N°: Ing-GInfo-09-2024

Département Génie Informatique

END-OF-STUDY PROJECT REPORT

Presented in order to obtain

Engineering degree in Computer Sciences

Specialty: **computer science engineering**

By Mr.: CHRIGUI Abdelbaki

Title of PFE:

Secure and automate CI/CD pipeline with enhanced monitoring

Realized within DOCIC

Defended on June 27, 2024, *in front of the jury composed of:*

President:	Mrs. GHRAIRI Salsabil
Reviewer:	Mrs. KRICHENE Hajer
University supervisor:	Mrs. BEN AZZOUZ Lamia
Industrial supervisor:	Mr. TRABELSI Sofiene

Academic Year 2023 – 2024

Acknowledgments

It is with great pleasure that I reserve these lines in order to thank everyone who has helped me to succeed in my internship and develop my report.

First, I would like to thank my academic supervisor Ms. Lamiya for sharing his experience and skills with me and for the time he devoted to me throughout this internship period.

I would also like to thank the entire DOCIC company team for the enormous helping hand and

the warm welcome within the company and for the support and valuable advice.

I would like to express my sincere gratitude to the pedagogical team of the National School of Engineers of Tunis (ENSIT) for the quality of the teaching and for having ensured the theoretical part of it.

Table of Contents

Table of figures

Table of tables

General Introduction

In a regular workflow, the developer makes the product available to the end users and it must work well, but not always. The software handed off to IT operations may not work on their side. This conflict between dev and ops is known as the Wall of confusion. The two teams work separately, with different goals and understandings of what is good and what is bad debts how software release over the wall to operations for deployment, and once the problem happens, both teams struggle to define why. To prevent such frustrating situations, the software development community has shaped a strategy called DevOps which aims to overcome this gap. DevOps combine development and operations through continuous integration and continuous deployment (CI/CD) pipelines, ensuring collaboration and restructuring the production cycle.

Our End of year project, "Secure and Automate CI/CD Pipeline with Enhanced Monitoring," addresses the needs of DOCIC, a Tunisian startup that focuses on innovative solutions for medical records management and healthcare optimization. The project aims to implement a comprehensive on one of the Devops practices (CI/CD) by leveraging a pipeline using GitLab, improving DOCIC's software infrastructure management.

This report synthesizes the work and is structured around five chapters:

- Chapter 1: Introduces DOCIC, its current software development challenges, and proposed solutions through virtualization, automation, and security integration.
- Chapter 2: Details project requirements, product backlog, sprint planning, and system architecture using Docker, Ansible, and OVHCloud.
- Chapter 3: Focuses on implementing the CI/CD pipeline, including GitLab runners, Docker containers, NGINX configuration, and automated monitoring.
- Chapter 4: Presents the deployment of the CI/CD pipeline, monitoring with Prometheus and Grafana, and security measures.
- Chapter 5: Discusses integrating DevSecOps practices, OWASP best practices, secret management with HashiCorp Vault, and integrity validation using HMAC.

We conclude our report with a general conclusion and future perspectives.

Chapter 1: Overview and Context

Introduction

In this chapter, we will discuss the current situation of the company's software development practices, identify the key challenges we face, and propose solutions through the implementation of a CI/CD pipeline using GitLab. We will also outline the project methodology and the selected approach to ensure a successful transition to DevOps practices.

1.1 Project Context

The project described in this report is entitled: "Secure and Automate CI/CD pipeline with Enhanced Monitoring". It was proposed as the subject of a final year project within the DOCIC company with a view to obtaining the national diploma of computer science engineer from the Higher National Engineering School of Tunis ,additionally, the company had requested this solution to address their own operational needs, highlighting the practical relevance and immediate applicability of the project's outcomes to DOCIC's infrastructure

1.2 Company Overview

DOCIC is a Tunisian startup founded in 2022 by Sami Kallel, a doctor passionate about technological innovation in the field of health. Convinced of the importance of improving medical record management to optimize patient care, Sami Kallel had the idea of creating DOCIC to fill this gap in the health system. As the company's founder and main investor, he brings valuable medical expertise as well as a strategic vision to develop solutions adapted to the needs of healthcare professionals and their patients.

Alongside him, Sofiene Trabelsi serves as DOCIC's Chief Technology Officer (CTO) and Chief Executive Officer (CEO). With her background in technology and business management, Sofiene oversees the development and implementation of DOCIC's technology solutions, ensuring their effectiveness and scalability. Her dynamic leadership and commitment to innovation are key pillars in achieving DOCIC's vision, following is their logo, the figure 1-1 shows the company logo.

Figure 1-1 logo DOCIC

1.3 Core Activities

DOCIC's field of activity lies at the intersection of technology and healthcare. As a startup specializing in the field of digital health, DOCIC focuses on developing and implementing innovative solutions to improve medical records management and optimize healthcare.

- Software development: DOCIC designs, develops and deploys a web and mobile application to enable healthcare professionals to efficiently manage their patients' medical records. This includes the development of web and mobile applications, as well as the creation of electronic medical records (EMR) systems.
- Data Security and Privacy: As a company operating in the healthcare field, DOCIC pays particular attention to the security and privacy of medical data. Its activities include implementing robust security measures and complying with regulations such as the Health Information Protection Act (HIPAA).

Research and Development: DOCIC invests in research and development to continually improve its products and services, anticipate the future needs of the healthcare sector and stay at the forefront of technological innovation

1.4 Customers and partners:

DOCIC's customers and partners represent key players in the healthcare and technology ecosystem, collaborating with the company to promote the adoption of its solutions and contribute to its success.

Healthcare institutions: Hospitals, clinics, medical practices and care centers that are looking to modernize their medical records management and improve the efficiency of their operations.

Healthcare professionals: Doctors, nurses, physiotherapists and other healthcare professionals who want to have advanced digital tools to facilitate the care of their patients and optimize their medical practice.

Insurers and health organizations: Health insurance companies, mutual insurance companies and other health service providers who can benefit from access to up-to-date medical data to evaluate

1.5 Current Situation Analysis

The current situation in our company is characterized by a significant lack of automation in the software development and deployment processes, leading to various issues:

- Manual Deployments: Releasing software manually is error-prone and time-consuming. It involves extensive documentation and repetitive tasks that increase the risk of human error, making it difficult to fully test the deployments and often requiring frequent calls to the development team to troubleshoot issues on release day.
- Manual Configuration: Managing production environments manually leads to unintentional configuration drifts and difficulties in maintaining consistent settings across different servers. This can cause last-minute changes during deployments and complicated rollback procedures

Figure 1-2 describes the deployment situation in the enterprise.

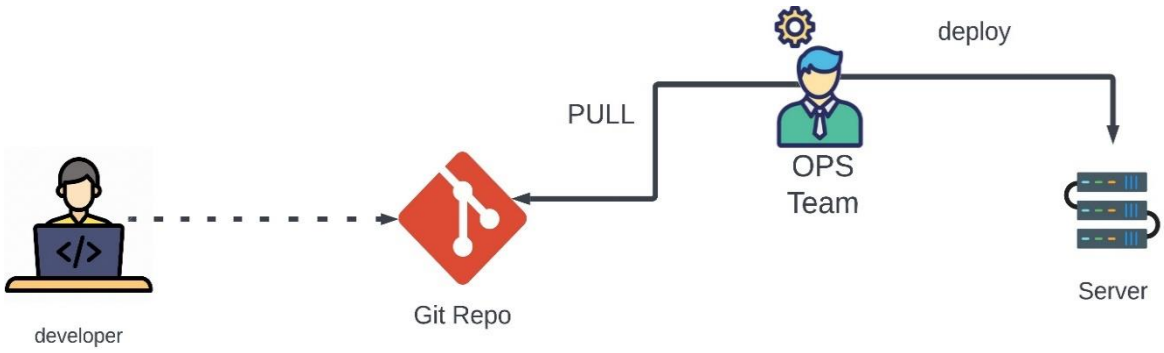


Figure 1-2:Current deployment methods

1.1 Current Situation issues

- Lack of virtualization: in the current workflow the developer directly pushes code changes to a Git repository, and the Ops team manually deploys the code to a physical server Without virtualization, the deployment process becomes tightly coupled with the underlying infrastructure, making it challenging to ensure consistent environments across different environments and increasing the risk of environment-specific issues.

- Lack of Automation: Without automated processes, our team spends a significant amount of time on deployment and configuration tasks, which could be better utilized in development and innovation. The absence of continuous integration and continuous delivery (CI/CD) practices leads to slower release cycles and delayed feedback on code changes. This lack of automation creates bottlenecks in our development workflow, making it challenging to keep up with market demands and deliver timely updates to our customers

1.6 Proposed Solutions

The primary challenge we face is implementing a secure and automated CI/CD pipeline using Devops practices This involves:

- Implementing virtualization using containerization technologies, such as Docker. Virtualization allows us to create isolated, consistent, and reproducible environments for our applications, ensuring that they run identically across different stages of the pipeline (development, testing, staging, and production). Containerization encapsulates our application and its dependencies into lightweight, portable containers, making it easier to deploy and manage applications across different environments.
- Ensuring automated, reliable processes for building, testing, and deploying code to reduce human errors and inconsistencies. Automating these processes will help us achieve more frequent and reliable releases, reducing the risk of introducing bugs and performance issues.
- Integrating security measures within the pipeline to protect the integrity of the code and the deployment environment. This includes implementing automated testing for security vulnerabilities and ensuring that any changes are thoroughly vetted before they reach production. Security is a critical aspect of modern software development, and it's essential to incorporate security checks and validations throughout the CI/CD pipeline.
- Facilitating seamless collaboration between development and operations teams to ensure smooth and efficient software releases. This involves adopting DevOps practices that promote shared responsibility and continuous feedback, enabling both teams to work together more effectively. Implementing a CI/CD pipeline requires close collaboration between development and operations teams.

the figure 1-3 show the company logo.

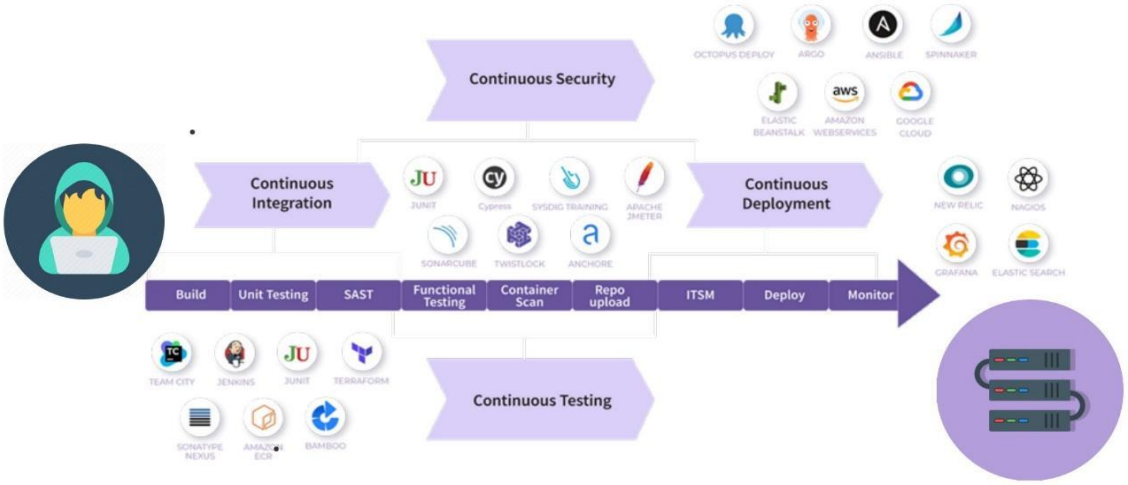


Figure 1-3: pipeline features

1.7 Project Methodology

In this section, we present the methodology chosen for this project

1.7.1 Agile Methodology

To achieve our goals, we will adopt Agile methodology, which is a term used for many incremental and iterative software development approaches such as Scrum (1995), Kanban, and Extreme Programming (1996). These methods are processes that support the Agile philosophy, emphasizing incremental delivery, team collaboration, face to face communication, constant planning and constant learning. In Agile development approach, applications should be developed with an incremental and iterative method that shortens its life cycle by making smaller increments (called sprints) which generally lasts from 1 to 4 weeks. During a sprint, the team has to be able to develop, test, and validate the software created. In contrast to the waterfall approach, in the agile methods the testing is done altogether with the programming and the team should release new application versions in these defined sprints. The way to achieve it is through automation The Following steps outline our Agile methodology:

Figure 1-4 : Agile Keynote [1]

1.7.2 Scrum

To properly manage the production of applications and guarantee the quality of the business and following the search for an adequate and suitable method, we have favored that the DOCIC uses the Scrum method which is the most proven and the most documented. Indeed, the Scrum approach follows the principles of the Agile methodology, in other words the involvement and active participation of the client throughout the project.

1.7.3 Scrum Roles

This approach is divided into several self-organized and multidisciplinary teams which are:

- Product Owner

The product owner plays a crucial role in the Scrum workflow, guiding agile team discussions about product backlog items and features. Additionally, product owners oversee quality assurance to ensure deliverables meet standards.

- Scrum Master

The Scrum Master adheres to the principles in the agile manifesto to support sprint planning. Scrum Masters guide development teams through agile methods to add value for stakeholders.

- Software Development Team

Development teams are skilled and cross-functional, typically comprising designers, developers, testers, and others to avoid the need for external assistance in agile software development environments. Figure 1-5 shows the scrum team struture

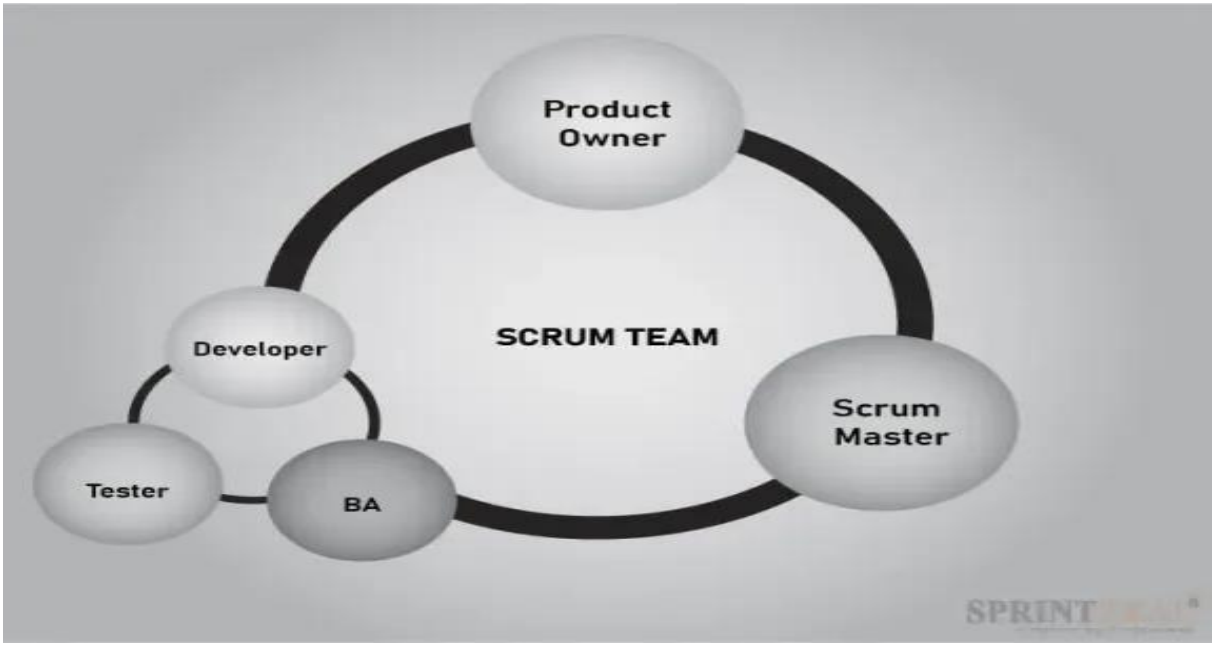


Figure 1-5:Scrum Team structure [2]

Scrum workflow steps can be used or customized into:

- Backlog Development: A product roadmap guides the creation of user stories and product requirements, forming the sprint backlog. Teams propose features or user stories to deliver, with product owners deciding which features to include in the backlog.
- Backlog Release: Collaboration between product owners and teams determines which user stories will be included in each backlog release. Each backlog release comprises a set of activities that eventually form a sprint release
- Sprint Work: Team members complete a set of backlog tasks within predetermined timeframes (usually 14-28 days) during a sprint. The agile team builds product features from a specific sprint backlog during this period.
- Daily Stand-Ups: Agile teams use daily standup meetings to track their workflow towards meeting sprint goals. These meetings, typically lasting no more than 15 minutes, help teams discuss solutions to daily work issues.
- Sprint Retrospective and Follow-Up Planning: The final phase of the Jira workflow is the sprint retrospective, a post-mortem of the previous workflow. Agile teams

evaluate what went well, what didn't, and what changes to implement in the next sprint. These retrospectives focus on delivering better value through continuous improvement.

The figure 1-6 sum up the whole workflow and roles in scrum methodology

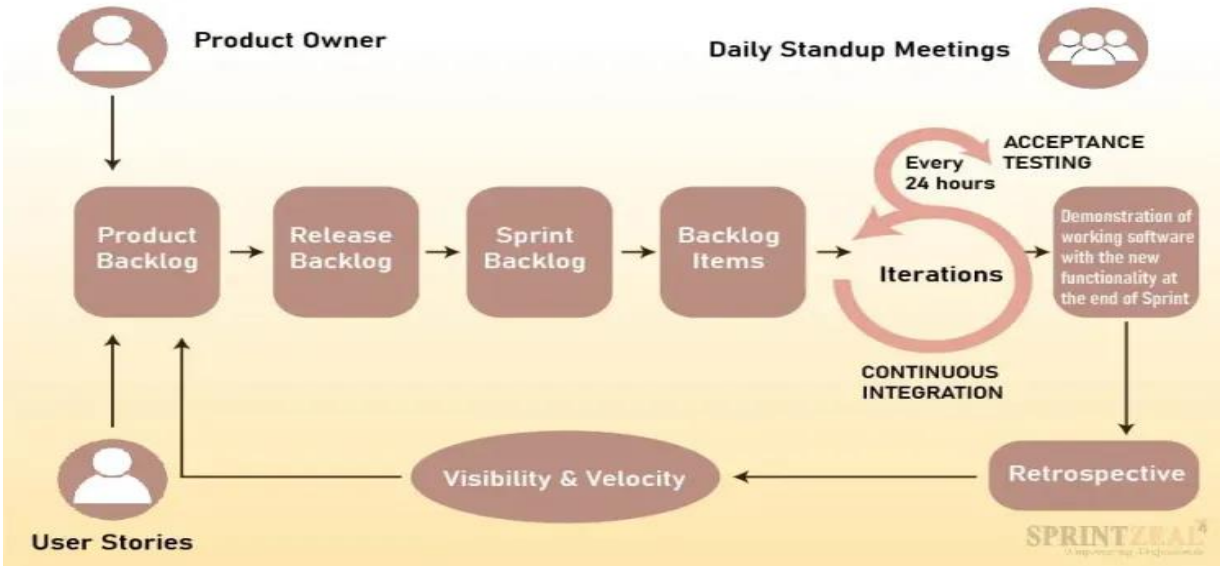


Figure 1-6 : Scrum Workflow [2]

1.7.4 Devops

Our selected approach involves adopting DevOps practices through GitLab CI/CD pipelines. This approach is done with:

- **Version Control**

Version control plays an important role in DevOps, From enabling efficient collaboration among development teams, to allowing multiple developers to work concurrently on the same codebase, tracking all changes with details on what have been modified, and for what purpose, not mentioning that version control systems support branching and merging, this would allow developers to work independently on separate flux of changes for features or future releases.

- Continuous Integration

Daily, all the developers integrate their code in a shared repository. They divide the work into small parts. which helps to manage the small code, identify bugs, and detect potential merge conflicts.

- Continuous Delivery

Since the code will be constantly integrated, it will be consistently delivered to the customer. Furthermore, through the smaller contribution, we can release the software update promptly. This gives better customer satisfaction.

- Continuous Deployment

DevOps automation enhances the speed of development. Plus, the continuous deployment will automate the release of minor updates that don't represent a substantial threat to the present architecture.

- Continuous Testing

DevOps focuses on continuous testing in every step of development. As a result, we will have valuable feedback with automated testing.

- Continuous Operations

DevOps engineers are always doing their best for the better modification and updating of software with a new release.

- Collaboration

This mythology is known for its collaboration and feedback sharing. Thus, Devs and Ops need to communicate and share feedback to streamline the efficiency of DevOps.

Figure 1-7 shows the Devops practices.

