CLIENT NAME
Computer Science Department
Swinburne Vietnam Alliance Program HCMC

FINAL
Technical Solution

Secure CI/CD Pipeline for Application Development

Version 1.0

Prepared by: GROUP 1

- Dang Vi Luan - 103802759
- Nguyen Linh Dan – 103488557
- Nguyen Duy Khang - 104056476
- Tran Bao Huy - 103505799

TABLE OF CONTENTS

1. BUSINESS SOLUTION

1.1. Business Plan

The Business Plan was created in the planning phase of the project as a brief overview of the business objectives, technology & tools breakdown, cost analysis, risks & recommendations for the project from a business perspective.

The Business Plan helps our team ensure that our solution will best meet the business needs.

Please refer to Appendix 1 – Business Plan.

1.2. Communication Plan

The purpose of the communication plan is to outline the key steps that clients (the Computer Science department) should take to effectively tranfer from their old system to the new proposed solution. The plan will provide a clear roadmap and timeline for how to communicate the changes, gather feedback from the end-users (IT students), and ensure a smooth implementation of the new system.

Please refer to Appendix 2: Communication Plan.

1.3. Project proposal

The Initial Project Proposal was created in the planning phase of the project to provide a complete and comprehensive summary of the requirements, scope, methodology, deliverables, and other aspects of the project.

This is crucial to ensure the project's operational efficiency and project alignment.

Please refer to Appendix 3 – Project Proposal.

1.4. Progress review report

The Progress review report was created on the 5th week of the project to record and analyze the results of the progress review conducted between our team and the client. We agreed and documented the achieved progress in the first 5 weeks and any additional requests from the client for the remaining weeks.

This document keeps our team on track with the client's request and helps improve client satisfaction.

Please refer to Appendix 4 – Progress Review report.

2. TECHNICAL SOLUTION

2.1. Revisit client's requirements

- **Functional requirements**

A. DevOps CI/CD Pipeline

- The DevOps CI/CD pipeline integrates and automates development, testing, and deployment stages to streamline software delivery.
- Integrate the codebase with Git for version control, using platforms like GitHub, Terraform Registry, and DockerHub for repositories.
- Automated build and testing processes are implemented using ArgoCD to trigger builds upon code commits or merges.

- Unit tests, integration tests, and static code analysis validate functionality and quality, providing feedback to developers.
- The pipeline uses Docker to build and push container images with versioning for traceability, and Dockerfiles are configured to automate the build and publish process.
- Terraform is leveraged for cloud infrastructure provisioning, integrating Terraform code into the CI/CD pipeline for consistent provisioning across environments and deploying the containerized application to a Kubernetes cluster.

B. DevSecOps Integration

- DevSecOps integration embeds security into the CI/CD pipeline. Pre-commit hooks using git-secrets prevent accidental commits of sensitive information, and AWS Secrets Manager controls access to credentials.
- SonarQube is used for Static Application Security Testing (SAST)
- GitHub for Source Composition Analysis (SCA), and OWASP ZAP for identifying vulnerabilities in running applications
- Docker images are scanned for vulnerabilities using tools like Clair.
- Prometheus collects and stores time-series metrics, and Grafana creates dashboards for monitoring system health and performance. Prometheus is configured to set up alerts based on predefined thresholds or anomalous behavior for timely incident response.

• Non-functional requirements

- Non-functional requirements ensure the CI/CD pipeline is robust, scalable, and reliable.
- The pipeline must handle increasing workloads, support horizontal scaling, and optimize resource utilization to minimize delays and bottlenecks.
- High availability and fault tolerance are ensured through redundancy and failover mechanisms.
- Compatibility with various development tools and platforms is maintained by adopting standard protocols and interfaces.
- Observability is achieved using Prometheus for metrics collection and Grafana for real-time dashboard visualization. Real-time alerting with Prometheus and Telegram ensures prompt incident response.

• Out of scope

- Out of scope components include:
- Training on cloud services and additional integrated service features
- Marketing or user adoption plans
- Ongoing management and support for third-party tools like Jenkins, GitHub, Docker, and Terraform.
- The project focuses on integration and deployment, excluding application bug fixing and enhancements. This means addressing application code bugs and developing new features are not covered within the project's scope.

2.2. Solution Package – A secure CI/CD pipeline

Our approach automates the deployment process, reduces human error, and ensures consistent and reproducible builds. By integrating security practices into the pipeline, students can protect their applications from vulnerabilities and threats, fostering a culture of security-first development. The solution includes 5 main components, CI Pipeline, CD Pipeline, Monitoring Feature, Infrastructure Provisioning Process, and Security Implementation.

