Project Report – Group 9

Eren Aydoslu (4997778)
Daniel Chou Rainho (5336643)
Giovanni Fincato de Loureiro (4926854)
Samuel Ribaric (6119115)

1 ABSTRACT

This report talks about the comprehensive process of developing a robust machine learning pipeline application. The project uses modern DevOps practices so that the CI/CD pipelines are seamless. We integrated Docker, Ansible, Poetry, etc. for an efficient and consistent deployment of the application. Each service is designed in a way that makes modularity, scalability, and maintainability better. Critical parts of the process (like building, deploying, testing) are automated to minimise human error during the process. Along with the documentation of the setup, this report reflects on the current state of the application as well as the strategies used for test coverage. The report also proposes enhancements to address the shortcomings of these strategies. These proposed improvements are meant to improve the quality and performance of the entire application as a whole.

2 INTRODUCTION

The purpose of this report is to give a comprehensive overview of the development process, architectural design, and operational strategies used to create our multi-service software project. This project has multiple interconnected services: app-service, app-frontend, model-service, lib-ml, etc., each playing a individually important role in delivering a cohesive and functional application.

Modern software development robust coding practices and also efficient and reliable deployment mechanisms. For these requirements to be met, our project integrated contemporary DevOps tools and practices, such as continuous integration and deployment (CI/CD), Docker for containerization, and Ansible for configuration management. These tools collectively ensure that the software can be developed, tested, and deployed easily across various environments.

The report begins by outlining the Release Pipeline Documentation. It then transitions into Deployment Documentation to document the final deployment of our application. We then reflect on the current state of our project and identify the points that we find the most critical or error prone. We then document how our pipeline is set up and the decisions we had to make to configure
our project.

Furthermore, the report addresses the limitations of the current testing strategy, identifying areas that are prone to errors or insufficient coverage. Suggestions for enhancing the testing framework are provided, emphasizing the need for broader test coverage and the inclusion of performance testing.

This report aims to serve as a detailed guide for understanding the development and deployment processes of the project, so that we can offer insights into the decisions made and challenges we encountered. It also provides recommendations for future improvements in the project, ensuring that the project can improve to meet higher standards of quality and reliability. Our report aspires to contribute valuable knowledge to the field of software development, particularly in the realms of DevOps and continuous delivery.

3 RELEASE PIPELINE

3.1 *lib-ml* Python Package

Our release pipeline deploys the *lib-ml* Python package directly to the Python Package Index (PyPI), the primary repository for Python libraries. This platform was chosen for its straightforward integration. We initially thought of releasing the package on GitHub Packages, but unfortunately found out that this one does not have a Python registry.

The pipeline is triggered by the addition of version tags that adhere to semantic versioning, such as `v1.0.1`. This activation starts the automated process on an Ubuntu server using Python 3.10.

Here's a simplified description of the workflow:

- Initially, the repository code is checked out, and the Python environment is prepared with the necessary settings and tools, primarily using Poetry, a tool for Python project management and packaging.
- The pipeline then verifies the consistency of version numbers, ensuring that the version specified in the project's `pyproject.toml` file matches the version in the Git tag. This step is crucial for maintaining version accuracy across the development and distribution phases.
- Finally, if the versions match, the package is built and uploaded to PyPI using a secure API token. This ensures that the package is available for installation via pip by any user globally.

3.2 *model-service* Docker Image

The second pipeline builds and releases a docker dontainer image for model-service. It starts off similar to the first pipeline, with code checkout and setting up the environment. After the environment is set up, the code quality checks are carried out to ensure coding standards (flake8, pylint). Unit testing is lastly done.

The docker image is then built, creating an image with the dockerfile. Once the image is built, the docker image is pushed to a container registry. The pipeline can also have a deploy to kubernetes step where the docker image is deployed to a Kubernetes cluster. Data flow in this pipeline includes source code, dockerfile, configurations, liting reports, test reports, and docker image.

