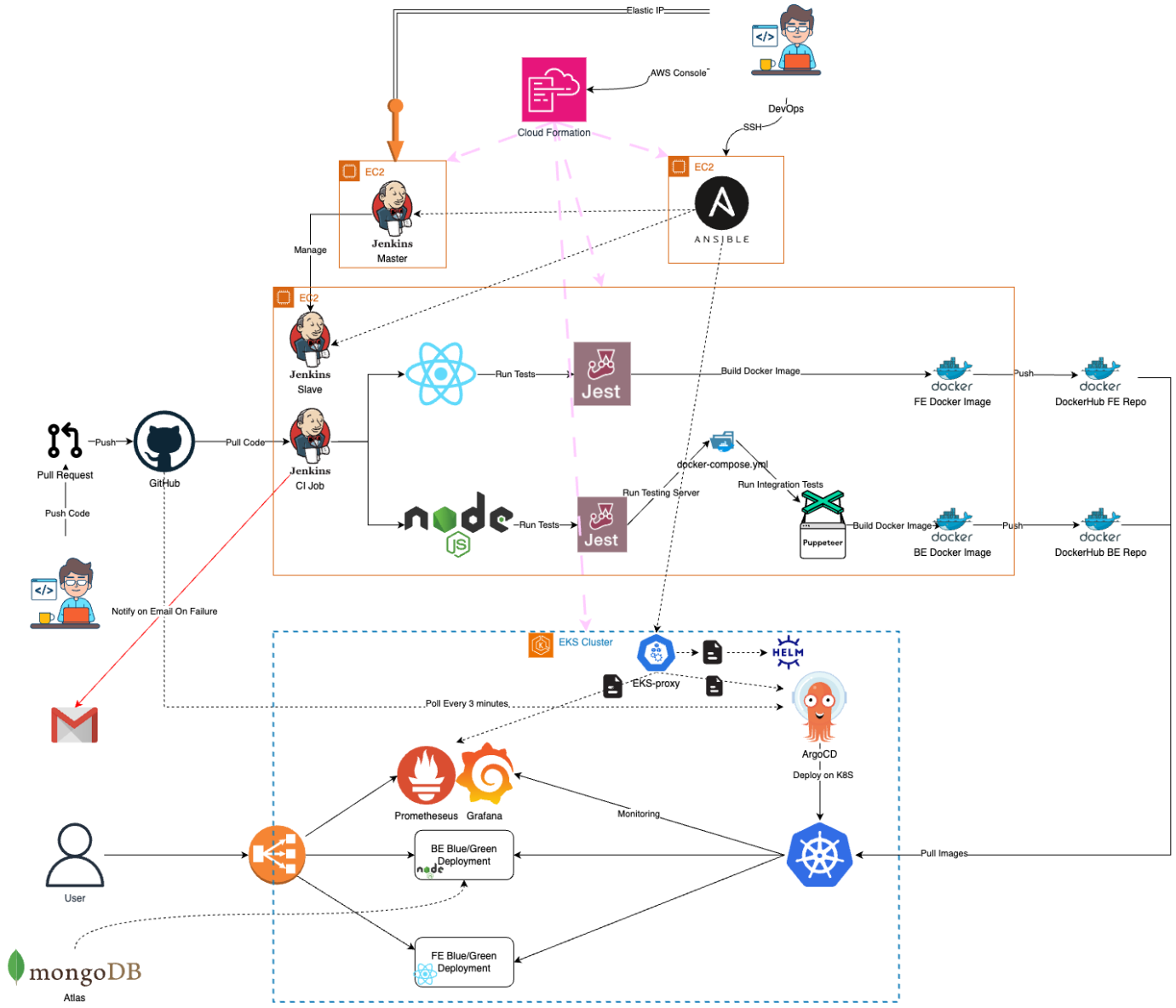


Figure 14. Application Flow

From the diagram above, our DevOps process begins with provisioning cloud infrastructure using AWS CloudFormation and configuring servers via Ansible playbooks. The application follows a Git-based workflow, where developers create Pull Requests (PRs) to the protected main branch. Upon approval, the CI pipeline orchestrated by Jenkins handles building, testing, and pushing Docker images for both frontend and backend services to Docker Hub. In the Continuous Deployment (CD) phase, the DevOps user manually updates image tags, and ArgoCD automatically deploys the updated application using a blue/green strategy. Real-time monitoring is handled by Prometheus, with dashboards provided by Grafana. Alerts ensure that any deployment issues are promptly addressed. For a detailed breakdown of the DevOps process, refer to APPENDIX E.

5. Known Bugs/Problems

Currently, the only significant issue we have encountered is related to resource management when spreading workloads across multiple accounts. This can lead to complexities in tracking resource utilization, managing permissions, and ensuring efficient collaboration. While it does not directly impact the system's functionality, it adds overhead to maintaining consistency and visibility across environments. Addressing this issue will require streamlining resource allocation and potentially consolidating management workflows in the future.

6. Evaluation, Reflection and Responsibilities

6.1. The Evaluation and Reflection

6.1.1. Evaluation

Our pipeline leverages AWS services (EKS, EC2, CloudFormation) to create a scalable cloud infrastructure with automated provisioning. Dockerized microservices deployed on Kubernetes ensure consistent environments and high availability. The CI process with Jenkins automates builds, tests, and image creation, while CD with ArgoCD ensures zero-downtime deployment using a blue-green strategy.

Multi-level testing (unit, integration, and E2E) ensures application reliability. Ansible and Helm simplify configuration management, while Prometheus, Grafana, and email alerts provide real-time monitoring and quick issue resolution. We met most advanced requirements, but future improvements could include canary deployment and expanded automation for better control and scalability. Overall, this solution provides a robust, automated CI/CD pipeline that enhances development efficiency, deployment reliability, and system scalability. For details of how we assess our pipeline solution, refer to Appendix A.

6.1.1. Challenges and how we overcame them

During the project, we encountered several key issues, most of which were related to the constraints of the lab account (e.g., inability to create new IAM roles, lack of IAM credential-based access, and budget limitations). Additionally, we faced technical challenges in learning new technologies and adapting existing ones to fit our context and solution. Below is a summary of the key issues and how we addressed them:

Issue 1: Budget Constraints when centralizing resources in one sandbox

Problem Definition: The sandbox budget was limited to $50. Running an EKS cluster proved costly, with expenses averaging $3–$4 per day. Centralizing all configurations into a single sandbox would quickly deplete the budget.

Solution: To mitigate this, we created multiple sandboxes, each dedicated to specific purposes (e.g., Ansible, Jenkins, etc.). This approach distributed the budget load across multiple sandboxes, allowing us to stay within budgetary limits.

Issue 2: Restrictions on IAM Role Creation and credential generation

Problem Definition: The lab environment did not allow the creation of new IAM roles or the generation of IAM credentials. This prevented programmatic interactions with AWS using Terraform or CloudFormation in our pipelines. Additionally, using the same LabRole for both EC2 instances within the EKS Node Group and as a Standard Access Entry caused a bug where the EKS CSR (Certificate Signing Request) was not issued. [23]

Solution: For CloudFormation, we manually imported configurations via the AWS Management Console. For EKS, we accepted the limitation that we would be unable to view logs or execute commands on pods after the CSR expired.

Issue 3: Computing resources are not enough for our pipeline

Problem Definition: Initially, we used a single Jenkins instance for both the master and build processes. However, we encountered an issue where the builds were unable to complete due to insufficient computing resources. To address this, we created agent instances with larger instance sizes to distribute the load. However, even the *t2.medium* instances occasionally became unresponsive due to high load.

Solution: Through trial and error, we observed that instances with more RAM performed better than those with larger CPUs but less RAM. Additionally, we realized that regular workspace pruning was essential, as Docker builds generated large artifacts that could quickly consume all available storage on an instance. To address this, we incorporated a cleanup step into the Jenkins pipeline to regularly clean up the workspace and free up space.

Further plan: To further optimize performance, we could implement an autoscaling mechanism for Jenkins slave instances. This would automatically scale the number of slave nodes based on the build load, ensuring that resources are dynamically adjusted and preventing instances from becoming overloaded.

Issue 4: Inexperience on GitOps development flow with ArgoCD

Problem Definition: While setting up continuous deployment to EKS, we identified ArgoCD as a potential solution. However, our limited experience with EKS posed significant challenges in configuring and using ArgoCD effectively.

Solution: We referenced ArgoCD example pipelines and deploy them into sandboxes to understand how it works and replicated to current solution.

6.2. Reflection & Experience from the Guest Lecturer

Despite not being able to attend the NAB guest lecture, we learn a lot from it. It emphasized the need for scalable infrastructure to handle traffic spikes, particularly in industries like banking and e-commerce. This aligns with the project, where deployment on **AWS EKS** with Kubernetes ensures automatic scaling through replica sets for the frontend and backend, maintaining high availability during peak loads. Workflow optimization was another key point, reflected in our fully automated **CI/CD pipeline** using **Jenkins**, **Ansible**,

and **ArgoCD**. Jenkins handles orchestration, Ansible manages server configurations, and ArgoCD enables GitOps-based continuous delivery, reducing manual effort and improving efficiency.

Security practices, such as token rotation and managing sensitive data, were also highlighted. While our project does not currently implement token rotation, adopting such practices in the future would enhance security. The speaker also discussed emerging DevOps fields like **FinOps** and **DevSecOps**. The project incorporates **DevSecOps** principles by using **Prometheus** and **Grafana** for real-time monitoring and alerting. Additionally, FinOps practices can be integrated to track cloud usage and optimize costs, ensuring a more efficient and cost-effective infrastructure.

6.3. Project Responsibilities

Refer to **Appendix A** for information on the contributions of team members across the project's deliverables (as reflected in the rubric) and detailed breakdown of the work.

7. Conclusion

Through this assignment, we gained valuable hands-on experience in designing and implementing a complete CI/CD pipeline using widely adopted DevOps tools, including **Jenkins, Ansible, Docker, ArgoCD, and Kubernetes (AWS EKS)**. We learned how automating key processes—such as code builds, testing, and deployments—can significantly enhance efficiency, consistency, and reliability. By deploying containerized microservices and managing them via Kubernetes, we developed a strong understanding of scalability, resource management, and maintaining high availability.

Additionally, working with Infrastructure as Code (IaC) using AWS CloudFormation demonstrated how automation can simplify infrastructure provisioning and management. The integration of Prometheus and Grafana for real-time monitoring and alerting reinforced the importance of observability in maintaining system stability. This assignment also provided us with insights into implementing blue-green deployment strategies to minimize downtime during updates. Overall, we learned how combining these tools and practices can streamline the software delivery lifecycle and improve operational efficiency.

The project also enhances our understanding of core DevOps practices and acquired critical skills in automation, orchestration, and infrastructure management. Setting up and managing a CI/CD pipeline with Jenkins for orchestration, Ansible for configuration management, and ArgoCD for GitOps-based continuous delivery gave us practical insight into automating deployments and reducing manual intervention. Deploying containerized applications using Docker and orchestrating them with Kubernetes helped us learn how to manage scalable, reliable, and fault-tolerant applications.

Furthermore, the integration of monitoring and alerting tools like Prometheus and Grafana helped us understand the importance of DevSecOps principles in maintaining system health and ensuring timely

incident response. We also gained experience with GitOps workflows, which emphasized the value of version-controlled infrastructure and application state. Beyond technical skills, we improved our ability to solve complex problems, manage tasks efficiently, and collaborate effectively in DevOps workflows. This assignment provided us with a comprehensive understanding of real-world DevOps implementations.

8. References

[1] MongoDB, "MongoDB Atlas - Amazon Web Services (AWS)," MongoDB, [Online]. Available: https://www.mongodb.com/docs/atlas/reference/amazon-aws/. [Accessed 16 January 2025].

[2] B. Cotton, "5 Benefits of a Container-First Approach to Software Development," 14 August 2023. [Online]. Available: https://www.docker.com/blog/5-benefits-of-a-container-first-approach-to-software-development/. [Accessed 12 January 2025].

[3] S. Pittet, "How to setup continuous integration," Atlassian, [Online]. Available: https://www.atlassian.com/continuous-delivery/continuous-integration/how-to-get-to-continuous-integration. [Accessed 14 January 2025].

[4] K. Zettler, "Trunk-based development," Atlassian, [Online]. Available: https://www.atlassian.com/continuous-delivery/continuous-integration/trunk-based-development. [Accessed 14 January 2025].

[5] AWS, "What is SDLC (Software Development Lifecycle)?," AWS, [Online]. Available: https://aws.amazon.com/what-is/sdlc/. [Accessed 14 January 2025].

[6] Microsoft, "Shift testing left with unit tests," 29 November 2022. [Online]. Available: https://learn.microsoft.com/en-us/devops/develop/shift-left-make-testing-fast-reliable. [Accessed 12 January 2025].

[7] Jest-yepitschunked, "Jest Documentation | v29.7," 16 January 2024. [Online]. Available: https://jestjs.io/docs/getting-started. [Accessed 12 January 2025].

[8] S. Bekkhus, "Snapshot Testing," 25 September 2023. [Online]. Available: https://jestjs.io/docs/snapshot-testing. [Accessed 12 January 2025].

[9] Puppeteer, "What is Puppeteer? v24.0.0," [Online]. Available: https://pptr.dev/guides/what-is-puppeteer. [Accessed 12 January 2025].

[10] Jenkins, "Jenkins Docs - Architecting for Scale - Distributed Builds Architecture," [Online]. Available: https://www.jenkins.io/doc/book/scaling/architecting-for-scale/#distributed-builds-architecture. [Accessed 12 January 2025].

[11] Jenkins, "Jenkins Docs - Branches and Pull Requests," [Online]. Available: https://www.jenkins.io/doc/book/pipeline/multibranch/. [Accessed 12 January 2025].