CI pipeline

Our CI pipeline will encompass automation from the point where the developers commit their code onto GitHub until it is built into a Container image and pushed onto a remote image registry. During this process, the Kubernetes manifest files will also be updated for future deployment steps. The overall flow and explanation for this step are as follows:

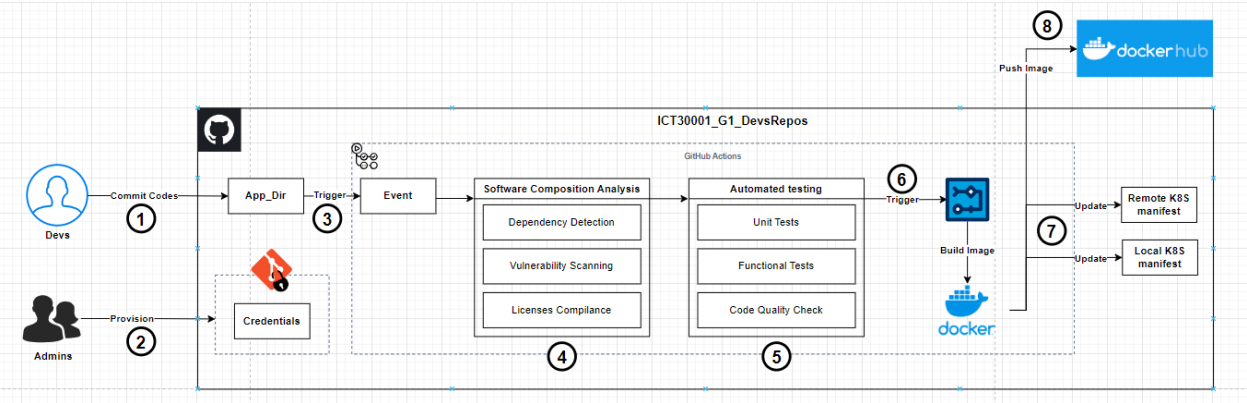

Figure 1: The overall CI flow

[1] The developers can make changes to their code on their local machine. Once they are done and satisfied with the changes, code commits can be made to a remote **GitHub repository**. This will store the code in a directory for collaboration and trigger an automation process.

[2] The credentials for the SaaS platform and cloud services will be provided by the operation administrators and stored as variables on **GitHub Secret**. By doing this, the developers can simply refer to these credentials in their code without exposing such sensitive assets to the public.

[3] Once the code has been committed, it will trigger a **GitHub Actions workflow** for integration. Various tests will be conducted before the code can be built into an image.

[4] **Software Composition Analysis** will check for securities malpractices in the developers' code and decline the CI flow should there be any anomalies.

[5] **Automated testing** written by the testers will be included in the directory for testing before the code can be containerized.

[6] After completing all defined testing, developers' code will trigger another workflow that will containerize their code into a **Docker** image file.

[7] After the completion of the containerization process, **Kubernetes manifest files** will be rewritten to reflect the changes in both the local and remote environments.

[8] Finally, the container image will be pushed to **Docker Hub -** a remote registry for container images.

The completion of this process will result in a Docker image containing developers' code changes on Docker Hub, as well as modifications to the Kubernetes manifest file both in local and remote environments.

CD pipeline

After the Continuous Integration (CI) pipeline has finished executing, it will be triggered to reflect the changes made by developers in their code across the local and remote environments. The overall flow and explanation for this process are as follows:

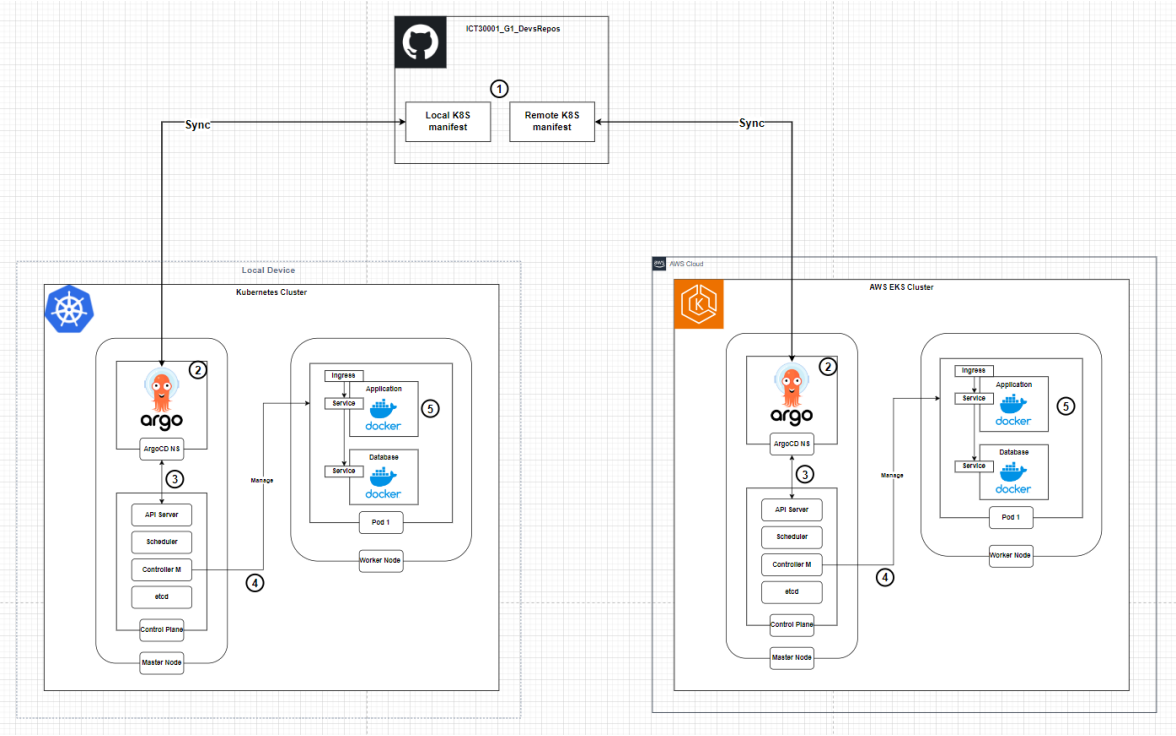

Figure 2: The overall CD flow

[1] As mentioned in the CI flow, once the GitHub Action workflow is completed, Kubernetes manifest files in the developer repository will be modified to reflect new codes. There will be two manifest files, one for the local **environment** and the other for the remote **environment,** on the AWS Elastic Kubernetes Service (EKS).