4 DEPLOYMENT

The (CI/CD) pipeline for the project (including the app-service, app-frontend, model-service, etc.) automates the process of building

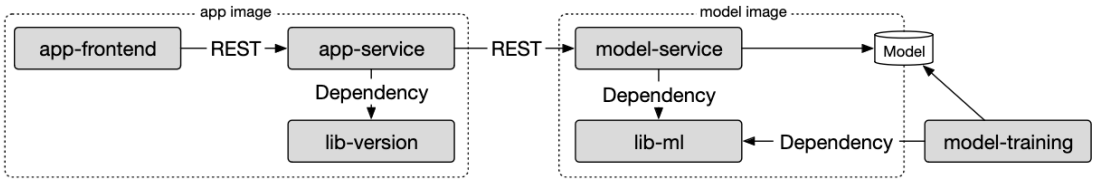

Figure 1: Deployment Overview

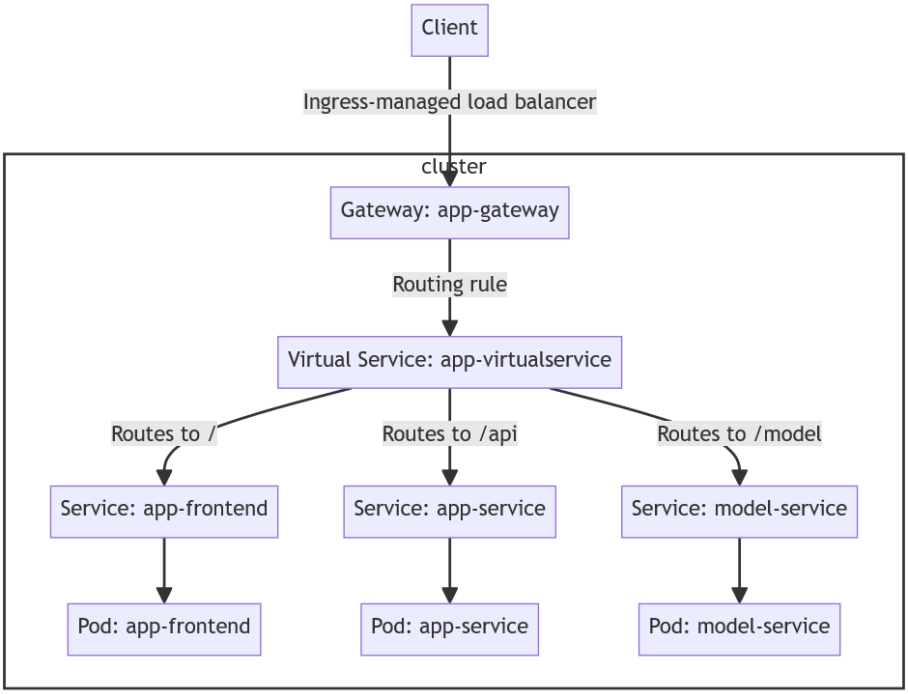

Figure 2: Kubernetes Architecture

and testing the components, as well as deploying them. It ensures consistency, reliability, and efficiency across the whole application. The main components of the pipeline are the Continuous Integraion for code verification and Continuous Deployment to automate deployment for all the different services involved.

We use Docker for creating the consistent and isolated environments, as well as ansible for managing configurations and Poetry for managing the python dependencies.

The design of he pipeline is to maintain modularity, and has the distinct stages throughout so that each service is responsible for particular tasks (including building, testing, deploying, cleaning). The modularity makes it easier to maintain the application as well as providing scalability. Ansible is used for automated configuration management with the Ansible playbooks, so there are reduced manual errors and consistency is better. Docker makes the apps run in consistent environment throughout all the different pipeline stages.

The steps of the pipeline itself beings with code checkout and the code is pulled from the repository (for all the services). Docker images are created using the Dockerfiles in the build for each service. The images have the application code and all dependencies. The test stage runs unit and integration tests inside Docker containers to check code correctness. The configure environment stage uses Ansible to set up all the dependencies.

In Figure 2 we have illustrate our Kubernetes architecture for serving the requests. The client requests are directed to an ingress-managed load balancer, which distributes traffic within the cluster. The initial entry point within the cluster is managed by a gateway, named 'app-gateway'. This gateway directs the incoming traffic to a virtual service named 'app-virtualservice', which applies routing rules to determine the appropriate service based on the request path.