[12] Ansible, "Ansible Docs - Introduction to Ansible," [Online]. Available: https://docs.ansible.com/ansible/latest/getting_started/introduction.html. [Accessed 12 January 2025].

[13] Ansible, "Ansible Playbook - Desired state and 'idempotency'," 10 January 2025. [Online]. Available: https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_intro.html#desired-state-and-idempotency. [Accessed 12 January 2025].

[14] AWS, "EKS Control Plane," AWS, n.d. [Online]. Available: https://docs.aws.amazon.com/eks/latest/best-practices/control-plane.html. [Accessed 12 January 2025].

[15] R. Powell, "Docker Swarm vs Kubernetes: how to choose a container orchestration tool," 8 April 2024. [Online]. Available: https://circleci.com/blog/docker-swarm-vs-kubernetes/. [Accessed 12 January 2025].

[16] HashiCorp, "Infrastructure as Code: What Is It? Why Is It Important?," 9 October 2023. [Online]. Available: https://www.hashicorp.com/resources/what-is-infrastructure-as-code. [Accessed 12 January 2025].

[17] ArgoCD, "Architectural Overview," [Online]. Available: https://argo-cd.readthedocs.io/en/stable/operator-manual/architecture/. [Accessed 12 January 2025].

[18] ArgoCD, "App Of Apps Pattern," [Online]. Available: https://argo-cd.readthedocs.io/en/stable/operator-manual/cluster-bootstrapping/#app-of-apps-pattern. [Accessed 12 January 2025].

[19] Prometheus, "Prometheus Docs - Overview," [Online]. Available: https://prometheus.io/docs/introduction/overview/. [Accessed 12 January 2025].

[20] Grafana, "Grafana Docs," [Online]. Available: https://grafana.com/docs/grafana/latest/. [Accessed 12 January 2025].

[21] R. Powell, "Canary vs blue-green deployment to reduce downtime," 23 December 2024. [Online]. Available: https://circleci.com/blog/canary-vs-blue-green-downtime/. [Accessed 12 January 2025].

[22] ArgoCD, "BlueGreen Deployment Strategy," [Online]. Available: https://argo-rollouts.readthedocs.io/en/stable/features/bluegreen/. [Accessed 12 January 2025].

[23] Github, "[EKS] [BUG]: CSR is not returning the certificate after approval in 1.21 · Issue #1604 · aws/containers-roadmap," 2024. [Online]. Available: https://github.com/aws/containers-roadmap/issues/1604. [Accessed 17 Jan 2025].

9. Appendices

APPENDIX A – PROJECT RESPONSIBILITIES

Table 2. Project contribution summary table

Team member	Overall contribution (%)
Le Dinh Ngoc Quynh - s3791159	50
Vuong Gia An - s3757287	12.5
Le Minh Duc - s4000577	12.5
Tran Minh Nhat - s3926629	12.5
Vu Tien Quang - s3981278	12.5

Table 3. Project responsibilities breakdown table

Name - sID	Responsibilities	Notes
Le Dinh Ngoc Quynh - s3791159	**Project Management:** - Main point of contact with the lecturer, addressing any issues and providing timely updates. - Organized meetings to finalize workload distribution and devise solutions. - Created and managed the project backlog, assigned tasks to the team, and consistently followed up on progress. **Frontend Testing:** - Developed 95% of unit and snapshot tests for the frontend, ensuring high code quality and reliability. **Backend Testing/Integration testing:** - Responsible for 70% of the integration tests, ensuring proper functionality between system components. - Set up a fully functional pipeline for integration test execution, using Jest within Docker containers. **Jenkins Configuration:** - Created 95% of the Jenkins configuration, including master-agent setup and integration with GitHub through webhooks. - Implemented Jest GitHub status checks to monitor test results directly within the GitHub interface.	Number of tests written: at least 70 files

- Developed a declarative Jenkinsfile pipeline for the frontend, automating build and deployment processes.

Database Configuration:

- Set up a MongoDB Atlas instance and updated the application configuration to point to this new database.

Ansible Playbooks:

- Developed all Ansible playbooks for the setup of Jenkins master/agents, EKS proxy, installation of ArgoCD, and ArgoCD namespaces.
- Configured EKS addons, including EBS, to ensure seamless cluster operations.

CloudFormation:

- Created all CloudFormation templates for EKS deployment and EC2 proxy configuration, enabling scalable cloud infrastructure.

Orchestration and ArgoCD Deployment:

- Configured EKS environments, managing all Kubernetes manifests and deployment files.
- Led all ArgoCD deployments and integrated them with Git, ensuring smooth CI/CD pipelines and version control.

Blue-Green Deployment:

- Integrated ArgoCD rollouts to enable blue-green deployment strategies for both frontend and backend, ensuring zero-downtime releases and efficient rollbacks.

Monitoring and Alerting:

- Setup Grafana/Prometheus on EKS
- Setup backend to send Prometheus metrics
- Create Grafana dashboards to visualize Prometheus metrics
- Create Grafana alerting rules
- Create ArgoCD rollouts alerting rules

HPA:

- Configure EKS metrics server and HPA for EKS cluster

Report:

	- Wrote 85% of main requirements and advanced requirements section. - Wrote challenges and how we overcame them section.	
Vuong Gia An - s3757287	- Frontend Tester - Video editor	Number of tests written: 4 files
Le Minh Duc - s4000577	Containerization and Deployment Engineer	Number of tests written: 8 files
Tran Minh Nhat - s3926629	Backend Tester, Project Secretary (in charge of Report and Minutes) - Develop the backend testing for the Backend Section of the Pipeline - Organization of the Report (along with Mr Quang) - Preparation of Slides and Host of the Presentation - Writing minutes for all project's meetings	Number of tests written: 7 files
Vu Tien Quang - s3981278	Integration and Backend Tester Report	Number of tests written: 8 files

Self-evaluation of CI/CD Pipeline Solution

Table 4. Self-Evaluation of CI/CD Pipeline solution

Requirement	Our Pipeline Proposal	Assessment
Cloud Integration	Utilize AWS services: **EKS** for container orchestration, **EC2** for Jenkins/Ansible servers, and **CloudFormation** for IaC.	Fully met. AWS services are leveraged effectively to ensure scalability, performance, and automation.
Containerized Microservices	Deploy frontend and backend as separate **Docker containers** managed by Kubernetes with 2 replica sets each.	Fully met. Containers ensure consistent environments, independent scalability, and high availability.
Continuous Integration	Implement CI with **Jenkins**, triggered by GitHub webhooks to run automated builds and tests, then push Docker images to **Docker Hub**.	Fully met. Automated CI process ensures consistent builds and code quality with minimal manual intervention.
Continuous Delivery	Use **ArgoCD** for GitOps-based deployments, with manual image tag updates triggering automated deployment using a **blue-green strategy**.	Fully met. The CD process ensures controlled and reliable updates with zero-downtime deployment.

Requirement	Our Pipeline Proposal	Assessment
Testing Framework	Integrate **Jest** for unit/integration tests and **Puppeteer** for E2E tests to validate core functionalities and user interactions.	Fully met. The pipeline covers multiple testing levels to ensure application functionality and reliability.
Configuration Management	Use **Ansible** to automate server setup and **Helm** to manage Kubernetes deployments.	Fully met. Ansible and Helm streamline configuration management and ensure consistency across environments.
Automated Alerting System	Implement Jenkins email notifications for pipeline failures and **Prometheus + Grafana** for real-time monitoring and alerting.	Fully met. Alerts and dashboards help detect issues early and enable quick resolution.
Orchestration	Deploy microservices on **Kubernetes (EKS)** using Helm for scalability and availability.	Fully met. Kubernetes ensures effective orchestration, scaling, and high availability of services.
Infrastructure as Code (IaC)	Automate resource provisioning using **CloudFormation** templates for AWS infrastructure setup.	Fully met. IaC ensures that infrastructure is consistently provisioned and easily reproducible.
Monitoring and Alerting	Use **Prometheus** for metrics collection, **Grafana** for dashboards, and email notifications for critical alerts.	Fully met. Comprehensive monitoring ensures application health and performance are well-tracked.
Advanced Deployment Strategy	Use **blue-green deployment** via ArgoCD to ensure seamless updates with minimal risk.	Fully met. Blue-green strategy ensures zero-downtime deployment and easy rollback if issues occur.
Additional Enhancements	Employ GitOps workflow for automated Pull Request.	Partially met.

APPENDIX B – MORE ABOUT THE TESTING FRAMEWORKS

UNIT TESTS WITH JEST

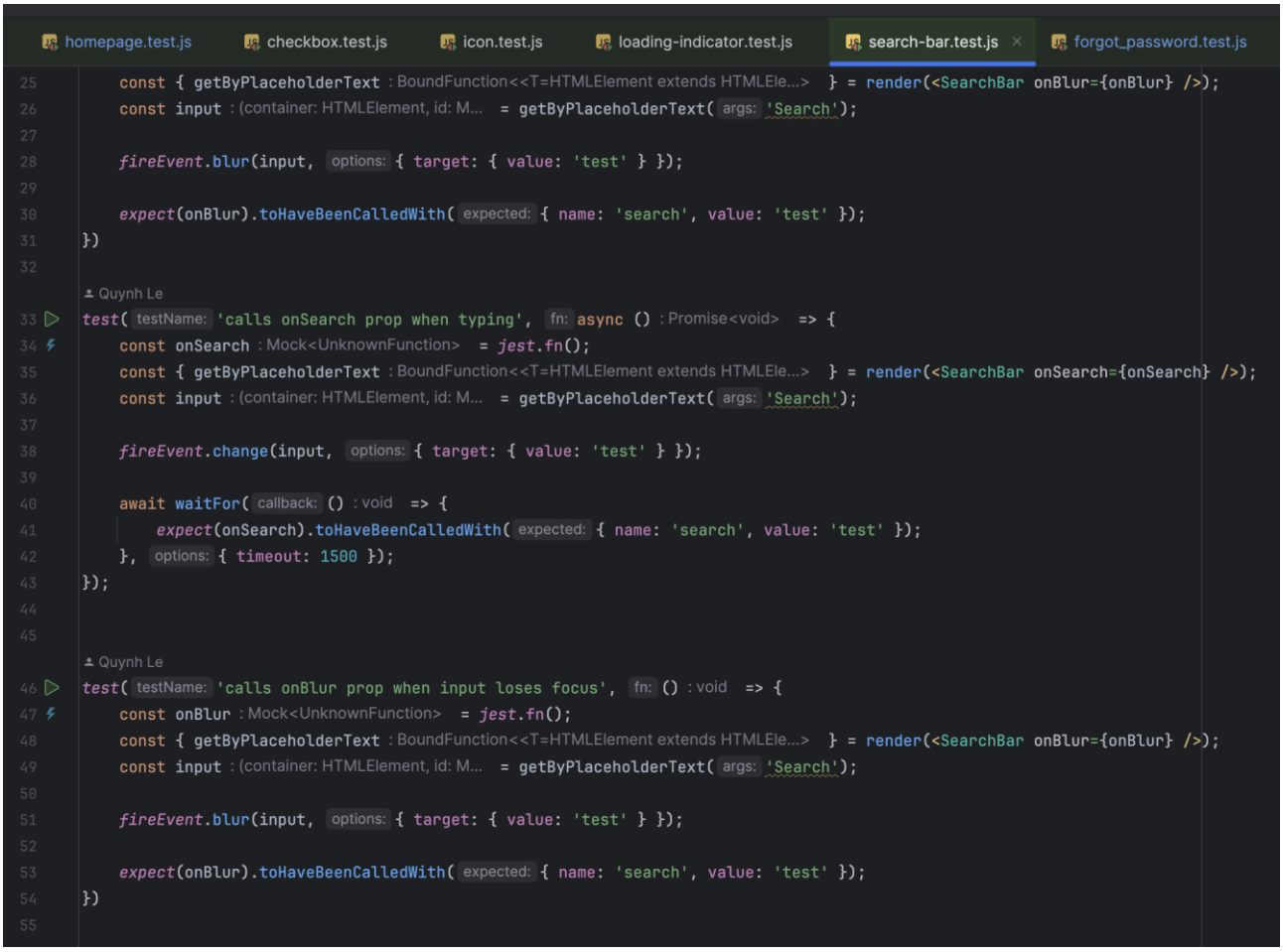


Figure 15. Search-bar unit test

In the Main Requirements part 2, we have mentioned the usage of Jest testing. The code for it is presented as above.

Snapshot testing

For example, we implemented a snapshot test for the website's footer, which reads "2024 RMIT Store." When the tests were rerun at the start of the new year, they failed because the component automatically updated to display "2025." (see Figure 16). This immediately alerted developers, who could then either update the snapshot to reflect the expected change or fix the issue if the update was unintended.

```
FAIL  __tests__/components/Common/footer.test.js (5.811 s)
  ● Footer component › Footer component matches snapshot

    expect(received).toMatchSnapshot()

    Snapshot name: `Footer component Footer component matches snapshot 1`

    - Snapshot  - 1
    + Received  + 1

    @@ -161,11 +161,11 @@
            </div>
            <div
              class="footer-copyright"
            >
              <span>
    -           © 2024 RMIT Store. All rights reserved.
    +           © 2025 RMIT Store. All rights reserved.
              </span>
            </div>
            <ul
              class="footer-social-item"
            >
```

Figure 16: Failed test (footer) with the Footer snapshot.

To view the test snapshots, we can navigate to the __**tests**__ folder, where **Jest** automatically saves the snapshots created during test execution (see Figure 17). During the initial test run, **Jest** generates baseline snapshots, which serve as the reference for future comparisons. In subsequent test runs, the generated output is compared against these baseline snapshots to detect any unintended changes or drifts.

