Project Report – Group 3

Jasper Bruin, Richard Wijers, Alexandru Dumitriu

1 INTRODUCTION

This report outlines the detailed processes for software package and container image release pipelines, deployment within a Kubernetes environment, and proposes an extension for automated security testing in CI/CD pipelines. It explains each step in the pipelines, the tools used, and how they contribute to ensuring a robust, scalable, and secure application infrastructure. The document also covers the deployment architecture, highlighting how different components interact to maintain high availability and operational efficiency.

2 RELEASE PIPELINE DOCUMENTATION

2.1 Software Package Release Pipeline

2.1.1 Overview. This section documents the release pipeline for the Python package `lib-ml`. It describes each step, the tools involved, and the data flow throughout the process. The pipeline is activated by pushes to the `main` branch, creation of tags prefixed with 'v*', and pull requests targeting the `main` branch. This configuration ensures that new versions undergo a rigorous build and test process before publication.

2.1.2 Pipeline Steps.

1. Checkout Repository
2. Extract Version Information
3. Create Release
4. Set up Python Environment
5. Install Poetry
6. Configure Poetry
7. Install Dependencies
8. Code Analysis with Pylint
9. Code Analysis with Flake8
10. Check Version on PyPI
11. Build and Publish to PyPI
12. Clean Up

2.1.3 Purpose and Implementation. The pipeline begins by checking out the repository using `actions/checkout@v4`. The purpose is to fetch the latest code from the repository, and it is implemented by cloning the repository into the runner environment with the full history by setting `fetch-depth` to `0`.

Next, version information is extracted using a custom script. This step involves determining the current version from git tags, calculating the next patch version, and setting the version environment variable. This ensures the versioning is consistent and automated. A release is then created using the extracted version information. The release is created using GitHub CLI, which sets the release title, generates release notes, and marks it as a pre-release if applicable. The Python environment is then set up using `actions/setup-python@v3` with Python version `3.9`. This step ensures the correct Python version is used throughout the pipeline.

The Poetry package manager is then installed to handle dependencies and packaging. This is implemented by running a command to download and install Poetry. Following the installation, Poetry is configured to not create virtual environments, implemented by running `poetry config virtualenvs.create false`, which simplifies the environment setup. Dependencies specified in `pyproject.toml` are installed using `poetry install`. The code is then analyzed for errors and compliance with coding standards using `pylint` and `flake8`. These steps are implemented by running `pylint` and `flake8` commands on the Python files to ensure code quality and style adherence.

A crucial step involves checking if the current version of the package already exists on PyPI. This is done by querying PyPI with the current version obtained from Poetry, and an output variable is set based on the version's existence. If the version does not exist on PyPI, the project version is incremented, and the repository is tagged with the new version. This involves updating `pyproject.toml`, committing the changes, tagging the new version, and pushing these updates to the repository, which is implemented through a series of git commands. The next step is to build the package and publish it to PyPI. This is accomplished by running `poetry build` to create distribution artifacts (`sdist` and `wheel`) and then publishing them using provided credentials. Finally, the pipeline performs a clean-up by removing the `dist` directory to maintain a clean working environment.

2.1.4 Data Flow. The data flows from the repository being cloned into the CI environment, through setting up the Python environment and installing dependencies, to analyse the code. If the version is new, the versioning data is updated, and the code changes are pushed back to the repository. The final artifact, a Python package, is built and published to PyPI. The clean-up step ensures no residual data is left in the environment.

2.1.5 Tools Used. The tools integral to this pipeline include GitHub Actions, Python, and Poetry for Python dependency management. Code analysis is performed using Pylint and Flake8.

2.1.6 Artifacts.

- Python package distribution files (`.tar.gz`, `.whl`)
- Published package on PyPI
- Github Release

2.2 Container Image Release Pipeline

2.2.1 Overview. The pipeline is triggered by pushes to the `main` branch or the creation of tags starting with 'v'. This ensures that the Docker image is updated and deployed in response to changes in the source code or new version releases.

2.2.2 Pipeline Steps.

1. Checkout Repository
2. Set up Docker Buildx
3. Log in to Registry
4. Extract Version Information
5. Create Release
6. Build and Push Docker Image

2.2.3 *Purpose and Implementation.*

The Docker Publish pipeline starts by checking out the repository using `actions/checkout@v4`. The purpose is to fetch the latest code from the repository, implemented by cloning the repository into the runner environment with the full history by setting `fetch-depth` to `0`.

Next, Docker Buildx is set up using `docker/setup-buildx@v2`. This step ensures that the advanced features of Buildx are available, allowing for a more flexible and efficient build process.

The pipeline then logs in to the Docker registry using the GitHub token to authenticate, utilizing `docker/login-action@v2`. This step is implemented by configuring the registry, username, and password to establish a session with the Docker registry.

Following this, version information is extracted using a custom script. This step involves determining the current version from git tags, calculating the next patch version, and setting the version environment variable. This ensures the versioning is consistent and automated.

A release is then created using the extracted version information. The release is created using GitHub CLI, which sets the release title, generates release notes, and marks it as a pre-release if applicable.

Finally, the pipeline builds and pushes the Docker image to the registry. This step involves building the Docker image from the Dockerfile and tagging it with the derived version. The image is then pushed to the registry using `docker push` $IMAGE_ID$ `:VERSION`, ensuring that the latest version is available in the registry.