[2] **Argo CD** will track changes made to manifest files; it will then compare the state of the manifest files with the current state of the Kubernetes cluster. This applies to both local and remote environments.

[3] Argo CD will make API calls to **Kubernetes Control Plane** should there be any changes. Kubernetes Control Plane will make appropriate modifications depending on the manifest file for the Worker Nodes.

[4] **Controller M** will then make API calls to **Worker Nodes** to modify them depending on the code's changes.

[5] After a moment, changes from the developer will be reflected in both the local environment and the remote environment.

The decision to have both the local environment and the remote environment is to maximize the efficiency and flexibility of the CI/CD pipeline for the developer's usage. The developer can have a local environment for their Git's feature branch. The remote server can then only be used for showing changes in Git's main branches. The diagram below shows the infrastructure of the remote server on AWS.

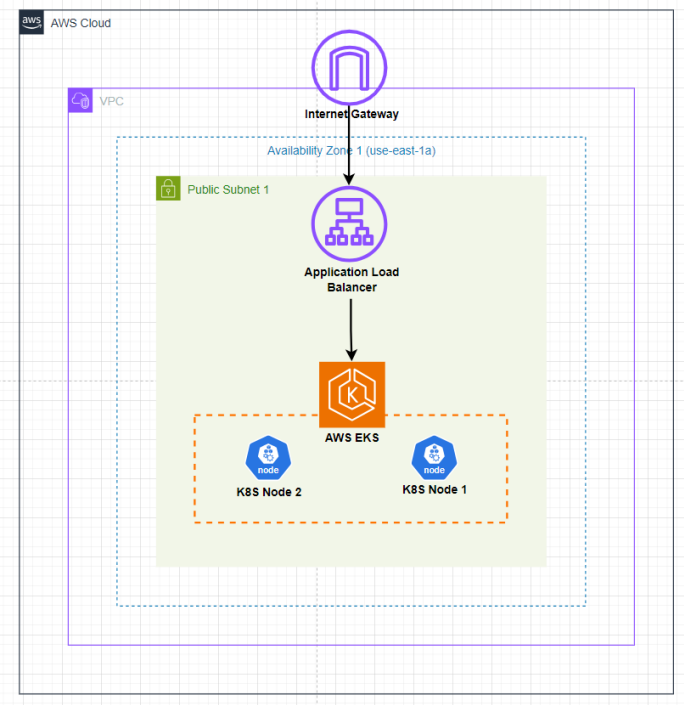

Figure 3: AWS EKS infrastructure

After setting up the CI and CD flows:

The combination of CI and CD flow will automate the integration and deployment process of developers' codes, easing the burden on developers and enabling them to focus only on meaningful development tasks.

Infrastructure Provisioning Process

Our solution utilizes the GitOps approach to automate the integration and deployment processes. GitOps is a modern operational framework that leverages Git as the single source of truth for managing infrastructure and application deployments. By using Git repositories to store and version-control configuration files, GitOps enables automated and continuous deployment processes.

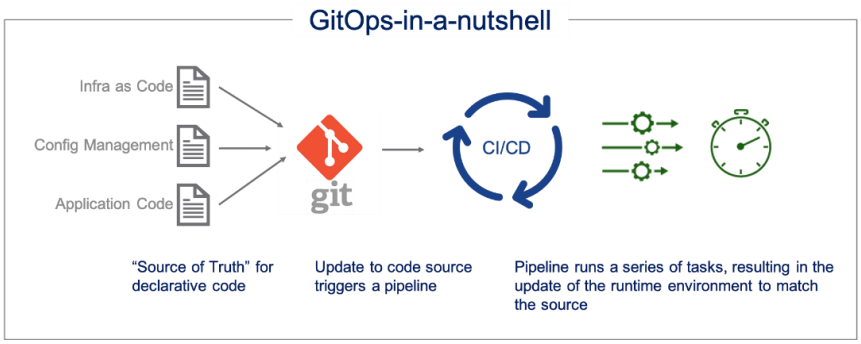

Figure 4: GitOps summary

A massive advantage of GitOps in conjunction with Infrastructure as Code (IaC) is that our infrastructure on remote servers can be expressed as codes, which can therefore be controlled using Git. Making use of this advantage, our solution uses Terraform to provision infrastructure. Below is the overall flow of our infrastructure provisioning.