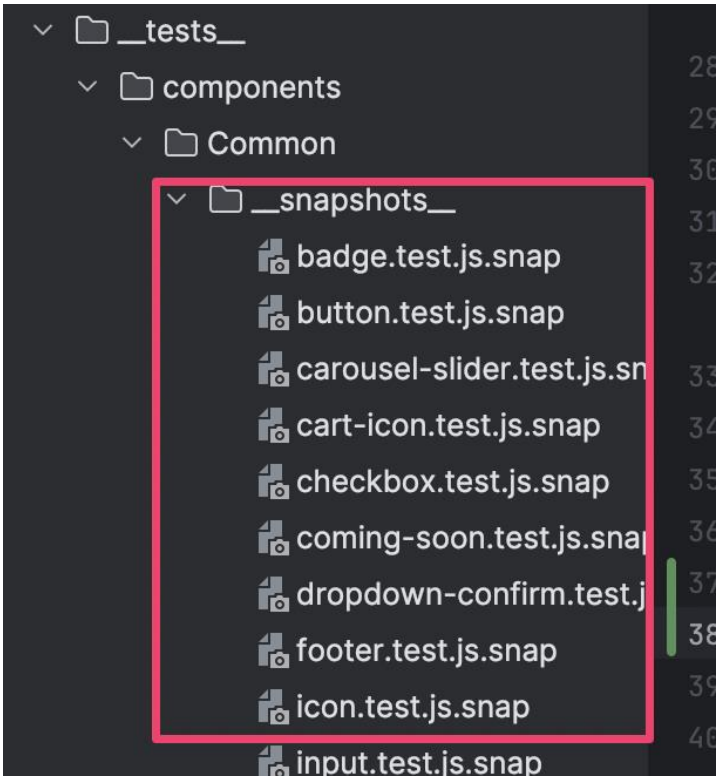
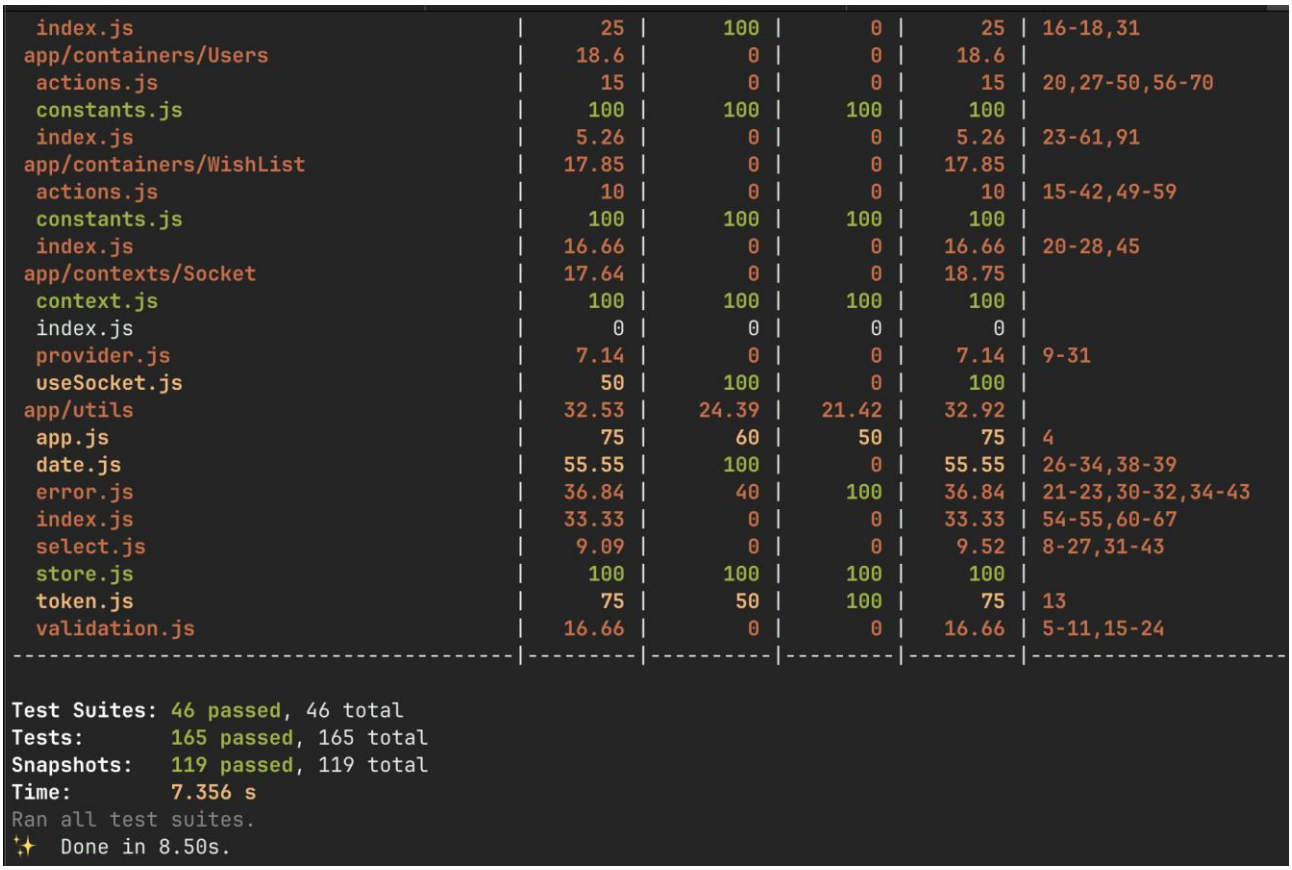


Figure 17. Snapshots folder

Unit tests coverages



```
 index.js                                |      25 |      100 |       0 |      25 | 16-18,31
app/containers/Users                     |    18.6 |        0 |       0 |    18.6 |
 actions.js                              |      15 |        0 |       0 |      15 | 20,27-50,56-70
 constants.js                            |     100 |      100 |     100 |     100 |
 index.js                                |    5.26 |        0 |       0 |    5.26 | 23-61,91
app/containers/WishList                  |   17.85 |        0 |       0 |   17.85 |
 actions.js                              |      10 |        0 |       0 |      10 | 15-42,49-59
 constants.js                            |     100 |      100 |     100 |     100 |
 index.js                                |   16.66 |        0 |       0 |   16.66 | 20-28,45
app/contexts/Socket                      |   17.64 |        0 |       0 |   18.75 |
 context.js                              |     100 |      100 |     100 |     100 |
 index.js                                |       0 |        0 |       0 |       0 |
 provider.js                             |    7.14 |        0 |       0 |    7.14 | 9-31
 useSocket.js                            |      50 |      100 |       0 |     100 |
app/utils                                |   32.53 |    24.39 |   21.42 |   32.92 |
 app.js                                  |      75 |       60 |      50 |      75 | 4
 date.js                                 |   55.55 |      100 |       0 |   55.55 | 26-34,38-39
 error.js                                |   36.84 |       40 |     100 |   36.84 | 21-23,30-32,34-43
 index.js                                |   33.33 |        0 |       0 |   33.33 | 54-55,60-67
 select.js                               |    9.09 |        0 |       0 |    9.52 | 8-27,31-43
 store.js                                |     100 |      100 |     100 |     100 |
 token.js                                |      75 |       50 |     100 |      75 | 13
 validation.js                           |   16.66 |        0 |       0 |   16.66 | 5-11,15-24
-----------------------------------------|---------|----------|---------|---------|--------------------

Test Suites: 46 passed, 46 total
Tests:       165 passed, 165 total
Snapshots:   119 passed, 119 total
Time:        7.356 s
Ran all test suites.
✨  Done in 8.50s.
```

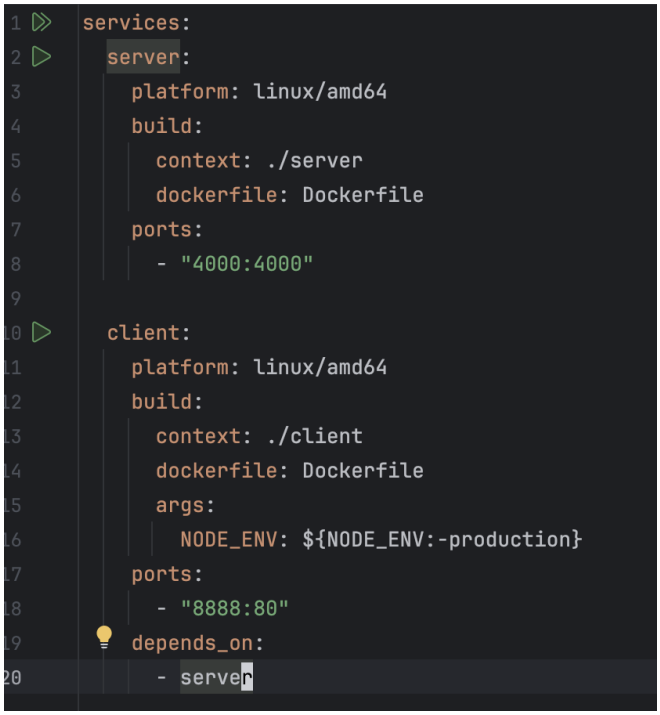
Figure 18. FE test coverage

```
 merchant.js    |     100 |      100 |     100 |     100 |
 order.js       |     100 |      100 |     100 |     100 |
 product.js     |     100 |      100 |     100 |     100 |
 review.js      |     100 |      100 |     100 |     100 |
 user.js        |     100 |       50 |     100 |     100 | 12
 wishlist.js    |     100 |      100 |     100 |     100 |
routes/api      |   55.78 |    34.78 |   56.75 |   56.08 |
 address.js     |   84.21 |       50 |     100 |   84.21 | 25,40,53,62,83,99
 auth.js        |    62.5 |    45.45 |   57.14 |    62.5 | ...,77,88,94,98,104,111-114,157,168,17
 brand.js       |   64.77 |       50 |   77.77 |   64.36 | 20,39,56,72,83,100,109,120-138,161,173
 cart.js        |      80 |      100 |   66.66 |      80 | 24-26,45,62,79,86-95
 category.js    |   73.13 |       50 |    87.5 |   73.13 | 19,33,68,85,94,112,124,131-155,175
 merchant.js    |   48.81 |    36.66 |   45.45 |    49.2 | ...7,33,41,63,71-90,117,125-143,174,19
 newsletter.js  |     100 |      100 |     100 |     100 |
 order.js       |   47.78 |    20.83 |      40 |   49.54 | 35-45,59-121,129-156,194,208,236-251,2
 product.js     |   26.61 |    11.36 |      20 |   26.61 | 44-48,65,74,82-158,165-172,185-243,256
 wishlist.js    |   70.83 |       50 |      50 |   70.83 | 24,45,53-67
services        |    61.7 |    27.27 |      80 |    61.7 |
 mailchimp.js   |      70 |      100 |      50 |      70 | 21-27
 mailgun.js     |   59.45 |    27.27 |     100 |   59.45 | 58-86
utils           |   31.39 |     2.85 |   18.75 |   31.39 |
 auth.js        |   27.27 |        0 |       0 |   27.27 | 5-23
 queries.js     |   23.07 |        0 |       0 |   23.07 | 6-97,102-129
 store.js       |   33.87 |     4.34 |   23.07 |   33.87 | 7-16,22-74,80-84,103-106,120,129
----------------|---------|----------|---------|---------|-------------------------------------
Test Suites: 27 passed, 27 total
Tests:       99 passed, 99 total
Snapshots:   0 total
Time:        49.866 s
Ran all test suites.
✨  Done in 52.12s.
```

Figure 19. BE test coverage

INTEGRATION TEST

Testing Server (docker-compose)



```yaml
services:
  server:
    platform: linux/amd64
    build:
      context: ./server
      dockerfile: Dockerfile
    ports:
      - "4000:4000"

  client:
    platform: linux/amd64
    build:
      context: ./client
      dockerfile: Dockerfile
      args:
        NODE_ENV: ${NODE_ENV:-production}
    ports:
      - "8888:80"
    depends_on:
      - server
```

Figure 20. Docker-compose file for testing server

Sign Up Test Case

To test the user sign-up flow, we navigate to the registration page, input a desired username and password, and verify whether the user can successfully complete the sign-up process.

```
Quynh Le *
describe( blockName: "Sign Up Integration Test", blockFn: () : void => {
    it( testName: "should sign up a new user successfully", fn: async () : Promise<void> => {
        // Navigate to the sign-up page
        await page.goto( url: `${baseUrl}/register`);
        // wait for element to be rendered
        await page.waitForSelector( selector: 'form');
        // Fill out the form
        await page.type( selector: 'input[name="email"]', text: "testuser@example.com");
        await page.type( selector: 'input[name="firstName"]', text: "Test");
        await page.type( selector: 'input[name="lastName"]', text: "User");
        await page.type( selector: 'input[name="password"]', text: "SecurePassword123");

        // Submit the form
        await page.click( selector: 'button[type="submit"]');
        await page.waitForResponse( urlOrPredicate: response : HTTPResponse => {
            return response.url().includes('/api/auth/register') && response.status() === 200;
        });

        const user : Query<...> & {...} = await User.findOne( filter: { email: "testuser@example.com" });

        expect(user).toBeDefined();
        expect(user?.firstName).toBe( expected: "Test");
        expect(user?.lastName).toBe( expected: "User");
    });
});
```

Figure 21. Integration test case - User Registration

```
PASS  __tests__/integration/store.test.js
PASS  __tests__/integration/homepage.test.js

Test Suites: 4 passed, 4 total
Tests:       7 passed, 7 total
Snapshots:   0 total
Time:        9.802 s
Ran all test suites.
✨  Done in 25.78s.
```