2.2.4 *Data Flow.*

The data flow in this pipeline starts with the GitHub repository as the source, moves through the GitHub Actions runner environment where the Docker image is built and ends at the GitHub Container Registry where the image is stored. This streamlined process ensures that new or updated Docker images are readily available for deployment or further development stages.

2.2.5 *Tools Used.*

Throughout this process, tools such as Docker for building and pushing images, and GitHub Actions for automating these steps, play a crucial role.

2.2.6 *Artifacts:*

- Docker image - ghcr.io
- Github Release

3 DEPLOYMENT DOCUMENTATION

3.1 Visual Representation of the Deployment Structure

The provided diagram in Fig. 1 depicts the deployment architecture of our application within a Kubernetes cluster, showcasing the interactions and data flow among various services and pods. The architecture is designed to ensure high availability, scalability, and efficient monitoring of the application.

3.2 Description of Data Flow for Incoming Requests

The deployment architecture begins with the client sending a request, which is managed by an Ingress-managed load balancer. This load balancer distributes the traffic across multiple instances of the Ingress Controller within the cluster, ensuring load balancing and redundancy.

The Ingress Controller directs the traffic based on predefined routing rules to either the **app_frontend** service or the **app_service** service within the default namespace. The **app_frontend** service, which manages the frontend application, forwards the request to one of its associated pods, either **app_frontend** pod or **app_frontend_experiment** pod, based on internal balancing mechanisms. These pods handle the user interface and interactions, returning the appropriate response to the client through the same route.

The Ingress Controller routes requests that require backend processing to the **app_service** service. This service directs traffic to the **app_service** pod, which performs backend operations. For tasks involving machine learning models, the `app_service` interacts with the **model_service** service, which manages the **model_service** pod responsible for executing model-related tasks.

3.3 Overview of All Resources and Their Connections

- **Ingress Controller**: Manages incoming HTTP requests and routes them to the appropriate service within the cluster.
- **app_frontend Service**: Handles HTTP requests directed at the frontend application, distributing them between the `app_frontend` and `app_frontend_exp` pods.
  - **app_frontend Pod**: Standard frontend pod handling user interface operations.
  - **app_frontend_experiment Pod**: Experimental frontend pod for testing new features or versions.
- **app_service Service**: Manages requests requiring backend processing and interacts with the `model_service` for tasks involving machine learning.
  - **app_service Pod**: Handles backend business logic and processing.
- **model_service Service**: Provides model-related services and computations.
  - **model_service Pod**: Executes machine learning models and returns the results to the `app_service` pod.
- **Monitoring Namespace**: Dedicated for monitoring services, it contains the `Monitoring Service` and the `Prometheus Dashboard`.
  - **Monitoring Service**: Collects metrics and data for monitoring the health and performance of the application.
  - **Prometheus Dashboard**: Visualizes the collected data, providing insights into the application's performance and operational status.

4 EXTENSION PROPOSAL

4.1 Identification of Shortcoming

While our current setup covers various aspects of deployment, performance, and monitoring, it lacks comprehensive automated security testing and vulnerability management. This omission can leave the system vulnerable to security threats that could be identified and mitigated through automated testing and continuous monitoring.

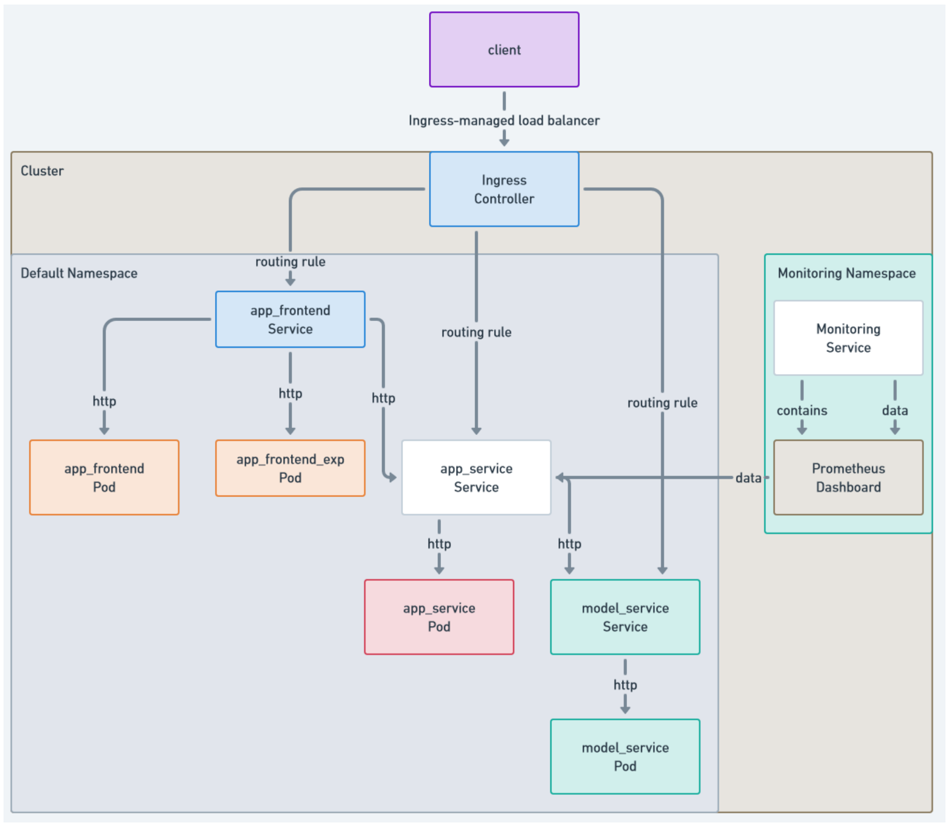


Figure 1: Deployment Architecture

4.1.1 *Effect of the Shortcoming.*

- **Exposure to Security Vulnerabilities**: Without automated security testing, vulnerabilities can go unnoticed, exposing the system to potential attacks.
- **Manual Security Audits**: Relying on manual security audits can be time-consuming and prone to human error, potentially missing critical issues.
- **Compliance Risks**: Lack of automated security checks can result in non-compliance with industry standards and regulations, leading to legal and financial consequences when the application reaches a much larger scale.