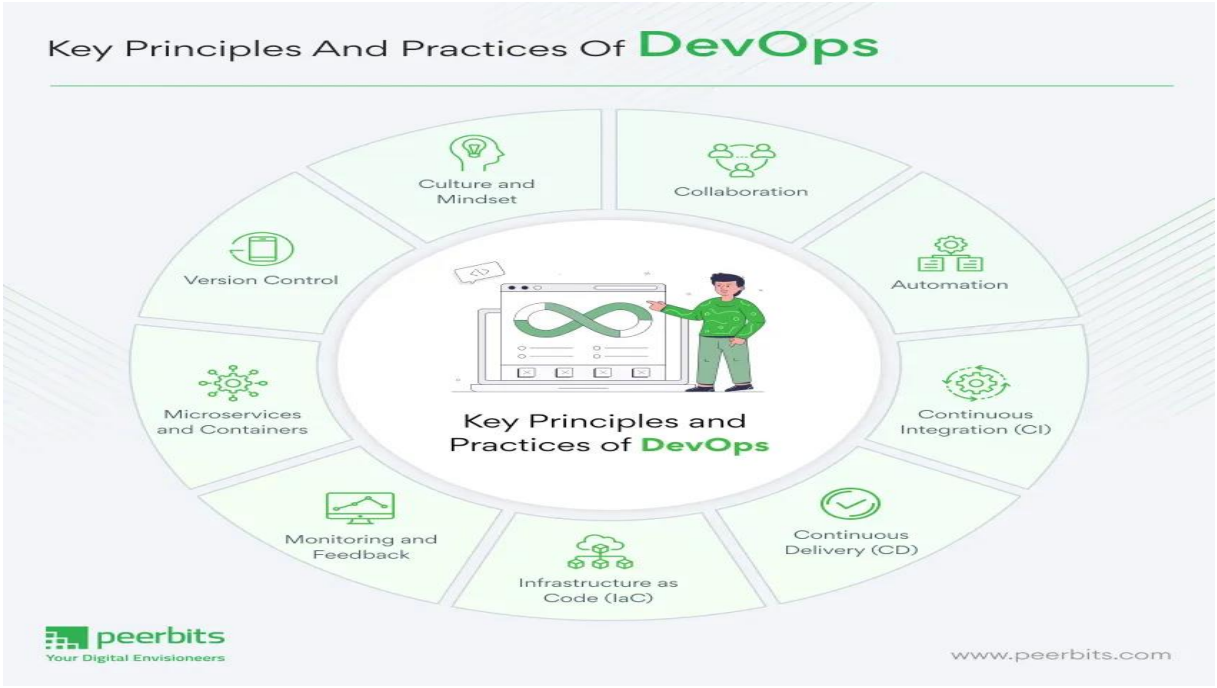


Figure 1-7 : Devops practices [3]

1.7.5 Devops vs scrum

The table below provides a comprehensive comparison between DevOps and Scrum, highlighting their core aspects and differences. While Scrum emphasizes iterative development and teamwork within an Agile framework. DevOps focuses on integrating development and operations to achieve continuous delivery and automation. By understanding these distinctions, teams can better implement practices that suit their specific project needs and organizational goals. The table 1-1 explain the Devops vs scrum methodology over certain critters

Table 1-1: Devops vs Scrum

Aspect	**DevOps**	**Scrum**
Focus	Integration of development and operations	Iterative development and teamwork
Goal	Continuous delivery and deployment	Incremental delivery of project requirements
Methodology	Emphasizes automation and collaboration	Follows Agile principles with defined roles
Practices	CI/CD, automated testing, infrastructure as code	Sprints, daily stand-ups, retrospectives
Challenges	Requires high skill levels and tool integration	Cultural resistance and maintaining discipline

1.8 Project terminologies

In this section, we will be describing different project terminologies

1.8.1 Continuous Integration

Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day [4]. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible [5]. Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly. [6]

The continuous integration workflow begins with a developer creating a new branch from the main codebase to work on changes. Once the modifications are complete, automated tests are run on the branch to verify the new code does not break existing functionality. If the tests pass, the system checks for any conflicts with the current main codebase that may have arisen from concurrent changes. With no conflicts or after resolving them, the branch can be merged into the main codebase, integrating the latest changes. This automated and iterative process of frequent code integration, testing, conflict resolution, and merging ensures early detection of errors, maintains code quality, and facilitates collaboration among the development team, ultimately enabling more efficient and reliable software deliveries. Figure 1-8 shows the ci steps.

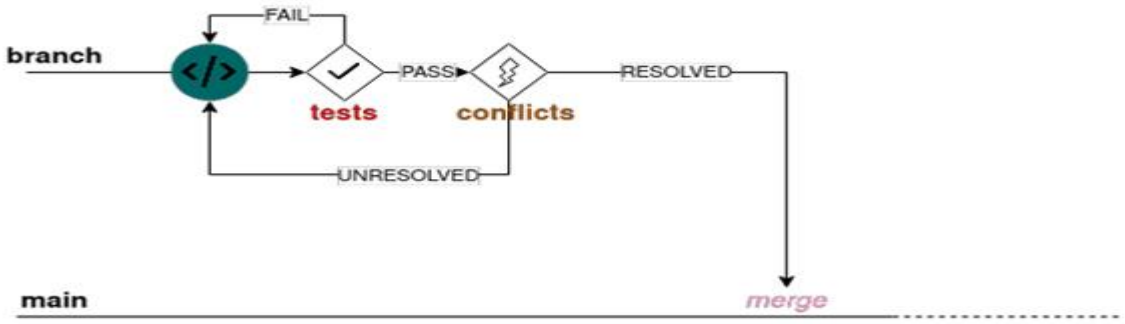


Figure 1-8: CI steps

When implementing these steps we guarantee:

- Increased Efficiency: CI/CD pipelines streamline the software development lifecycle, from development to deployment. This efficiency is achieved through automation, which minimizes the time and effort required to deliver software to end-users. It also facilitates faster iterations, allowing teams to respond quickly to feedback and make necessary adjustments.
- Enhanced Collaboration: CI encourages regular code integration, which fosters better collaboration among team members. This practice ensures that everyone is working with the most up-to-date codebase, reducing the chances of merging conflicts and facilitating smoother teamwork.

1.8.2 Continuous Deployment

Continuous Deployment is a software delivery model that relies on automated, reliable processes and tools. It's all about delivering value to customers faster by continuously improving the quality of software through automation and process improvements.

It means that developers can release changes without waiting for code reviews, which makes it easier to get feedback from customers often and early.

Since each change is tested thoroughly using an automated testing pipeline before being released, there are fewer bugs when the change gets deployed into production which means less risk of introducing problems during deployment or downtime. Because these deployments happen frequently, they become a vital part of an ongoing development cycle. Also needless to say that this workflow will provide:

• Improved Speed of Development: Since Development does not pause for deployment and release, the coding process is much faster.

Figure 1-9 shows the diagram of the CD cycle, with phase description.

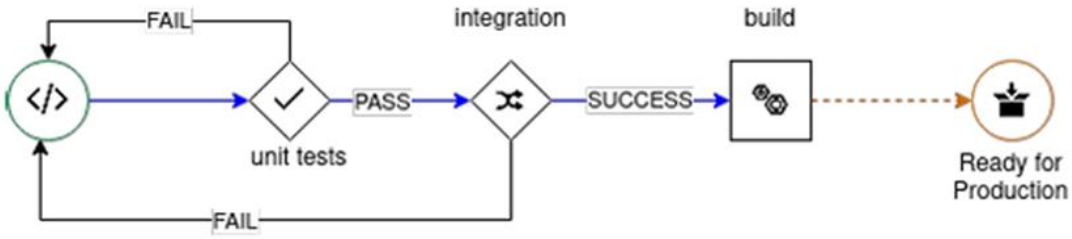


Figure 1-9 : CD steps

1.8.3 CI/CD pipeline

The CI/CD pipeline (In software engineering) is a set of processes used to manage and deploy software. If carried out properly, it decreases guided mistakes and enhances the comment loops during the SDLC (software dev lifecycle), permitting devs to supply smaller chunks of releases in a shorter time. For the implementation of the pipeline, a script needs to be developed and deployed, containing all the required details about the run time environment and actions these methods provide:

- Quality Code: One technical gain of non-stop integration is that it makes it possible to combine small portions of code at one time.

• Simplify Maintenance and Updates With the option of zero downtime updates, the process of software upgrading gets reduced by a huge amount, benefitting both user and customer in terms of time and money.

1.8.4 CI/CD monitoring

The CI/CD pipeline is distinct from the software environment that hosts the application, but it's nonetheless linked inextricably to it. A healthy pipeline is one that

allows the team to write, build, test, and deploy code and configuration changes into the production environment on a continuous basis.

An unhealthy CI/CD pipeline can hamper our ability to achieve optimal application performance in a variety of ways. For example:

- Ensuring System Availability:

Monitoring is pivotal in upholding system availability and safeguarding against disruptions that could inconvenience users.

- Spotting Performance Concerns:

Through vigilant monitoring, performance glitches and bottlenecks can be identified promptly, allowing for timely improvements.

- Alerts and Automated Responses:

Monitoring facilitates the setup of alerts triggered by predefined conditions, enabling automated responses and swift issue resolution.

Conclusion

In this chapter, we focused on the theoretical part of the solution as well as a definition of the host company Docic

Chapter 2: Project Planning and Architecture

Introduction

In this chapter we will discover different project requirements (functunal and non-function),as well as actors identification and global use case diagram.

2.1 Project requirements

As part of the crucial requirements analysis phase, we will identify the actors involved in our system and outline the functional and non-functional needs related to our project. We then identify the actors involved in our system and conclude with Global use case diagrams represented using UML, accompanied by a detailed textual description of the experimentation platform

2.1.1 Actors Identification

An actor represents the abstraction of a role played by external entities, user, hardware device or other system, which interact directly with the system studied. Then, an actor can directly consult or modify the state of the system, by sending and receiving messages that may carry data. Our project revolves around two main actors and four secondary actors, the list describes first the main then the secondary actors

- DevOps engineer: The DevOps engineer is responsible for setting up an adequate environment to ensure the smooth running of an IT project. He often communicates with the development team to specify and review their requirements.

We have also three secondary actors

- GitLab centralizes version control, CI/CD automation, and collaboration in this DevOps system. It manages and monitors pipelines for efficient software delivery while integrating security features like code scanning and secure runners. This ensures fast, high-quality, and secure software development and deployment.
- HCP Vault (HashiCorp Vault): Vault is managing secrets like API keys and credentials. It offers encrypted storage, strict access controls, and auditing, ensuring sensitive data is protected throughout its lifecycle. Vault's secure secret management is crucial for system-wide security, from user authentication to secure CI/CD pipelines.
- Ansible: Ansible automates server configuration and management, eliminating manual errors and inconsistencies. It enables infrastructure-as-code, ensuring

servers are provisioned and configured consistently and securely. This automation supports reliable pipeline management and CI/CD security, allowing teams to focus on strategic tasks rather than repetitive configurations.

- OVH (OVHcloud): OVH provides the foundational cloud infrastructure for the entire system. It hosts the server components like Vault for secret storage, supports GitLab's pipeline execution and monitoring, and will be managed by Ansible.

2.1.2 Product Backlog

The backlog is a list of prioritized tasks that define the characteristics of the solution. It is one of the basic elements of the agile method. It is the main working tool of the Product owner, responsible for collecting the requirements of the stakeholders and converting them into a list of features to be developed by the team. We then divided the work into three sprints. The priority is considered according to the importance of the task dedicated to accomplishing the final product. For each task, we estimated an initial workload measured in terms of days.

The product backlog, detailed in the Table, contains the following information:

- Task: this is a simple and clear description of the interest of a task as seen by the user. It generally follows the following syntax: "As <who>, I want <what>, in order to <why>"
- Sprint Duration: The estimate represents the amount of work required to develop the item. It is determined by the team itself and considers the development time and the complexity of the work.

Table 2-1 shows the product backlog.

Table 2-1: Product Backlog

ID	User Story	Description	priority
1	Set up and maintain pipeline	As a DevOps engineer, I can set up and maintain a CI/CD pipeline to automate the building, testing, and deployment processes	1
2	Automate Server Configuration	As a DevOps engineer, I automate server configuration without the need to interfere manually	3
3	Automate Pipeline Monitoring	As a DevOps engineer, I can automate pipeline monitoring to ensure continuous oversight and management of the CI/CD pipeline.	2
4	secure pipeline management	Ensures that sensitive information is securely managed using HCP Vault, and the integrity and authenticity of container images are maintained using HMACs, providing a robust and secure software delivery process.	4

2.2 Sprint planification

In the SCRUM method, the sprint planning meeting is an essential step in the project. In this meeting, the team will choose the necessary duration for each sprint considering the complexity of the latter and considering the objective of the project. In our case, we decided to divide the sprints by themes, as shown in the following image. Table 2-2 shows the sprint planification

Table 2-2:Sprint planification

sprint	User Story	Priority
1	Set up and maintain pipeline	1
2	Automate Server Configuration	3
2	Automate Pipeline Monitoring	2
3	secure pipeline management	3

2.3 Functional Requirements

For the proper functioning of our platform, it is necessary to concretely define

the functionalities that will be implemented within our IS to make them more appropriate to the needs of the company. To do this, we will assume that our platform must be able to provide the following functionalities:

- Authenticate

The CI/CD system must implement robust authentication mechanisms to ensure that only authorized users can access and interact with the pipeline. This includes integrating with existing authentication systems like LDAP or OAuth to provide seamless and secure user authentication. By enforcing strict access controls, the system protects sensitive operations and data from unauthorized access, thereby maintaining the integrity and security of the entire CI/CD process.

- Manage Secret Store

Secret management is crucial for the security of the CI/CD pipeline. Integrating HCP Vault allows for secure storage and management of sensitive information such as API keys, passwords, and certificates. Vault ensures that secrets are encrypted at rest and in transit, providing secure access through fine-grained access controls. This integration simplifies secret management by automating the injection of secrets into

the pipeline as needed, thus eliminating the risks associated with hardcoding secrets in the codebase or configuration files.

- Manage Pipeline

Effective pipeline management is key to a successful CI/CD implementation. This involves automating the build and deployment of Docker containers, configuring NGINX settings, and managing DNS records. The pipeline should be defined in a .gitlab-ci.yml file, detailing each stage from code commit to production deployment. Automation tools like GitLab CI runners handle the building and deploying of containers, while Ansible scripts can manage NGINX configurations and auto-create DNS entries. This comprehensive management ensures that each deployment is consistent, reliable, and efficient, reducing manual intervention and potential errors.

- Monitor Pipeline

Monitoring is essential to maintain the health and performance of the CI/CD pipeline. By integrating monitoring tools and exporters on the server, the system can track various metrics such as build times, resource utilization, and error rates. This data provides valuable insights into the pipeline's performance, helping to identify and resolve issues promptly. Effective monitoring ensures that the pipeline runs smoothly, with minimal downtime and optimal performance, thereby supporting continuous delivery and deployment processes.

- Secure pipeline credentials

Security is a critical aspect of the CI/CD pipeline. Implementing HMAC for container integrity ensures that containers are verified for authenticity and have not been tampered with during the build and deployment process. Additionally, the pipeline must secure all communications and operations, protecting sensitive data and preventing unauthorized access. By integrating security checks and controls at every stage of the pipeline, the system can detect and mitigate potential security threats, ensuring a secure and trustworthy CI/CD environment.

- Remove Manual Server Configuration

Automating server configuration using Ansible reduces the need for manual interventions, ensuring consistency and reliability in server setups. Ansible playbooks define the desired state of the server configurations, which can be applied across multiple environments. This automation simplifies the management of server configurations, making it easier to deploy updates and maintain the infrastructure. By eliminating manual configurations, the system minimizes the risk of human errors and ensures that the servers are always configured correctly and consistently.

2.4 Non-Functional Requirements

To identify an application development framework, non-functional needs must meet the following criteria:

Scalability

The CI/CD system must be designed to scale efficiently to handle an increasing number of builds and deployments. This involves ensuring that the underlying infrastructure can accommodate growing workloads without performance degradation.

Performance

Performance is critical for a CI/CD pipeline to ensure that builds and deployments are completed in a timely manner.
This can be achieved through efficient resource allocation, caching mechanisms, and parallel processing of pipeline stages.
Monitoring tools should also be optimized to provide real-time insights without adding significant overhead, ensuring that the pipeline runs smoothly and efficiently.

Reliability

the system should have mechanisms in place to handle failures gracefully, with automatic retries and failover strategies to minimize downtime.
Regular backups and disaster recovery plans are essential to restore the system quickly in the event of a failure.

Maintainability

The CI/CD system should be easy to maintain, with clear documentation and automated processes for routine tasks.

This includes well-documented pipeline configurations, automated testing and validation scripts, and streamlined processes for updating and patching the system.
By prioritizing maintainability, the system reduces the effort and complexity involved in managing the CI/CD pipeline, enabling smoother operations and quicker troubleshooting

2.5 Global Use Case Diagram

Figure 2-1 diagram exposing the global Use Case Diagram.

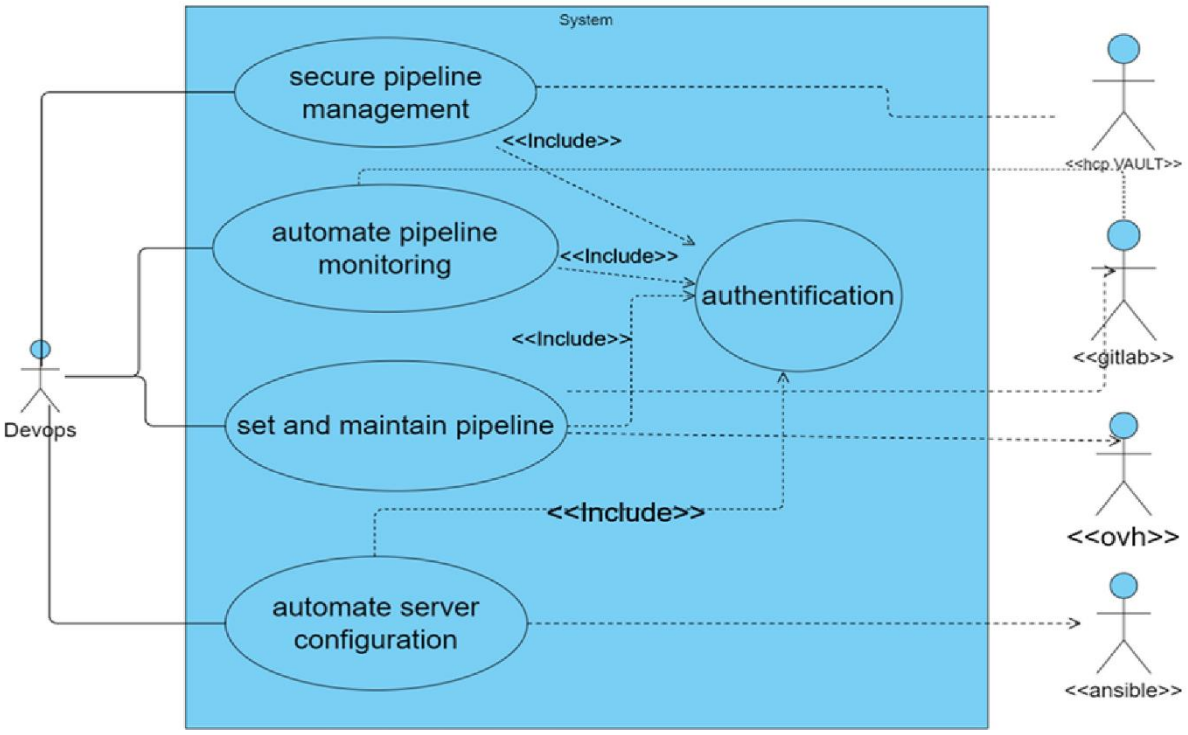


Figure 2-1 : Global use Case Diagram

- Authentication

Authentication is required for accessing the CI/CD system, ensuring that only authorized users can interact with the pipeline. This is the first step in maintaining security and integrity within the development and deployment processes.

- Secure pipeline management

Secret store management involves integrating HCP Vault to securely manage sensitive data, such as API keys and passwords, within the pipeline. This ensures that secrets are stored and accessed securely, reducing the risk of exposure. implementing measures such as HMAC for container integrity, ensuring that the containers have not been tampered with. It also involves securing communication and operations within the pipeline to maintain a high level of security throughout the process.

- Set up and maintain pipeline

Pipeline management encompasses various tasks essential for building, deploying, and configuring applications. This includes:

- ✓ Building Containers: Automating the creation of Docker containers triggered by code commits.
- ✓ Deploying Containers: Ensuring the deployment of built containers to specified environments.
- ✓ NGINX Configuration: Managing NGINX settings automatically to ensure consistency and optimization.
- ✓ AutoDNS Creation: Automating the creation of DNS records for new deployments, ensuring seamless accessibility.

- Automate Pipeline Monitoring

Pipeline monitoring involves adding exporters to the server to track and monitor various aspects of the pipeline and infrastructure. This includes integrating tools to observe performance, health, and overall operation of the CI/CD pipeline.

- Automate Server Configuration

Removing manual server configuration is achieved by using Ansible to automate these configurations. This reduces the need for manual intervention, ensures consistency, and simplifies the maintenance of server configurations through predefined playbooks.

2.6 Project architecture

The architecture is divided into two main sections:

We have nginx component acts as a reverse proxy or load balancer, routing incoming requests to the appropriate backend services. It interacts with two applications, which handle backend and frontend respectively. Also for data storage MongoDB instance was created. Below that, the front-end component serves as a standalone application that reads data.The next layer consists of three React containers: These containers represent different user interfaces each responsible for specific functionality such as reading, writing, or administering data.These React containers interact with a shared Node component, which is a data orchestration engine and a service responsible for coordinating data operations.This architecture leverages containerization to achieve scalability, and separation of concerns. Different components are isolated into separate containers, allowing for independent deployment, scaling, and maintenance. The figure 2-2 illustrates more details about the architecture.

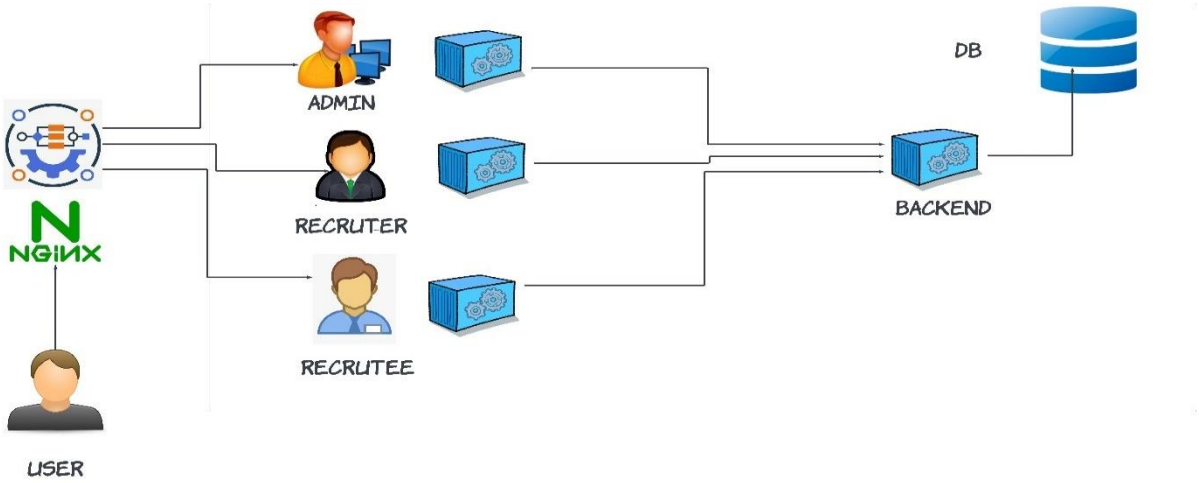


Figure 2-2: project architecture

2.7 Pipeline design

The pipeline workflow has various steps involved in building, testing, and deploying a project. Let us walk through an explanation of each step:

• Build: This step involves building the project into container images, which are self-contained executable packages that include the application code, runtime, system tools, and libraries necessary to run the application.

• Push images to an image repository: After building the container images, this step pushes them to an image repository, which is a centralized location for storing and managing container images.