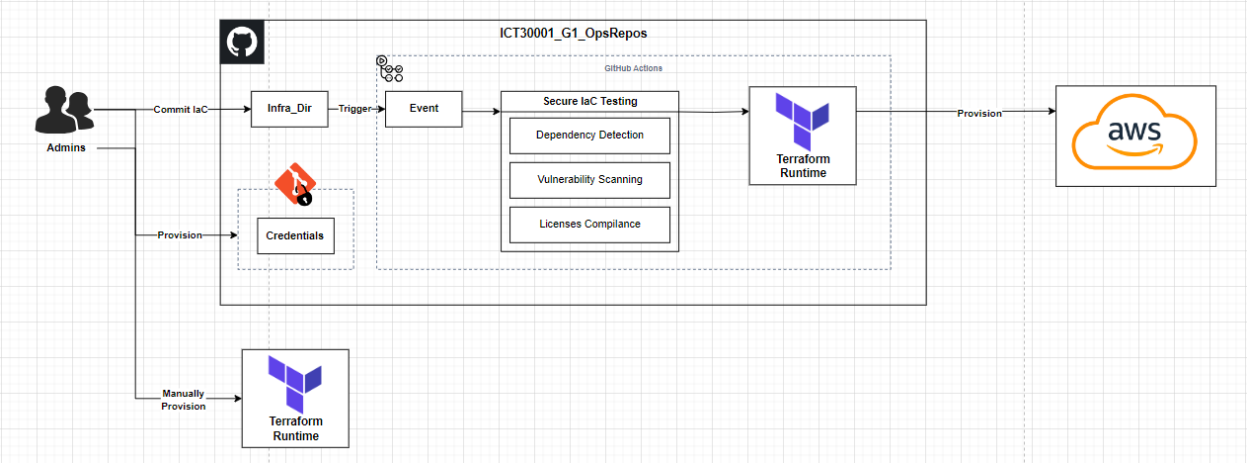

Figure 5: Infrastructure provisioning process

This process is quite similar to how the developers will manage their code. The operation administrators will make appropriate changes to the infrastructure based on business requirements using Terraform. They will commit the code to the Operation Team's repository; this will trigger a GitHub Actions workflow and provision infrastructure on AWS according to Terraform's codes.

Security implementation

During the interaction with industry standards, our team reckons the importance of implementing security for our solution. Since then, security implementation has been our primary focus, besides the functional requirements for the pipeline. We implemented security features in three main areas.

- **Credentials Security using GitHub Secrets**

Credential security using GitHub Secrets is crucial for protecting sensitive information such as API keys, passwords, and tokens in your repositories. Below is an example of storing API passwords and keys for GitHub action usage.

This practice mitigates the risk of unauthorized access and potential breaches, safeguarding both the application's integrity and the data it handles. As GitHub Secrets integrates seamlessly with GitHub Actions, this allows for secure and automated workflows without compromising sensitive information.

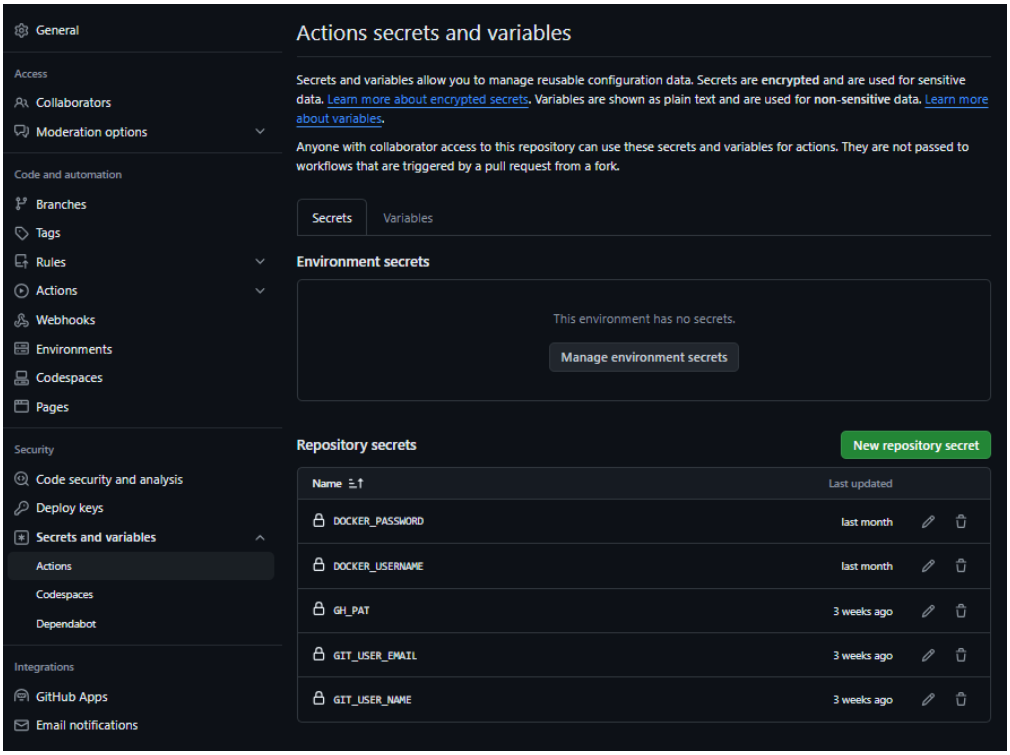

Figure 6: GitHub Secret

- **Static Application Security Testing (SAST) using Sonarqube**

Static Application Security Testing (SAST) is essential for identifying and mitigating security vulnerabilities early in the software development lifecycle.