Figure 22. Integration test coverage

LOAD TESTING

Load Testing with Hey for Autoscaling feature

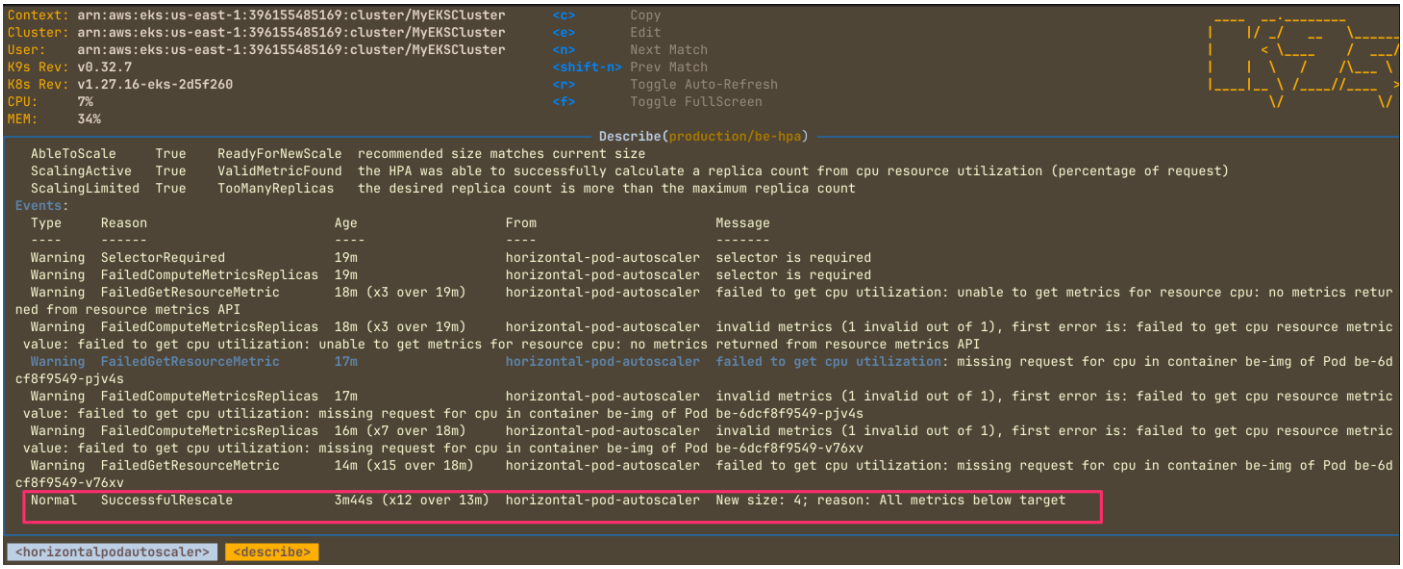


Figure 23 – Scaling event due to simulated traffic

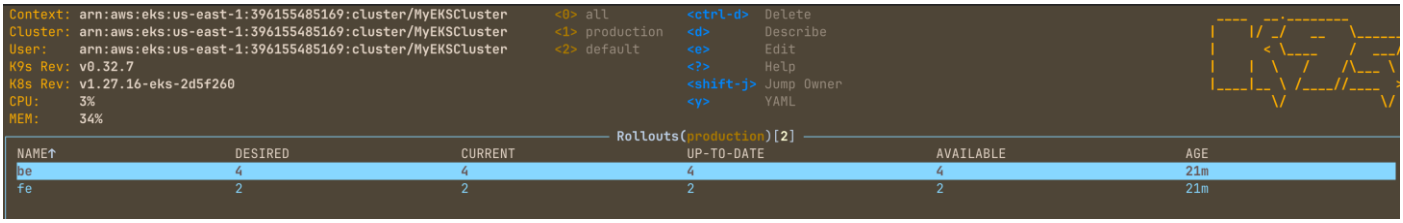


Figure 24 - BE pod scale as a result of traffic

```
~/Downloads base ❯ hey -z 5m -c 50 -q 10 http://k8s-producti-appingre-8eca7f2e72-505836489.us-east-1.elb.amazonaws.com


Summary:
  Total:        301.8700 secs
  Slowest:      4.2373 secs
  Fastest:      0.2677 secs
  Average:      0.3243 secs
  Requests/sec: 153.4502

  Total data:   82499482 bytes
  Size/request: 1781 bytes

Response time histogram:
  0.268 [1]     |
  0.665 [45225] |■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
  1.062 [146]   |
  1.459 [902]   |■
  1.856 [41]    |
  2.252 [4]     |
  2.649 [1]     |
  3.046 [1]     |
  3.443 [0]     |
  3.840 [0]     |
  4.237 [1]     |


Latency distribution:
  10% in 0.2890 secs
  25% in 0.2959 secs
  50% in 0.3038 secs
  75% in 0.3120 secs
  90% in 0.3222 secs
  95% in 0.3323 secs
  99% in 1.1112 secs

Details (average, fastest, slowest):
  DNS+dialup:   0.0004 secs, 0.2677 secs, 4.2373 secs
  DNS-lookup:   0.0001 secs, 0.0000 secs, 0.0915 secs
  req write:    0.0000 secs, 0.0000 secs, 0.0111 secs
  resp wait:    0.3214 secs, 0.2676 secs, 4.2363 secs
  resp read:    0.0025 secs, 0.0000 secs, 1.6332 secs

Status code distribution:
  [200] 46322 responses
```

Figure 25. Hey load testing command

APPENDIX C - JENKINS PIPELINE

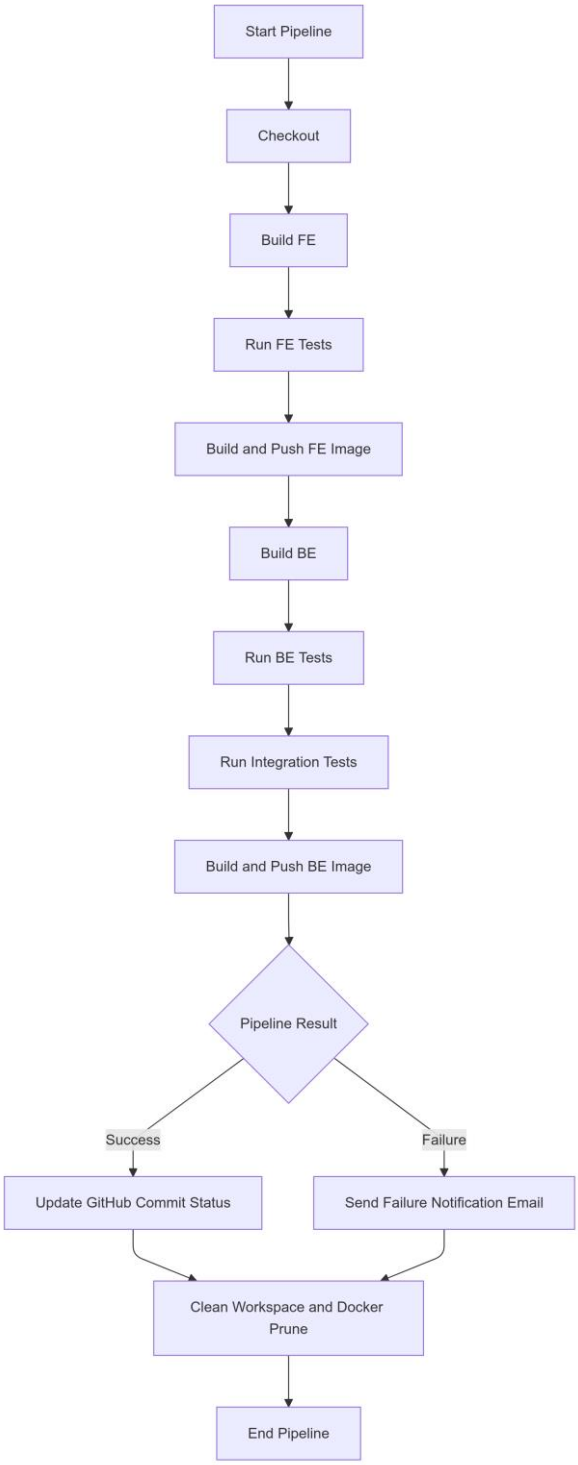


Figure 26. CI pipeline diagram

Our **Jenkinsfile** follows these steps:

- Checkout the branch that triggered the build.
- Run all tests in the branch, ensuring early detection of issues.
- Build and push Docker images if all tests pass successfully.
- If tests fail, send an email notification to the team to promptly alert them about the failure.

- Update GitHub status for the branch at the end of the pipeline, regardless of whether the pipeline succeeds or fails. This enables users to navigate to any commit and see the build status for that specific commit.

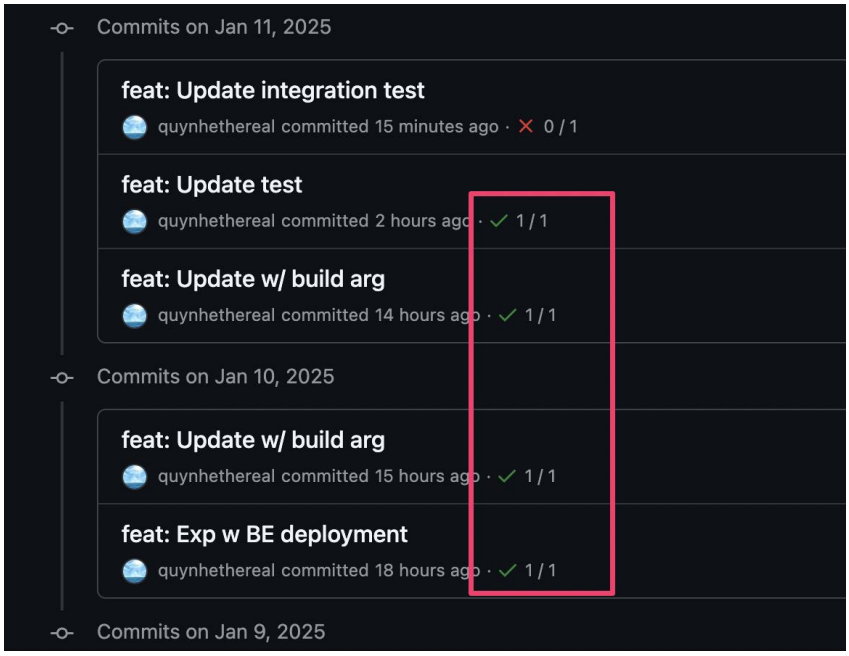


Figure 27. Continuously update Git commit

- Finally, the pipeline includes a cleanup step to remove all workspaces. This involves using docker prune to delete unused Docker artifacts and clearing temporary files created during the pipeline run. This cleanup is essential because we observed that storage on agent instances can quickly become depleted if artifacts are retained, leading to flaky builds due to leftover artifacts interfering with subsequent runs.

APPENDIX D - MONITORING AND ALERT

EMAIL NOTIFICATIONS

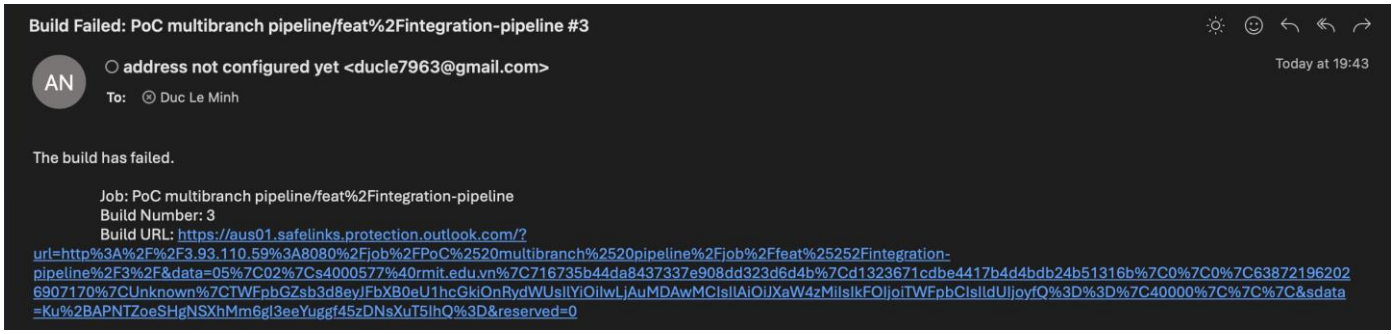


Figure 28. Email Notification on CI Failure

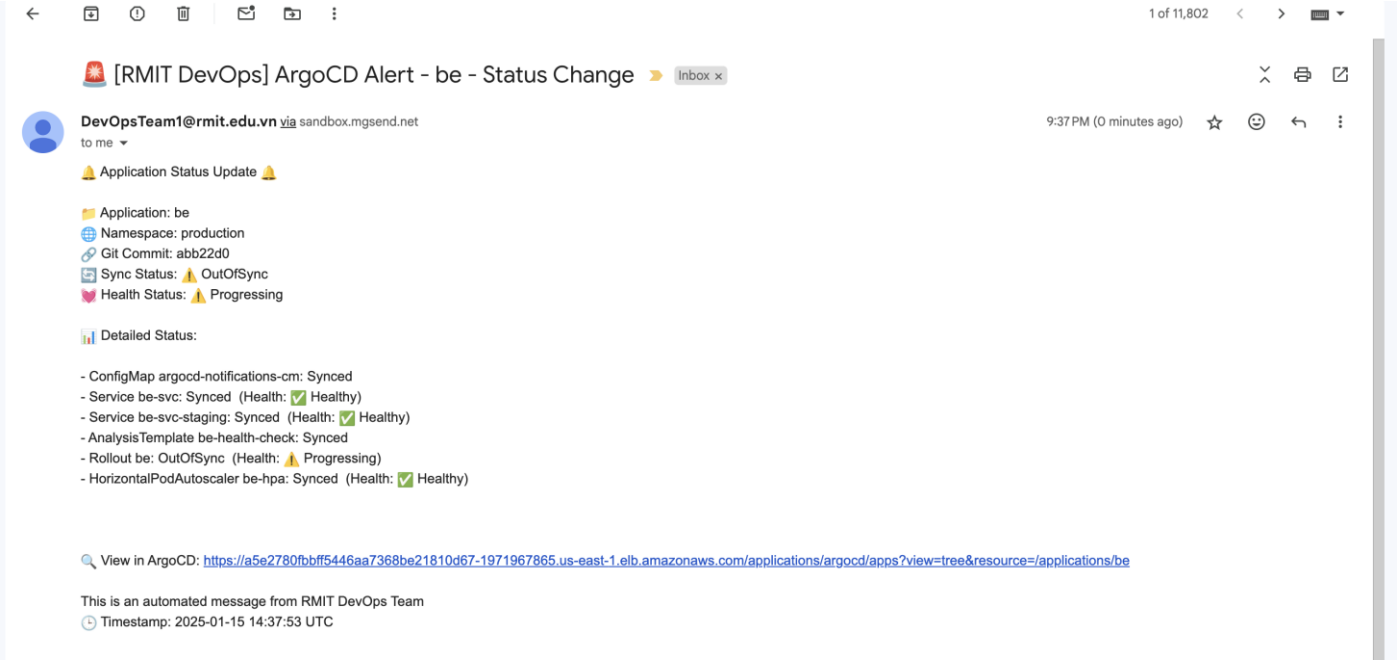


Figure 29. Email Notifications on CD Failure

DASHBOARDS

Figure 30. Monitoring dashboard for the RMIT application

For the cluster, it will be like this:

Figure 31. The cluster controlling center

The alerting system:

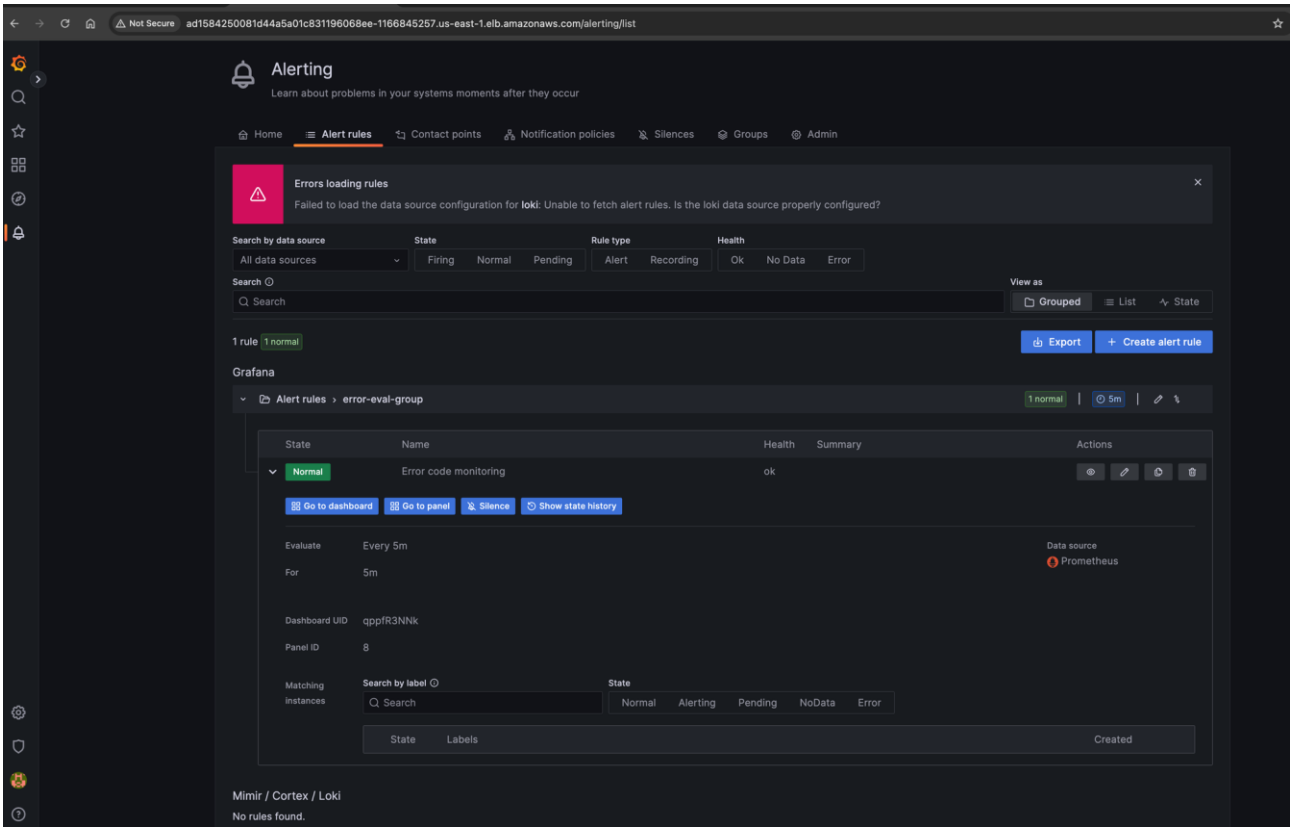


Figure 32. Alerting center

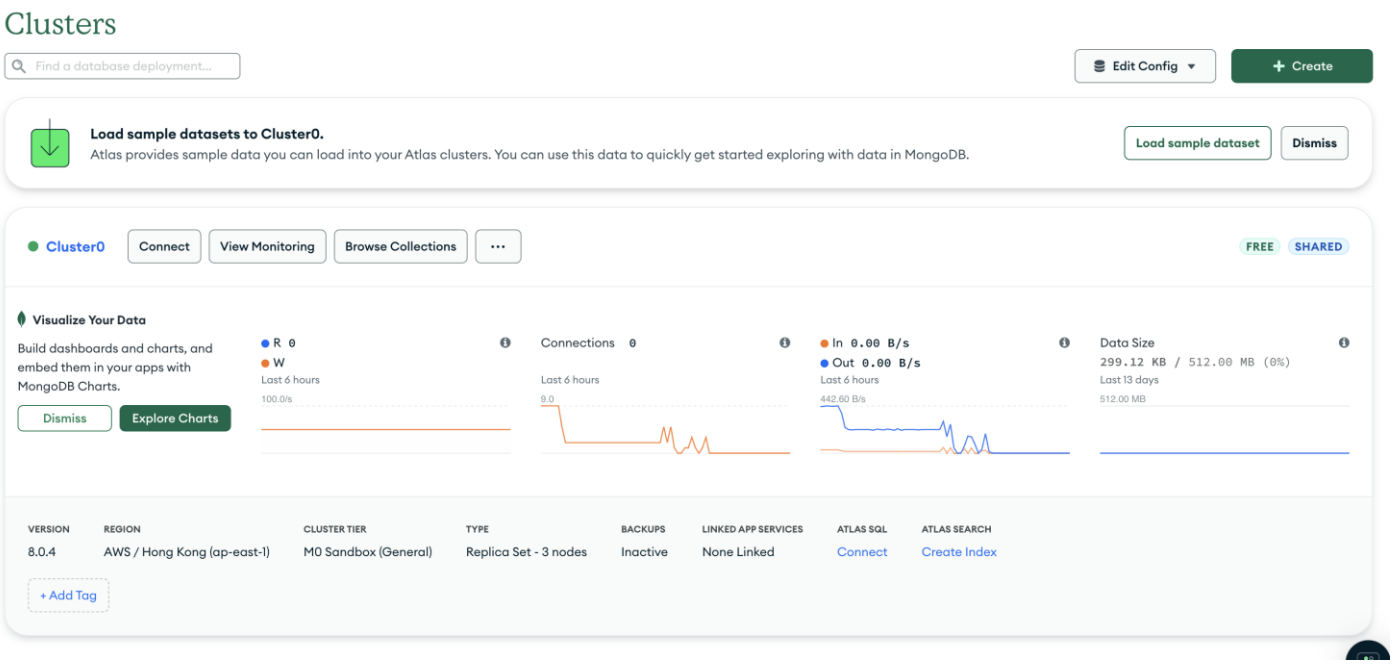


Figure 33. MongoDB Atlas Dashboard

```
nodejs_heap_space_size_available_bytes{space="new",app="rmit-app"} 13021048

# HELP nodejs_version_info Node.js version info.
# TYPE nodejs_version_info gauge
nodejs_version_info{version="v18.17.1",major="18",minor="17",patch="1",app="rmit-app"} 1

# HELP nodejs_gc_duration_seconds Garbage collection duration by kind, one of major, minor, incremental or weakcb.
# TYPE nodejs_gc_duration_seconds histogram
nodejs_gc_duration_seconds_bucket{le="0.001",app="rmit-app",kind="incremental"} 1
nodejs_gc_duration_seconds_bucket{le="0.01",app="rmit-app",kind="incremental"} 1
nodejs_gc_duration_seconds_bucket{le="0.1",app="rmit-app",kind="incremental"} 1
nodejs_gc_duration_seconds_bucket{le="1",app="rmit-app",kind="incremental"} 1
nodejs_gc_duration_seconds_bucket{le="2",app="rmit-app",kind="incremental"} 1
nodejs_gc_duration_seconds_bucket{le="5",app="rmit-app",kind="incremental"} 1
nodejs_gc_duration_seconds_bucket{le="+Inf",app="rmit-app",kind="incremental"} 1
nodejs_gc_duration_seconds_sum{app="rmit-app",kind="incremental"} 0.00022566699981689452
nodejs_gc_duration_seconds_count{app="rmit-app",kind="incremental"} 1

# HELP http_request_count Count of HTTP requests made to my app
# TYPE http_request_count counter

# HELP http_request_duration_seconds Duration of HTTP requests in seconds
# TYPE http_request_duration_seconds histogram

# HELP nodejs_active_connections Number of active connections
# TYPE nodejs_active_connections gauge
nodejs_active_connections{app="rmit-app"} 2

# HELP nodejs_failed_requests_total Total number of failed requests
# TYPE nodejs_failed_requests_total counter

# HELP nodejs_memory_usage_bytes Memory usage in bytes
# TYPE nodejs_memory_usage_bytes gauge
nodejs_memory_usage_bytes{app="rmit-app"} 0
```

Figure 34. Metrics exposed via Prometheus client

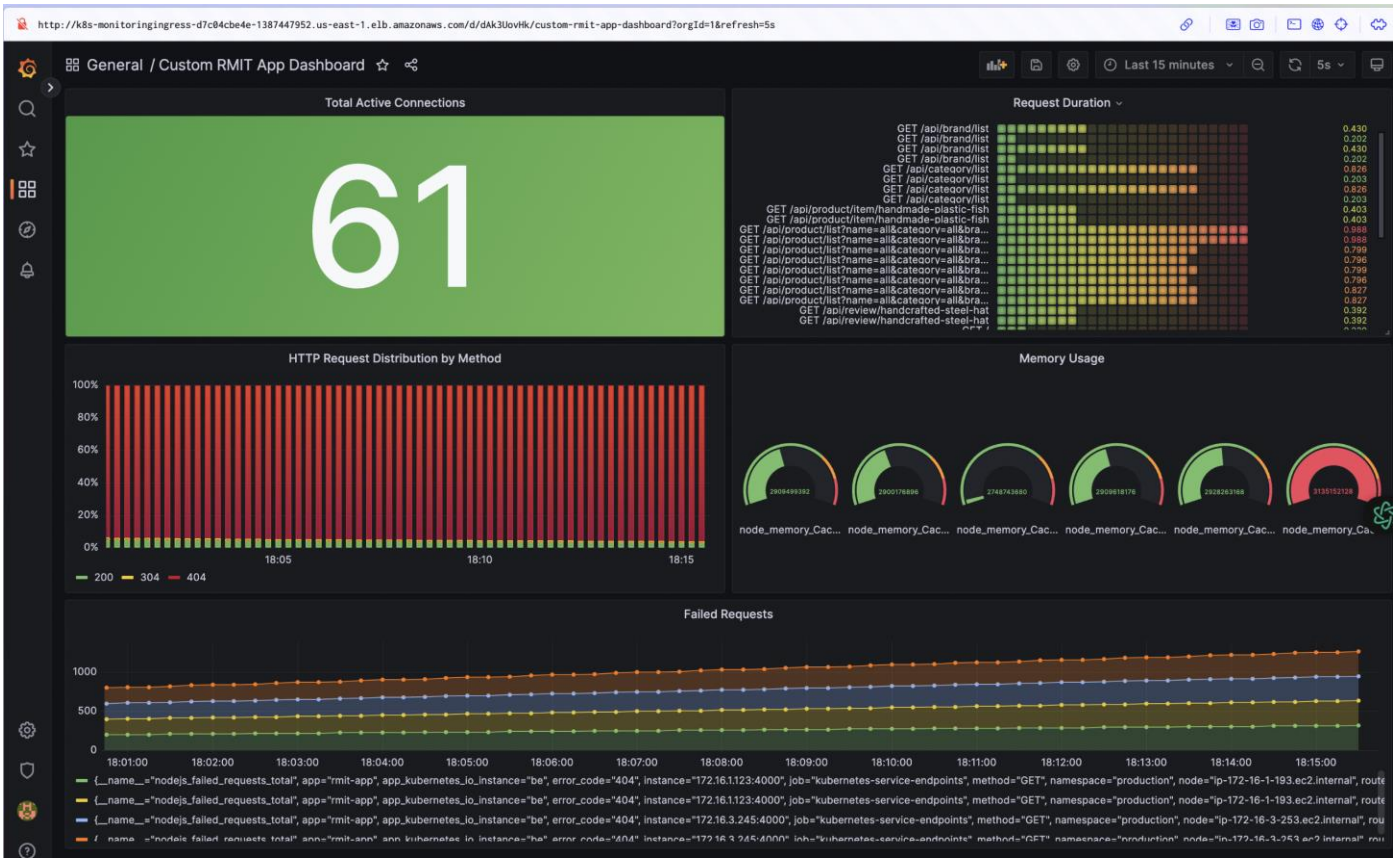


Figure 35. Grafana dashboard exposed with public URL

APPENDIX E - APPLICATION FLOW

DETAILED APPLICATION FLOW

The DevOps process as illustrated in Figure 14 begins with the **initial setup**, where the DevOps team provisions the necessary cloud infrastructure using AWS CloudFormation templates. These templates define and create resources such as EC2 instances, VPCs, subnets, and an EKS cluster. Once the infrastructure is up and running, the team connects to the Ansible Server via SSH and executes Ansible playbooks to configure the Jenkins servers for continuous integration (CI) and set up the EKS cluster for deploying the application.