• Code integration: This step involves verifying the Nginx syntax, which is a popular open-source web server used for reverse proxying, load balancing, and serving static content.

• Reverse Proxy Config/Create domain names: This step is responsible for configuring the Reverse Proxy and creating domain names.

Creating domain names allows for easier access and identification of the application or service.

• Add main config for static branches: This step adds the main configuration for static branches, which are typically used for hosting stable, long-lived versions of the application or service.

• Create dynamic domain for feature branches: This step creates dynamic domains for feature branches, which are temporary branches used for developing and testing new features before merging them into the main branch.

• deploy the images: This final step runs the container images, effectively deploying and starting the application or service in the target environment Image the figure 2-3 more about the interaction between these steps

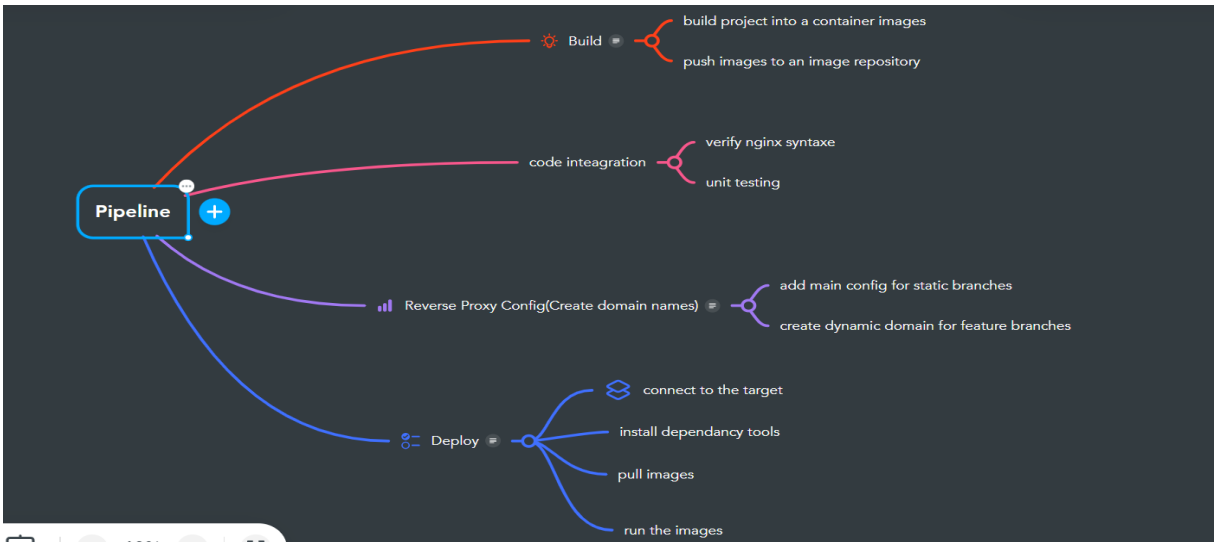


Figure 2-3:Pipeline steps design

2.8 Building the experimentation platform

In this section, we will be describing different tools utilized in our project

2.8.1 Hardware platform

To support our application's robust functionality, we've deployed it on a server equipped with ample hardware resources. This includes a spacious storage capacity of 132 GB, ensuring ample room for data storage and processing. Additionally, we've allocated 8 GB of RAM to facilitate smooth and efficient operation, enabling seamless multitasking and handling of concurrent user requests. Furthermore, the server boasts a powerful processing capability, with a 4-vCPU configuration enhancing computational speed and responsiveness. This comprehensive hardware setup ensures optimal performance and reliability for our application, allowing it to effectively meet the demands of our users

2.8.2 Software platform

Complementing our hardware infrastructure, our software configuration is designed to maximize efficiency and performance, integrating with our robust hardware setup

- Gitlab

GitLab is a widely used open-source platform that leverages Git, the distributed version control system created by Linus Torvalds. It offers an extensive array of tools for managing Git repositories, project planning, continuous integration, and deployment (CI/CD), code reviews, issue tracking, and more

Designed as a comprehensive DevOps platform, GitLab facilitates concurrent collaboration across Product, Development, QA, Security, and Operations teams within a unified application.

- Hashicorp Vault

HashiCorp Vault is designed to help organizations manage access to secrets and transmit them safely within an organization. Secrets are defined as any form of sensitive credentials that need to be tightly controlled and monitored and can be used to unlock sensitive information. Secrets could be in the form of passwords, API keys, SSH keys, RSA tokens, or OTP. [7]

HashiCorp Vault makes it very easy to control and manage access by providing us with a unilateral interface to manage every secret in our infrastructure. Not only that, but we can also create detailed audit logs and keep track of who accessed what.The figure 2-4 shows vault main functionality

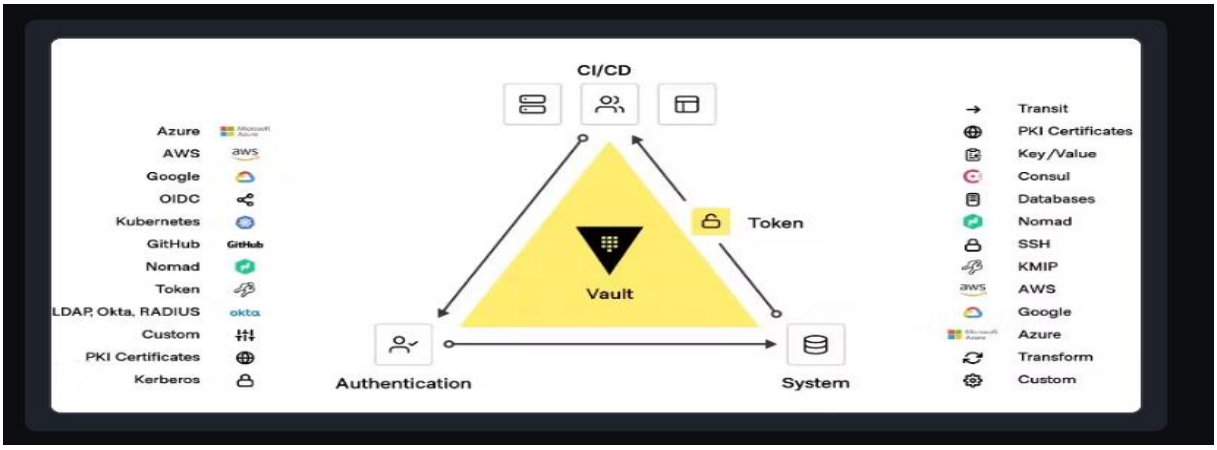


Figure 2-4 : HCP Vault LOGO [7]

- OvhCloud

As Europe's leading cloud provider, we deliver public and private cloud products, shared hosting and dedicated server solutions in 140 countries worldwide. We also offer domain name registration, telephony services and internet access to our customers [8]. Founded in 1999, OVHcloud is a French company with an international presence, based on a backbone of datacentres and points of presence spread across the globe.

- Ansible

Ansible is an open-source IT automation platform from Red Hat. It enables organizations to automate many IT processes usually performed manually, including provisioning, configuration management, application deployment and orchestration.

The Red Hat Ansible Automation Platform is a Red Hat-supported platform built around Ansible Core. It provides an Agentless framework to create, test and manage automation content. Written in the Python programming language, this command-line IT automation software application provides organizations with a flexible, secure way to automate many IT tasks and complex processes. [9]

Conclusion

In this chapter, we have defined the different functional and non-functional needs of our solution and presented the different technical and physical architectures of the pipeline, which we have illustrated in the form of use Case diagrams in order to better schematize its dynamic aspect.

Chapter 3: leveraging pipeline

Introduction

This chapter will explore the step-by-step process of constructing a continuous integration and continuous deployment (CI/CD) pipeline, dramatically transforming our workflow efficiency. It will detail the tools, techniques employed to automate code integration, into the solution across environments.

3.1 Containerization with docker

Containerization is a type of application-level virtualization that allows the creation of multiple isolated user space instances, called containers, on the same core. A container system stores and isolates objects before transporting or deploying them in an extended operating environment, including applications, software, and libraries. It also facilitates the safe and easy transport of application code from development to production, eliminating the need for hypervisors and guest operating systems on virtual machines. figure 3-1 shows the docker defintion.

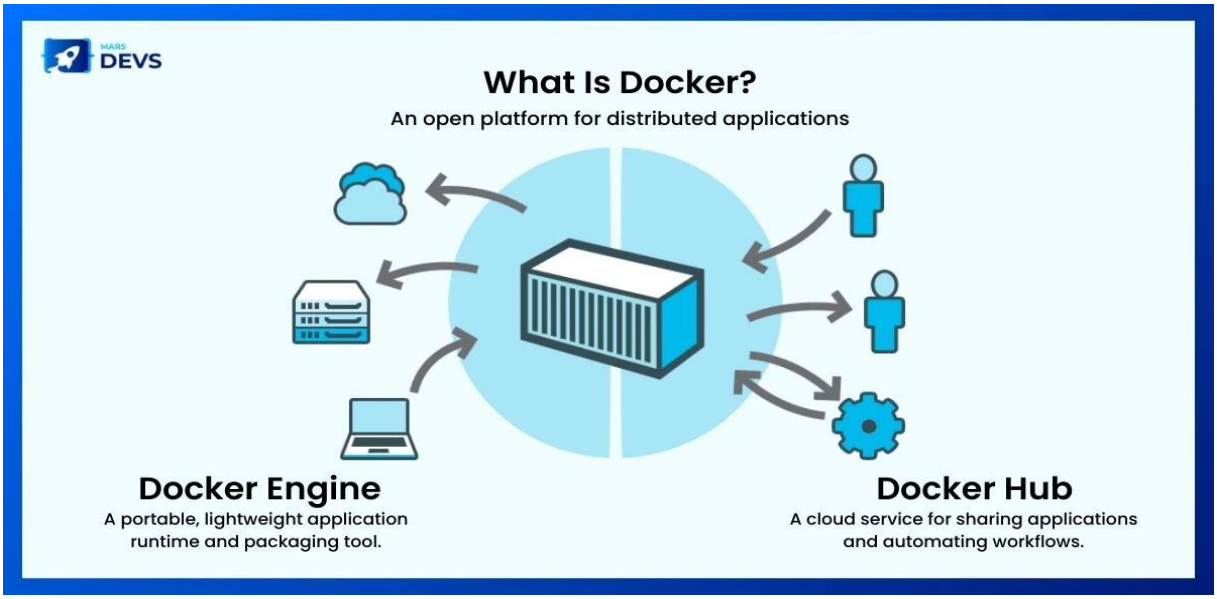


Figure 3-1 : Docker interaction [10]

These containers bundle the application's code along with all its dependencies - like libraries and system settings - into a single, portable unit. This approach streamlines the workflow by ensuring consistency across different environments, from a developer's machine to testing and ultimately production deployment.

The Docker workflow consists of two main stages: building and running. First, a Dockerfile is created. This file contains instructions for building a Docker image, which is a self-contained archive that includes the application code, its runtime environment, and any system libraries or settings needed to run the application.

Dockerfiles contain a set of instructions that specify the operating system base for the container, the installation of any necessary dependencies, the copying of our application code, and the configuration of the environment to run our application host system. The figure bellow illustrates more details about how docker container are created

3.2 Sprint Backlog

It is interesting to automate the build, test, and deployment processes for our applications, ensuring faster and more reliable releases. The Table 3.1 describes the sprint 1 Backlog.

Table 3-1 : Sprint 1 backlog

ID	User story	Technical Story
2	As a DevOps engineer, I want to set up and maintain a CI/CD pipeline.	Configure GitLab runner for automating CI/CD tasks
		Build application containers using Docker for consistency across environments
		Deploy containers on the target server for production or staging environments
		Create and configure Nginx config files for web server setup and routing
		Create automation scripts for ovhCloud to manage cloud infrastructure as code

3.3 Sprint analysis

In this section, we begin by a description of the use case diagram of Sprint 1 and give, then, a textual description of this use case.

3.3.1 Use case diagram

We isolated the "Set Up and Maintain CI/CD Pipeline" for further expanding, the bellowed figure 3-2 explains how it is going to be devised

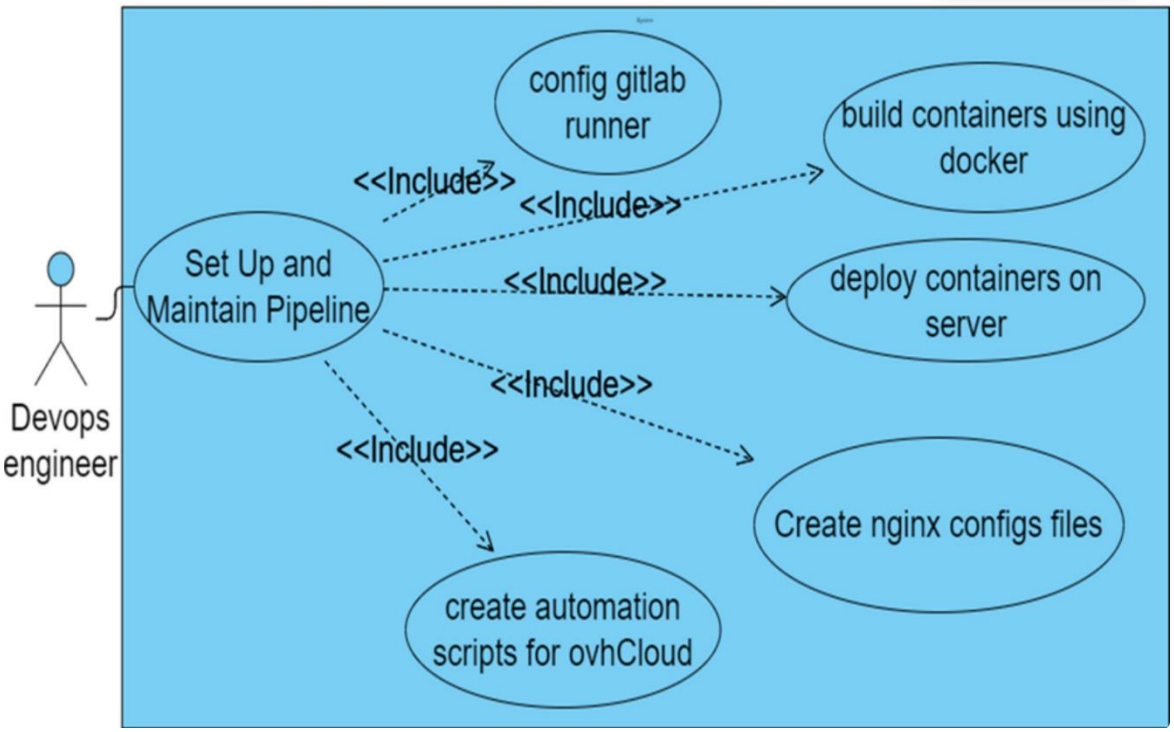


Figure 3-2 :set up and maintain pipeline use Case

3.3.2 Use case textual explanation

The table 3.2 contain a full textual description of the previously discussed

Table 3-2 : Sprint 1 use case explanation

Case	Set Up and Maintain CI/CD Pipeline
Description	This use case describes the process of setting up and maintaining a CI/CD pipeline by configuring GitLab runner, building containers using Docker, deploying containers on a server, creating Nginx configuration files, and automating scripts for OVHCloud.
Primary Actor	Devops engineer
Preconditions	• GitLab runner is available and can be configured. • Docker is installed and accessible for building containers. • Server environment is ready for deploying containers. • OVHCloud environment is accessible for automation • Nginx is available for configuration.
Main Success Scenario	• DevOps engineer sets up and maintains the CI/CD pipeline • Configures GitLab runner for pipeline execution. • Builds containers using Docker with necessary configurations. • Deploys the built containers on the server. • Develops automation scripts for OVHCloud to streamline deployment and management.
alternatives	• If GitLab runner configuration fails, the engineer logs the error and revisits the configuration steps. • If container deployment fails, the engineer checks the server environment and resolves deployment issues. • If creating Nginx config files fails, the engineer troubleshoots and corrects the configurations. • If automation script creation for OVHCloud fails, the engineer debugs and corrects the scripts.
Post-conditions	• CI/CD pipeline is properly set up and maintained. GitLab runner is configured correctly. • Docker containers are built and deployed successfully. • Nginx configuration files are created and properly set up. • Automation scripts for OVHCloud are created and functional

3.4 Sprint implementations

In this section, we begin by arranging the step of pipeline implementation

3.4.1 How does Gitlab run pipeline

GitLab CI/CD is defined using the YAML format file called "**gitlab-ci.yml**". A job within a pipeline is to be defined using a predefined structure, as declared in Code 1. In this example the name of the job is provided, after which the stage of the job is defined. Script section defines the commands to be ran on this step. The figure can also be visualized to allow for a more complete understanding of the current state. Figure 3-3 shows a job example.

```
# Define a job with a literal script example
example_job:
  script:
    - echo "This is a literal example script."
  stage: test  # Assign the job to the test stage
```

Figure 3-3 : job example

➢ Adding custom runner for the project

GitLab Runners are key to GitLab's CI/CD capabilities, executing jobs in a project's pipeline like building, testing, and deploying code. They can be hosted as shared Runners by GitLab.com, dedicated Runners by organizations, or group-level Runners shared among projects. This flexibility allows jobs to run on various platforms and environments, ensuring consistent application development. Features like parallel job execution, caching, and secure execution streamline and automate the software development lifecycle, validating and deploying code changes reliably the figure illustrate how Gitlab instance orchestrate jobs loads. figure 3-4 shows the Gitlab runner architecture.

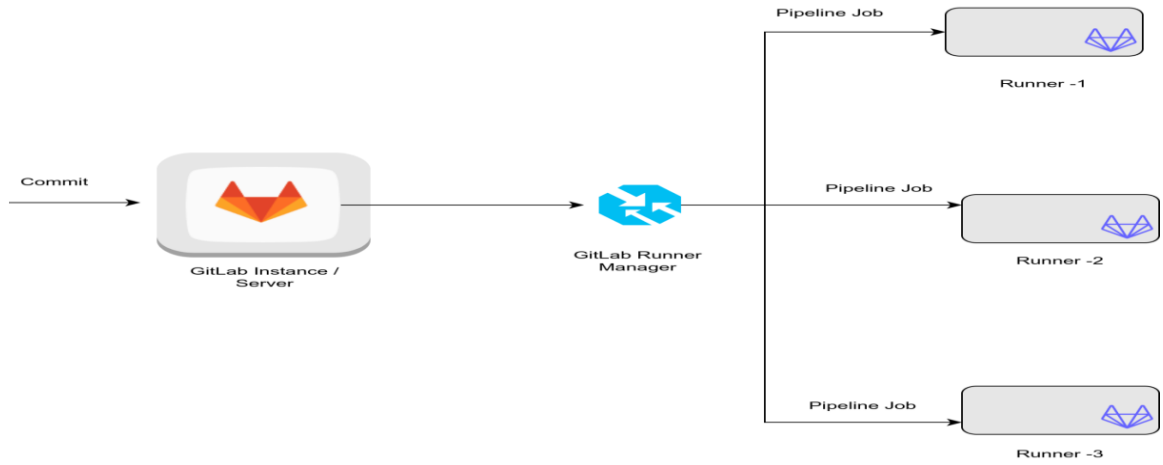


Figure 3-4: how Gitlab organize jobs [11]

3.4.2 Add runner for the project

Before going into the steps lets us install Gitlab Runner binaries in the server, figure 3-5 shows the steps to add it:

Figure 3-5: pre-configuration steps

Once the GitLab Runner is installed, the next step is to register the Runner with our GitLab instance. This registration process is crucial as it establishes a secure connection between the Runner and our GitLab project or organization. During the registration, we'll need to provide certain details, such as the GitLab instance URL and a registration token, which we can obtain from our GitLab project or admin settings.

figure 3-6 explain the runner creation.

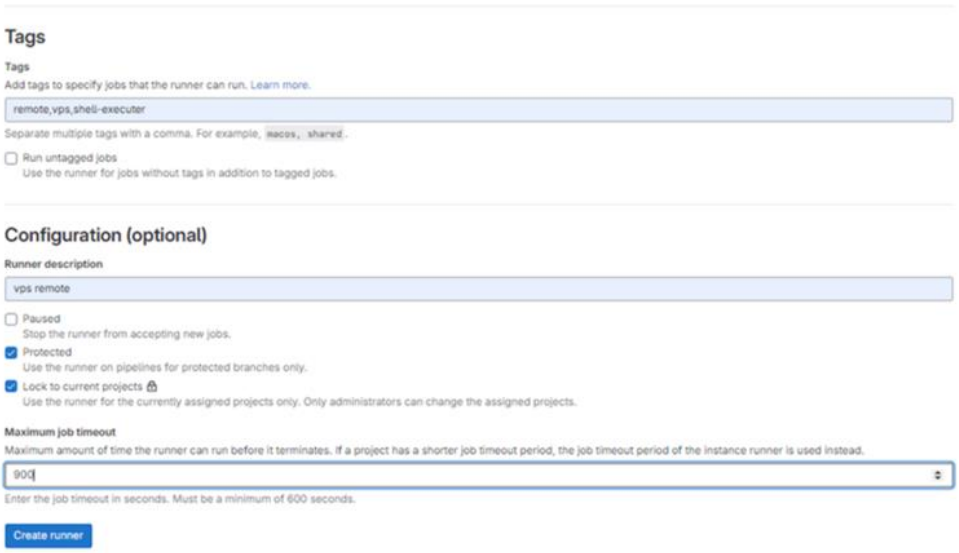


Figure 3-6: creating Gitlab Runner

we can initiate the Runner registration process. Execute the command-line utility or script provided by GitLab, which will guide us through the registration steps. As shown in the figure We 'll need to enter the GitLab instance URL, registration token, and any additional configuration options. figure 3-7 shows the runner registration.

```
ubuntu@vps-b77b9b50:~$ gitlab-runner register  --url https://gitlab.com  --token glrt-ynsczRm8_bYaGjkeHtdV
Runtime platform                                    arch=amd64 os=linux pid=2149467 revision=91a27b2a version=16.11.0
WARNING: Running in user-mode.
WARNING: The user-mode requires you to manually start builds processing:
WARNING: $ gitlab-runner run
WARNING: Use sudo for system-mode:
WARNING: $ sudo gitlab-runner...

Created missing unique system ID                    system_id=s_f7250ca8d8b2
Enter the GitLab instance URL (for example, https://gitlab.com/):
[https://gitlab.com]:
Verifying runner... is valid                        runner=ynsczRm8_
Enter a name for the runner. This is stored only in the local config.toml file:
[vps-b77b9b50]: vps-project-1
Enter an executor: ssh, parallels, docker, docker+machine, docker-autoscaler, custom, shell, virtualbox, docker-windows, kubernetes, instance:
docker
Enter the default Docker image (for example, ruby:2.7):
python:3.10
Runner registered successfully. Feel free to start it, but if it's running already the config should be automatically reloaded!

Configuration (with the authentication token) was saved in "/home/ubuntu/.gitlab-runner/config.toml"
```

Figure 3-7 : Register Gitlab Runner

After successfully registering the GitLab Runner, we can start or restart the Runner service. From the figure we can ensure it's running and ready to execute jobs from our GitLab CI/CD pipelines. Depending on our setup, we may need to configure additional settings, figure 3-8 shows the runner start up.