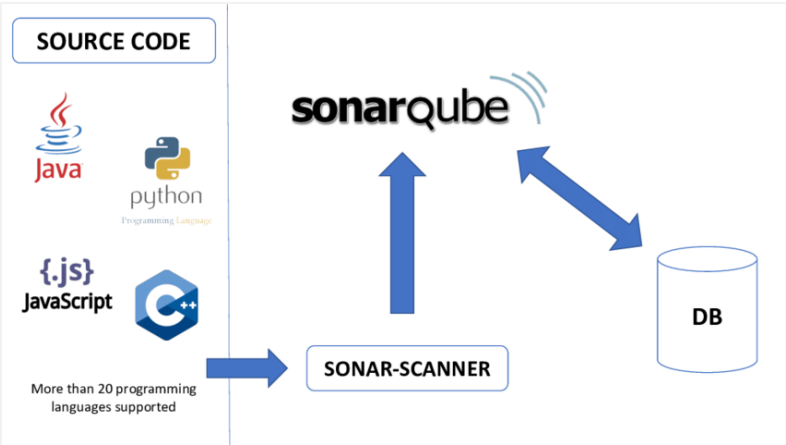

Figure 7: Sonarqube for SAST

Our tool of choice is Sonarqube. It can analyze source code for potential security issues, such as SQL injection, cross-site scripting (XSS), and other common vulnerabilities, providing detailed insights and recommendations for remediation.

- **Dynamic Application Security Testing (DAST) using OWASP ZAP**

Dynamic Application Security Testing (DAST) is critical for identifying and addressing security vulnerabilities in running applications.

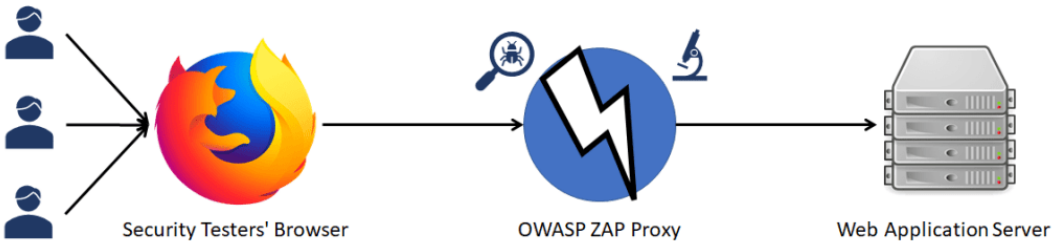

Figure 8: OWASP ZAP

ZAP simulates real-world attacks, scanning web applications for issues such as cross-site scripting (XSS), SQL injection, and other security flaws. By incorporating DAST with ZAP into the development and testing process, teams can detect vulnerabilities that static analysis might miss, ensuring comprehensive security coverage.

Monitoring feature

Apart from delivering the CI/CD pipeline for the solution, we also tackle the monitoring aspect of this pipeline in order to provide a holistic solution for the application development process. Below is a diagram of how our solution integrates Prometheus and Grafana for monitoring purposes.

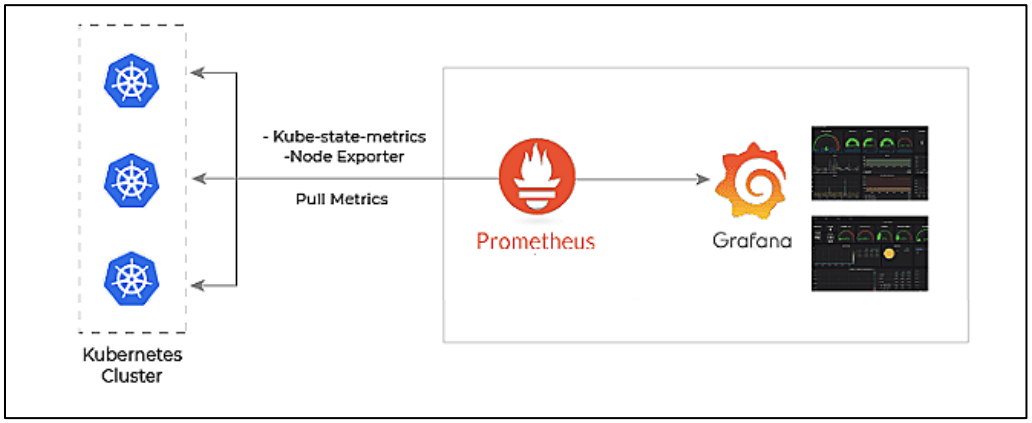

Figure 9: Prometheus and Grafana for monitoring

Using Prometheus and Grafana on Kubernetes provides powerful monitoring and visualization capabilities that are essential for managing and optimizing containerized applications. Prometheus excels at collecting and storing time-series data from various Kubernetes components, enabling detailed performance metrics and alerting. Grafana complements Prometheus by offering a versatile and user-friendly interface for visualizing this data through customizable dashboards.

Together, they enable real-time monitoring, proactive issue detection, and insightful analytics, helping teams maintain the high availability and performance of their applications.