The application flow follows a typical Git-based workflow. Developers push their changes to a shared repository, where they create a Pull Request (PR) to the protected main branch. If the PR is approved, the CI pipeline proceeds with further stages to ensure code quality before merging the changes into the main branch.

The **Continuous Integration (CI) process** is divided into two main parts: frontend (FE) and backend (BE). For the frontend, Jenkins installs all necessary dependencies, runs unit tests, builds the application, and pushes the resulting Docker image to the Docker Hub frontend repository. For the backend, Jenkins installs dependencies, runs unit tests, starts a testing server using Docker Compose, and performs integration tests. Once the backend application passes these tests, Jenkins builds a Docker image and pushes it to the Docker Hub backend repository. If any step in the pipeline fails, Jenkins immediately sends an email notification to the team to ensure prompt attention to the issue.

In the **Continuous Deployment (CD) phase**, the DevOps user updates the image tags in the configuration files, such as Kubernetes manifests or Helm chart values. ArgoCD, configured for GitOps-based deployment, automatically detects these changes and triggers an automated deployment to the EKS cluster. The deployment strategy used is blue/green, where a new version (green) is deployed alongside the existing one (blue) to ensure a smooth transition with minimal downtime. If any issues are detected during the deployment, a quick rollback to the stable version is possible. During the deployment, real-time metrics and performance data are collected and monitored using Prometheus and visualized in Grafana. Alerts and notifications are configured to ensure any issues are promptly addressed.

Once the deployment is complete, the website and monitoring service remain accessible via the blue version using its assigned public URLs.

APPENDIX F - BLUE/GREEN DEPLOYMENT


```
apiVersion: argoproj.io/v1alpha1
kind: Rollout
metadata:
  name: fe
  labels:
    app: fe
spec:
  replicas: 2
  revisionHistoryLimit: 1
  selector:
    matchLabels:
      app: fe
  template:
    metadata:
      labels:
        app: fe
    spec:
      containers:
        - name: fe-img
          image: guynhethereal/fe-client-devops-course:c4a1e7d
          imagePullPolicy: Always
          ports:
            - name: http
              containerPort: 80
  strategy:
    blueGreen:
      autoPromotionEnabled: true
      activeService: fe-svc
      previewService: fe-svc-staging
      prePromotionAnalysis:
        templates:
          - templateName: fe-health-check
```

Figure 36. ArgoCD Rollout template

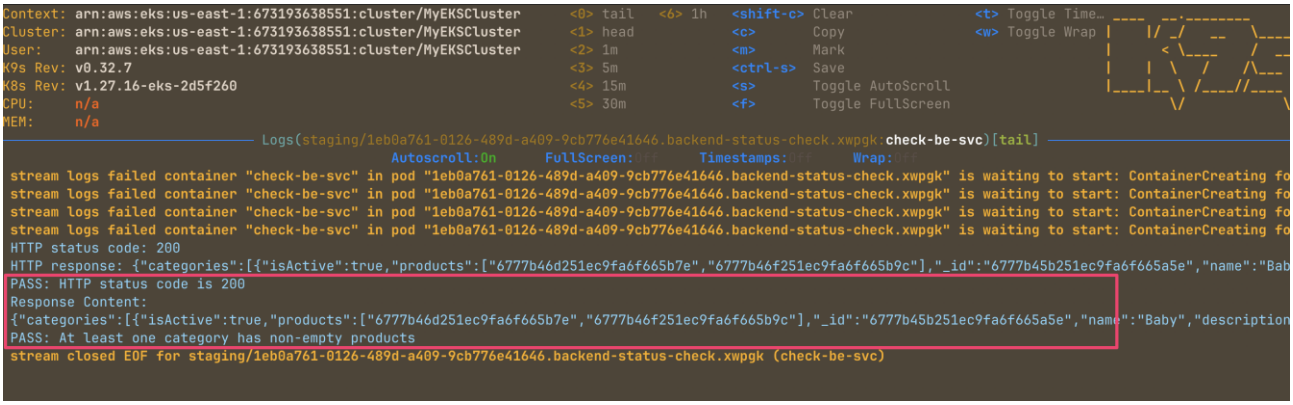


Figure 37. Analysis Template for Backend image

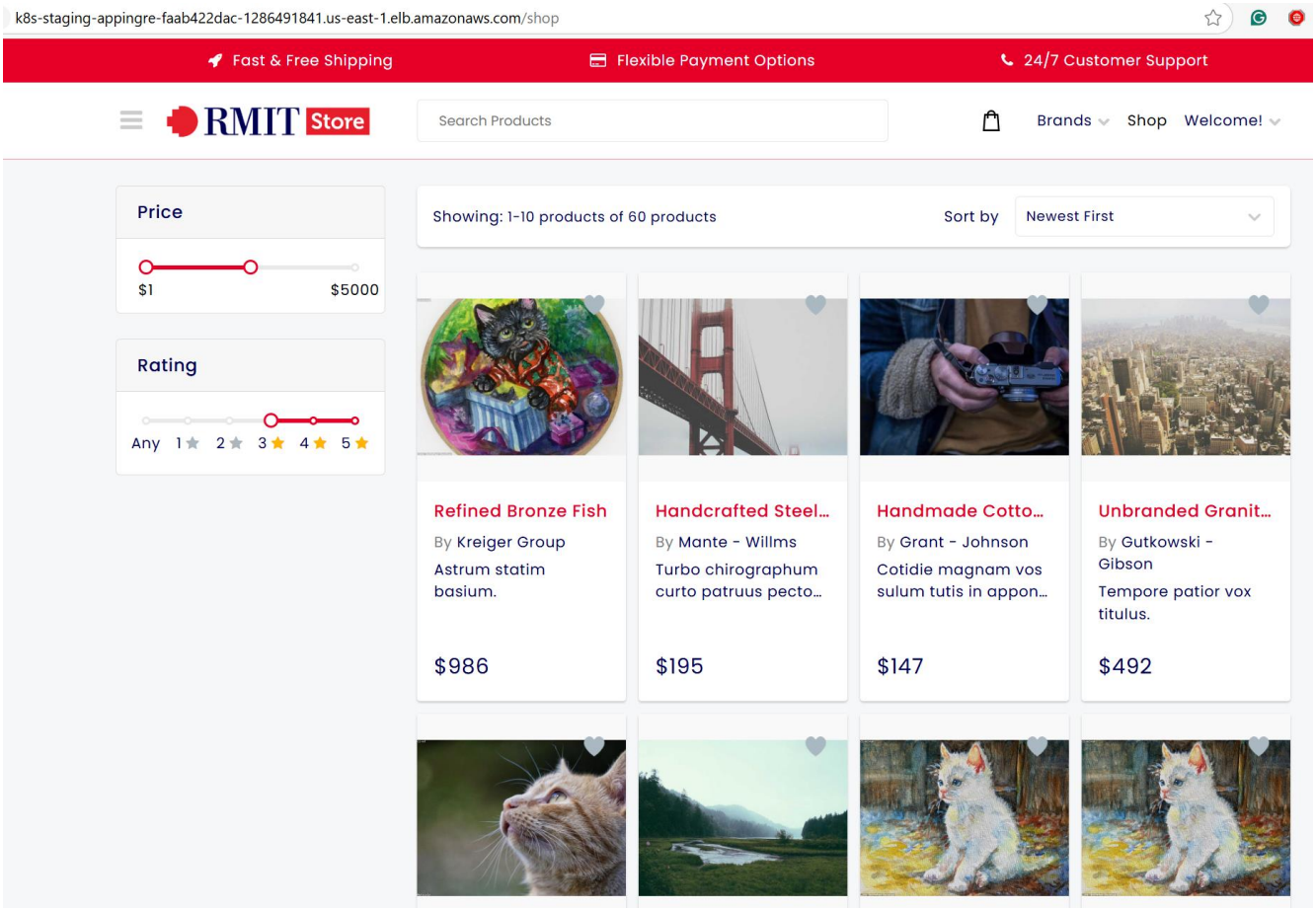


Figure 38. Main application is accessible using Application Load Balancer public URL

APPENDIX G - PROMETHEUS AND GRAFANA CONFIGURATION

The ArgoCD applications for Grafana and Prometheus are stored in Git as below.

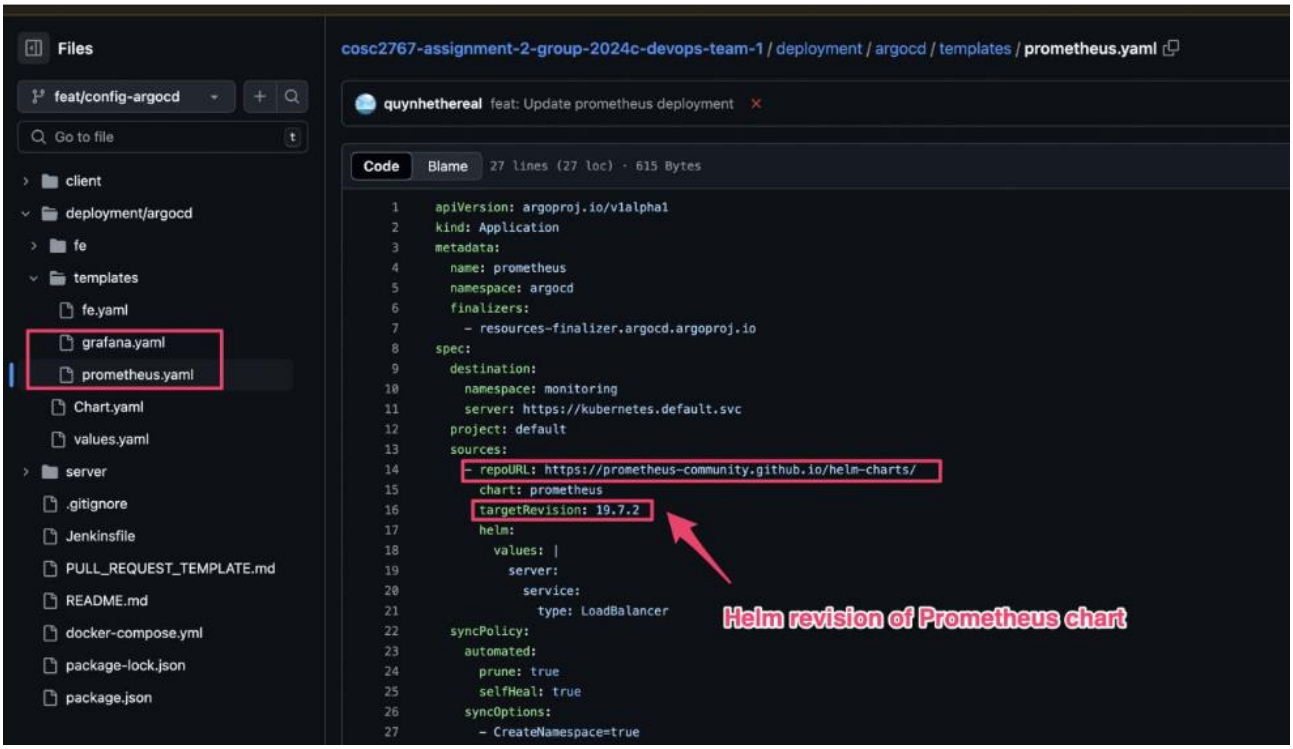


Figure 39. The Prometheus configurations inside the project.

After deploying the parent application as outlined in the **ArgoCD** section, the monitoring applications, such as Prometheus and Grafana, automatically initialize themselves.

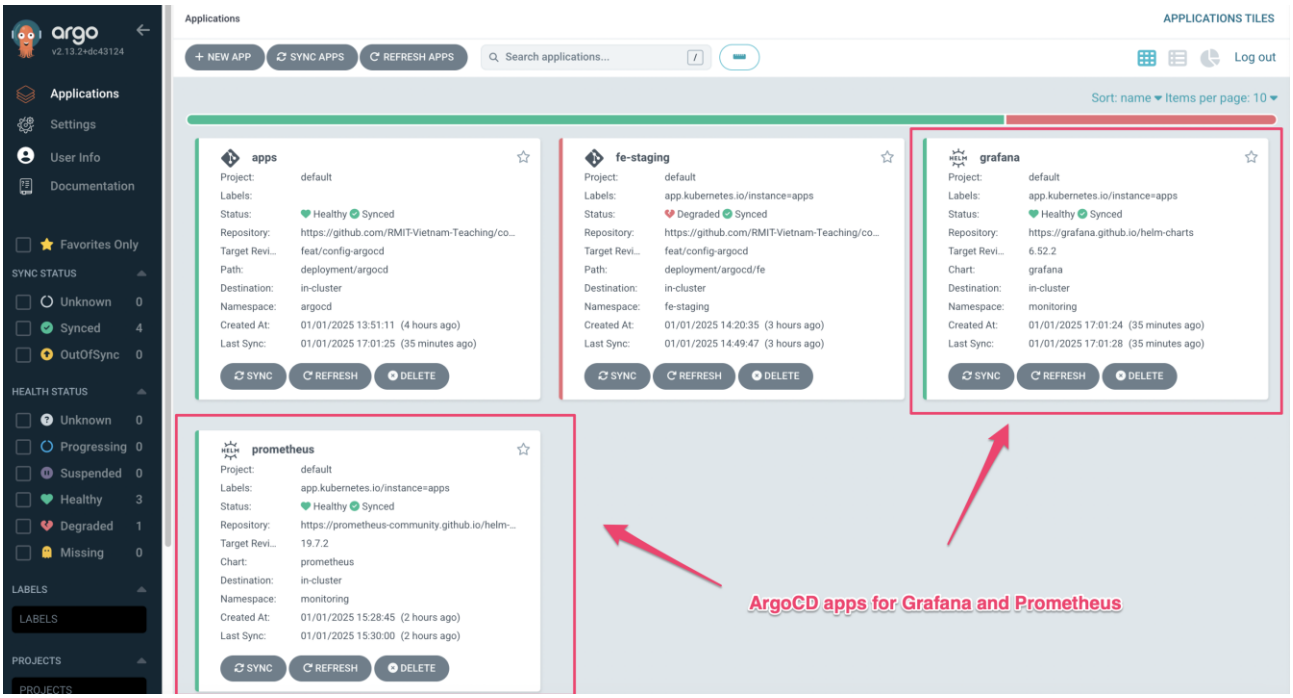


Figure 40. Prometheus and Grafana inside the ArgoCD dashboard.

