Project Report: DevOps and Continous Deployment

Table of Contents

[I. Introduction and Background 1](#_Toc187802073)

[1. Project objectives 1](#_Toc187802074)

[2. Architecture and technical choices 2](#_Toc187802075)

[II. Project Structure 2](#_Toc187802076)

[III. Phased achievements 2](#_Toc187802077)

[Part 1 2](#_Toc187802078)

[1. Architecture and Toolchain 2](#_Toc187802079)

[2. Jenkins Initial Setup and Launch Script 5](#_Toc187802080)

[3. Build and deploy the application locally 5](#_Toc187802081)

[4. Deploying on Kubernetes 6](#_Toc187802082)

[Part 2 8](#_Toc187802083)

[1. Setup of environment: Prometheus, Grafana 8](#_Toc187802084)

[2. Exposing the Tools through Kubernetes 9](#_Toc187802085)

[3. Alert Manager testing 11](#_Toc187802086)

[Part 3 : Grafana-Loki 12](#_Toc187802087)

[1. Installation du chart Helm Grafana/Loki 12](#_Toc187802088)

[2. Log Verification 13](#_Toc187802089)

[Conclusion 15](#_Toc187802090)

# Introduction and Background

We produced this report as part of a graduation project, mobilizing our skills acquired in the field of software development and DevOps practices. This project allowed us to design and implement a complete CI/CD pipeline that can automatically manage the lifecycle of an application, from its construction to its deployment in production. The different stages of this work were carried out with contemporary tools such as Docker, Kubernetes and Jenkins, aligned with current standards.

## Project objectives

The objective of this project is to set up a complete CI/CD pipeline to:

1. Build, test, and deploy an app written in Go, accessible on the /whoami endpoint with a customization displaying the team name.
2. Perform a gradual deployment of the application on a Kubernetes cluster ("development" environment, then "production") by guaranteeing automated validation tests.

## Architecture and technical choices

The tools and technologies selected for this project are:

1. **Docker** : For creating container images and deploying locally.
2. **Kubernetes (Minikube):** For container orchestration and management of "development" and "production" environments.
3. **Jenkins** : As a CI/CD platform to automate the deployment process.
4. **GoLang** : Application programming language, used for code, build, and testing.
5. **Prometheus** : For monitoring and alerting (integration possible).
6. **Grafana:** For visualization of the metrics collected by Prometheus
7. **Loki:** For centralizing and consulting application logs

The use of these tools ensures an efficient, adaptable, and DevOps-compliant continuous integration and continuous deployment chain.

# Project Structure

Here are the descriptions of the main folders and files of the project:

1. **Dockerfile** : File to build the Docker image of the Go application.
2. **Jenkinsfile** : A script describing the Jenkins pipeline for building and deploying the application.
3. **k8s-config/** : Contains the YAML deployment files for Kubernetes (development and production environments).
4. **scripts\_project/** : Here we have the main bash scripts to run to configure the virtual machine with all the installations and most of the tool configurations. The rest are mostly utility files for their execution. These scripts are:
   1. *setup\_project.sh*: The first file to run to initialize the VM
   2. *script\_Part1.sh:* To be initiated to run the pipeline for part 1 of the policies.
   3. *Script\_Part2.*SH: To be launched to set up the Prometheus Alert Manager linked to Grafana
   4. *script\_Part2&3.sh*: Run to install and run the Grafana/Loki log tool
5. **webapi/** : Contains the source code of the Go app and its unit tests.

# Phased achievements

## Part 1

### Architecture and Toolchain

In terms of tools, we will use what we mentioned in the technical choices, except for the log tools: Prometheus, Grafana and loki.

The following is the planned pipeline architecture diagram:

Une image contenant texte, capture d’écran, diagramme, conception

Description générée automatiquement

### Jenkins Initial Setup and Launch Script

#### Jenkins Preparation

To start Jenkins, we used a pre-configured Docker container (launched in setup\_project.sh). Then we carried out the following configuration steps in the interface:

1. **Installation of the required plugins** : the "Docker Pipeline" plugin to manage containers more easily.
2. **Configuration de l'agent Jenkins** :
   1. Created a Jenkins agent named "Jenkins-slave" that is configured to run pipeline tasks.
3. **Creating the "DevOps\_project\_pipeline" pipeline with these configurations** :
   1. Pipeline linked to the url of our git as well as its Jenkinsfile
   2. Credentials settings: added information from the "efrei\_2023" Docker Hub used in the previous labs to push the built images.
   3. Added the option to launch remotely via the CRUMB to be able to do it with the script\_Part&.sh file: the token is included in this script.