Websites

There will be 2 main web application UI that our solution revolves around, the first is the web UI of ArgoCD used to create applications, syncing changes from GitHub to various environments, and tracking the status of Kubernetes' entities. The other would be the Monitoring website from Grafana, this web UI is crucial to have a holistic overview of the CI/CD pipeline as well as the deployed environment.

- **ArgoCD's Web UI**

On this Web UI, the developers as well as the operation staff can create applications from the GitHub repository in their chosen environment. Refer to the user manual's video for detailed steps for this process.

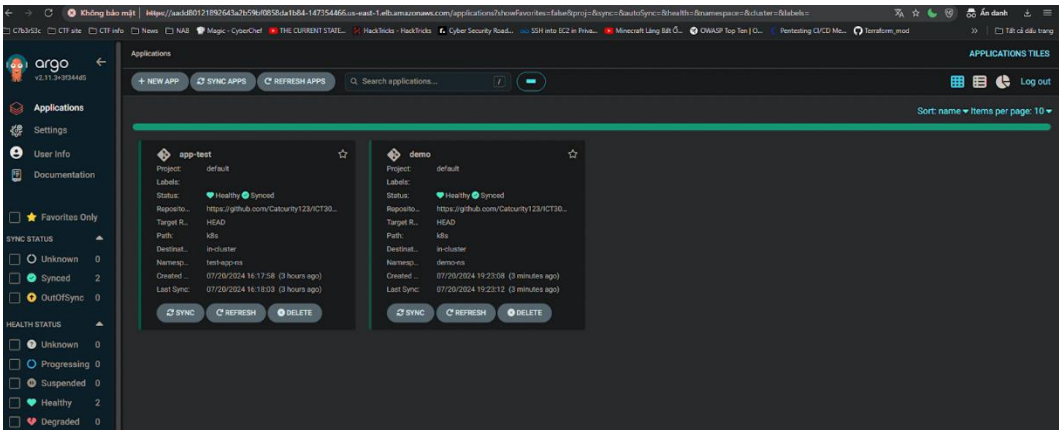

Figure 10: ArgoCD UI

- **Prometheus and Grafana Monitoring Web UI**

Monitoring and visualizing application performance using Prometheus and Grafana is vital for maintaining the health and efficiency of modern software systems. Refer to the user manual's video for detailed steps for this process.

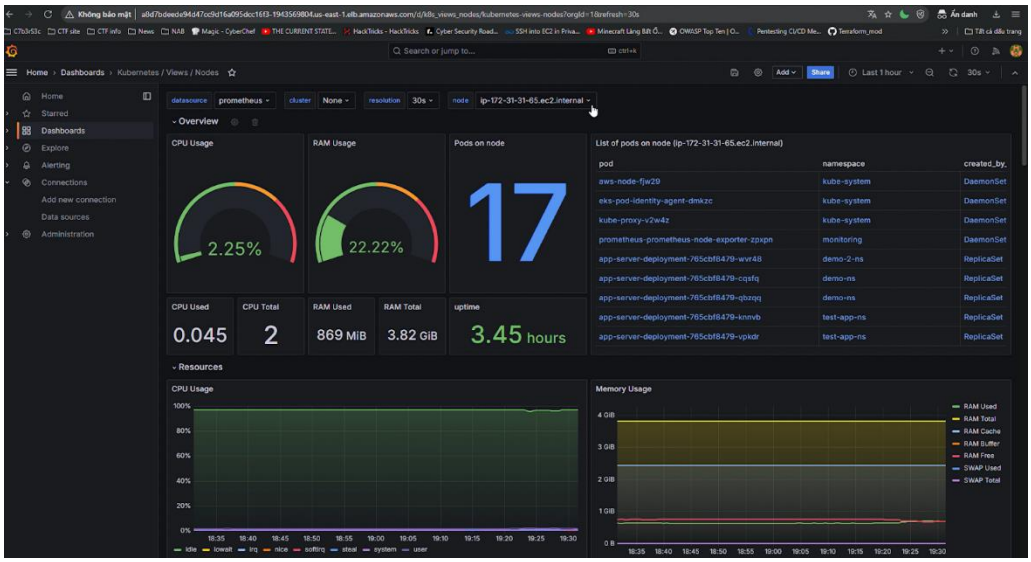

Figure 11: Monitoring tool – Grafana UI

User documentation

- **The planning of user documentation**

In the planning phase, outline the entire CI/CD pipeline setup and gather necessary tools like Docker, Minikube, and GitHub Actions. Decide on AWS EKS for Kubernetes management and choose monitoring tools such as Prometheus and Grafana. Plan the necessary IAM roles and permissions in AWS, draft YAML manifest files for Kubernetes, and organize the sequence of installation and configuration steps. A clear roadmap ensures a smooth implementation.

- **The execution of user documentation**

During the writing phase, create detailed instructions and scripts based on the plan. Write the user manual with step-by-step guides for installing Docker, and Minikube, and configuring CI/CD pipelines using GitHub Actions. Draft YAML scripts for Kubernetes resources and document commands for creating IAM roles, setting up the EKS cluster, and configuring compute resources. Include installation and configuration steps for ArgoCD, Prometheus, and Grafana with necessary Helm commands.

- **The review of user documentation**

In the review phase, we invite some stakeholders to review the documentation by redoing the steps in the documentation as well as the video to ensure it is comprehensible to all audiences.