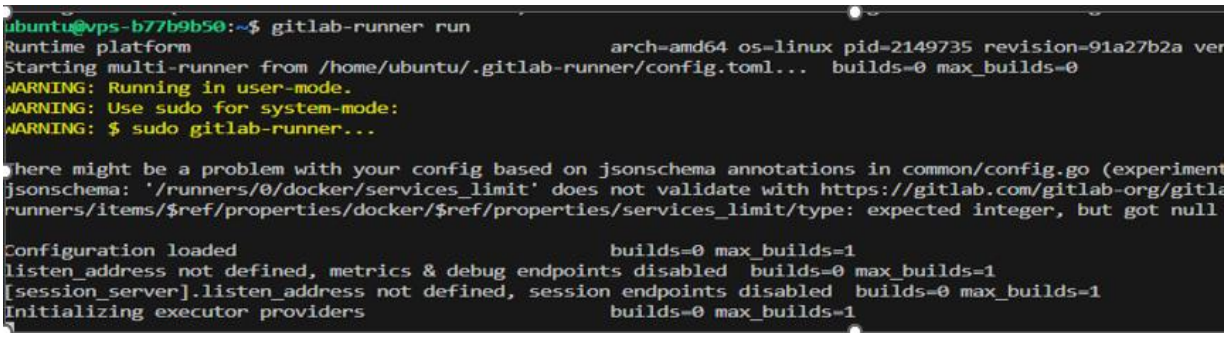


Figure 3-8 : Start-up the Runner

The GitLab Runner's configuration is managed through a file named `config.toml`, which defines settings to customize the Runner's behavior and execution environment. This file specifies the number of concurrent jobs, intervals for checking and shutting down, and the session timeout. It also includes configurations for individual Runner instances, such as the Runner's name, the GitLab instance URL, a unique identifier, and an authentication token. The execution method is set to Docker, allowing jobs to run inside Docker containers. Docker-related settings like the base image, privileges, entrypoint overwriting, and mounted volumes are also defined here. The file includes options for custom build directories, caching mechanisms, and other advanced configurations. Overall, the figure 3-9 fine-tunes the GitLab Runner's operations, ensuring seamless integration with the GitLab instance.

.gitlab-runner > config.toml

```
concurrent = 1
check_interval = 0
shutdown_timeout = 0

[session_server]
  session_timeout = 1800

[[runners]]
  name = "vps-project-1"
  url = "https://gitlab.com"
  id = 35338221
  token = "glrt-ynsczRm8_bYaGjkeHtdV"
  token_obtained_at = 2024-04-24T18:16:12Z
  token_expires_at = 0001-01-01T00:00:00Z
  executor = "docker"
  [runners.custom_build_dir]
  [runners.cache]
    MaxUploadedArchiveSize = 0
    [runners.cache.s3]
    [runners.cache.gcs]
    [runners.cache.azure]
  [runners.docker]
    tls_verify = false
    image = "python:3.10"
    privileged = false
    disable_entrypoint_overwrite = false
    oom_kill_disable = false
    disable_cache = false
    volumes = ["/cache"]
```

Figure 3-9: Runner config file

Finally, the runner is available in the Gitlab settings section as shown in the figure 3-10 and could be used according to an already attached tags which we will talk about later

Figure 3-10:Gitlab Runner state on the cloud

3.4.3 Create environmental variables

The table 3-3 serves as a valuable resource, offering a overview of essential configuration variables along with their respective data types and detailed descriptions.

Table 3-3 :Gitlab CI/CD variables

VARIABLES	**type**	**description**
SERVER_HOST_IP	Integer	Public Ip address of the production server
SERVER_HOST_PRIVATE_KEY	file	Private key of ssh connection
DOCKER_USERNAME	string	DOCKER HUB CREDENTIALS
DOCKER_PASSWORD	string	DOCKER HUB CREDENTIALS

3.4.4 Preparing dockerfiles for the project

To prepare the Docker File, we need information about dependencies of the project. The figure 3-11 shows an example of the file given by the developer project.

Figure 3-11 : requirement file

The next figure 3-12 shows the Dockerfile of previous example

```
# Create image based on the official Node 10 image from dockerhub
FROM node:14-slim as build-production

# Create a directory where our app will be placed
RUN mkdir -p /app

# Change directory so that our commands run inside this new directory
WORKDIR /app

# Copy dependency definitions
COPY package*.json /app/

# Install dependecies
RUN npm install

# Get all the code needed to run the app
COPY . /app/
# Serve the app
RUN npm run build  --verbose

# NGINX
FROM nginx:1.18

COPY  --from=build-production /app/dist/ /usr/share/nginx/html

COPY --from=build-production /app/nginx.conf /etc/nginx/conf.d/default.conf

CMD ["nginx", "-g", "daemon off;"]
```

Figure 3-12 : Dockerfile snippet

- **FROM NODE**:14 Since Docker allows us to inherit existing images, we install a node image and install it in our Docker image. Alpine is a Linux distro that serves as the OS on which we install our image.
- **COPY ./package.json /app/package.json**: Here, we copy the requirements file and its content (the generated packages and dependencies) into the app folder of the image.
- **WORKDIR /app**: We proceed to seting the working directory as /app, which will be the root directory of our application in the container
- RUN npm install: This command installs all the dependencies defined in the previous file into our application within the container
- COPY . /app: This copies every other file and its respective contents into the app folder that is the root directory of our application within the container

- Npm run build :This is the command that create the served file in the container
- CMD ["nginx",”-g”,”deamon off”]: Finally, this appends the list of parameters to the EntryPoint parameter to perform the command that runs the web server.

3.4.5 Preparing pipeline jobs

Implementations between different pipelines should follow a somewhat similar structure, in order to provide some level of comparison between the different systems, we will be going to proceed with creating “gitlab template” that are going to be serving as “a layers of abstraction” and make the jobs less specific ,For this, all the jobs will be extended from abstract job configs.

3.4.6 Continuous Integration jobs

The BUILD stage is responsible for building a Docker image and pushing it to a container registry. The configuration starts by defining the required inputs or variables, such as CI_REGISTRY_IMAGE, which refers to the base Docker image or registry path where the final built image will be pushed. Additionally, it specifies the IMAGE_TAG input, the tag to be used for the Docker image being built. The DOCKER_PASSWORD and DOCKER_USER inputs provide the credentials required to authenticate with the Docker registry for pushing the built image.

We are using the custom runner added earlier, with docker installed, **manually**, **for now.** The figure 3-13 bellow illustrates more details about the job.

```
spec:
    inputs:
      CI_REGISTRY_IMAGE:
      IMAGE_TAG:
      DOCKER_PASSWORD:
      DOCKER_USER:
---
.build:
  script:
    - export TAG=$[[ inputs.IMAGE_TAG ]]
    - export TAG=$( echo ${TAG} | sed 's/\//\-/g' )
    - export DOCKER_IMAGE_TAG=$[[ inputs.CI_REGISTRY_IMAGE ]]:${TAG}
    - echo ${DOCKER_IMAGE_TAG}
    - docker build -t ${DOCKER_IMAGE_TAG} .
    - docker login -u $[[ inputs.DOCKER_USER ]] -p $[[ inputs.DOCKER_PASSWORD ]]
    - docker push $DOCKER_IMAGE_TAG
```

Figure 3-13: abstract BUILD job

In a more general sense, the export command in a shell script is used to create and define environment variables, the **envsubst** command takes the input template file (nginx-template.conf) and substitutes placeholders with the values of corresponding environment variables. This filename is constructed to avoid conflicts and provide some level of uniqueness for each NGINX configuration file. The figure 3-14 bellow illustrates more details about the job.

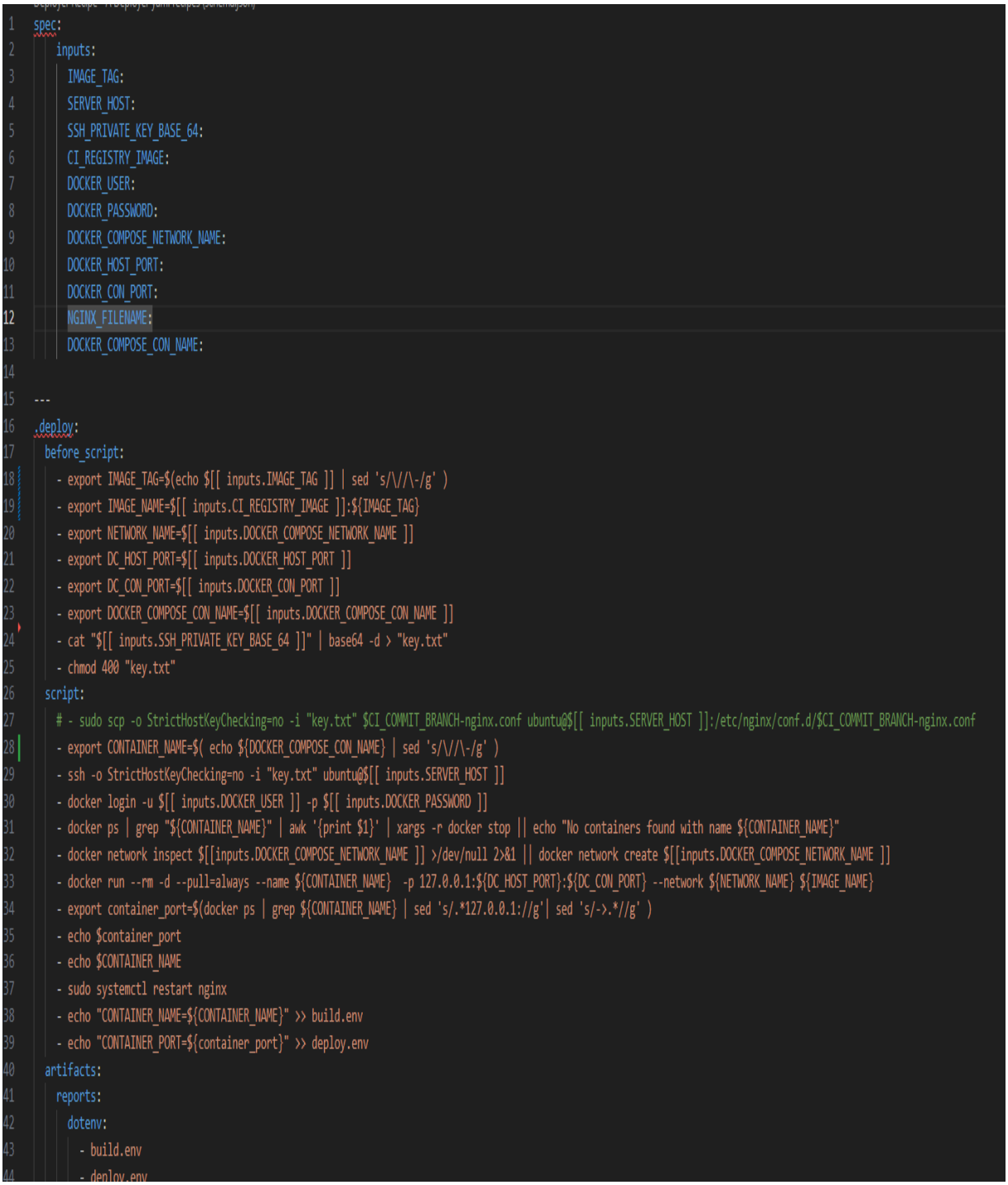
```
.create_nginx_conf_job:
  before_script:
    - export SSL_CERTIFICATE_KEY_DOMAIN_PATH=$[[ inputs.SSL_CERTIFICATE_KEY_DOMAIN_PATH ]]
    - export SSL_CERTIFICATE_DOMAIN_PATH=$[[ inputs.SSL_CERTIFICATE_DOMAIN_PATH ]]
    - export SERVER_DOMAIN_NAME=$[[ inputs.SERVER_DOMAIN_NAME ]]
    - export DC_APP_PORT=${CONTAINER_PORT}
    - export DOLLAR="$"
    - export NGINX_FILENAME=$( echo $CI_COMMIT_BRANCH | sed 's/\//\-/g' )
    - envsubst < nginx-template.conf > ${NGINX_FILENAME}-nginx.conf
  script:
    - sudo cp -f ${NGINX_FILENAME}-nginx.conf /etc/nginx/conf.d/${NGINX_FILENAME}-nginx.conf
```

Figure 3-14 : abstract nginx config job

3.4.7 Continuous Deployment jobs

For the sake of simplicity, we will be representing only the deploy job. The job begins by preparing the environment for deployment. It sets up essential variables that will be used throughout the process. These variables include Docker image tag, server connection details. Additionally, it decodes the provided SSH private key. Once inside, the script takes care of the Docker-related tasks

The script proceeds to launch the Docker container housing the application. It configures the container to expose specific ports on the host machine and connects it to the Docker network established earlier. To keep track of the dynamically assigned port of the container, we retrieve this information. Finally, we restart NGINX to ensure that any configuration changes are applied correctly. The figure 3-15 bellow illustrates more details about the job.



```yaml
spec:
  inputs:
    IMAGE_TAG:
    SERVER_HOST:
    SSH_PRIVATE_KEY_BASE_64:
    CI_REGISTRY_IMAGE:
    DOCKER_USER:
    DOCKER_PASSWORD:
    DOCKER_COMPOSE_NETWORK_NAME:
    DOCKER_HOST_PORT:
    DOCKER_CON_PORT:
    NGINX_FILENAME:
    DOCKER_COMPOSE_CON_NAME:

---
.deploy:
  before_script:
    - export IMAGE_TAG=$(echo $[[ inputs.IMAGE_TAG ]] | sed 's/\//\-/g' )
    - export IMAGE_NAME=$[[ inputs.CI_REGISTRY_IMAGE ]]:${IMAGE_TAG}
    - export NETWORK_NAME=$[[ inputs.DOCKER_COMPOSE_NETWORK_NAME ]]
    - export DC_HOST_PORT=$[[ inputs.DOCKER_HOST_PORT ]]
    - export DC_CON_PORT=$[[ inputs.DOCKER_CON_PORT ]]
    - export DOCKER_COMPOSE_CON_NAME=$[[ inputs.DOCKER_COMPOSE_CON_NAME ]]
    - cat "$[[ inputs.SSH_PRIVATE_KEY_BASE_64 ]]" | base64 -d > "key.txt"
    - chmod 400 "key.txt"
  script:
    # - sudo scp -o StrictHostKeyChecking=no -i "key.txt" $CI_COMMIT_BRANCH-nginx.conf ubuntu@$[[ inputs.SERVER_HOST ]]:/etc/nginx/conf.d/$CI_COMMIT_BRANCH-nginx.conf
    - export CONTAINER_NAME=$( echo ${DOCKER_COMPOSE_CON_NAME} | sed 's/\//\-/g' )
    - ssh -o StrictHostKeyChecking=no -i "key.txt" ubuntu@$[[ inputs.SERVER_HOST ]]
    - docker login -u $[[ inputs.DOCKER_USER ]] -p $[[ inputs.DOCKER_PASSWORD ]]
    - docker ps | grep "${CONTAINER_NAME}" | awk '{print $1}' | xargs -r docker stop || echo "No containers found with name ${CONTAINER_NAME}"
    - docker network inspect $[[inputs.DOCKER_COMPOSE_NETWORK_NAME ]] >/dev/null 2>&1 || docker network create $[[inputs.DOCKER_COMPOSE_NETWORK_NAME ]]
    - docker run --rm -d --pull=always --name ${CONTAINER_NAME}  -p 127.0.0.1:${DC_HOST_PORT}:${DC_CON_PORT} --network ${NETWORK_NAME} ${IMAGE_NAME}
    - export container_port=$(docker ps | grep ${CONTAINER_NAME} | sed 's/.*127.0.0.1://g'| sed 's/->.*//g' )
    - echo $container_port
    - echo $CONTAINER_NAME
    - sudo systemctl restart nginx
    - echo "CONTAINER_NAME=${CONTAINER_NAME}" >> build.env
    - echo "CONTAINER_PORT=${container_port}" >> deploy.env
  artifacts:
    reports:
      dotenv:
        - build.env
        - deploy.env
```

Figure 3-15 : abstract deploy job

3.4.8 Unifying Jobs into a Cohesive Pipeline

The first thing to do is configuring that all the jobs must run on the custom runner, configured earlier, and with help the tags keyword we can tell the GitLab-ci that the

executor of the jobs must be of tags : remote", "shell-executer", and "vps" It's like setting the ground rules for how things should work in those situations. The figure 3-16 bellow illustrates more details about the job.

Figure 3-16: Custom Gitlab Runner tags

Next, we need to import the previously created templates in order to use them in the main pipeline so instead of defining all configurations in a single monolithic file, we can split them into smaller, reusable templates and include them as needed in different pipelines or stages.

- The first file **(build.yml)** handles building the Docker image and pushing it to a registry. It takes inputs such as the registry image name, image tag, Docker credentials, etc.
- The second file **(deploy.yml)** deploys the Docker container to a server. It requires inputs like the server host, SSH private key, Docker image details, network settings, port configurations, NGINX configuration filename, etc.
- The third file **(nginx-setup.yml)** handles NGINX setup, possibly configuring it for serving web content or proxying requests. It takes inputs related to server domain name, SSL certificate paths, etc.

The local keyword indicates that the included YAML files are located within the same repository as the pipeline configuration file (.gitlab-ci.yml).Each included YAML file is specified with its local file path relative to the root of the repository.the figure elaborates all the details .The figure 3-17 illustrates more details about the job.

```yaml
include:
  - local: 'gitlab-templates/build.yml'
    inputs:
      CI_REGISTRY_IMAGE: $CI_REGISTRY_IMAGE
      IMAGE_TAG: $IMAGE_TAG
      DOCKER_PASSWORD: $DOCKER_PASSWORD
      DOCKER_USER: $DOCKER_USER
  - local: 'gitlab-templates/deploy.yml'
    inputs:
      SERVER_HOST: $SERVER_HOST
      SSH_PRIVATE_KEY_BASE_64: $SSH_PRIVATE_KEY_BASE_64
      CI_REGISTRY_IMAGE: $CI_REGISTRY_IMAGE
      IMAGE_TAG: $IMAGE_TAG
      DOCKER_USER : $DOCKER_USER
      DOCKER_PASSWORD : $DOCKER_PASSWORD
      DOCKER_COMPOSE_NETWORK_NAME: $DOCKER_COMPOSE_NETWORK_NAME
      DOCKER_CON_PORT: $DOCKER_CON_PORT
      DOCKER_HOST_PORT: $DOCKER_HOST_PORT
      NGINX_FILENAME: nginx-frontend.conf
      DOCKER_COMPOSE_CON_NAME: $DOCKER_COMPOSE_CON_NAME
  - local: 'gitlab-templates/nginx-setup.yml'
    inputs:
      SERVER_DOMAIN_NAME: $SERVER_DOMAIN_NAME
      SSL_CERTIFICATE_DOMAIN_PATH: $SSL_CERTIFICATE_DOMAIN_PATH
      SSL_CERTIFICATE_KEY_DOMAIN_PATH: $SSL_CERTIFICATE_KEY_DOMAIN_PATH
```

Figure 3-17 : jobs inclusion

Now let's rewind to the main workflow of the pipeline, since we have three main static branch, we will be separating each one of them and set environment variables accordingly. For the 'develop' branch, it configures a development environment with specific settings for Docker Compose network, container registry... When the branch is 'main' or 'pre-prod', it establishes a pre-production environment with designated variables. For any other branch, it assigns a unique domain name, SSL certificate paths. However, if none of the branch conditions are met, a default rule prevents any workflow from executing. The figure 3-18 bellow illustrates more details about the job.

```yaml
workflow:
  rules:
    - if: $CI_COMMIT_BRANCH == 'develop' # main branch
      variables:
        DOCKER_COMPOSE_NETWORK_NAME: $CI_DEFAULT_BRANCH
        CI_REGISTRY_IMAGE: abdelbaki1/develop
        DOCKER_HOST_PORT: 9002
        DOCKER_CON_PORT: 80
        DOCKER_COMPOSE_CON_NAME: $CI_PROJECT_NAME--$CI_COMMIT_BRANCH-container
        IMAGE_TAG: $CI_PROJECT_NAME-latest


    - if: $CI_COMMIT_BRANCH == 'main'
      variables:
        DOCKER_COMPOSE_NETWORK_NAME: prod
        CI_REGISTRY_IMAGE: abdelbaki1/prod
        DOCKER_HOST_PORT: 9000
        DOCKER_CON_PORT: 80
        DOCKER_COMPOSE_CON_NAME: $CI_PROJECT_NAME--$CI_COMMIT_BRANCH-container
        IMAGE_TAG: $CI_PROJECT_NAME-latest
    - if: $CI_COMMIT_BRANCH == 'pre-prod'
      variables:
        DOCKER_COMPOSE_NETWORK_NAME: pre-prod
        CI_REGISTRY_IMAGE: abdelbaki1/pre-prod
        DOCKER_HOST_PORT: 9001
        DOCKER_CON_PORT: 80
        DOCKER_COMPOSE_CON_NAME: $CI_PROJECT_NAME-$CI_COMMIT_BRANCH-container
        IMAGE_TAG: $CI_PROJECT_NAME-latest

    - if: $CI_COMMIT_BRANCH
      variables:
        SERVER_DOMAIN_NAME: $CI_COMMIT_BRANCH.test.com
        SSL_CERTIFICATE_DOMAIN_PATH: '/etc/letsencrypt/live/pre-prod.djubatusr.com/fullchain.pem'
        SSL_CERTIFICATE_KEY_DOMAIN_PATH: '/etc/letsencrypt/live/pre-prod.djubatusr.com/privkey.pem'
        DOCKER_COMPOSE_NETWORK_NAME: $CI_COMMIT_BRANCH
        CI_REGISTRY_IMAGE: abdelbaki1/features
        DOCKER_HOST_PORT: 0
        DOCKER_CON_PORT: 80 #auto-mapping
        DOCKER_COMPOSE_CON_NAME: $CI_PROJECT_NAME-$CI_COMMIT_BRANCH-container
        IMAGE_TAG: $CI_PROJECT_NAME-$CI_COMMIT_BRANCH-latest
    - when: never
```

Figure 3-18 pipeline rules

Once the workflow has been set up, the pipeline proceeds to execute the jobs according to their order, which is called stage. These are defined below are used to elaborate the

jobs dependency chain order and for clarity when observing the logs at the end, The figure 3-19 bellow illustrates more details about the stages

```
stages:
  - ci_lint_nginx
  - build
  - add_nginx_frontend_config
  - deploy
  - create_nginx_conf
```

Figure 3-19: pipeline stages

Now let's get deeper to some jobs mechanisms for each one of the phase:In the build phase of the pipeline we have the job "build_extended" stage that extends the configuration from ".build" section defined previously in job templates. It also sets up caching for this stage, where the cache will be downloaded before the job starts, and uploaded after the job completes successfully.The "policy: pull-push" line specifies that the cache should be pulled (downloaded) before the job runs, and pushed (uploaded) after the job succeeds.