The virtual service directs traffic to three different services based on the URL path. Requests with the root path '/' are routed to 'app-frontend', which handles the user interface and is backed by the

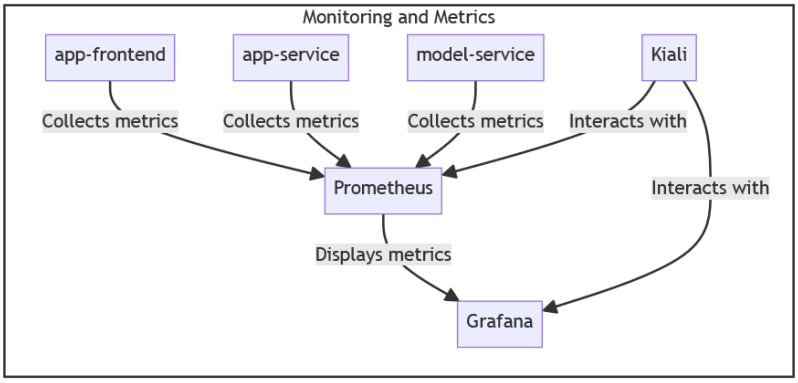

Figure 3: Monitoring

app-frontend pod. Requests with the /api path are routed to app-service, which served by the 'app-service' pod and is responsible for the communication between 'app-frontend' and 'model-service'. Finally, requests with the /model path are routed to 'model-service', which contains the REST endpoint for reaching the deep learning model.

We have set up a monitoring architecture (see Figure 3) within out cluster to gather information about the services. The 'app-frontend', 'app-service', and 'model-service' components each collect their respective metrics, Prometheus collects these metrics, which serves the central monitoring system. It aggregates and stores these metrics, so that they can be reached real-time.

To visualize the collected metrics, Grafana works with Prometheus to display the data. Additionally, Kiali interacts with both Prometheus and Grafana to provide an interface to observe the service mesh within the cluster.

5 EXTENSION PROPOSAL

An identified shortcoming is in the model-service project, in the process of managing dependencies and configurations across the different environments. This issue can be seen in the Dockerfile and the manual setup for installing dependencies. The absence of a streamlined and automated approach can lead to inconsistencies and environment-specific issues (as well as more work on the maintenance side) this problem is worse when deploying the application across multiple environments.

The proposed refactoring to address this would be to further the implementation of IaC tools like Ansible (or Terraform alternatively) along with a CI/CD pipeline.

6 ADDITIONAL USE CASE (ISTIO)

We chose to implement a shadow deployment for the `model-service` to evaluate two different models and compare how they react to user data. The shadow deployment receives the same requests as the actual `model-service`, but it does not send responses back to the `app-frontend`.

6.1 General Description of the Use Case

In the shadow deployment use case, we deploy a second instance of the `model-service` (referred to as `model-service-shadow`) that contains a newer version of the model. This setup allows us to mirror the incoming traffic to both the primary `model-service` and the `model-service-shadow`. The purpose of this deployment is to test the new model version under real-world conditions without exposing it to the users. The results from the shadow service are logged and analyzed to compare the performance and behavior of the new model against the currently deployed model.

6.2 Changes Compared to the Base Design in the Deployment Section

The main changes to the base design is the addition of the `model-service-shadow` and the configuration for traffic mirroring. The following components were added or modified:

- **Deployment of model-service-shadow**: A separate deployment was created to deploy the shadow model service.
- **Service for model-service-shadow**: A service file was added to expose the shadow model service within the cluster.
- **VirtualService Configuration**: The VirtualService was updated to include rules for mirroring traffic to the `model-service-shadow`.

7 EXPERIMENTAL SETUP (ISTIO)

To monitor the performance and usage of our services as well as the shadow deployment of mode-service, we introduced several domain-specific metrics in our application using Prometheus. These metrics help us understand how users interact with our services and assess the performance of different components, including the shadow deployment. The metrics implemented are as follows:

- **REQUEST_COUNT**: This counter metric tracks the total number of requests received by the `app-service`. It provides a quantitative measure of the service usage over time.
- **IN_PROGRESS_REQUESTS**: This gauge metric tracks the number of in-progress requests in the `app-service`. It helps in understanding the current load on the service at any given moment.
- **REQUEST_LATENCY**: This summary metric tracks the request latency in seconds for the `app-service`. It provides insights into the responsiveness of the service and helps in identifying performance bottlenecks.

These metrics are exposed via the `/metrics` endpoint in each service and are automatically scraped by Prometheus. The configuration of Prometheus to scrape these metrics ensures that we continuously collect data on the performance and usage of our application.