Please refer to Appendix 5 for more information about the artifact.

Usability testing report

The usability testing plan aimed to evaluate the ease of setting up the CI/CD pipeline, assess user navigation and understanding, identify pain points, and measure the effectiveness of security and monitoring features. The methodology included one-on-one moderated sessions, think-aloud protocols, task completion times, and post-test feedback through questionnaires and interviews. Target participants were IT students and software developers with varying levels of CI/CD knowledge.

- **The Scenario for Testing**

Testing was conducted across six scenarios:

1) Initial setup and GitHub integration.

2) CI pipeline configuration.

3) CD pipeline setup with ArgoCD.

4) Infrastructure provisioning with Terraform.

5) Monitoring setup with Prometheus and Grafana.

6) Implementing security measures.

- **The Summary of Results of All Participants**

Participant 1 (Junior Developer): Completed 5 of 6 tests successfully but struggled with Terraform and had some issues interpreting security scan results. Feedback indicated familiarity with CI/CD but challenges with infrastructure and security aspects.

Participant 2 (IT Student, Final Year): Completed 5 of 6 tests, needing minor assistance with ArgoCD and Terraform. Demonstrated a good understanding of CI/CD concepts but found infrastructure and advanced monitoring challenging.

Participant 3 (IT Student, 2nd Year): Completed 3 of 6 tests, with difficulties in CI/CD processes and ArgoCD. Had some understanding of Terraform and monitoring but struggled with SAST and DAST. Feedback suggested a need for more background information and glossaries.

Participant 4 (IT Student, Freshman): Completed 0 of 6 tests, struggling with basic concepts and setup. Feedback revealed that the process was overwhelming and disconnected from their current knowledge.

- **The Outcome and Feedback After Testing**

The usability testing revealed that participants with more CI/CD experience could complete most tasks, though some struggled with advanced topics like Terraform and ArgoCD. Freshmen and less experienced users found the pipeline too complex. Feedback suggested a need for more detailed guides and educational resources for beginners, as well as potential simplifications for those new to CI/CD concepts.

Please refer to Appendix 6 for more information about the artifact.

Note:

Besides the User Manual and Usability Testing Report, the solution package also includes other documents/reports. However, they are not directly related to the technical aspects, so we did not include them in this report.

3. HOW THE SOLUTION MEETS THE REQUIREMENTS

Overall, the developed software package described in Section 2.2 addresses the key requirements identified in the project proposal. By building continuous integration (CI) and continuous deployment (CD) workflows and combining them to create the final CI/CD pipeline, the system is able to efficiently and reliably deliver updates to both the local development and remote production environments. This automated approach ensures that the evolving codebase is thoroughly tested and validated before being deployed to the live system. While the implemented CI/CD pipeline may lack some of the advanced features found in enterprise-level systems, it still satisfies the demands of the end-users - Swinburne University IT students who need to find a better way to manage their source code and automate parts of the deployment process of their university projects.

To demonstrate how our solution and the relevant artifacts created during the project can meet the client's requirements, we will provide a detailed overview in the following sections. This discussion will be divided into two parts:

- Our technical solution meets the core requirements.
- The relevant artifacts were developed throughout the project to support the solution.

By covering these three aspects, we aim to comprehensively showcase how our proposed solution effectively fulfills the project requirements.

Technical solution

The project's core objective was to design and develop a CI/CD pipeline to enable IT students to manage their code deployment more efficiently. The pipeline was required to have foundational CI/CD functionalities, such as automated build, testing, and container image management processes.

Additionally, the pipeline was expected to provide effective source code management and operational monitoring capabilities. Crucially, the CI/CD solution was required to automate the deployment process while also incorporating security measures, including vulnerability scanning, performance monitoring, and real-time risk alerting.

The following sections compare the specified requirements against the current implementation to provide a clear assessment of the project's outcome quality. This comparison ensures the delivered CI/CD pipeline meets the initial objectives and satisfies the specified functional and security-related requirements.

- **Part 1: General Functionalities of the CI/CD Pipeline (DevOps Methodology)**

Requirement 1.1: Use ArgoCD to automate build and testing processes.

Implementation 1.1: We successfully built a CI pipeline to automate the code integration and testing using GitHub Actions. Security checks and automated test cases are also included. Besides ArgoCD, we also integrated Sonarqube and SAST to support the feature.

Requirement 1.2: Integrate the pipeline with Docker to build and push the container images.

Implementation 1.2: We developed a feature to allow Docker images to be created and pushed to DockerHub, with automated updates to Kubernetes manifests.

Requirement 1.3: Use Git to support source code management.

Implementation 1.3: We have set up GitHub as a place for code commits and collaboration between project team members. The credentials are protected using GitHub secrets.

Requirement 1.4: Use Terraform for cloud infrastructure provisioning

Implementation 1.4: We applied the GitOps approach to our solution and wrote Terraform code to provision the AWS infrastructure (IaC).

- **Part 2: Increase the security level of the CI/CD pipeline (DevSecOps methodology)**

Requirement 2.1: Integrate tools to scan for security vulnerabilities.

Implementation 2.1: Our team integrated the CI/CD pipeline Sonarqube, which is used by SAST to scan the source code. We also implemented security checks in the CI flow. We also integrated DAST with ZAP into the development and testing process to identify the vulnerabilities of running applications.