Caching is a mechanism that allows reusing data from previous job runs, such as downloaded dependencies or compiled artifacts, to speed up subsequent runs and save time and resources.it optimize the build process by reusing previously cached data when possible, reducing the time and resources required for repetitive tasks. The figure 3-20 bellow illustrates more details about the job

```
build_extended:
  stage: build
  extends: .build
  cache:
    policy: pull-push
    when: on_success
```

Figure 3-20 : extended build job

The deploy stage called "deploy_extended" with a specific environment setup. The environment keyword is used to specify the target environment for the deployment. The on_stop parameter is set to "teardown_containers", which refers to manual job that will be executed when the deployment is stopped or terminated. This involve stopping

and removing any running containers or cleaning up resources associated with the deployed application. The figure 3-21 bellow illustrates more details about the job

```
deploy_extended:
  environment:
    name: feature/$CI_COMMIT_BRANCH
    url: https://$SERVER_DOMAIN_NAME
    on_stop: teardown_containers
  stage: deploy
  extends: .deploy
```

Figure 3-21 deploy job

By using the environment keyword and dynamically setting the environment name based on the Git branch, this configuration allows for easy management of multiple deployment environments, enabling developers to test and validate their changes in isolated environments before promoting them to higher-level environments or production.

This job configuration defines a task named "teardown_containers" that is responsible for stopping and removing Docker containers associated with a deployed environment. It is part of the "deploy" stage and is triggered manually (when: manual) for environment cleanup purposes.

The "environment" section defines the target environment for this task, with the name set to "feature/${CI_COMMIT_BRANCH}". This naming convention aligns with the environment setup in the "deploy_extended" job, allowing the teardown process to target the same environment created during deployment. The "action: stop" parameter indicates that the goal of this job is to stop the running containers, as opposed to other actions like starting or redeploying.The figure 3-22 bellow illustrates all related job detail

```yaml
teardown_containers:
  stage: deploy
  needs:
    - deploy_extended
  script:
  - docker stop $(docker ps -q --filter name=${CONTAINER_NAME} )
  environment:
    name: feature/$CI_COMMIT_BRANCH
    action: stop
  when: manual
```

Figure 3-22 cleaning job

Since this is manual job being created in GitLab. It requires user intervention must be started manually when its needed, in Gitlab a manual job will like the figure 3-23 below

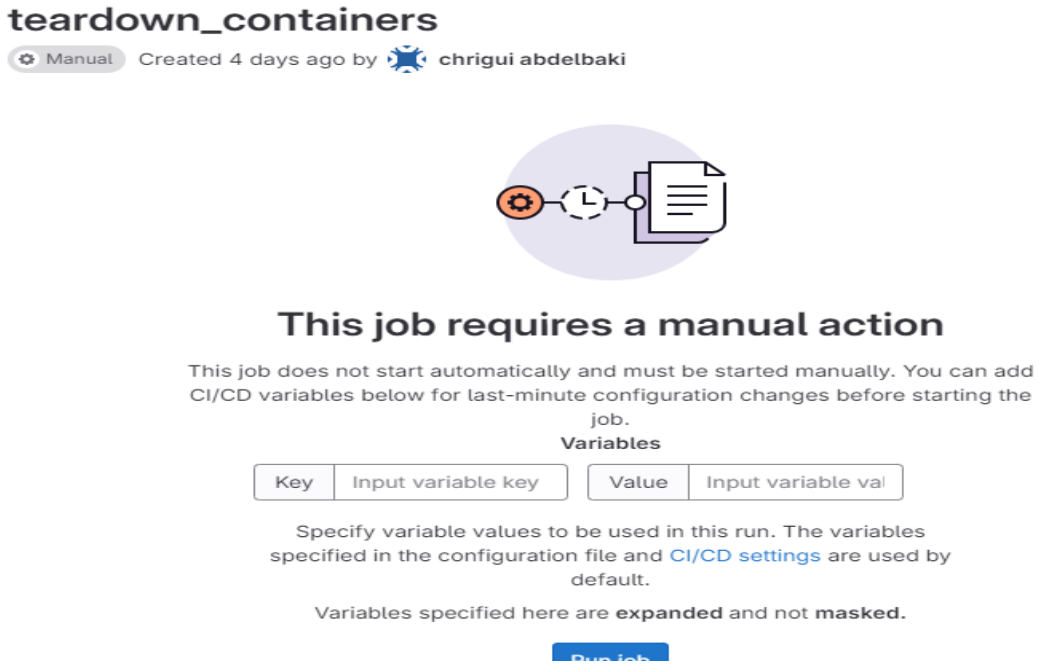


Figure 3-23 : Gitlab manual job trigger

Conclusion

In this chapter we started the realization of the CI/CD pipeline while emphasizing some jobs that show the major functionalities of the solution.

Chapter 4: leveraging pipeline

Introduction

In this chapter we present the automation phase of our solution, building upon the successful design and containerized environment implementation from the previous sprint. We'll describe the second sprint, concerned by the automation of the monitoring tasks and dependency workflow.

4.1 Monitoring tools architecture

In this section, we will be describing different tools architecture used for this sprint

4.1.1 Prometheus

At the heart of Prometheus lies the Prometheus server, responsible for data collection, storage, querying, and processing of metrics, also:

- Alerting and Alert Manager: Prometheus offers a built-in alerting system that enables users to define custom rules for generating alerts based on metric thresholds or complex conditions.
- Exporters and Instrumentation Libraries: Prometheus boasts a vast ecosystem of exporters and instrumentation libraries that facilitate the collection of metrics from a wide range of systems and applications.
- Grafana Integration: Prometheus seamlessly integrates with Grafana, a popular data visualization and exploration tool allowing users to build dynamic dashboards, visually representing Prometheus metrics and providing real-time insights into system performance.

The figure 4.1 illustrates more the relation between Prometheus components

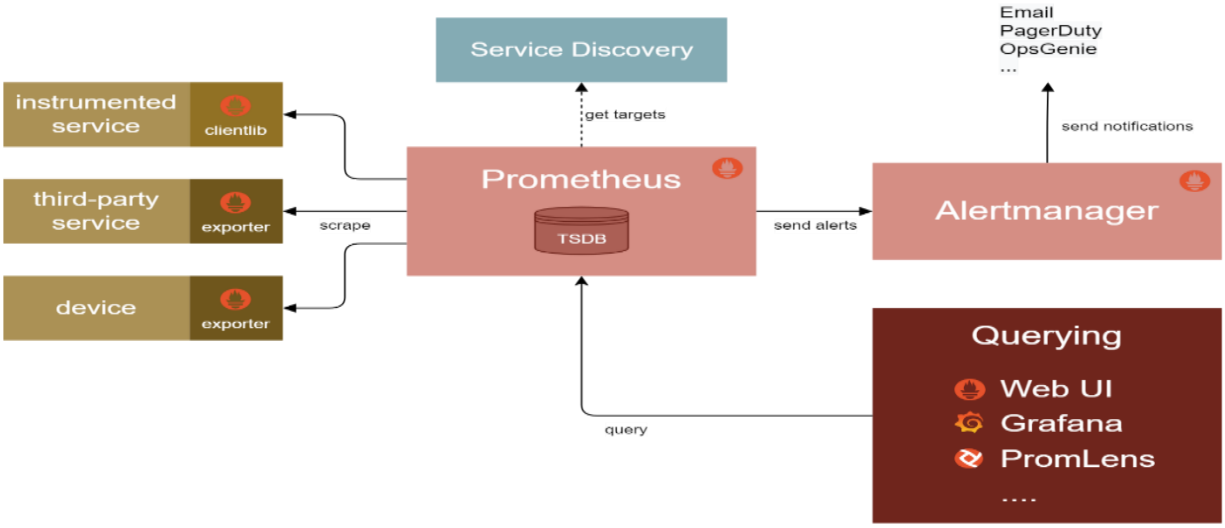


Figure 4-1:prometheus architecture [12]

4.1.2 Ansible

Ansible comprises several key components that form its automation framework:

- Control Node: The central machine where Ansible is installed and operated. Administrators run playbooks (YAML scripts outlining automation tasks) from this node. Ansible's agentless architecture enhances scalability and minimizes network resource usage.
- Managed Nodes: Servers, systems, or devices automated by Ansible. The control node accesses these via SSH (Linux/Unix), leveraging existing security and authentication frameworks without requiring agent installations.
- Inventory: A list of managed nodes that Ansible controls, detailing access methods and optionally categorizing nodes into groups for efficient management and targeting within playbooks.
- Playbooks: Fundamental to Ansible's automation, defining desired system states, tasks to achieve these states, and the sequence of task execution.

The Figure 4.2 shows the ansible architecture

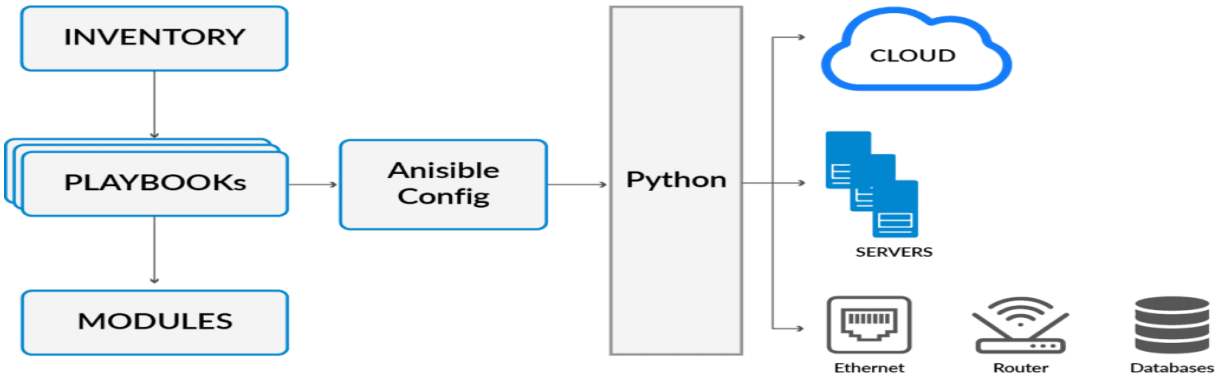


Figure 4-2 : ansible architecture [13]

4.2 Sprint Backlog

The table 4.1 explain the sprint backlog throughout the user stories

Table 4-1:sprint 2 backlog

ID	User story	Technical Story
2	As a DevOps engineer, I want to automate pipeline monitoring, so that I can proactively detect issues, optimize performance, and ensure high availability of our applications, thereby reducing downtime and improving user experience.	Install Ansible for infrastructure automation
		Install monitoring tools
		Deploy Node Exporter on all servers for system metrics
		Add alert management system
	automate server configuration, so that I can ensure consistent, reliable, and scalable infrastructure across our environments, reducing manual errors and deployment times	Create Grafana dashboards
		Set up Prometheus and Grafana for metric collection and storage
		Create infrastructure inventory for Ansible
		Create playbooks for pipeline dependencies and monitoring solution

4.3 Sprint analysis

In this section, we will be describing the use case diagram of this sprint and their textual description

4.3.1 Use case diagram

These figure 4.3 show in detail the use Cases of this sprint

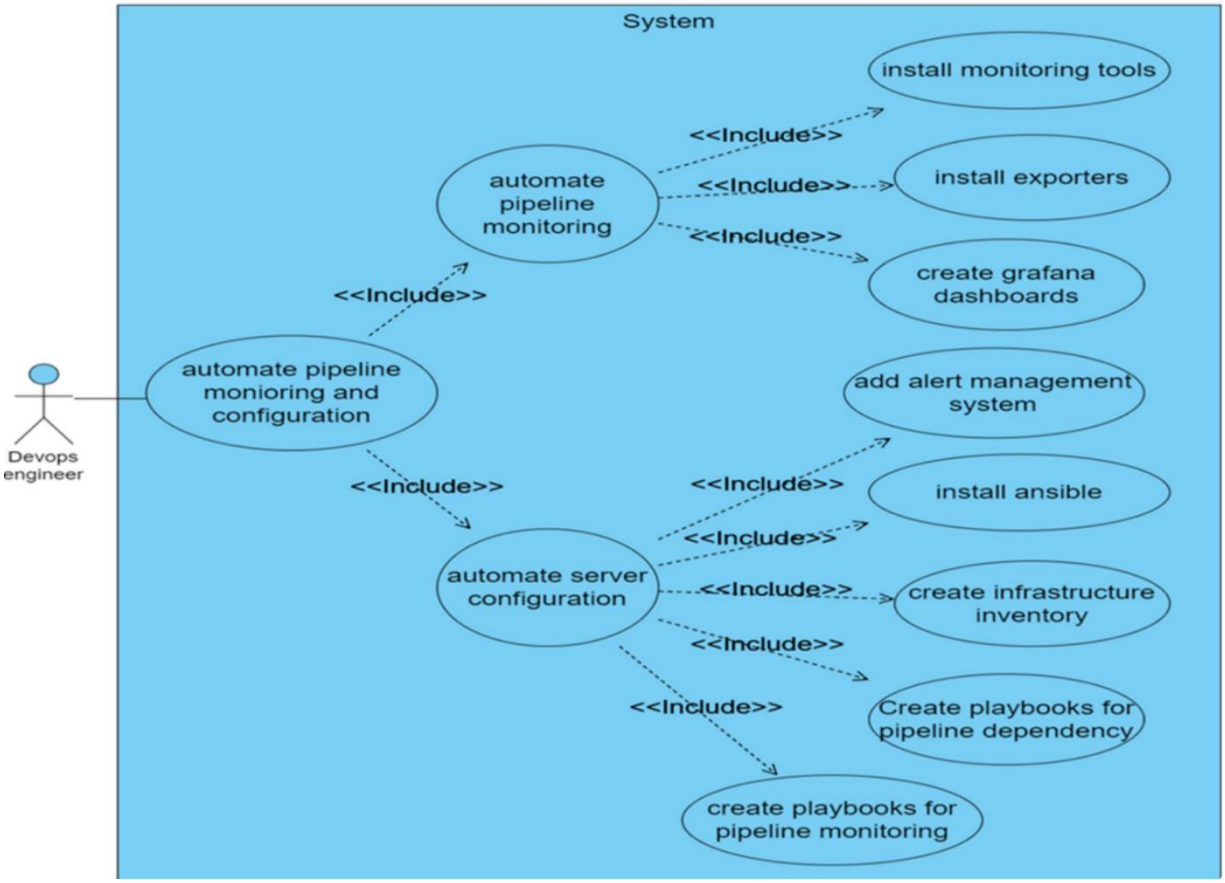


Figure 4-3: Automate pipeline monitoring and server Configuration Use Case

4.3.2 Use case textual explanation

The table 4.2 describes the use case "Automate Pipeline Monitoring" and "Automate Server Configuration"in text

Table 4-2 sprint 2 use case textual explanation

Case	Automate Pipeline Monitoring
Description	This use case describes the process of automating pipeline monitoring by installing monitoring tools, installing exporters, creating Grafana dashboards, and adding an alert management system.
Primary Actor	Devops engineer
Preconditions	• Monitoring tools are available for installation. • Exporters are ready to be set up. • Grafana is installed and accessible • Alert management system is available for integration.
Main Success Scenario	• DevOps engineer automates the pipeline monitoring. • Install necessary monitoring tools. • Creates Grafana dashboards for visualizing metrics. • Adds and configures an alert management system.
alternative Scenario	• monitoring tool installation fails, the engineer logs the error and retries. • If creating Grafana dashboards fails, the engineer revisits the configuration steps. • If adding the alert management system fails, the engineer debugs and corrects the integration.
Post-conditions	• Pipeline monitoring is automated and functional • Monitoring tools are installed and operational. • Exporters are set up and collecting data. • Grafana dashboards are created and display metrics. • Alert management system is functional

Table 4-3 automate Server configuration use case

Case	Automate Server Configuration
Description	This use case describes the process of automating pipeline monitoring by installing monitoring tools, installing exporters, creating Grafana dashboards, and adding an alert management system.
PrimaryActor	Devops engineer
Preconditions	• Monitoring tools are available for installation. • Exporters are ready to be set up. • Grafana is installed and accessible. • Alert management system is available for integration.
Main Success Scenario	DevOps engineer installs Ansible on the system. DevOps engineer creates an infrastructure inventory listing all servers to be managed. DevOps engineer creates playbooks for pipeline dependency management. DevOps engineer creates playbooks for pipeline monitoring. DevOps engineer uses the created playbooks and inventory to automate server configuration.
Scénario alternatif	If the ansible tool installation fails, the engineer logs the error and retries. If adding the playbooks fails, the engineer debugs and corrects the integration.
Post-conditions	• Pipeline monitoring is automated and functional. • Monitoring tools are installed and operational. • Exporters are set up and collecting data. • Grafana dashboards are created and display metrics. • - Alert management system is functional

4.4 Sprint implementations

In this section, we will be describing different sprint 1 implementations

4.4.1 Completing CI/CD lifecycle with monitoring

The figure 4-4 displays the initial configuration of the exporters and linking back to Prometheus server

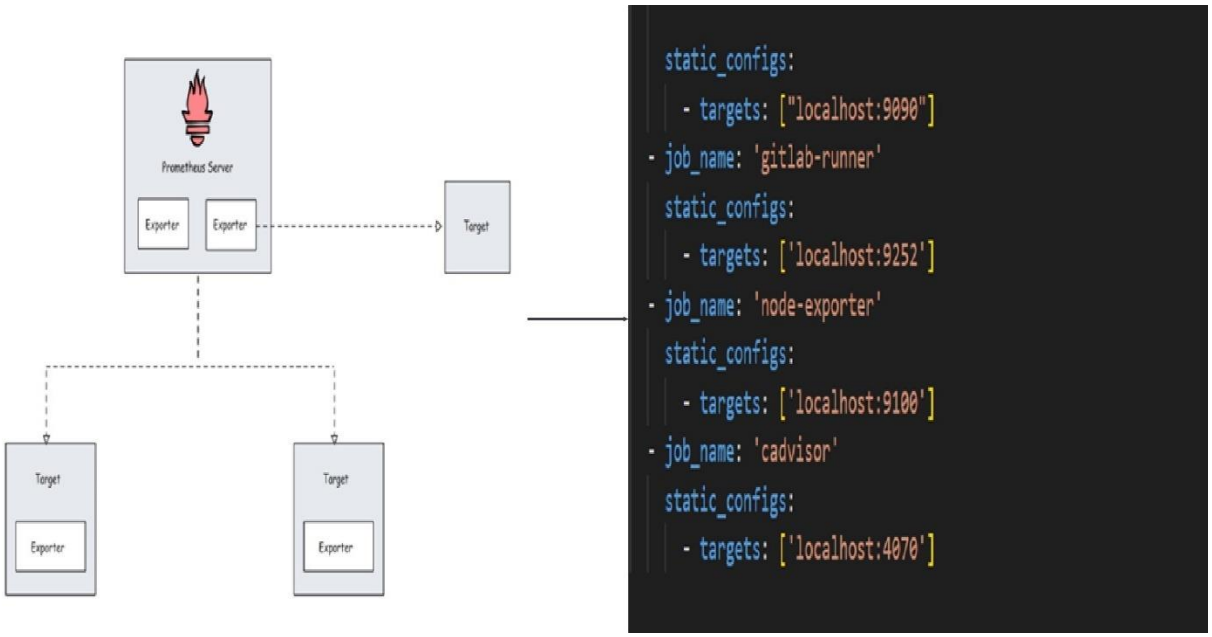


Figure 4-4 : Prometheus exporter design

4.4.2 Monitoring jobs

After setting up exporters we can verify if Prometheus is successfully scraping metrics from them by checking the Targets page. For example,

- "cadvisor" with the endpoint at http://localhost:4070/metrics. This target has the "UP" state, indicating Prometheus can reach and collect metrics from the cAdvisor instance running on the local host at port 4070.

The Targets page shows additional details like the last successful scrape time, scrape duration, and labels associated with each target. By monitoring this page, we can quickly identify if Prometheus encounters issues scraping metrics from any of the configured data sources. A target in an "Unhealthy" state would indicate a problem that needs investigation and resolution. Figure 4-5 bellow illustrates more details about the targets

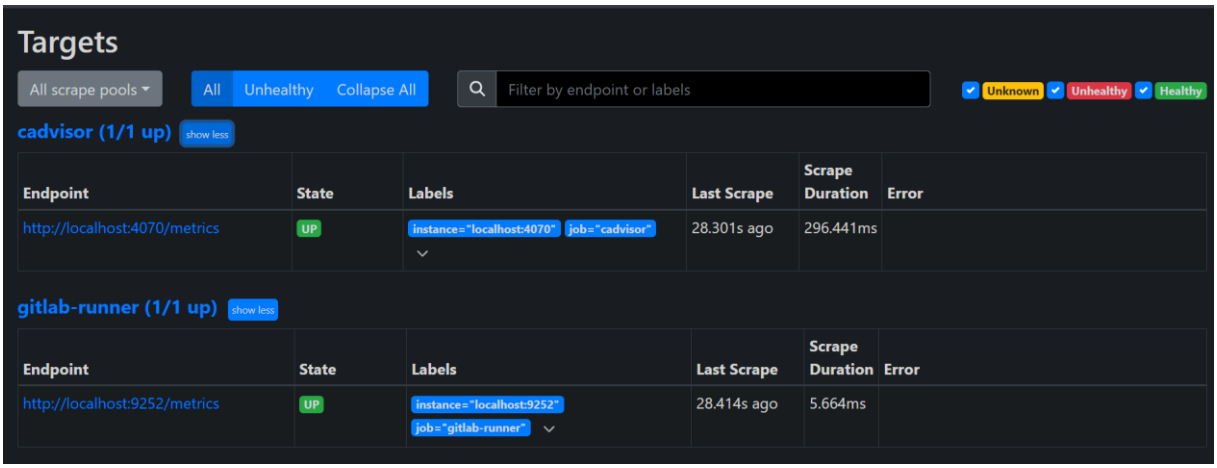


Figure 4-5: prometheus target endpoints

4.4.3 Alerting

We start by specifying the image name as "prom/alertmanager".The "volumes" section mounts the host directory "/alertmanager" to the container path '/etc/alertmanager'. the "network mode" is set to "host", which means that the alertmanager container will share the network namespace with the host machine, allowing it to access network resources as if it were running directly on the host.Figure 4-6 bellow illustrates more details about the job.

```
alertmanager:
  image: prom/alertmanager
  volumes:
    - ./alertmanager:/etc/alertmanager
    # - alertmanager_data:/alertmanager
  command:
    - '--config.file=/etc/alertmanager/config.yml'
    - '--storage.path=/alertmanager'
    - --log.level=debug
  network_mode: "host"
```

Figure 4-6 : alert manager configuration

In the global section, we can see three key settings:

• scrape_interval: 15s - This sets the interval at which Prometheus will scrape metrics from its configured targets (e.g., node_exporter, cAdvisor) to 15 seconds.