4.2 Proposed Extension: Integration of Automated Security Testing and Vulnerability Management

To address this, we propose integrating automated security testing and vulnerability management into the CI/CD release pipeline. This will ensure that the system is continuously monitored for security vulnerabilities and compliance issues, with automatic remediation where possible.

4.2.1 *Benefits of Automated Security Testing and Vulnerability Management.*

- **Continuous Security Assurance**: Automated tests ensure that security checks are performed consistently on every code change and deployment.
- **Early Detection and Mitigation**: Vulnerabilities can be detected and addressed early in the development lifecycle, reducing the risk of security incidents in production.

4.2.2 *Implementation Plan.*

(1) **Integrate Static Application Security Testing (SAST)**:
   - **Tools**: Use tools like SonarQube or Semgrep for static code analysis to detect security vulnerabilities in the codebase. Research indicates these tools perform well in detecting common vulnerabilities in both synthetic and real-world scenarios [3].
   - **Configuration**: Configure these tools to run automatically on every pull request and commit.

(2) **Set Up Dynamic Application Security Testing (DAST)**:
   - **Tools**: Integrate tools like Invicti for dynamic security testing to identify vulnerabilities in the running application. The effectiveness of DAST tools in identifying security weaknesses by simulating real-world attacks and providing detailed reports for remediation can be easily seen in the following article by Invicti, a well-known security company [2].

- **Automation**: Configure these tools to run as part of the CI/CD pipeline, especially in staging environments.

(3) **Configure Continuous Monitoring and Alerting**:
- **Tools**: Adapt Prometheus for monitoring security metrics and integrate it with Alertmanager for alerting on security incidents.
- **Dashboards**: Update Grafana dashboards to visualize security metrics and trends over time.

4.2.3 Testing the New Design. To ensure the new design works effectively, the following steps can be taken:

(1) **SAST and DAST Validation**: Validate that SAST and DAST tools correctly identify vulnerabilities and that reports are generated and reviewed.
(2) **Alerting and Monitoring Validation**: Test the alerting system by simulating security incidents and verifying that alerts are generated and received by the team.

By integrating automated security testing and vulnerability management, we can significantly enhance the security posture of our application, ensuring that it is robust against potential threats and compliant with security standards.

5 ISTIO USE CASE: RATE LIMITING

The EnvoyFilter targets the echoserver pods, applying a local rate limit to HTTP traffic. It configures the Envoy proxy to insert a rate-limiting filter before the existing HTTP connection manager. This setup includes A token bucket with a maximum of 10 tokens, replenishing 5 tokens every 60 seconds. The rate limit is always enforced, with every request checked against the current token count.

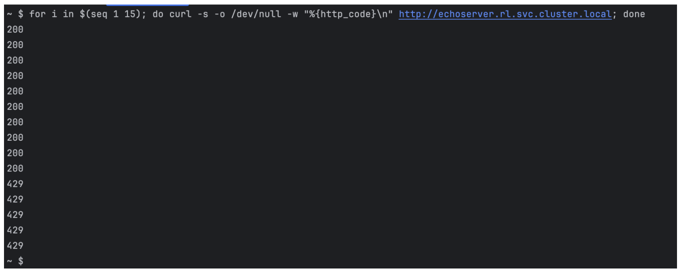
Responses for rate-limited requests modify headers to include x-rate-limited: TOO_MANY_REQUESTS and return an HTTP 429 status code, indicating that the rate limit has been exceeded. The Echoserver Service defines a network identity for the echoserver pods within the rate-limit namespace, directing traffic on port 80 to the pods. The StatefulSet for echoserver manages the deployment, using the ealen/echo-server:latest image, a simple server suitable for demonstrating Istio's capabilities. The Sleep Service supports network testing by routing traffic to the sleep pod, which runs indefinitely due to the sleep infinity command in its configuration.

The StatefulSet for sleep ensures it remains operational for ongoing testing purposes. This configuration exemplifies the effective management of service traffic using Istio within Kubernetes, emphasizing how rate limiting can prevent service overload and maintain fair usage standards. This streamlined approach to traffic management ensures service stability and responsiveness under varying load conditions.

6 ISTIO EXPERIMENTAL SETUP

6.1 Description of the Experiment

The Istio experiment aimed to assess the traffic management capabilities and performance monitoring of the system. Specifically, we implemented an IngressGateway and VirtualServices to control the traffic flow and monitored the application performance using Grafana dashboards.



```
~ $ for i in $(seq 1 15); do curl -s -o /dev/null -w "%{http_code}\n" http://echoserver.rl.svc.cluster.local; done
200
200
200
200
200
200
200
200
200
200
429
429
429
429
429
~ $
```

Figure 2: Screenshot of the terminal showing the rate limiting test results.