Requirement 2.2: Scan Docker images for vulnerabilities.

Implementation 2.2: Initially, we planned to use Clair to scan the images. However, we found that the images are scanned automatically when pushing the image to DockerHub. This scanning tool is a feature that is available on Docker Hub. Therefore, we decided to scan via DockerHub instead of Clair.

Requirement 2.3: Monitoring the pipeline performance

Implementation 2.3: Our team successfully integrated Prometheus and Grafana into the pipeline for real-time monitoring of the pipeline and sending alerts via email when any risks are detected.

The functional requirements outline the key features of the CI/CD pipeline. They define the pipeline's core functionality and enable the delivery of applications and services. On the other hand, non-functional requirements are crucial in determining the overall stability and reliability of the pipeline, which makes it essential for users to use the functional features. Similar to functional requirements, the description of what our team developed for the non-functional requirements is included to determine whether we satisfied the clients' demands written in the project proposal.

- **Part 3: Non-functional requirements of the secure CI/CD pipeline**

Requirement 3.1: Performance and Reliability

The CI/CD pipeline must be able to handle increasing workloads and maintain high availability to provide the developers (in the project team of IT students) with quick feedback and a clear development cycle.

Implementation 3.1: Our team has set up Prometheus and Grafana to monitor the Kubernetes cluster. We made changes to the service types of Prometheus and Grafana from NodePort to LoadBalancer. This provides a highly available and scalable solution, as the load balancer can distribute incoming traffic across multiple nodes, improving the overall performance and reliability of the monitoring infrastructure.

Requirement 3.2: Tool compatibility

Implementation 3.2: During the pipeline development, we have integrated several supportive tools into the CI/CD pipeline, such as GitHub, ArgoCD, Docker, Helm, Prometheus, Grafana, etc. We have resolved all conflicts between tools and ensured that the final solution package has a high compatibility level.

Requirement 3.3: Metric collection, alerting, and visualization

Implementation 3.3: Our team integrated Prometheus tool for metric collection and Grafana for metric visualization and alerting. We achieved this requirement as we have established a robust solution for performance observability to enable timely troubleshooting for the pipeline.

Relevant artifacts

- **Artifact 1: User documentation**

From the project scope, the final CI/CD pipeline solution will include user documentation and training materials. This will ensure that the pipeline's functionality and features are fully documented and that end-users receive the necessary guidance to use the CI/CD pipeline in their development workflows.

To support the deployment of IT student projects, our team has prepared a user manual, which provides step-by-step instructions for setting up and implementing the CI/CD pipeline. Additionally, the team has created tutorial videos to demonstrate how to perform basic tasks and use the pipeline's key features to support the deployment process of IT students' projects.

- **Artifact 2: Usability Testing Report**

While a usability testing report is not in the project scope, our team has included it as an additional deliverable. This report provides a comprehensive evaluation of the CI/CD pipeline's user interface, workflows, and overall user experience. This testing report could serve as a reference for further improvement and ensure the solution meets the end-users' expectations.

- **Artifact 3: Final Report and Implementation Strategy**

These documents are also out of scope. However, we decided to include them in the project outcomes to consolidate the project objectives, deliverables, development process, and recommendations gathered throughout the CI/CD pipeline implementation process. The implementation strategy outlines a detailed roadmap for deploying and integrating the CI/CD pipeline into the client's existing infrastructure and development workflows. These documents can demonstrate our commitment to delivering a complete and well-documented solution.

Note: The Implementation Strategy will be included in the Final Report.

4. ADDITIONAL FEATURES

Technical development

The implementation of local Kubernetes deployment as an additional feature to the initial cloud EKS deployment requirement improves flexibility in deployment and testing for the following reasons:

- Enhanced development workflow: Local deployments allow developers to work efficiently on feature branches. The developer can have a local environment for their Git's feature branch.
- Segregation of development stages: This approach creates a clear separation between feature development and main branch deployments, with "The remote server can then only be used for showing changes in Git's main branches."
- Faster iteration cycles: Local deployments enable quicker testing and development iterations without waiting for cloud deployments.
- Resource optimization: By enabling local testing, the solution reduces load on cloud infrastructure, potentially leading to cost savings and more efficient resource use.
- Improved flexibility: The decision to have both the local environment and the remote environment is to maximize the efficiency and flexibility of CI/CD Pipeline for the developer's usage.

Documentation preparation

Although the project is primarily focused on the technical aspects and successfully delivering a final technical solution package, the business components were considered to be out of scope. However, our team believes that analyzing the business context is indispensable to ensuring the successful delivery of the project. Therefore, in addition to the technical work, our team also conducted some business analysis to draft two key documents. Our business-related documents include:

- Business Plan
- Communication Plan

The first document summarizes the potential risks and mitigation strategies that may come into play when bringing the new technical solution to the business. This allows the client, the computer science department, to be prepared for any challenges that may arise during the transition. The second document outlines a detailed timeline of different steps, such as organizing kick-off meetings and training workshops, to help facilitate the end-user's acceptance and adoption of the new solution.

APPENDIX

1 - Business Plan

2 - Communication Plan

3 - Project Proposal

4 - Progress Review report

5 - User Documentation

6 - Usability Testing report