#### Script script\_Part1.sh

1. This script automates several crucial steps and is to be launched right after setup\_project.sh:
2. Checks the necessary imports and functions (written in utils.sh).
3. Retrieves the initial password from Jenkins and displays it: the script pauses while you copy it into the Jenkins interface and configure it (it restarts when you press a key).
4. Retrieves the Jenkins agent created before to run pipelines, and updates it as needed.
5. Initiates the main pipeline via a token and a REST request to the Jenkins API.
6. To launch it, it is necessary to change the following variables according to its local configuration:
7. DIR\_FOR\_AGENT: The agent's workspace that changes according to user preferences. To be put in a folder with permissions open to Jenkins as the working directory.
8. AGENT\_KEY: The key identifying the agent

The following variables can optionally be changed if needed: TOKEN\_NAME, JOB\_NAME, AGENT\_NAME, JENKINS\_URL.

### Build and deploy the application locally

#### Building the Docker image

The pipeline starts by cloning the Git repository containing the project, and then uses the Dockerfile to build the application image. Its main steps include:

1. **Definition of the working environment** : Creation of a "webapi" directory for the code.
2. **Dependency Management** : Installation of the required Go modules via go mod download.
3. **Compilation and execution** : The go build command handles compilation, and execution is done on port 81.

From the Dockerfile we can then build the image.

#### BONUS : Buildpack

We were able to quickly try the implementation of the buildpacks, with these steps:

1. Added commands to install the "pack-cli" buildpack program in setup\_project.sh
2. Choice of builder: several builders were compatible with Go, but after testing two of them we finally decided that Google's (the creator of Go) was the most suitable: the rest of the game proved us right.
3. Added the build step of the image in the pipeline by temporarily replacing the classic build.

In our case, the build and the pipeline went well overall but we didn't get the application in the browser at the end: we assume that it was missing configuration information such as the port to expose (to put in a config.toml...), as the application is on port 81. We were still able to note these differences compared to the classic docker build:

1. No need to write a Dockerfile, a good part of the build actions are automated and the image can be manipulated by Docker without any problem afterwards.
2. However, the build is slower and takes up more space. In addition, builders may have difficulty if they have to combine multiple languages in a project without coordination instructions.

#### Unit Testing

Before any deployment, the pipeline runs unit tests via the main\_test.go file, which we added ourselves, after launching the container locally. This ensures the stability of the application. Note that the "/whoami" endpoint is customized to return the team name, verified via an HTTP request sent by one of these local tests.

Finally after the tests, we push the image to the Docker Hub "efrei\_2023" for the rest.

### Deploying on Kubernetes

#### Deployment in a "Development" environment

The pipeline uses the dev-deployment.yaml file to deploy the application to Kubernetes. The actions carried out are:

1. **Initialization** : Creation of the "development" and "production" namespaces if non-existent.
2. **Configuration enforcement** : The deployment and services are created using kubectl apply. After a wait for this creation, Minikube then applies the service configuration to expose the application.
3. **Validation** : The pipeline runs tests on the endpoint "/healthcheck" and the root "/" to verify that the application is functional.

#### Transition to the "Production" environment

After the previous step is committed, the pipeline deploys the application to production via prod-deployment.yaml. We apply a configuration adapted to the "production" namespace, with more replicas.

So here is the proof of the deployed application:

Une image contenant logiciel, texte, Logiciel multimédia, ordinateur

Description générée automatiquement

## Part 2

### Setup of environment: Prometheus, Grafana

#### Prometheus setup

The first step involved adding Helm repositories for Prometheus and Grafana, which offer pre-packaged charts for easy deployment of monitoring services. The repositories were updated to ensure access to the latest versions.

Prometheus has been installed in the "production" namespace via a Helm chart (the same namespace of the app). Prometheus collects performance metrics and exposes that data for use by other tools like Grafana for visualization.

Une image contenant texte, capture d’écran, menu, Police

Description générée automatiquement

#### Grafana setup

Une image contenant capture d’écran, texte

Description générée automatiquement Grafana, a powerful visualization platform, was deployed via its Helm chart. For simplicity, a default administrator password was used during the installation. Grafana connects to Prometheus to access the performance metrics and build dashboards for visual representation.

### Exposing the Tools through Kubernetes

#### Pre-configuration of Prometheus

**Configuring Prometheus Alerts**

Une image contenant texte, capture d’écran, menu, Police

Description générée automatiquementPrometheus alerting rules were set up to monitor specific conditions, such as unexpected performance anomalies or breaches of predefined thresholds. These rules help identify critical situations requiring immediate attention.