• scrape_timeout - This setting is left at the default global value, which determines how long Prometheus will wait for a target to respond before marking it as unresponsively.

The Alertmanager is a crucial component in the Prometheus ecosystem, responsible for handling alerts generated by Prometheus's alerting rules.By configuring the Alertmanager integration, Prometheus can not only collect and visualize metrics but also actively monitor for specific conditions and send notifications when those conditions are met, enabling proactive incident response and maintaining the reliability of monitored systems. The figure 4-7 bellow illustrates more details about the alerting system.

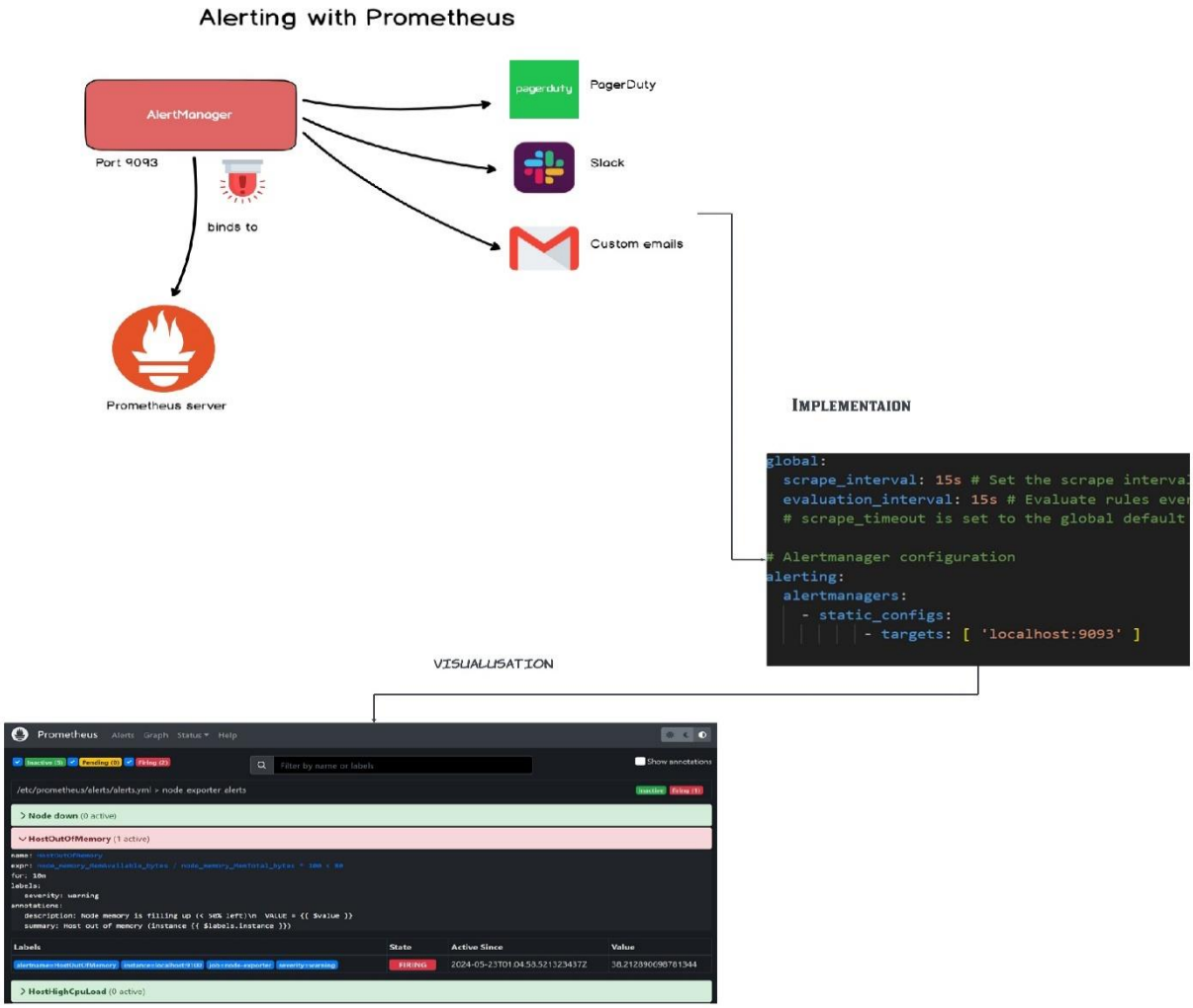


Figure 4-7 : alert manager workflow

Now Prometheus is up and running, with Alertmanager configured. An alert will be created and an email like the following will be sent to the specified email account.

The figure 4-8 bellow illustrates more details about the email.

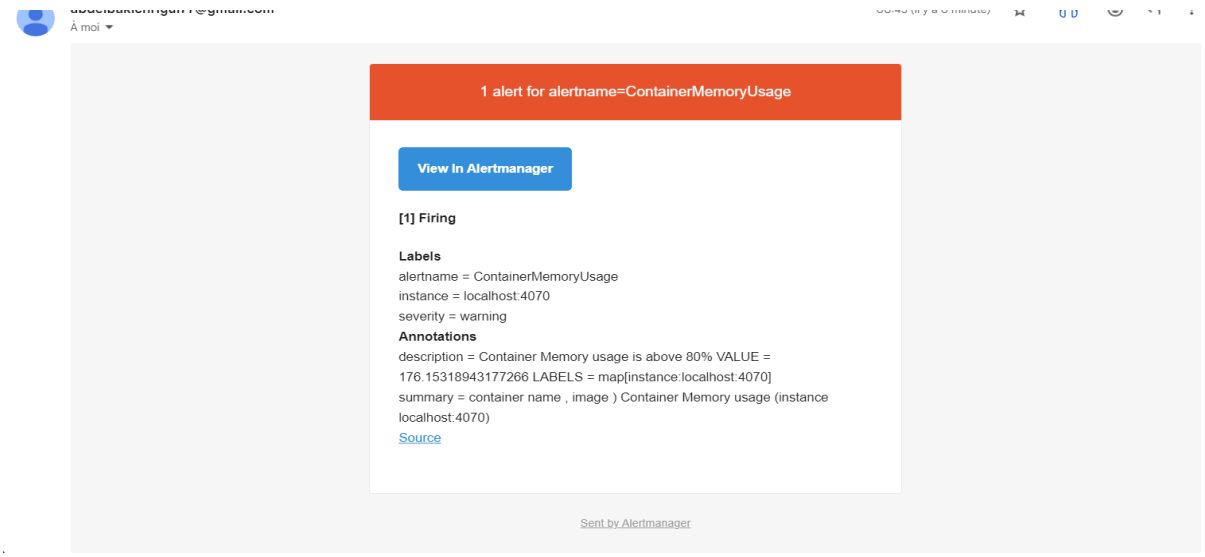


Figure 4-8 : alert message through Gmail

4.4.4 Dashboarding with Grafana

Grafana installation

Adding Prometheus as a data source in Grafana involves configuring the Grafana UI. Through Docker Compose. we launch Grafana and its data sources (like Prometheus) in one go. It simplifies deployment and keeps things running smoothly. The figure 4-9 illustrates more details about the job.

```
grafana:
  image: grafana/grafana
  volumes:
    - grafana_data:/var/lib/grafana
    - ./grafana/grafana.ini:/etc/grafana/grafana.ini
  network_mode: "host"
```

Figure 4-9 : Grafan image instalation

Adding data source to Grafana

Choose "Prometheus" to indicate we want to utilize data collected by our Prometheus server.Next, locate the field labeled "From" or "URL" and enter the web address (URL)

of our Prometheus server.The format typically follows http://:, where represents the IP address or hostname of Prometheus server, and specifies the port it listens on (often 9090).If Prometheus server requires authentication, look for a section labeled "Signature" or similar.Clicking a button like "Add new data source" might allow we to configure username and password credentials.

In the bottom corner, we might find a link offering more information about the Prometheus Data Source.Finally, after entering the server URL and configuring any authentication, remember to save the data source configuration. This establish a connection with Prometheus and retrieve data for visualizations. The figure 4-10 illustrates more the process

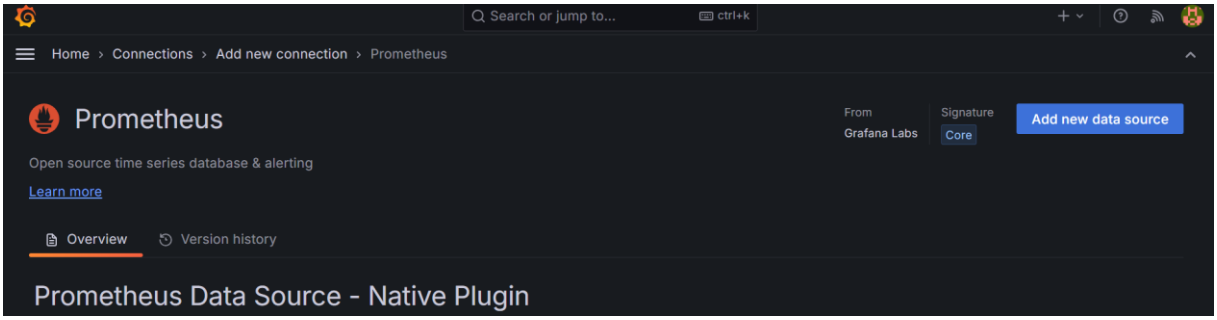


Figure 4-10: Grafana data source

Import dashboard template

Grafana offers a convenient way to import pre-built dashboard templates, saving us time and effort in creating visualizations for our data. This process allows us to leverage existing dashboards or share our own creations with others. Importing a template involves two methods:

- Public URL or ID: For publicly available templates on Grafana.com, we can simply paste the URL or unique identifier (ID) into the designated field. Grafana will then retrieve the template definition for import.
- Direct JSON Text: Alternatively, if we have the JSON text of the template readily available, we can copy and paste it directly into the provided text area within

Once imported successfully, we can save the dashboard with any further adjustments we desire.

The figure 4-11 illustrates more details about the dashboard import.

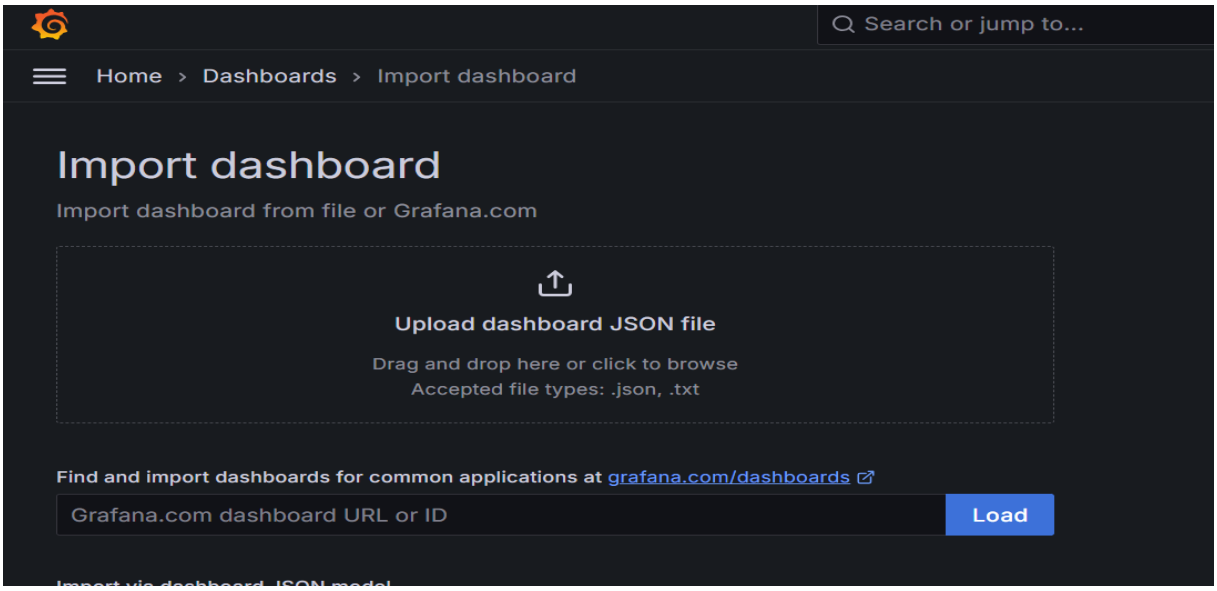


Figure 4-11: Dashboard import

Test the dashboards

This Visualization that contain info's like CPU utilization graphs, memory allocation heatmaps, disk I/O bar charts, and network traffic histograms can paint a clear picture of system health The figure 4-12 illustrates more details about the dashboard stats.

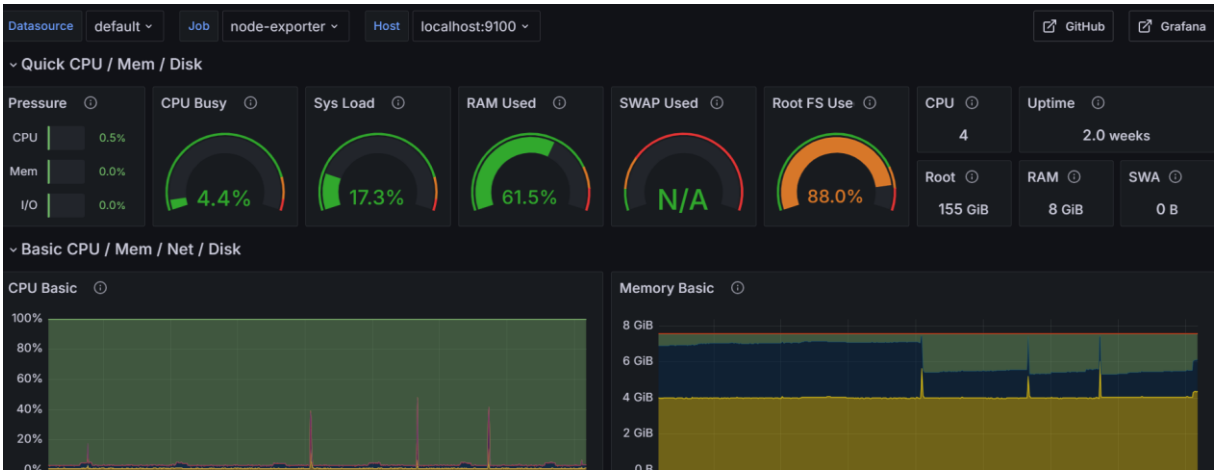


Figure 4-12 :Gitlab Runner statistics

Additionally the dashboards suite include pipeline-specific metrics like queue lengths (number of tasks waiting to be processed), The figure bellow illustrates more the other metrics The figure 4-13 illustrates more details about the dashboard stats.

.

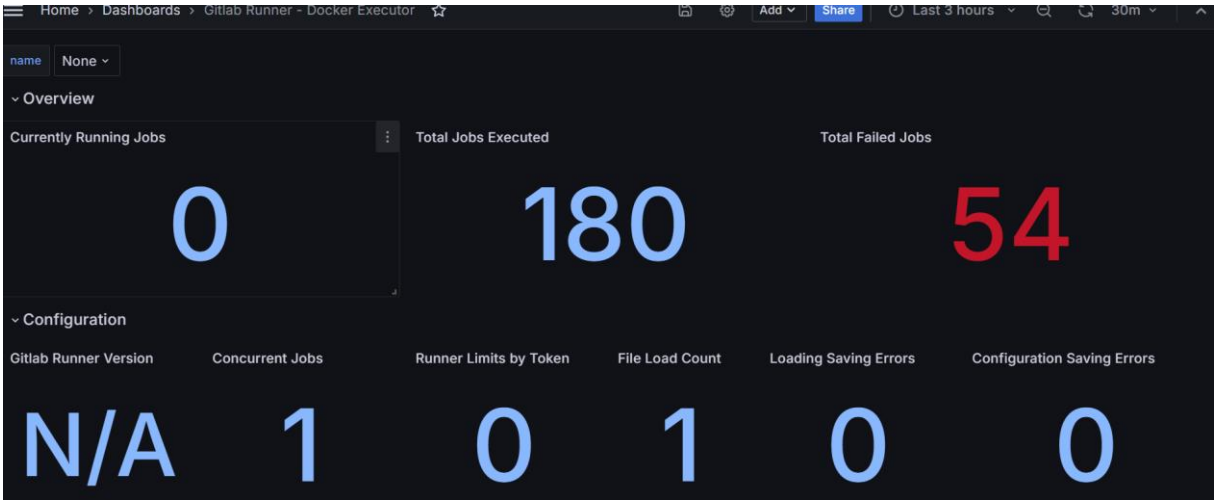


Figure 4-13: pipeline metrics dashboard

4.4.5 Create inventories for existing infrastructure

we can specify the location to this configuration file through an environment variable & set it to the path to the new config file.this time when playbook is run.Ansible picks up this file instead of the default configuration file.In the '[defaults]' section, it specifies the location of the Ansible inventory file as './inventory.yml', which is in the same directory as the ansible.cfg file. Additionally, the 'host_key_checking' option is set to 'false', disabling the host key checking feature in Ansible. This setting allows Ansible to connect to hosts without prompting for confirmation when encountering an unknown or different host key, which is often used in testing or development environments but not recommended for production environments due to security concerns. The figure 4-14 illustrates more the configuration file

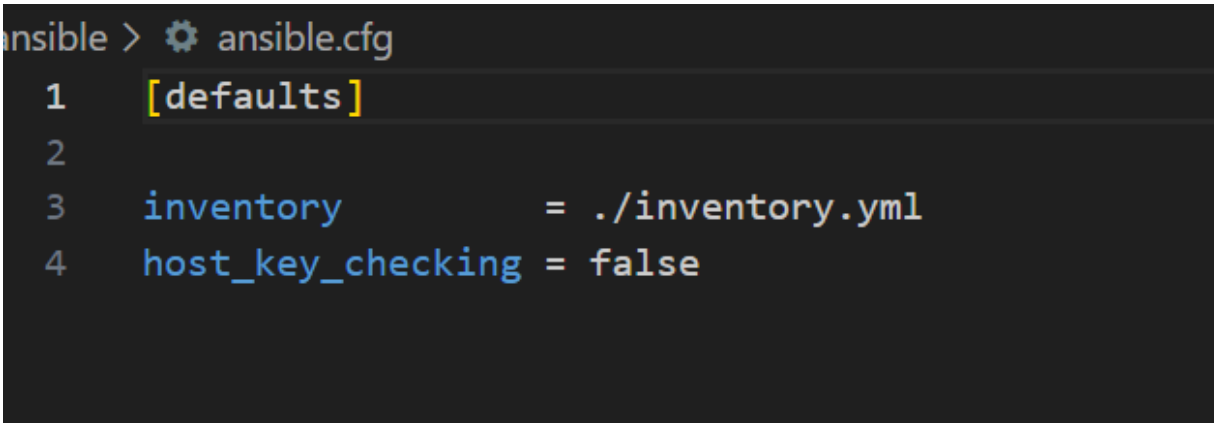


Figure 4-14: ansible.cfg

Once we setup ue configs,let us create Ansible inventory file in JSON format, which defines the hosts and associated connection details for Ansible to manage. Under the top-level "all" group, there is a single host named "server1" specified with the following attributes: "ansible_host" set to the IP address "51.4.140.152" for connecting to the host, "ansible_user" set to "ubuntu" as the username for SSH connections, and "ansible_ssh_private_key_file" specifying the path to the private SSH key file "~/private.key.pem" located in the user's home directory, which Ansible will use for authenticating and establishing secure connections to this host. The figure 4-15 bellow illustrates more the inventory

Figure 4-15: ansible inventory

4.4.6 Creating playbook

This playbook is designed to prepare an Ubuntu system for the installation of Docker, Nginx, and Certbot by ensuring the necessary package dependencies are met, and the required repositories are configured correctly.The playbook start by define the configuration as a tasks like our case ,named "Install Docker, Nginx, and Certbot", also the tasks will be executed on the host "server1". The playbook begins by updating the apt package cache, followed by installing several dependencies required for Docker, Nginx, and Certbot, such as "apt-transport-https", "ca-certificates", "curl", and "software-properties-common". It then adds the Docker repository to the system's APT sources list, The figure 4-16 illustrates more the playbook

Figure 4-16 dependency playbook

The "Install Docker" task uses the "apt" module to install the "docker-ce" package, which is the Docker. Next, the "Install Nginx" task becomes the root user and installs the "nginx" package using the "apt" module. Lastly, the "Install Certbot" task installs the "certbot" package, which is a tool for automatically obtaining and renewing SSL/TLS certificates from Let's Encrypt, a free and open certificate authority The figure 4-17 illustrates more the playbook

Figure 4-17 tools playbook

Next task is designed to deploy a monitoring and visualization stack using Docker containers on the host "server1". Here are the details:

The playbook starts by defining variables: "compose_src" pointing to "/root/compose.yml" and "compose_dist" pointing to "/home/ubuntu/compose.yml".

The first task is "Copy file with owner and permissions", which uses the "ansible.builtin.copy" module to copy files from the source ("{{ compose_src }}") to the destination ("{{ compose_dist }}"). The destination file is set to be owned by the user "ubuntu" and the group "ubuntu", with permissions set to '700'.