6.2 Configuration Details

We implemented several Kubernetes resources in the istio-system namespace to manage traffic and monitor application performance.

Gateway: We created a Gateway named my-gateway, using the selector istio: ingressgateway. It listens on port 80 for HTTP traffic and accepts requests from any host ("*").

DestinationRules: Three DestinationRules were set up for the frontend, backend, and model services, all using a round-robin load balancing strategy. The targets are app-frontend.default.svc.cluster.local, app-service.default.svc.cluster.local, and model-service.default.svc.cluster.local respectively.

VirtualService: The VirtualService named my-virtual-service routes traffic through my-gateway. It handles multiple routes, directing requests for various paths (/kiali, /jaeger, /grafana, /prometheus/, /backend, /model, and /) to their respective services. For example, requests to /kiali are routed to kiali-dashboard on port 20001, while requests to / are routed to app-frontend.default.svc.cluster.local on port 3000.

Service Definitions: We exposed monitoring dashboards using LoadBalancer services:

- kiali-dashboard on port 20001
- jaeger-dashboard on port 16686
- grafana-dashboard on port 3012
- prometheus-dashboard2 on port 9090

These configurations ensure that our application services and monitoring dashboards are properly exposed and accessible, with traffic management effectively handled by Istio.

6.3 Hypothesis and Metrics

The hypothesis tested in this experiment is that the new version of the app service will perform better under load compared to the existing version. The metrics used to evaluate this hypothesis include:

- **Request Duration**: Measures the time taken to process requests.
- **Active Sessions**: Tracks the number of active user sessions.
- **Prediction Requests Rate**: Monitors the rate at which prediction requests are made.
- **Total HTTP Requests**: Counts the total number of HTTP requests received.
- **Request Processing Time**: Tracks the time taken to process HTTP requests.

6.4 Data Decision Process

The decision process involves analyzing the Grafana dashboard data to determine if the new version meets the performance criteria. The key steps are:

- Monitor Request Duration: Ensure that the new version's request duration is lower or comparable to the current version.
- Analyze Active Sessions: Check for a stable or increased number of active sessions without performance degradation.
- Evaluate Prediction Requests Rate: Higher or stable prediction requests rate indicates better performance.
- Check Total HTTP Requests: Ensure that the total number of HTTP requests handled is consistent with expected traffic.
- Assess Request Processing Time: Lower processing time for requests indicates better performance.

6.5 Grafana and Istio Visuals

The Istio diagram 4 shows a well-defined setup with an IngressGateway and VirtualServices. The traffic is routed to different versions of the app frontend and app service, demonstrating a 90/10 split for the app service. The Grafana dashboard visualizes key metrics for monitoring the performance of the app service.

7 ML PIPELINE DOCUMENTATION

This section provides a overview of the Machine Learning (ML) pipeline used in our project. It covers the setup and configuration details, the decision-making process and design choices, automated tasks, and created artefacts.

7.1 Tools used

The project uses the Cookiecutter project template. Cookiecutter standardizes the project setup process with predefined directory structures and configuration files. This helps to maintain consistency and predictability for the project structure.

For managing Python dependencies, we use Poetry. Poetry simplifies dependency management by ensuring that all collaborators use the same versions of dependencies across different environments. It also automates the creation and management of virtual environments, which helps to isolate the project and its dependencies from other Python projects on the same machine.

To ensure code quality and consistency, we use Pylint and Flake8 as linters. Pylint checks for errors, bugs, and code that does not adhere to best practices, while Flake8 enforces style guidelines. This helps us maintain a high standard of code quality and readability.

Data Version Control (DVC) is an open-source tool that is designed to handle large data files, binary models, and metrics as well as code. It is built to make Machine Learning (ML) projects reproducible and shareable. DVC is compatible with Git repositories and has a similar command-line interface.

7.2 Pipeline stages

The ML pipeline consists of four main stages as shown in figure 5 . each interacting or producing some result as follows:

(1) **download_data**: This stage is responsible for downloading the dataset. It depends on the `download_dataset.py` script and produces the output in the `data/external` directory. The parameters `data_folder_id` and `data_folder` are used in this stage.
(2) **preprocess_data**: This stage is responsible for tokenizing the data. It depends on the `pyproject.toml` file and produces the output in the `artifacts/tokenized` directory. The parameters `tokenized_folder` and `data_folder` are used in this stage.
(3) **train_model**: This stage is responsible for training the model. It depends on the `train_model.py` script and produces the output in the `artifacts/trained` directory. The parameters `epochs`, `batch_size`, `categories`, `loss_function`, `optimizer`, `tokenized_folder`, and `trained_folder` are used in this stage.
(4) **predict_model**: This stage is responsible for making predictions using the trained model. It depends on the `predict_model.py` script and produces the output in the `artifacts/predicted` directory. The parameters `predicted_folder`, `trained_folder`, and `tokenized_folder` are used in this stage.