**Configure AlertManager**

Une image contenant texte, capture d’écran, menu, Police

Description générée automatiquementAlertManager was configured to handle the alerts generated by Prometheus. This included setting up the communication channels for notifications, such as email or Slack. For testing purposes, a sample alert was created via the Prometheus interface, while AlertManager was configured to send email alerts using Gmail’s SMTP server.

#### Service Exposure

Prometheus and Grafana were exposed locally using port-forwarding to provide access to their interfaces on specific ports:

1. **Prometheus:** Accessible at http://localhost:9090, offering insights into metrics and alert configurations.
2. **Grafana:** Available at http://localhost:3000, presenting an intuitive interface to construct dashboards using the collected data.

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

### 

#### Verification of exposure

##### Prometheus

The Prometheus interface was successfully accessed, verifying its functionality and the configured alerts.**Une image contenant texte, capture d’écran, logiciel, Page web

Description générée automatiquement**

##### Grafana

**Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Description générée automatiquement**Upon logging into Grafana for the first time with default credentials, the platform requested a password change. After this, the application’s main menu became accessible, allowing users to add data sources and build custom dashboards.

### Alert Manager testing

To validate AlertManager's performance, we tested it by deliberately deleting a Prometheus deployment using kubectl delete deployment. This action triggered an alert, which was displayed in the Prometheus interface.Une image contenant texte, capture d’écran, logiciel, Icône d’ordinateur

Description générée automatiquement

However, despite configuring Gmail’s SMTP settings correctly and according with the provided documentation, email alerts were not successfully delivered, returning an error: "Username and password not accepted." After several adjustments, manual testing of the SMTP server using OpenSSL commands confirmed that authentication was functioning, but the integration with AlertManager faced persistent issues.Une image contenant texte, capture d’écran, Police

Description générée automatiquement

Here is also a manual test at the smtp server, which worked, with the following commands:

‘’’

openssl s\_client -starttls smtp -crlf -connect smtp.gmail.com:587

AUTH LOGIN

<base64\_encoded\_username>

<base64\_encoded\_password>

‘’’

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

## Part 3 : Grafana-Loki

In this part of the project, we set up a log management system using Grafana and Loki, deployed in a Kubernetes namespace. The objective was to retrieve the logs of the "production" namespace where the application was running and to explore filtering by lines containing the word "error". So we ran the "script\_TP3.sh" file.

Initially, we wanted to try to reapply the method given in practical work. But even if we had followed the steps better, for example by retrieving the right link for the Loki datasource, we still had no logs in Grafana. After several searches in the documentation, we finally decided to use the grafana/loki-stack service instead of loki.

### Installation du chart Helm Grafana/Loki

We started by adding the official Grafana Helm repository and then updated the available charts. Next, we generated a custom configuration file loki-custom-values.yaml to adapt the deployment settings to our environment, specifically to restrict the logs to the "Production" namespace. The file has been reworked from the one provided by a command placed as a comment in the script.

We used this file to install Loki Stack in the "grafana-loki" namespace. Grafana's and Loki's pods were checked to make sure they were working properly.

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

### Log Verification

**Checking general logs**

Once the system was in place, we went into the grafana interface of the grafana-loki component. Here the loki datasource was already configured by default because it had been done in the loki-custom-values.yaml file.

Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Description générée automatiquement

To test the system, we first retrieved the general logs (all unfiltered logs) from the "Production" namespace via Grafana's query interface. This step allowed us to confirm that Loki was collecting logs and that the filter on the namespace was working as expected.

*Une image contenant capture d’écran, logiciel, texte, Logiciel multimédia

Description générée automatiquement*

**Searching for logs containing "error"**

After confirming the general collection of logs, we refined our query to look only for logs containing the word "error" in the "Production" namespace. This query was configured directly in the Grafana interface using the Loki query language (LogQL).

However, no logs containing "error" were found, which is a good indicator for our production environment.

Une image contenant capture d’écran, texte, logiciel, Logiciel multimédia

Description générée automatiquement

# Conclusion

This project successfully implemented a robust CI/CD pipeline and monitoring setup using state-of-the-art DevOps tools. The automated pipeline ensured seamless application building, testing, and deployment across development and production environments, utilizing Docker, Kubernetes, and Jenkins.

The monitoring stack with Prometheus and Grafana provided actionable insights into application performance and stability. Despite some challenges with email alerts in AlertManager, local alert configurations and notifications were effectively tested and verified.

The integration of Grafana-Loki enabled centralized log management, confirming the absence of critical errors in the production environment. This step reinforced the reliability of the deployed application while offering a scalable solution for log analysis.

In conclusion, this project demonstrated the value of combining CI/CD pipelines with comprehensive monitoring and logging frameworks to achieve efficient and resilient application management. The methodologies and tools applied here can serve as a reference for future projects aiming to adopt DevOps practices.