The second task is "start container with docker docker-compose", which utilizes the "community.docker.docker_compose" module to start containers based on the "compose.yml" file located at "/home/ubuntu". The "recreate" option is set to "always", ensuring containers are recreated every time the playbook runs, and the "state" is set to "present" to ensure the containers are running. The figure 4-18 illustrates more the playbook

```yaml
- name: Install Prometheus, Grafana, Alertmanager, cAdvisor, and Node Exporter using Docker
  hosts: server1
  vars:
    compose_src: "/root/compose.yml"
    compose_dist: "/home/ubuntu/compose.yml"
  tasks:
    - name: Copy file with owner and permissions
      ansible.builtin.copy:
        src: "{{ compose_src }}"

        dest: "{{ compose_dist }}"
        owner: ubuntu
        group: ubuntu
        mode: '700'

    - name: start container with docker docker-compose
      community.docker.docker_compose_v2:
        project_src: /home/ubuntu
        files:
          - compose.yml
        recreate : always
        state: present
```

Figure 4-18 monitoring playbook

Conclusion

In this chapter, we have completed the part of implementing monitoring and automating dependency using ansible. The last part will be the configuration of CI/CD security, which we will detail in the next chapter

Chapter 5: completing pipeline with security

Introduction

Once we're done with the automation phase of our solution, it is necessary to study the security of the system. In this chapter, we'll describe how we go on to the last sprint, that provides the security pitfalls of our pipeline workflow.

5.1 Security Throughout the Pipeline: Secret Management and Integrity Validation

To measure and understand security risks and vulnerabilities, organizations need some industry standards and guidelines as well to learn from previous security incidents and mistakes.The OWASP (Open Web Application Security Project) Foundation is an open community and non-profit organization formed by various large companies, governments, individuals, and people that helps organizations develop secure applications by providing various security tools, standards, and resources.

One of the popular and widely used OWASP projects is the OWASP Top Ten [ref], which is a list of the top ten security categories that companies should consider based on security breaches and important trends adjusted to keep it up-to-date and relevant. It serves as a high-level categorization of various security aspects of applications, and more detailed security issues can be mapped to the OWASP Top Ten categories. Among the critical security risks outlined the CICD-SEC and CICD-SEC-6.

In this section, we will go through the attack scenario and consequences

5.1.1 CI-CD-SEC6 attack scenario and consequences

During the CI process, an attacker exploits a vulnerability and upload credentials or sensitive data to gain unauthorized access to the build and deployment processes. They retrieve embedded credentials from container image layers and clear-text credentials from console logs. Also, Neglected credential rotation leaves many unchanged for years, allowing the attacker to maintain prolonged access and deploy

malicious code, compromising the company's high-value resources. The Figure 5-1 describes the attack.

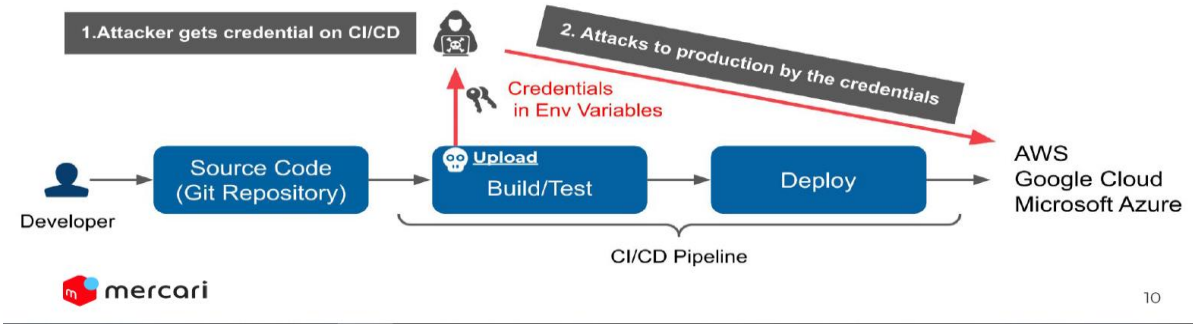


Figure 5-1: CI-CD-SEC6 [14]

If an attacker can inject malicious environment variables or other sensitive data, they can:

- gain unauthorized access to critical systems and bypass security controls.
- compromise or steel sensitive data stored or processed within the CI/CD pipeline
- exploit stolen credentials to compromise entire systems or networks.
- move laterally to escalate privileges and access additional resources.

5.1.2 CI-CD-SEC9 attack scenario and consequences

Improper artifact integrity validation permits to inject malicious code into the software delivery pipeline via artifacts. Artifact integrity validation ensures that digital artifacts, including software packages, containers, and configuration files, remain unaltered and authentic from their original state.

In an example scenario show in the figure below, an attacker with access to a system within the CI/CD process uploads an artifact that contains malicious code, replacing the correct artifact. Initially, an attacker identifies vulnerabilities and examines the company's CI/CD process, noting the combination of internal resources and third-party packages to identify potential weak points where artifact integrity validation

might be insufficient. By substituting a legitimate library with the tampered version and then re-packages them.

Due to insecure artifact validation practices, the system unknowingly fetches the tampered library from the compromised mirror repository.

the resulting application, now tainted with malicious code, progresses through the pipeline The figure 5-2 illustrates more details about the attack.

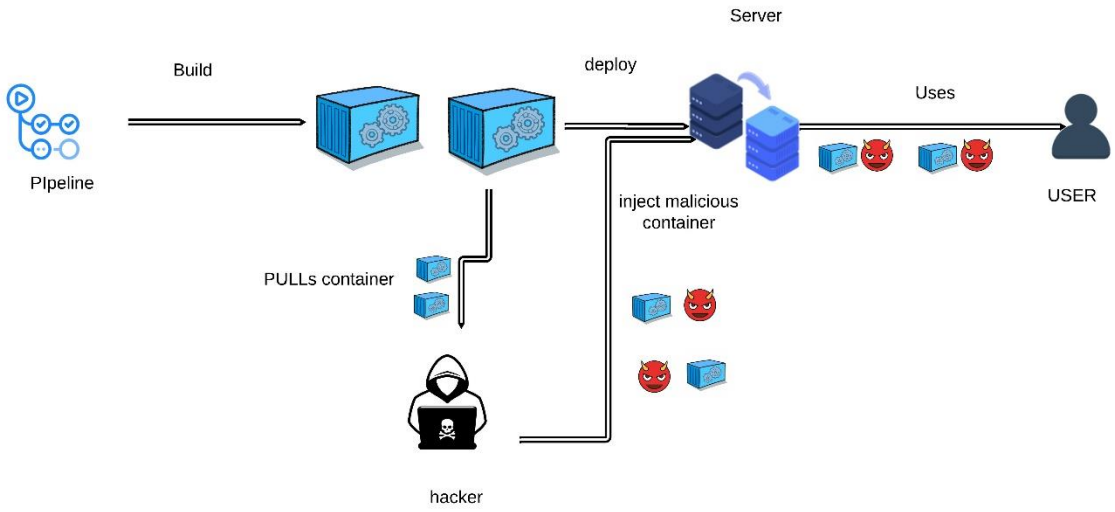


Figure 5-2: CI-CD-SEC-9 attack scenario

If an attacker can inject malicious environment variables or other sensitive data, they can:

- gain unauthorized access to critical systems and resources, bypassing security controls.
- **be compromised or stolen** sensitive data stored or processed within the CI/CD pipeline may.
- exploit stolen credentials to compromise entire systems or networks.
- move laterally to escalate privileges and access additional resources.

5.2 DevSecOps Revolution: Transforming Security in Devops

Today, developers can change or update new code in large applications several times per day, thanks to the CI/CD (Continuous Integration/Continuous Development) and DevOps tools, but in this process, we might also face undesirable security risks. That's where DevSecOps comes into the picture. For a DevOps framework to be successful, we need robust security integration. To keep the application code safe, businesses need to empower their developers with robust security tools. Enterprises should not wait for the security team to analyze and detect security vulnerabilities at a later stage. Instead, developers must utilize DevSecOps tools earlier to identify code problems right in the CI/CD pipeline. This approach will help developers fill up security gaps and mitigate security risks before reaching production infrastructure. DevSecOps processes and tools are deployed wherever our new applications or code are, thereby securing our CI/CD pipeline

DevSecOps extends DevOps by promoting collaboration between Security and DevOps professionals, focusing on security and stability alongside speed and efficiency.

The principles of DevSecOps include all DevOps principles with additional specific points:

- Shift-left Security (collaboration with Security professionals) - DevSecOps makes security a shared responsibility among 'Dev', 'Ops', and 'Sec' teams by introducing security early in the DevOps process.
- Automating Compliance and Policy Enforcement - DevSecOps enforces policies and ensures compliance, crucial for large enterprises and regulated industries.
- Continuous Monitoring and Incident Response - Continuous monitoring of applications and infrastructure is crucial in DevSecOps. Automated monitoring tools scan for performance anomalies and security vulnerabilities, enabling rapid response and mitigation.

The figure 5-3 illustrates more details about the DevSecOps lifecycle

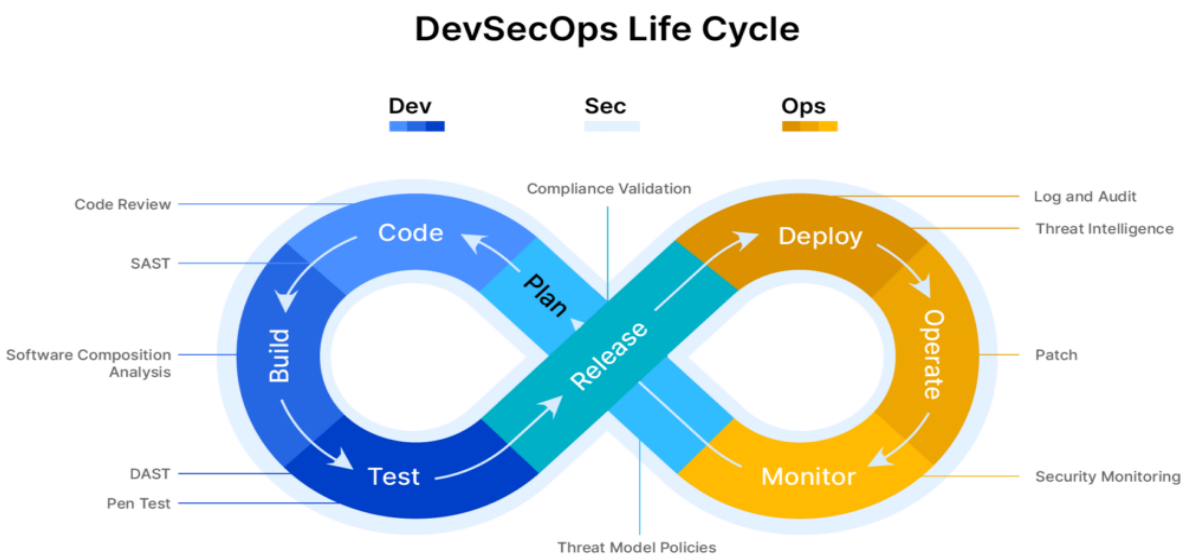


Figure 5-3 DevSecOps lifecycle [15]

5.3 Sprint Backlog

The table 5.1 below explains the sprint backlog in form of user stories

Table 5-1 : Sprint 3 Backlog

ID	User story	Technical Story
2	As a DevOps engineer, I want to set up secure pipeline management, so that I can ensure the integrity, authenticity, and security of our CI/CD processes and artifacts throughout the development lifecycle	Deploy and configure chosen Vault server with high availability and disaster recovery
		Generate and securely store pipeline authentication credentials in Vault
		Implement integrity checks using HMACs to verify artifacts in the pipeline
		Set up GitLab CI/CD jobs for security scans (SAST, DAST, dependency checks)
		Create and configure pipeline roles in HCP Vault with least privilege access

5.4 Sprint analysis

In this section we will provide the use case diagram and its textual description

5.4.1 Use case diagram

Figure 5-4 shows the use case diagram of secure pipeline management.

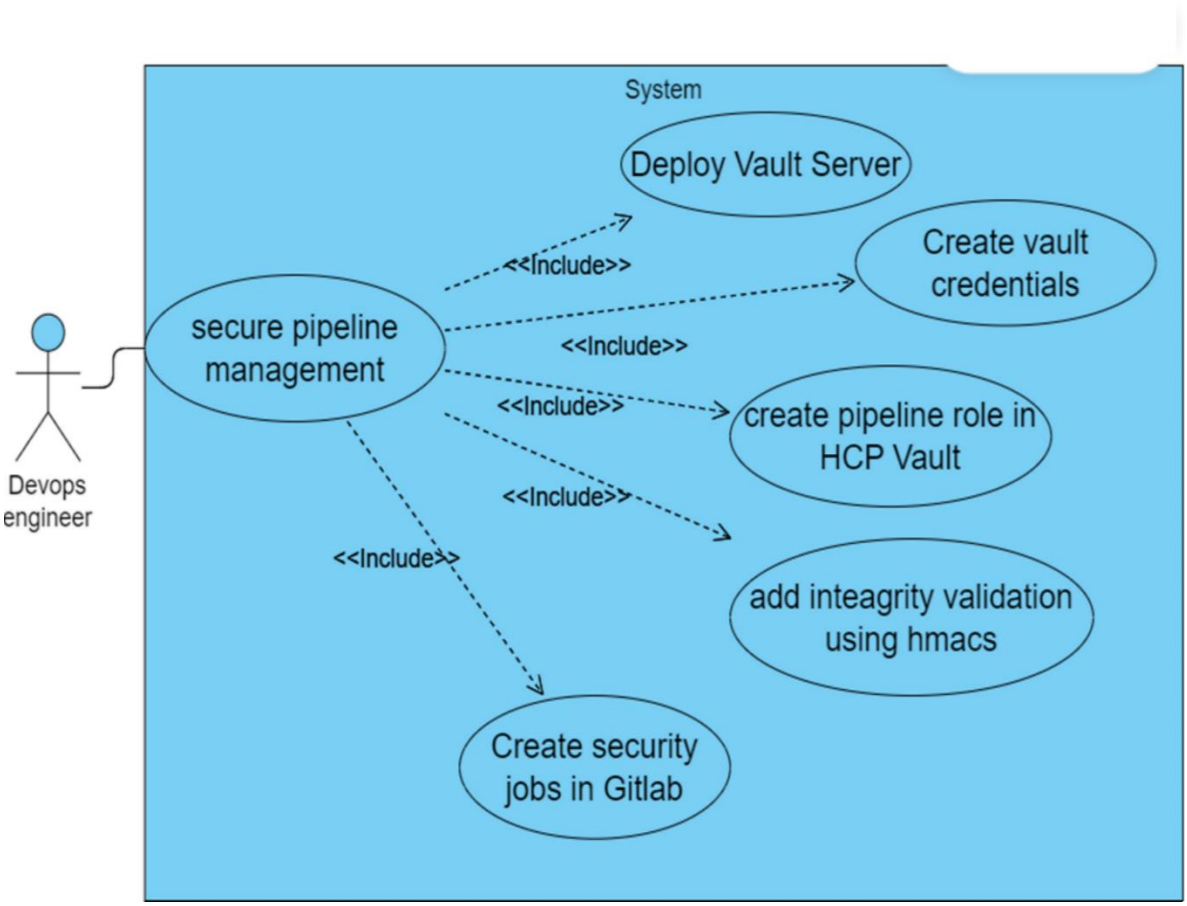


Figure 5-4 : secure pipeline management use case

5.4.2 Use case textual explanation

The table 5.2 shows the use case diagram of secure pipeline management.

Table 5-2 : Sprint 3 use case textual explanation

Case	secure pipeline management
Description	ensures that sensitive information is securely managed using HCP Vault, and the integrity and authenticity of container images are maintained using HMACs, providing a robust and secure software delivery process
Primary Actor	Devops engineer
Preconditions	• HCP Vault is set up and configured with AppRole authentication. • CI/CD pipeline is configured to authenticate with HCP Vault. • Container registry is available for storing container images and HMACs.
Main Success Scenario	• DevOps engineer commits code, triggering the CI/CD pipeline. • pipeline authenticates with HCP Vault using approle. • pipeline retrieves necessary secrets from HCP Vault. • pipeline builds the container image using these secrets. • pipeline generates an HMAC for the container image using a secret key from HCP Vault. • pipeline stores the container image and HMAC in the container registry. • Deployment environment retrieves the container image and HMAC. • Deployment environment validates the HMAC against the container image to ensure integrity and authenticity.
alternative Scenario	-If HCP Vault authentication fails, the pipeline logs the error and notifies the DevOps engineer If secret retrieval from HCP Vault fails, the pipeline aborts the build and deployment process, logging the incident and notifying the DevOps engineer. If the container image build fails, the pipeline logs the error and notifies the DevOps engineer. If HMAC generation fails, the pipeline logs the error and halts the deployment process. If HMAC validation fails during deployment, the container is not deployed, and an alert is triggered for the DevOps team to investigate
Post-conditions	Container image is securely deployed if HMAC validation is successful. Alerts are generated for any failed HMAC validations, preventing potentially compromised containers from being deployed

5.5 applying sprint 3:

In this section, we will be describing different sprint 1 implementations

5.5.1 CI-CD-SEC6 attack solution: HCP Vault AppRole

implementing a robust security framework for our CI/CD environment is crucial to protect sensitive credentials and maintain the integrity of software development process. That is why a Secret Manager with temporary tokens or keyless access, as depicted in the image, ensures that sensitive credentials are retrieved securely Implementing comprehensive audit logging and monitoring mechanisms is essential to track activities, detect anomalies, and facilitate incident response and forensic analysis The figure 5-5 illustrates more details about the secure template.

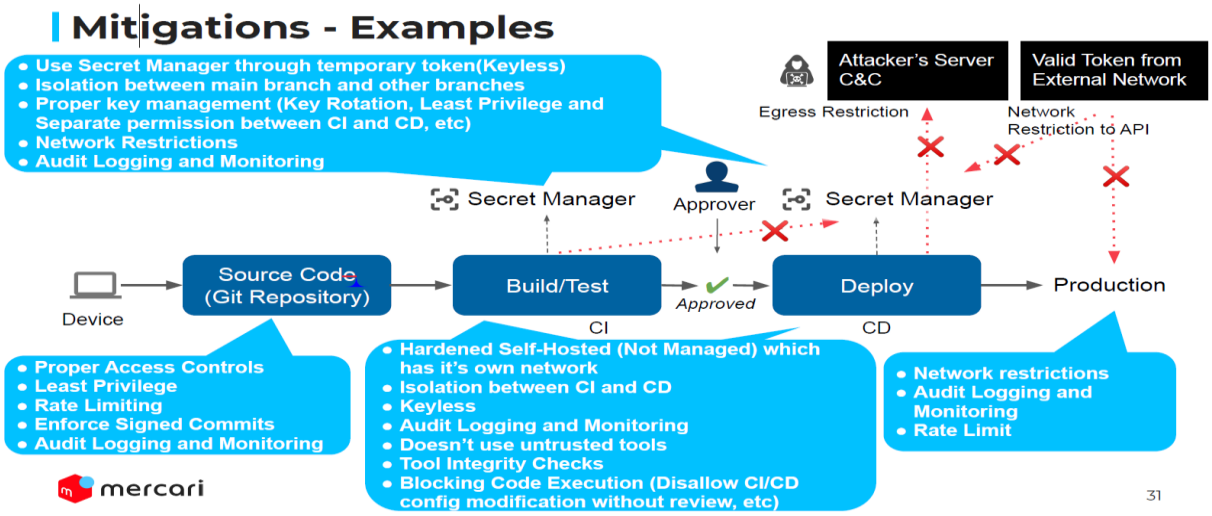


Figure 5-5 : secure pipeline template [14]

Using this new architecture, we will define our work into multiple steps for easier establishment this first one illustrates how the created secret is stored and managed in the Vault, the next steps will involve configuring access policies and authentication methods to enable applications to securely retrieve these secrets, The figure shows the interface for creating secrets or information.

The main section is titled "Create secret" and provides fields to specify the path or identifier for the secret, as well as the actual secret data itself.

The figure 5-6 illustrates more details about the dashboard import.

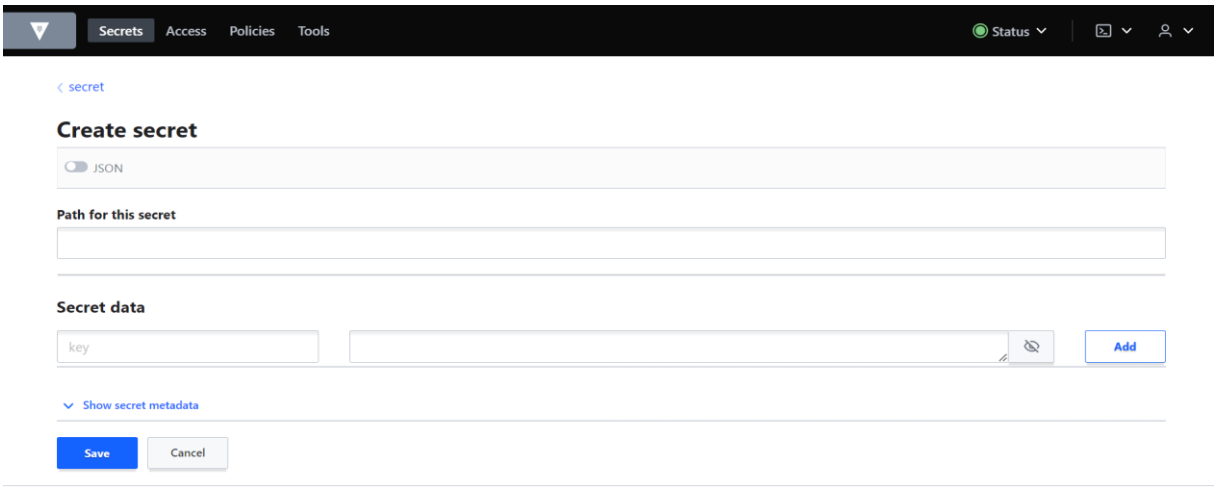


Figure 5-6 vault secrets engine interface

Let discuss how we will be configuring the Approle feature for our benefits The most essential feature of Vault is AppRole that makes it better than direct token assignment is that the credential is split into a Role ID and a Secret ID, delivered through different channels.