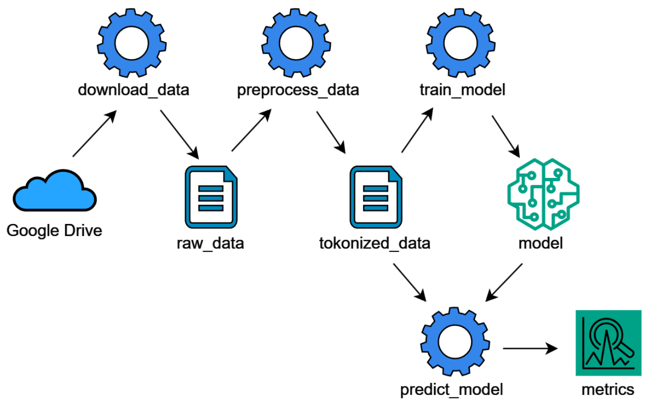


Figure 5: ML Pipeline

7.3 Automation

To ensure quality, two automatic GitHub actions are used:

- pylint.yml defines a GitHub Actions workflow that checks Python code quality and linting on every push event. It runs on the latest Ubuntu, sets up Python 3.9, installs and configures Poetry for dependency management, installs project dependencies, and then runs Pylint and Flake8 for static code analysis. Pylint checks for programming errors, while Flake8 enforces coding style guidelines.
- The `pytest.yml` file defines a GitHub Actions workflow that runs unit tests on every pull request. It operates on the latest Ubuntu, sets up Python 3.9, checks out the code, installs dependencies using Poetry, downloads the dataset by running a Python script, and then runs the unit tests