8 ML PIPELINE

8.1 Pipeline (DVC)

Our pipeline, as visibile in Figure 2, represents a comprehensive data processing and machine learning workflow, structured around multiple stages.

Initially, data is downloaded and split into training, testing, and validation sets. These sets are further processed to separate features from labels. Next, the features undergo tokenization, and labels are encoded, preparing them for input into the training phase. A model is then trained using the tokenized features and encoded labels, optimized through specified parameters such as loss functions and optimizer choices.

Once the model is trained, it is uploaded to a cloud storage. Simultaneously, the tokenizer used during feature preparation is also uploaded for consistency in future data processing. The trained model is evaluated through a series of tests assessing various aspects such as accuracy, memory usage, repeatability, and latency. Additional tests check for implicit bias, integration robustness, and parity with expected outcomes. Each stage generates specific output files that facilitate tracking of the process and outcomes.

This workflow is managed using DVC (Data Version Control), allowing the entire pipeline to be executed with the command *dvc repro*, or a single stage can be run independently using dvc *repro <stageName>*.

8.2 Data Version Control (DVC)

DVC manages version control by tracking and storing artifacts produced at each stage of the pipeline in a designated folder within the project, which is ignored by Git. This setup allows users to effectively push and pull data to and from this folder using *dvc push* and *dvc pull* commands. By using DVC's version control capabilities, users can avoid rerunning the entire pipeline when making changes; instead, they can selectively execute a specific stage to test modifications directly.

8.3 Dependency Management (Poetry)

For dependency management, our project uses Poetry, a tool that manages library versions to prevent conflicts among different users of the project. Unlike a traditional *requirements.txt* file, Poetry provides several advantages: it ensures consistent environments by locking dependencies to specific versions through the poetry.lock file. This allows for reproducible installations and mitigates issues arising from differing library versions.

8.4 Code Standards (Pylint)

In our project, we use `PyLint` to enforce good coding standards and ensure consistency across our codebase. `PyLint` is used to check our code for a variety of potential issues, such as:

- **Naming Conventions:** We enforce specific styles for variables, functions, classes, and more to adhere to the same naming guidelines, such as using `snake_case` for functions and `PascalCase` for classes.
- **Code Complexity:** We limit complexity by restricting the number of nested blocks, the size of functions, and the number of arguments in functions, helping in maintaining code readability and simplicity.
- **Formatting Rules:** Specific rules for line length and indentation are set, ensuring uniform formatting across the codebase.

We considered using `Flake8`, which was very likely the next best choice. However, the team is more experienced with configuring `PyLint`. Moreover, the integration of `dslinter`, which is specialized for TensorFlow, NumPy, and pandas, provided a strong reason to use `PyLint` as it aligns with our project's needs and the team's desire to experiment with this setup.

8.5 Deployment Strategy (Google Cloud)

Our deployment strategy automates model updates by integrating a pipeline stage that uploads the trained model to Google Cloud Storage whenever the pipeline is executed. This model is then accessed by a *model-service* Docker container in production. This approach ensures that our production environment always uses the latest model, facilitating seamless updates.

9 TESTING DESIGN

The current testing strategy for the entire project includes the unit tests and integration tests executed during the CI stage of the pipeline for all the services. Unit tests verify individual components while in isolation while the integration tests are responsible for ensuring the various components work together properly. End-to-end tests are performed on the application deployed to see the functionality in a production environment. The limitations of the strategy include insufficient coverage and the lack of automated performance testing.

The testing strategy should be improved to overcome these issues by expanding the test coverage and automating the performance testing. Expanding the test coverage is generally laborious but straightforward, we increase the number of unit test ot cover more edge cases and more different scenarios, and use tools to test the code coverage and identify which parts of the codebase remain untested. Automating the performance tests using a tool like JMeter or Locust would also help find the bottlenecks in application performance.

Figure 4: DVC Pipeline

Baseline metrics for code coverage and performance should be collected to test the effectiveness of the improvements (i.e. compare before/after the improvements).

10 CITATIONS

You can cite papers, e.g., [1]. To make the references appear, make sure to compile the latex sources, then bibtex, and then latex twice.

REFERENCES

[1] First Author, Second Author, and Third Author. 2018. An Examplary Paper For The References. In *International Conference on Silly Walks*.