Further, the Secret ID is delivered to the application only at the expected time of use (usually at application startup).This pattern of authorization by using knowledge delivered just in time,

The Role ID is not sensitive and can be used for any number of instances of a given application, the Secret ID, by contrast, is intended to be access-limited so it can be used only by authorized applications it may be usable by only a single application or even a single app instance

- Step 1

We Enable the AppRole authentication method. This step involves considering role management. Determine if delegation of role management is necessary and whether the delegated party needs access to all roles. If not, enabling AppRole at multiple paths can facilitate granting permissions for managing specific sets of roles. The figure 5-7 shows the logs of the command inside the vault server

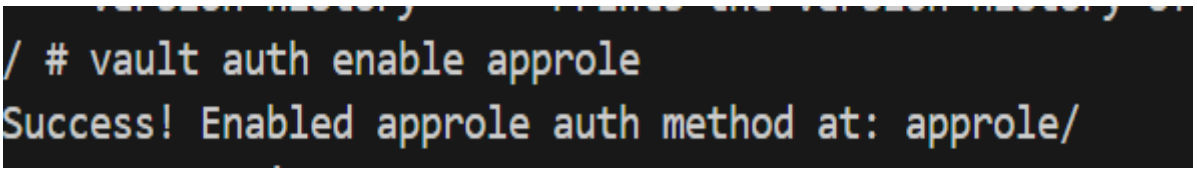


Figure 5-7 activating vault AppRole

- Step 2

We Create the role and policies for the app. A role is a reusable unit containing Vault policies that define the secrets an app can access. It can have multiple policies attached; thus, if apps share some but not all secrets, create one policy for common secrets and app-specific policies for unique needs. the process of creating an access policy in HashiCorp Vault explained in the figure. Here we define a new policy named Gitlab. Within, several paths are specified along with their respective access capabilities. The paths include **secret/data/docker**, secret/data/cosign, and secret/data/hosts, each granting read capabilities. This setup allows entities assigned this policy to read secrets stored under these paths, ensuring they cannot write or delete the data. The figure 5-8 illustrates more details about the policies.

```
/ # vault policy write gitlab -<<EOF
>
> path "secret/data/docker" {
>   capabilities = [ "read" ]
> }
> path "secret/data/cosign" {
>   capabilities = [ "read" ]
> }
> path "secret/data/hosts" {
>   capabilities = [ "read" ]
> }
>
> EOF
Success! Uploaded policy: gitlab
```

Figure 5-8 pipline policies

After saving the policy, we proceed to create a role using the policy, and specifying the token total time to limit long-term abuse. The figure 5-9 shows the result of the action.

```
/ # vault write auth/approle/role/gitlab token_policies="gitlab" token_ttl=100h token_max_ttl=400h secret_id_ttl=100h se
cret_id_num_uses=40 token_num_uses=10
Success! Data written to: auth/approle/role/gitlab
```

Figure 5-9 create gitlab pipeline role in vault

- Steps 3 and 4

Request and receive the role's Role ID. This Role ID is inserted into the pipeline later

The figure 5-10 illustrates more details about the role_id import.

```
/ # vault read auth/approle/role/gitlab/role-id
Key        Value
---        -----
role_id    c6425874-4f64-22f0-cfe7-06657a4f7ef9
```

Figure 5-10 : reading pipeline role_id

• Steps 5: we generate and deliver the Secret ID, This minimizes the duration the Secret ID is unconsumed and vulnerable to interception The figure 5-11 illustrates more details about the secret_id.

```
/ # vault write -f  auth/approle/role/gitlab/secret-id
Key                   Value
---                   -----
secret_id             27be31eb-ffa6-c413-1e48-d2e09ba39479
secret_id_accessor    2652daea-5370-5b16-6f6f-6e02c2a0b48e
secret_id_num_uses    40
secret_id_ttl         100h
```

Figure 5-11 secret_id generation

- Step 6

We Deliver the retrieved Secret ID wrapping token to the authorized application. The method of delivery can vary but we ensure it is not delivered through the same channel as the Role ID.

- Step 7

We add vault kv get commands to retrieve secrets from different paths within the vault. First, it retrieves secrets from Docker secrets or configuration. it retrieves secrets from the secret/hosts path, containing information about host configurations like IP addresses or hostnames.which is appended to the secrets.env file. The figure 5-12 illustrates more details about the job.

```
read_secrets:
  stage: read_secrets_step
  image: vault:1.13.3
  inherit:
    default: false
  before_script:
    - apk add jq
    - export VAULT_ADDR=$VAULT_ADDR_IN_NETWORK
    - export TOKEN="$(vault write -format=json auth/approle/login role_id=$VAULT_ROLE_ID secret_id=$VAULT_SECRET_ID | jq .auth |jq .client
  script:
    - echo $TOKEN | xargs vault login
    - vault kv get -format=json secret/docker |jq .data | jq .data |jq -r 'to_entries|map("\(.key|ascii_upcase)=\(.value|tostring)")|.[]'
    # - vault kv get -format=json secret/cosign |jq .data | jq .data |jq -r 'to_entries|map("\(.key|ascii_upcase)=\(.value|tostring)")|.[]
    - vault kv get -format=json secret/hosts |jq .data | jq .data |jq -r 'to_entries|map("\(.key|ascii_upcase)=\(.value|tostring)")|.[]' >
    - vault kv get -format=json secret/ovh |jq .data | jq .data |jq -r 'to_entries|map("\(.key|ascii_upcase)=\(.value|tostring)")|.[]' >>
    - echo COSIGN_PRIVATE_KEY_BASE64=$(vault kv  get -format=json secret/cosign |jq .data.data.COSIGN_PRIVATE_KEY_BASE64) >> secrets.env
    - cat secrets.env
  artifacts:
    reports:
      dotenv: secrets.env
```

Figure 5-12: read_secrets job

- Step 8

define a trigger for the "upload" stage, which consists of building, deploying, and running a pipeline.

The trigger stage includes a list of variables, which is environment variables or credentials required for the pipeline execution. some include:

1. `DOCKER_PASSWORD` and `DOCKER_USER`: Credentials for authenticating with a Docker registry.

2. `SERVER_HOST`: The hostname or IP address of a server or target environment.

3. `SSH_PRIVATE_KEY_BASE_64`: An SSH private key encoded in Base64 format, likely used for securely accessing remote servers or repositories.

4. `OVH_CONSUMER_KEY`, `OVH_APPLICATION_SECRET`, and `OVH_APPLICATION_KEY`: Credentials for interacting with OVH (a cloud service provider) services or APIs.

The trigger itself includes a child pipeline defined in a separate file (`child-pipeline.yml`) using the `include` directive. The `strategy` is set to `depend`, indicating that the child pipeline is dependent on the parent pipeline's execution. The figure 5-13 illustrates more details about the job.

```yaml
trigger_upload:
  stage: triger_build_deploy_pipeline
  variables:

    DOCKER_PASSWORD: "${DOCKER_PASSWORD}"
    DOCKER_USER: "${DOCKER_USER}"
    SERVER_HOST: "${SERVER_HOST}"
    SSH_PRIVATE_KEY_BASE_64: $SSH_PRIVATE_KEY_BASE_64
    OVH_CONSUMER_KEY: $OVH_CONSUMER_KEY
    OVH_APPLICATION_SECRET: $OVH_APPLICATION_SECRET
    OVH_APPLICATION_KEY: $OVH_APPLICATION_KEY

  trigger:
    include:
      - local: child-pipeline.yml
    strategy: depend
  needs:
    - job: read_secrets
      artifacts: true
```

Figure 5-13 trigger job

- Step 9

After creating the job script, we proceed to gitlab logs to see the token creation for given pipeline execution . The figure 5-14 illustrates more details about the job logs.

```
26 $ export TOKEN="$(vault write -format=json auth/approle/login role_id=$VAU
   LT_ROLE_ID secret_id=$VAULT_SECRET_ID | jq .auth |jq .client_token)"
27 $ echo $TOKEN | xargs vault login
28 Success! You are now authenticated. The token information displayed below
29 is already stored in the token helper. You do NOT need to run "vault logi
   n"
30 again. Future Vault requests will automatically use this token.
31 Key                    Value
32 ---                    -----
33 token                  hvs.CAESIBqyCj84LZwV3kt5dEVQ-GFZ7o2L5xfQTySeRP3xF-
   o_Gh4KHGh2cy54bjgyNnRMOWFCZ1JrZmw3UElRV1pJdEQ
34 token_accessor         2tq9OaQ6xp3OMnixZ0im9iSM
35 token_duration         99h59m59s
36 token_renewable        true
37 token_policies         ["default" "gitlab"]
38 identity_policies      []
39 policies               ["default" "gitlab"]
40 token_meta_role_name   gitlab
```

Duration: 20 seconds
Finished: 1 week ago
Queued: 0 seconds
Timeout: 1h (from proje
Runner: #12270852 (JI
green.saas-lin
amd64.runner
manager.gitlab

Job artifacts
These artifacts are the l
not be deleted (even if
newer artifacts are avai

Commit 0c59d951
test

Pipeline #1301682621

Figure 5-14 :read_secrets job logs

The figure 5-15 shows the final pipeline architecture after these adjustment

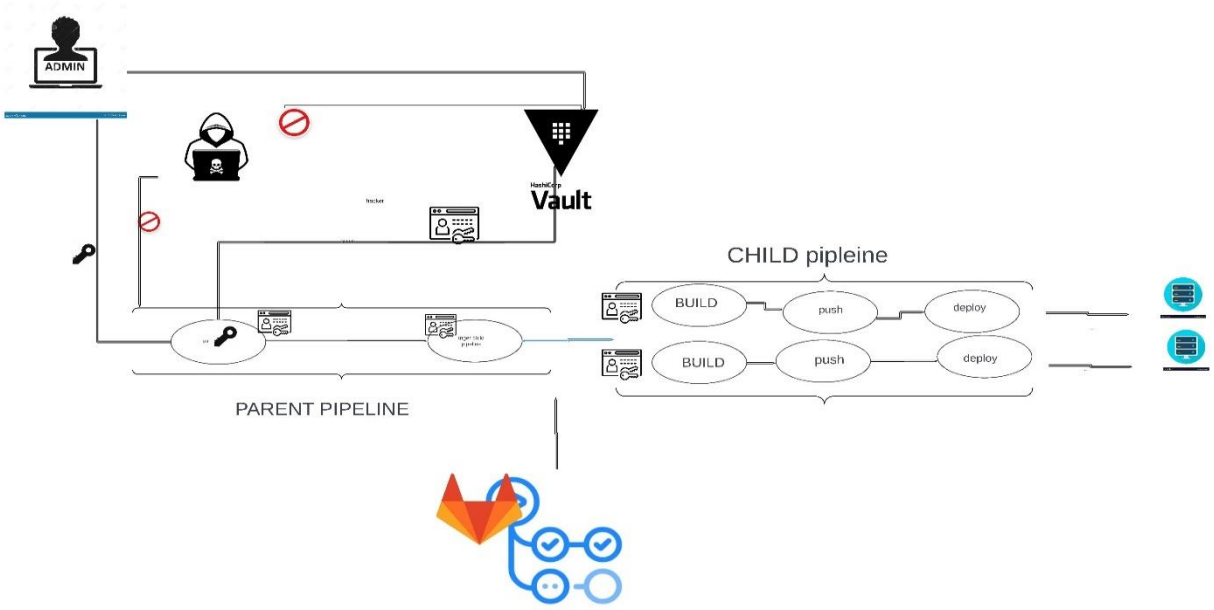


Figure 5-15 pipeline new architecture

5.5.2 Leveraging HMAC for Secure Artifact Integrity

HMAC, which stands for Hash-based Message Authentication Code or Keyed-hash Message Authentication Code, is used to ensure the authenticity and integrity of transmitted data.

The HMAC algorithm enhances security by combining cryptographic keys with hashing. Hashing transforms data into a fixed-size output, maintaining determinism by producing the same hash for identical inputs. A crucial property of hashing is its one-way nature, making it impossible to reverse-engineer the original data from the

hash, which is ideal for ensuring data integrity. When transmitting data, both the data and its hash are sent, allowing the recipient to verify integrity by re-hashing the received data and comparing it to the provided hash.

HMAC extends this concept to provide authenticity in addition to integrity. It achieves this by incorporating a secret key into the hashing process. First, HMAC combines the data with the cryptographic key and creates an initial hash. This hash is then combined again with the key to produce a second hash, which is transmitted alongside the data. This two-step process, utilizing the same secret key for both generation and verification, enables HMAC to simultaneously verify the authenticity and integrity of the received data, as only parties possessing the correct shared key can produce or validate the HMAC. Figure 5-16 illustrates more details about the algorithm.

Figure 5-16 hmacs verification algorithm [16]

In the "Build/Test" stage, the pipeline retrieves the source code and performs necessary build operations and automated testing using a hashing key specifically designated for this phase. In the "Deploy" stage, the pipeline uses the credentials to verify the authenticity of the artifacts so the attacker would still not have access to the deployment container to mess with them. Figure 5-17 shows the illustration of this patches

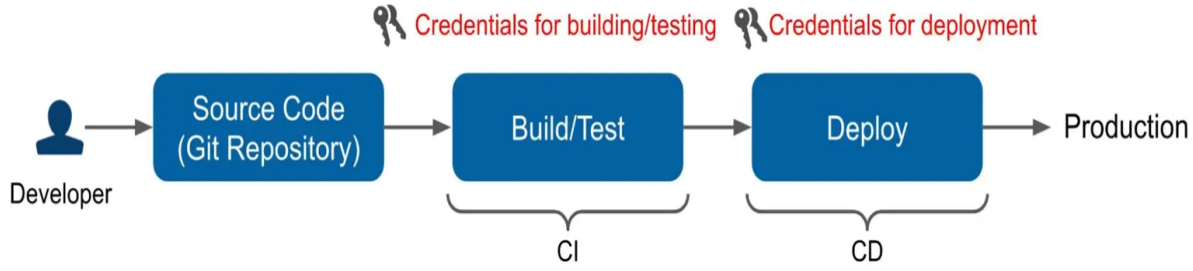


Figure 5-17 pipeline patches

Bringing theory to practice, the digest will be generated and saved in a text file to be later on verified using the Seceret_key retrieved from the vault server(figure below)

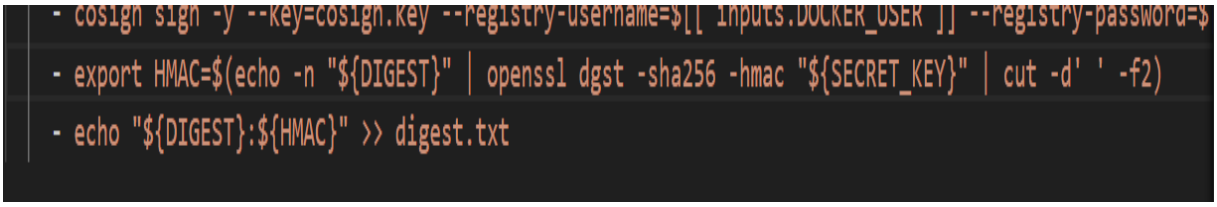


Figure 5-18 : hashing job

In the deployment job we check for any invalid signature using the hmac generated in the build stage . The figure 5-19 illustrates more details about the job.

```
- export container_port=$(docker ps | grep ${CONTAINER_NAME} | sed 's/.*127.0.0.1://g
-  export CALCULATED_HMAC=$(echo -n "${DIGEST}" | openssl dgst -sha256 -hmac "${SECRE
- if [ "${HMAC}" == "${CALCULATED_HMAC}" ]; then
-   echo "Image integrity verified. Deploying..."
-   docker pull myimage:${CI_COMMIT_SHORT_SHA}
-   # Deploy the image
- else
-   echo "Image integrity check failed. Aborting deployment."
-   exit 1
- fi
```

Figure 5-19: verifying hash

Finally, figure 5-20 shows the Gitlab logs, indicate that the signature was verified successfully and the image ‘s integrity is not conserved

```
   ry-password=$DOCKER_PASSWORD $IMAGE_NAME
30 Verification for index.docker.io/abdelbaki1/features:dj
   dd-gitlab-templates-latest --
31 The following checks were performed on each of these si
32   - The cosign claims were validated
33   - Existence of the claims in the transparency log was
34   - The signatures were verified against the specified
35 [{"critical":{"identity":{"docker-reference":"index.doc
   eatures"},"image":{"docker-manifest-digest":"sha256:195
```

Figure 5-20 : pipeline logs

Conclusion

This last chapter deals with the realization of our work, it allowed us to present the hardware and software environment of our project, it also allowed us to present the different technologies that we used for the completion of this work, and finally, we exhibited some graphical interfaces relating to our project.

General Conclusion

This end-of-studies internship carried out within DOCIC aims to implement a solution based on the GitLab tool that allows developers and admins of the team to automate the production cycle easily and very quickly, without being confronted with complicated configurations and tasks. To achieve this solution, we started by implementing the pipeline steps like dockerizing the applications, by creating docker images and storing them in a private repository which allowed us to better understand our application and therefore have a clearer idea of the containerized architecture to implement.we focused on adding monitoring to the pipeline and eliminate any manual configs using ansible, To enhance the solution, we integrated monitoring tools such as Prometheus and Grafana into the pipeline

The last part of the project aimed to secure the pipeline, helping developers to improve against the most notorious ci/cd vulnerabilities such as injection and unauthorized access.DevSecOps methodology. It allowed me to easily integrate into collaborative work and to face difficulties such as task distribution and time and effort.

As we look to the future, the integration of DevOps practices could go right along with AI-driven technologies. AI can enhance automation, predictive analytics, and anomaly detection, making the CI/CD pipelines more resilient and secure. Emphasizing the use of machine learning models, organizations can better adapt to changing market demands and technological advancements.

To conclude, this internship was very enriching technically and professionally which allowed me to practice many concepts such as networking; security, Dockerisation etc

Bibliography

[1] "DevOps vs Agile: Understanding the Key Difference," 06 2024. [Online]. Available: https://radixweb.com/blog/agile-vs-devops.

[2] "Scrum Workflow - A Step by Step Guide," 06 2024. [Online]. Available: https://www.sprintzeal.com/blog/scrum-workflow.

[3] "An Ultimate Guide to DevOps – Principles, How it works, and Real-life Examples," 06 2024. [Online]. Available: https://www.peerbits.com/blog/ultimate-guide-to-devops-principles-woks-and-examples.html.

[4] link.springer.com, "Continuous Integration Flows," 06 06 2024. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-11283-1_9.

[5] jhall.io, "You're probably using the wrong definition of Continuous Integration," 06 06 2024. [Online]. Available: https://jhall.io/archive/2021/02/18/youre-probably-using-the-wrong-definition-of-continuous-integration/.

[6] Martin Fowler, "Continuous Integration," 18 1 2024. [Online]. Available: https://www.martinfowler.com/articles/continuousIntegration.html.

[7] "What is HashiCorp Vault and How it works? An Overview and Its Use Cases," 06 06 2024. [Online]. Available: https://www.devopsschool.com/blog/what-is-hashicorp-vault-and-how-it-works-an-overview-and-its-use-cases/.

[8] ovhCloud, "What is OVHcloud?," 06 06 2024. [Online]. Available: https://www.ovhcloud.com/en/about-us/.

[9] medium, "What is Ansible & how to Install and Configure Ansible on RHEL 9," 06 06 2024. [Online]. Available: https://medium.com/@priyajoshi03082003/what-is-ansible-how-to-install-and-configure-ansible-on-rhel-9-bdac3168de14.

[10] "Introduction to Docker," 06 2024. [Online]. Available: https://sebastien-sime.medium.com/introduction-to-docker-e748624a0069.

[11] linkedin, "Understanding GitLab Runner," 06 2024. [Online]. Available: https://www.linkedin.com/pulse/understanding-gitlab-runner-nikila-fernando/.

[12] promlabs, "Query Language," 06 06 2024. [Online]. Available: https://training.promlabs.com/training/introduction-to-prometheus/prometheus-an-overview/query-language/. [Accessed 06 06 2024].

[13] "Ansible for Beginners | Overview | Architecture & Use Cases," 06 2024. [Online]. Available: https://k21academy.com/ansible/ansible-for-beginners/.

[14] speakerdeck.com, "Attacking and Securing CI/CD Pipeline," 06 06 2024. [Online]. Available: https://speakerdeck.com/rung/cd-pipeline.

[15] opsmx.com, "DevOps vs. DevSecOps: Similarities, Differences, and the Right Time to Transition from One to the Other," 06 06 2024. [Online]. Available: https://www.opsmx.com/blog/devops-vs-devsecops-similarities-differences-right-time-to-transition/.

[16] .encryptionconsulting, "Understanding Docker Image Signing," 06 06 2024. [Online]. Available: https://www.encryptionconsulting.com/docker-image-signing/.

Summery

The project "Secure and Automate CI/CD Pipeline with Enhanced Monitoring" at DOCIC addresses key software development challenges by implementing a CI/CD pipeline using GitLab, Docker, and Ansible. It integrates Prometheus and Grafana for monitoring, and HashiCorp Vault for secure secret management. The solution aims to improve deployment efficiency, reduce manual errors, and enhance security, ultimately optimizing DOCIC's software infrastructure management

Keywords: CI/CD pipeline, GitLab, Docker, Ansible, Prometheus, Grafana, HashiCorp Vault, automation, monitoring, security, software development.

Résumé

Le projet "Sécuriser et automatiser le pipeline CI/CD avec une surveillance améliorée" chez DOCIC répond aux principaux défis du développement logiciel en mettant en place un pipeline CI/CD utilisant GitLab, Docker et Ansible. Il intègre Prometheus et Grafana pour la surveillance, et HashiCorp Vault pour la gestion sécurisée des secrets. La solution vise à améliorer l'efficacité des déploiements, à réduire les erreurs manuelles et à renforcer la sécurité, optimisant ainsi la gestion de l'infrastructure logicielle de DOCIC.

Mots clés : Intégration continue, monitoring, pipeline.

ملخص

هذا التقرير يقدم ملخص للعمل الذي تم تنفيذه في إطار مشروع التخرج بالمدرسة الوطنية العليا للمهندسين بتونس للحصول على شهادة الهندسة الوطنية في علوم الحاسوب. تم تنفيذ المشروع داخل شركة DOCIC,باستعمال Gitlab وAnsible. Prometheus وGrafana وHashiCorp Vault، لإدارة الأسرار بأمان. تهدف الحلول إلى تحسين كفاءة النشر، وتقليل الأخطاء اليدوية، وتعزيز الأمان، مما يعزز إدارة البنية التحتية للبرمجيات في DOCIC.

الكلمات المفتاحية: CI/CD pipeline, GitLab, Docker, Ansible, Prometheus, Grafana