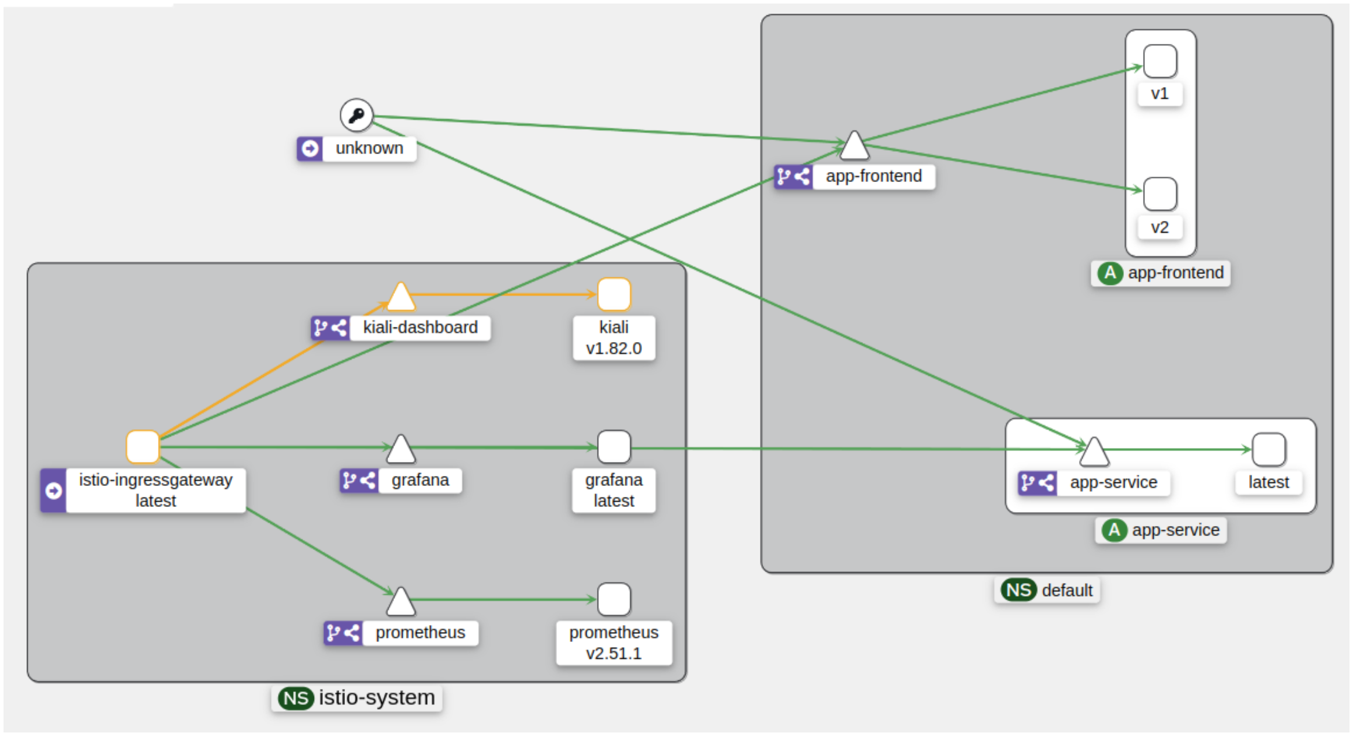


Figure 3: Istio Architecture

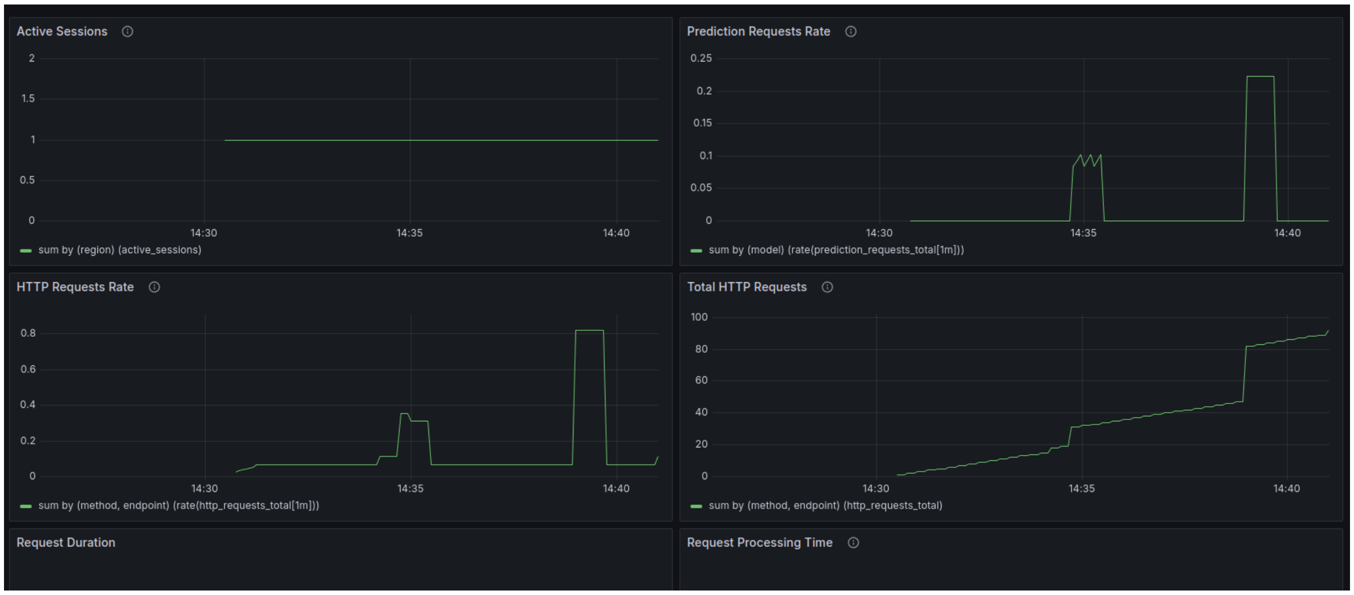


Figure 4: Grafana Dashboard

using pytest. This ensures that any changes made in a pull request do not break the functionality of the existing code.

7.4 Artifacts

The pipeline produces several artifacts that are used in different stages of the pipeline or for evaluation purposes. These include the downloaded data, tokenized data, trained model, and prediction results. These artifacts are stored in the artifacts directory, with each type of artifact stored in its respective subdirectory (tokenized, trained, predicted). This organization makes it easy to locate and use the artifacts in different stages of the pipeline or for evaluation purposes.

8 TESTING STRATEGIES ANALYSIS

Table 1 outlines our comprehensive testing strategy. Each type of test and phase in the table is mentioned for quick reference. The table systematically categorizes the sequence of testing phases: Initiate, Data Validation, Feature Handling, Evaluation, Assessment, Infrastructure, Monitoring, Completion, and End. This structured approach ensures that each test is properly set up, executed, reviewed for performance, and analyzed for outcomes. Detailed information about these tests can be found in our `model-training` repository.

8.1 Automation of Testing Processes

Our testing processes are fully automated to ensure reliability and efficiency. We leverage our Data Version Control (DVC) pipeline to manage and download datasets from remote sources. This automation ensures that the most current and relevant data is used during testing, reflecting real-world conditions as closely as possible.

8.1.1 Continuous Integration Setup. We employ GitHub Actions to automate our testing workflow, as detailed in our `pytest.yml` configuration. This setup is triggered on every pull request, ensuring that all changes are thoroughly tested before integration. Here's how our testing pipeline operates:

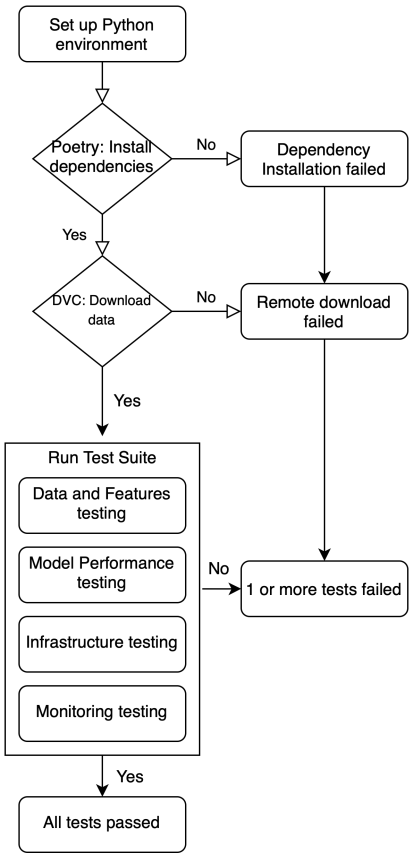


Figure 6: Workflow of the automated CI Testing after each Pull Request

- **Code Checkout:** GitHub Actions checks out the latest code for testing.
- **Environment Setup:** Sets up Python and installs dependencies using Poetry, which ensures that our testing environment is consistent and replicable.
- **Dataset Download:** The dataset is automatically downloaded using a script facilitated by our DVC setup. This step is for testing with the most up-to-date data.
- **Test Execution:** Tests are run using Pytest, which systematically executes various test cases, including unit tests, integration tests, and system tests.

8.2 Tests for Features and Data

Our testing framework is designed to ensure data integrity and feature quality, particularly for the detection of phishing URLs. This section explores the specific testing procedures, highlighting their critical roles and outcomes with brief code snippets.