We patched the Prometheus and Grafana manifests to expose their services via **AWS LoadBalancer**, enabling access to the dashboards through a public URL.

To persist the monitoring configurations, we used the **AWS EBS CSI driver** for Prometheus and Grafana. From there, we configured a **PersistentVolumeClaim** (PVC) and **PersistentVolume** (PV), which enables the storage of monitoring data and configuration across pod restarts or failures. This setup allows us to maintain time-series data in Prometheus and retain dashboards, configurations, and user preferences in Grafana, ensuring data persistence, scalability, and high availability.

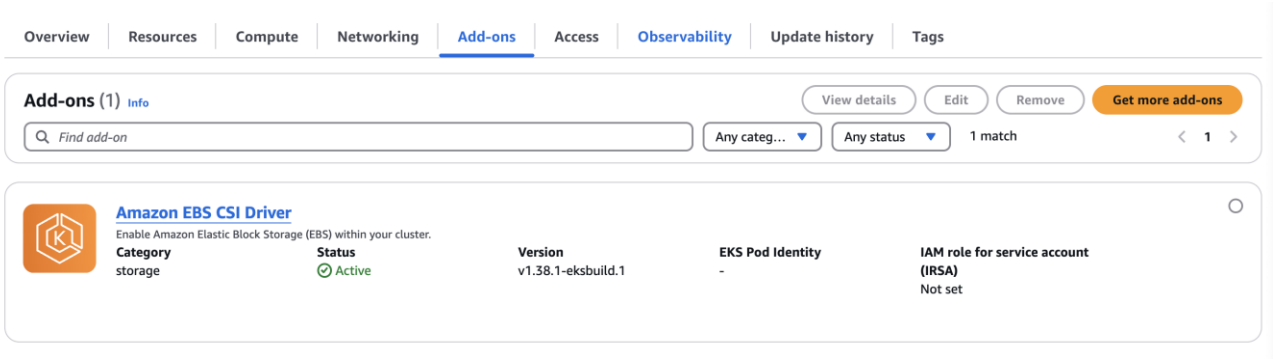


Figure 41. AWS EBS CSI Driver for Monitoring apps on AWS Console

APPENDIX H - INFRASTRUCTURE AS CODE - CLOUDFORMATION

EKS CLUSTER SETUP USING CLOUDFORMATION ON AWS CONSOLE WITH AWS LEARNER LAB

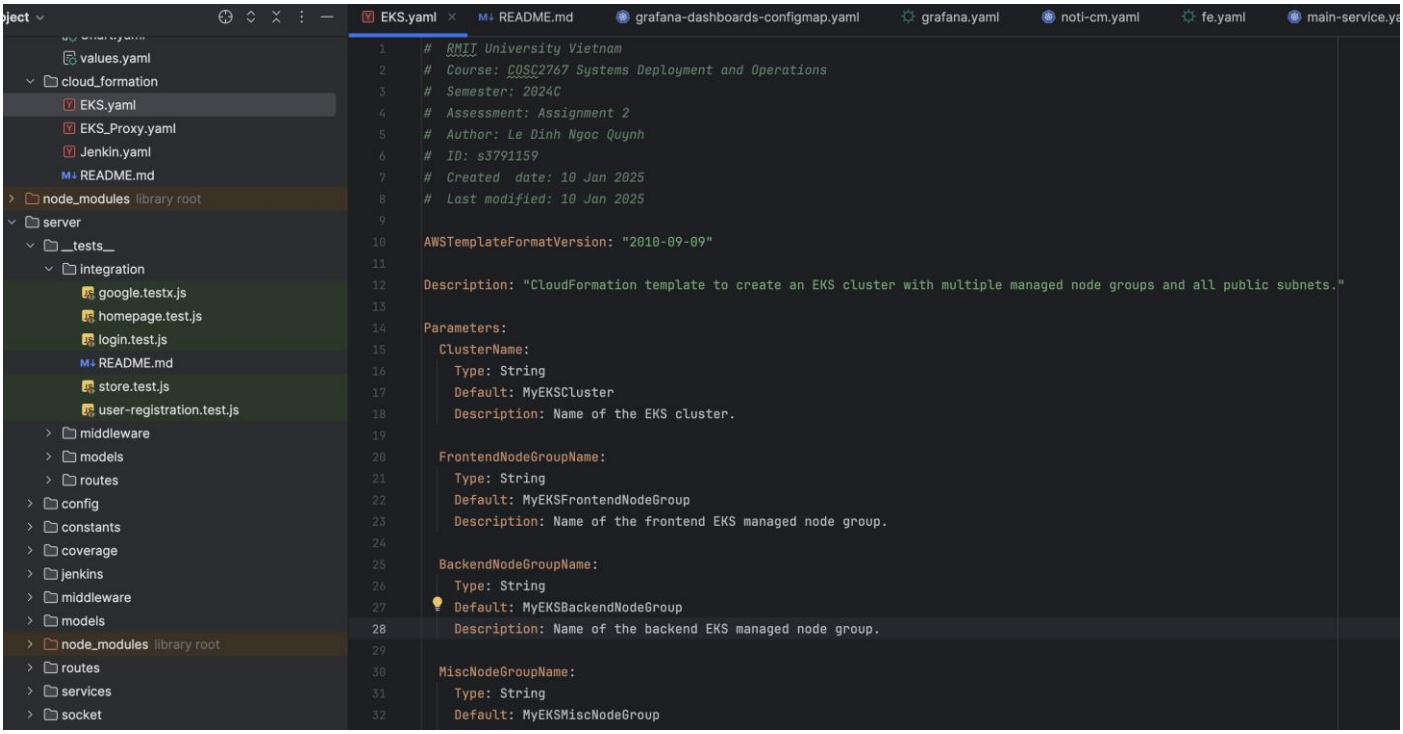


Figure 42. CloudFormation templates

1. We will navigate to CloudFormation in AWS Portal and upload the CloudFormation file into it.

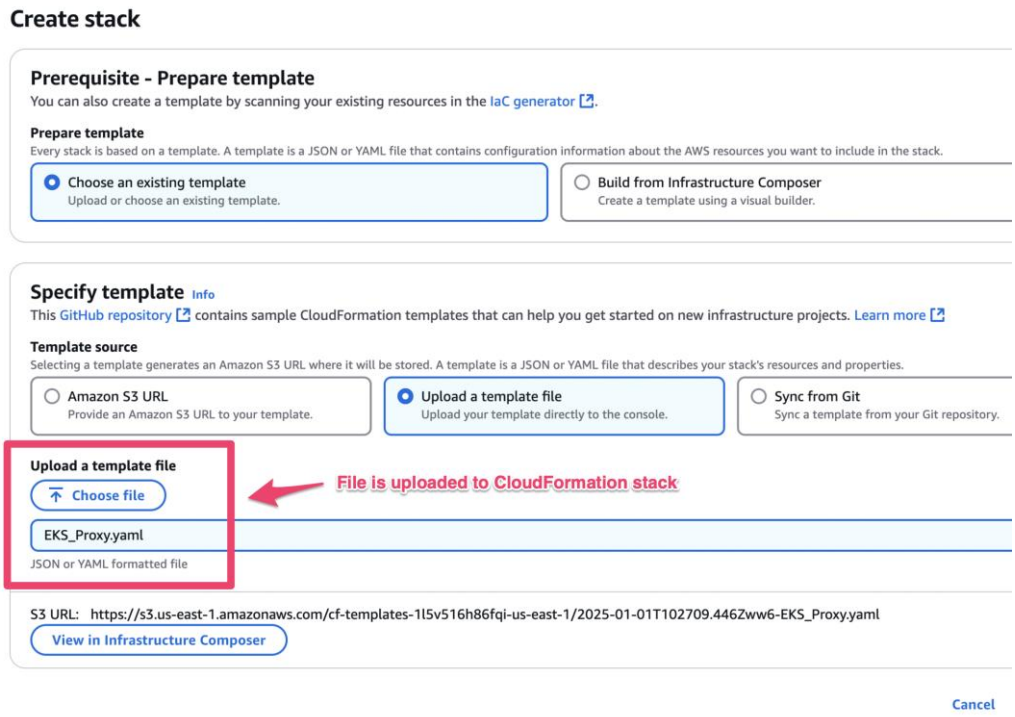


Figure 43. EKS Proxy file construction

2. Click "Next" and fill in the required parameters for the EKS setup. In our stack, the user needs to provide the following:
   - RoleArn: Obtain the ARN URI for the LabRole in the sandbox and input there.
   - KeyName: Select the key created before.
   - Leave the remaining fields as default or modified as needed.

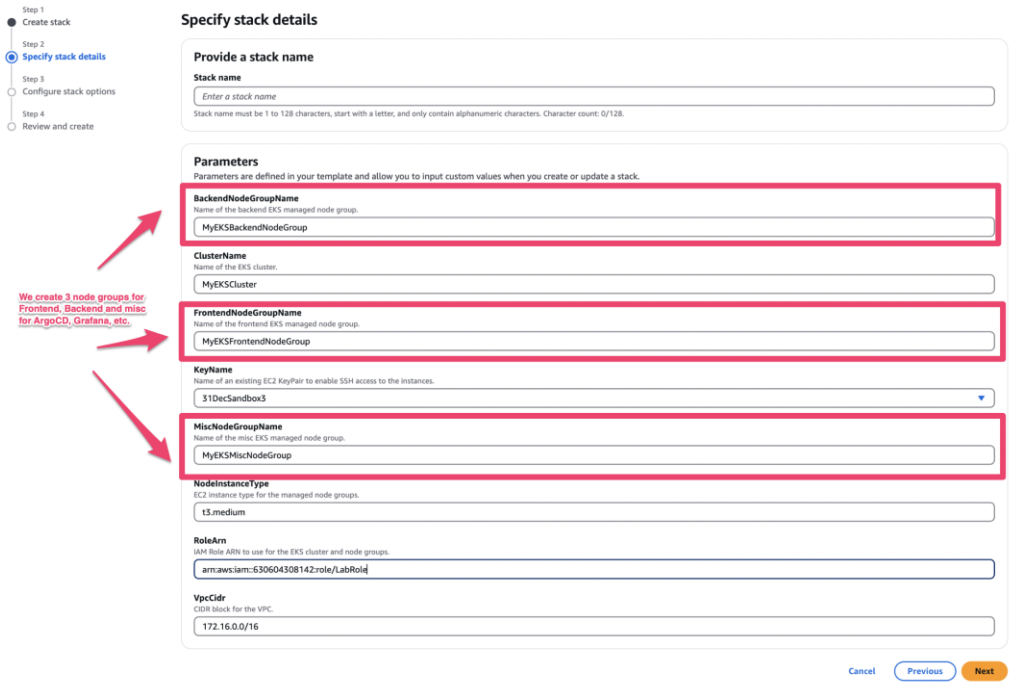


Figure 44. EKS Server Process Creation for the Project

3. Next, click **Submit** to deploy the resources. This CloudFormation stack will provision the VPC and its subnets, as well as the network security groups. It will then create the EKS cluster and configure access entries for the LabRole and voclabs roles, enabling us to interact with the cluster's API server using kubectl and view the resources from the AWS portal, respectively.

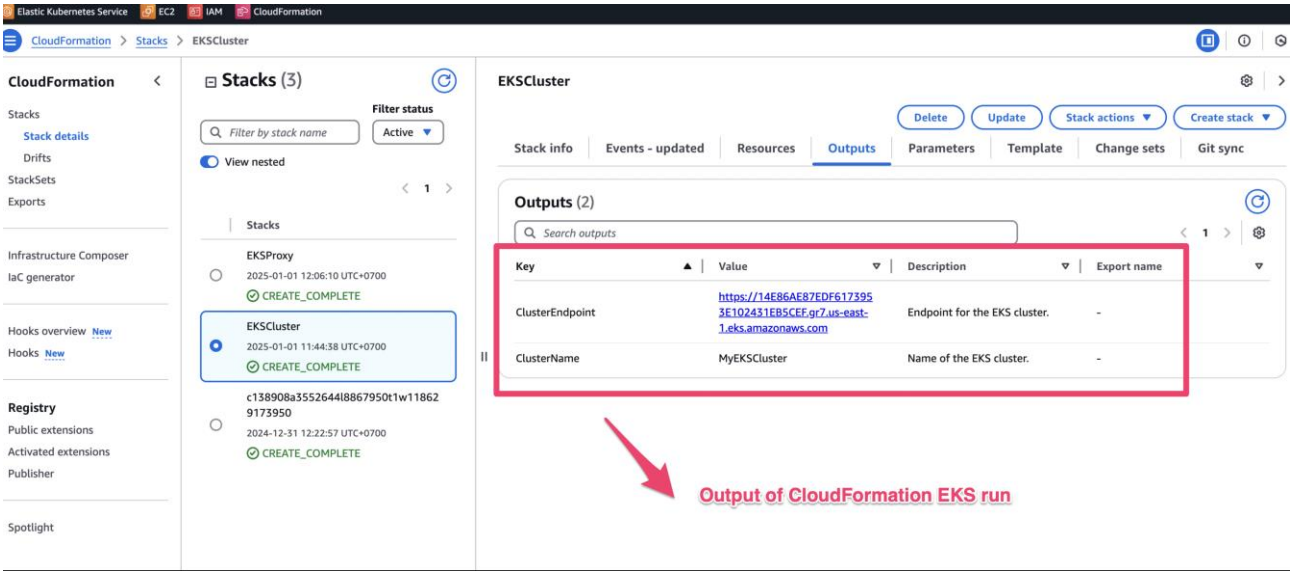


Figure 45. Completion of the Deployment Process for the IaC

We can deploy other resources using IaC following the same method.