8.2.1 Data Handling and Random Sampling. We have developed functions to efficiently read and randomly sample data from datasets. These functions not only streamline the process of loading data but also incorporate randomness to simulate real-world scenarios. For example, data sampling is performed using the `random.shuffle` method, ensuring unbiased data selection:

Phase	Tasks
Data Validation	Ensure Data Integrity, Efficient Data Reading, Manage File Operations
Feature Handling	Incorporate Randomness, Remove Sensitive Information, Verify Tokenization and Encoding
Evaluation	Feature Tests, Evaluate Data Slice Mechanisms
Assessment	Check Feature Distribution, Examine Feature-Target Relationships, Assess Computational Cost, Uphold Privacy Considerations, Model Performance Tests
Infrastructure	Verify GPU Resources, Conduct Memory Usage Tests, Validate Model Persistence, Integration Tests, Monitoring Tests
Monitoring	Ensure that model is up-to-date
Completion	Trigger Alerts or Retraining, Results Analysis

Table 1: Workflow Overview

Test Name	Statistic	Expected Value/Condition	Result
Tokenizer	Number of sequences	Match number of input texts	300
	Word index	Non-empty	1659
Label Encoder	Transformed labels	Match number of input labels	300
	Unique classes	More than one	2
Feature Distribution	Mean sequence length	Greater than zero	11.33
Feature-Target Relationship	Number of sequences	Match number of labels	300
Feature Cost	Feature size	Less than 10,000	1874
Feature Privacy	Sensitive patterns	Not present in word index	True
Feature Code	URL cleaning	All URLs start with 'http://' or 'https://'	True
	Input feature type	All features are strings	True
	Input feature length	No empty strings	True

Table 2: Summary of Test Statistics and Expected Conditions

```
random.shuffle(lines)
```

8.2.2 Sensitive Information Removal. Sensitive data is redacted from URLs using a regex pattern defined as follows:

```
re.compile(r"(@|token|session|user|userid|"
           "password|auth|files|pro)")
```

This ensures compliance with data privacy standards and prevents any potential data leakage.

8.2.3 Tokenization and Label Encoding. The robustness of our tokenization and label encoding processes is thoroughly tested. The tokenizer transforms URLs into a machine-readable format, which is verified by checking:

```
assert len(data["tokenizer"].word_index) > 0
```

Similarly, label encoding converts categorical labels into numerical format, ensuring the model receives correctly formatted input.

8.2.4 Feature Distribution and Target Relationship Analysis. We evaluate the distribution of features and examine the relationships between features and targets. This includes checking that feature lengths are consistent and meaningful:

```
sequence_lengths = [len(seq) for seq in sequences]
assert np.mean(sequence_lengths) > 0
```

Such tests confirm the logical alignment of data inputs to their respective outputs, essential for effective learning.

8.2.5 Computational Cost Assessment. The computational efficiency of processing features is assessed by evaluating the size of the tokenizer's word index:

```
assert len(data["tokenizer"].word_index) < 10000
```

This step ensures that the memory usage and processing time remain within practical limits for real-time applications.

These tests, structured with direct references to the code logic, validate not only the functionality and efficiency of our data processing methods but also uphold stringent privacy standards, ensuring that our phishing detection system is both effective and ethical.

8.3 Tests for Model Development

In the development of our URL Publishing Detection CNN Phishing Detector, we implement a series of rigorous tests aimed at evaluating model performance metrics including accuracy, precision, recall, and F1 scores. These metrics are computed for different data subsets—short, medium, and long text slices—to ensure that our model performs consistently across various types of input data.

8.3.1 Accuracy Testing Across Text Lengths. To assess the robustness and uniformity of our model's predictive accuracy, we perform tests across three defined text length categories: short, medium, and long. We ensure that the model's accuracy does not vary significantly between these categories, which is crucial for maintaining reliable detection across different URL lengths. The test checks if the differences in accuracy are within acceptable limits:

```
assert abs(short_metrics['accuracy'] -
medium_metrics['accuracy']) < 0.25

assert abs(medium_metrics['accuracy'] -
long_metrics['accuracy']) < 0.25
```

From the results in 3, we can see that short slices exhibit significant variability in all metrics, indicating that the model may struggle with shorter data segments. This could be due to insufficient information within short inputs, leading to inconsistent performance. Medium and long slices demonstrate consistently high performance across all metrics. This suggests that the model performs well when given more context or information within the input data, resulting in better precision, recall, and overall accuracy. The F1 Score for medium and long slices remains high and stable, indicating a well-balanced trade-off between precision and recall, which is crucial for reliable model performance.

Slice	Accuracy	Precision	Recall	F1 Score
Short	0.88	0.8864	0.88	0.8817
Short	0.6957	0.7846	0.6957	0.7075
Short	0.7647	0.8371	0.7647	0.7469
Short	0.6667	0.8000	0.6667	0.6250
Medium	0.9759	0.9759	0.9759	0.9759
Medium	0.9114	0.9177	0.9114	0.9108
Medium	0.8421	0.8895	0.8421	0.8448
Medium	0.8333	0.8719	0.8333	0.8361
Long	0.9688	0.9691	0.9688	0.9688
Long	0.9242	0.9336	0.9242	0.9249
Long	0.9517	0.9545	0.9517	0.9517
Long	0.9286	0.9368	0.9286	0.9291

Table 3: Summary of Model Development Statistics

8.3.2 Precision, Recall and F1-Evaluation. Precision and recall are measured to ensure that the model not only accurately identifies phishing attempts but also minimizes false positives and negatives, crucial for user trust. We calculate these metrics for each text slice, using the scores metrics from `sklearn.metrics`, this allows us to tailor our model's training to improve these specific metrics, depending on the needs of the deployment environment. By monitoring these metrics, we can adjust our model to ensure that it equally weighs both false positives and false negatives, which is essential for phishing detection.