APPENDIX I - AWS RESOURCES

AWS Resources Specifications

Table 5. EC2 Specifications

Name	Specs	EIP	AZ
Jenkins_Server	Instance Type: t2.micro AMI: Amazon Linux 2023 Key pair: devops_key Storage: 25GiB Security Group: Source Type: Anywhere 22 (for SSH) 8080 (for Jenkins GUI)	3.224.2.251	us-east-1d
Jenkins_Slave	Instance Type: t2.medium AMI: Amazon Linux 2023 Key pair: devops_key Storage: 40GiB Security Group: Source Type: Anywhere 22 (for SSH)	Public IP	us-east-1d
Ansible_Server	Instance Type: t2.micro AMI: Amazon Linux 2023 Key pair: devops_key Storage: 8GiB Security Group: Source Type: Anywhere	Public IP	us-east-1d
EKSProxy_Server	Instance Type: t2.micro AMI: Amazon Linux 2023 Key pair: devops_key Storage: 8GiB Security Group: Source Type: Anywhere 22 (for SSH)	Public IP	us-east-1d

APPENDIX J - CONTAINERIZED MICROSERVICES

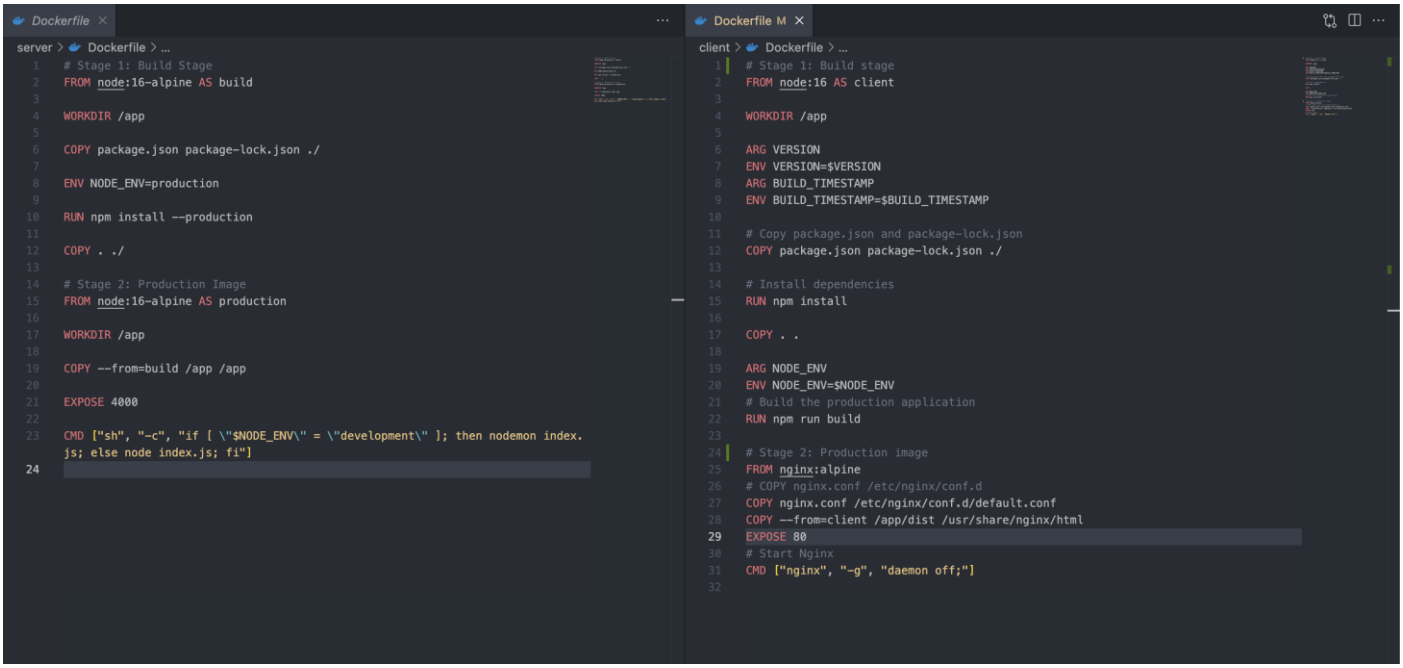


Figure 46. Dockerfiles for Frontend and Backend from client and server directories, respectively

For the frontend, a multi-stage Dockerfile is employed to streamline the build and runtime processes. The build stage uses the node:16 image to install dependencies, build the application (npm run build), and optimize the output for production. The runtime stage uses a lightweight nginx:alpine image to serve the static files, incorporating a custom Nginx configuration and production assets to ensure efficient delivery of content. The application is configured to run on port 80, making it production-ready and capable of handling user requests seamlessly.

For the backend, the Dockerfile also adopts a multi-stage approach. The build stage uses the node:16-alpine image to install production dependencies and prepare the application for deployment. The runtime stage then copies the built application into a clean node:16-alpine container. The backend runs on port 4000, leveraging nodemon during development for live reloading and node for a streamlined production environment. This multi-stage strategy minimizes image size while ensuring both the frontend and backend containers are optimized for their respective use cases.

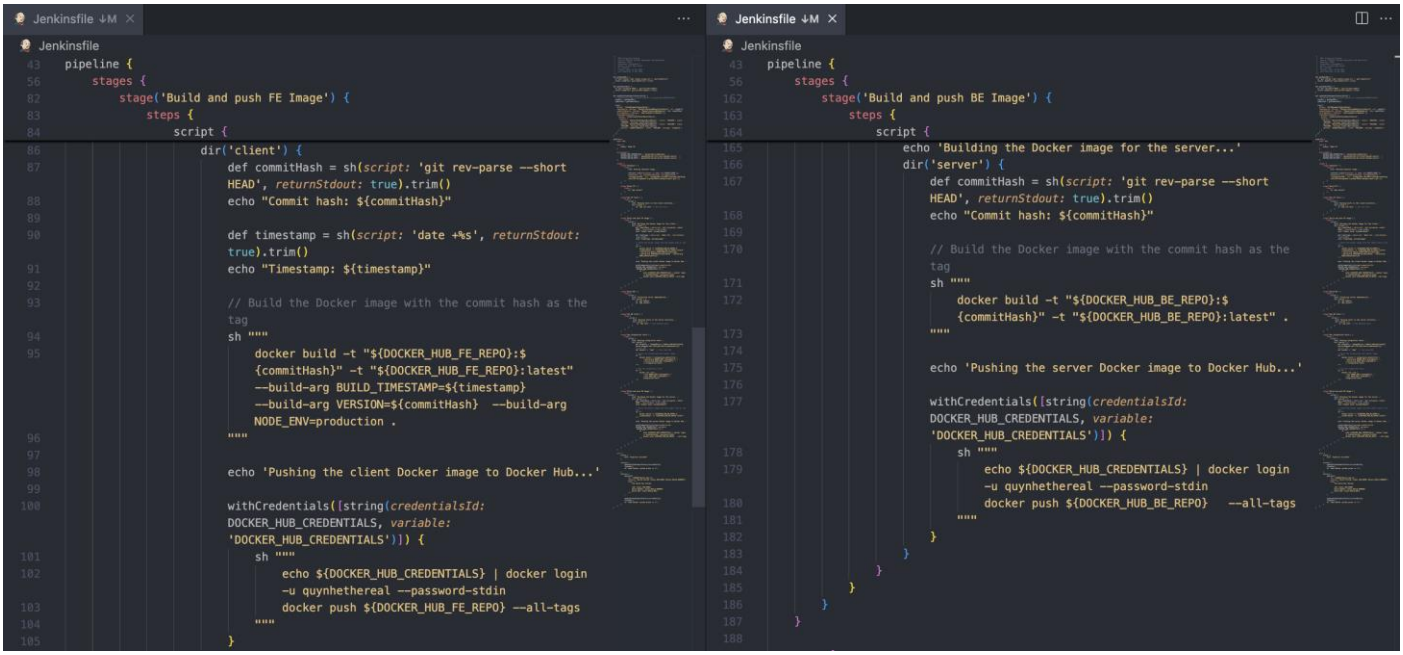


Figure 47. Jenkinsfile stage Build and Push Frontend and Backend Docker images to their respective DockerHub Repositories

The Jenkins pipeline includes dedicated stages for building and pushing Docker images for both the frontend (FE) and backend (BE). During each stage, the pipeline retrieves the current commit hash and timestamp, embedding these details as tags in the Docker image. This approach facilitates version tracking and ensures consistent tagging for easier deployment and debugging. The pipeline uses Jenkins' withCredentials block to securely manage Docker Hub credentials, enabling authenticated uploads of images to their respective repositories. This process not only enhances security but also ensures the integrity and traceability of the images.

APPENDIX K - ORCHESTRATION (EKS + ArgoCD)

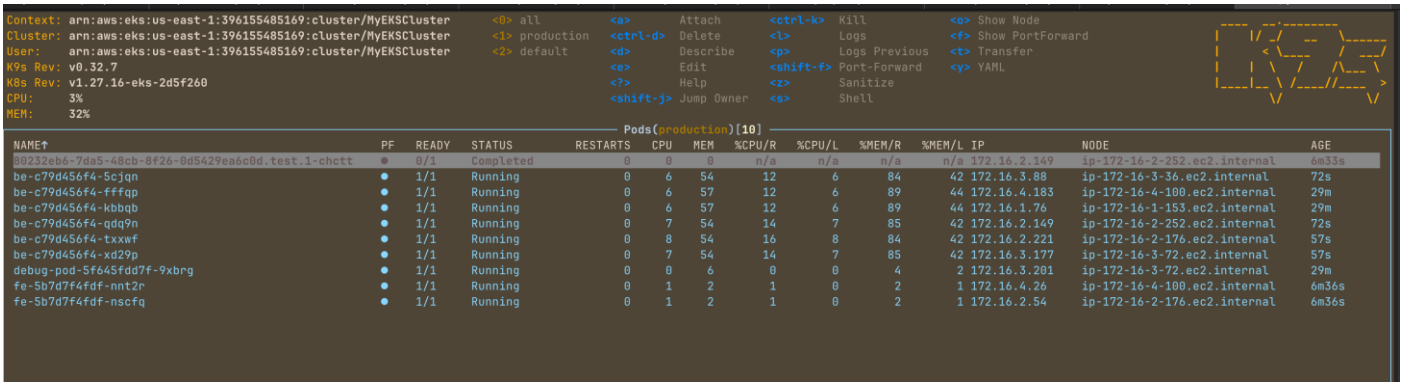


Figure 48. EKS Pods in K9S CLI

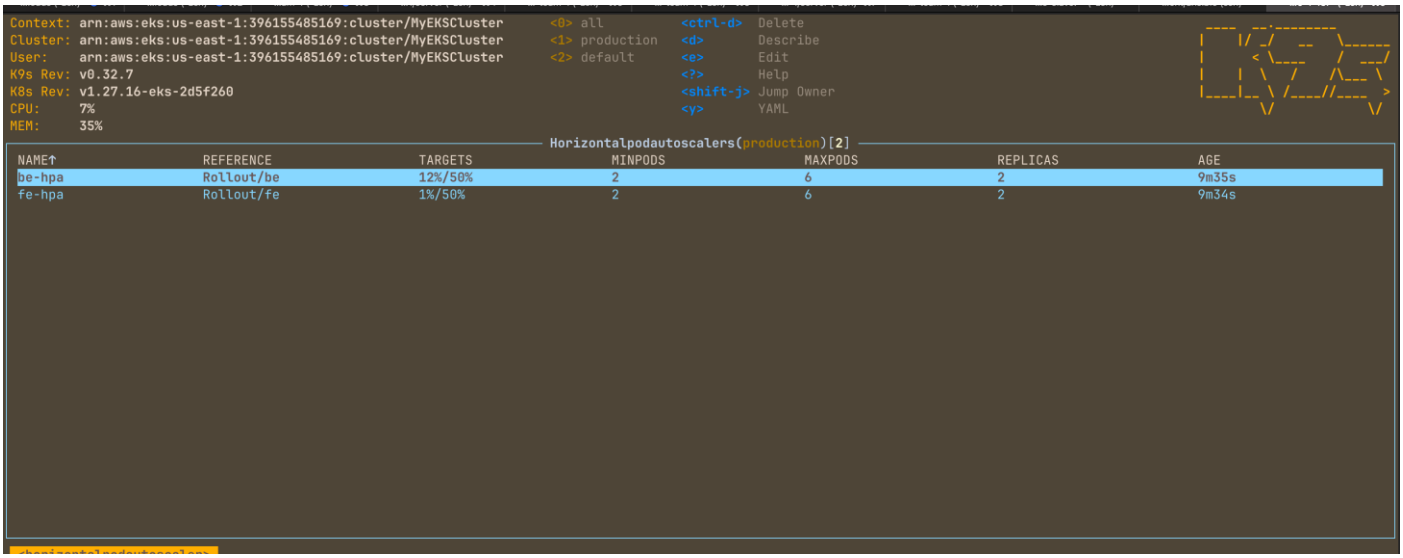


Figure 49. HPA Configuration for backend/frontend