8.3.3 Robustness to Input Perturbations by Metamorphic and Differential Testing. We further test the model's robustness by introducing slight perturbations to the text data. This simulates urls where input data might not be perfectly formatted or might contain minor errors. The test verifies that even with these changes, the model's performance remains stable:

```
perturbed_x_train = [perturb_text(text)
for text in raw_x_train]
```

Ensuring that our model can handle such inconsistencies is critical for deploying a reliable phishing detector.

Metamorphic testing is an effective approach for this purpose, as it involves generating new test cases from existing ones by applying transformations that should not affect the expected outcomes. For example, by adding uninformative code elements or renaming variables, we can test the robustness of ML models to these variations [1]. This approach has been shown to reveal weaknesses in models that are not evident through traditional accuracy metrics alone [1].

Additionally, in the context of phishing detection, metamorphic testing can simulate various real-world perturbations, such as typos, synonyms, or changes in text structure. This ensures that the model's performance is stable under these conditions and helps identify potential vulnerabilities that could be exploited [1, 4].

By incorporating metamorphic testing, we can enhance our model's resilience and reliability, making it better suited for deployment in environments where data inconsistencies are common [1].

8.3.4 Model Retraining and Evaluation. Finally, the model's ability to retrain and adapt to new data while maintaining its performance is tested. We retrain the model using different subsets of data, seeded randomly to ensure diverse training scenarios. This step is needed to ensure that the model does not overfit to a particular data configuration and can generalize well across various data distributions.

8.4 Tests for ML Infrastructure

Our infrastructure testing is necessary for ensuring the efficient operation of our phishing detection models, particularly in handling the intensive tasks of training and evaluating large datasets. This section details the specific tests we conduct to verify infrastructure readiness and performance.

8.4.1 GPU Resource Availability. We verify the availability of GPU resources, which are essential for accelerating the computation-intensive processes. This is conducted via the following test:

```
# "GPU is not available"
assert tf.config.list_physical_devices('GPU')
```

This ensures that our models run on appropriate hardware that can support their computational needs.

8.4.2 Memory Usage Monitoring. Memory usage is critically monitored before and after model training sessions using `psutil` to ensure it remains within acceptable limits. The relevant code snippet is:

```
process = psutil.Process(os.getpid())
# Memory in MB
memory_before = process.memory_info().rss / 1024 ** 2

# Model training here
memory_after = process.memory_info().rss / 1024 ** 2
assert memory_after - memory_before < 1000
```

This test helps in identifying potential memory leaks or excessive memory consumption that could hinder deployment.

8.4.3 Model Persistence. Testing the model's persistence involves rigorous save and load procedures to ensure that the model retains its configuration and weights across sessions, critical for deployment scenarios. We perform:

```
model.save('model.h5')
loaded_model = tf.keras.models.load_model('model.h5')
```

Test Suite	**Objective**	**Statistics**
test_memory_usage	Measure memory usage before and after training to ensure it is within acceptable limits	Memory before: 691.14 MB, Memory after: 761.17 MB, Difference: 70.03 MB
test_robustness_to_noise	Evaluate the model's robustness to noisy input by comparing performance on clean and noisy validation data	Original val loss: 0.699, Noisy val loss: 0.699, Relative change: 0.00

Table 4: Summary of Test Suites, Objectives, and Statistics

```
# Model mismatch on load"
assert model.get_config() == loaded_model.get_config()
```

This step verifies that the saved model when loaded, remains identical to the original, maintaining its integrity over time.

Each of these tests is integrated into our continuous integration pipeline to automatically verify system capabilities and readiness as part of our development process. This automated testing ensures that any infrastructure-related issues are identified early and can be addressed before they impact the production environment.

8.5 Monitoring Tests for ML

The test_check_for_staleness function plays a role in the monitoring phase of model management. It serves to ensure that the deployed models remain up-to-date and effective over time. Specifically, this test verifies that the age of the model file does not exceed the predefined threshold of 50 days, as determined by the MAX_MODEL_AGE constant. By evaluating the staleness of the model, we can trigger necessary updates or retraining processes to maintain the model's accuracy and relevance. This approach is particularly important in dynamic environments where data characteristics can shift rapidly, thereby necessitating frequent model evaluations and updates. The test is strategically integrated into our broader testing framework, which is outlined in our model training repository, ensuring that each aspect of the model's lifecycle is systematically monitored and maintained.

8.6 Suggestions for Improvement

To enhance the testing framework, a multi-faceted approach is essential. Monitoring tests need to be developed to ensure the model's long-term reliability in production environments. These tests should continuously evaluate the model's performance and integrity, alerting the team to any degradation or unexpected behaviors that may surface over time.

Additionally, the scope of robustness checks should be expanded. Incorporating stress testing with extreme values and including adversarial examples where applicable will test the model's resilience under unusual or unexpected conditions. This expansion will help identify potential weaknesses that could be exploited under real-world operating conditions, thereby strengthening the model's defenses.

Moreover, a clearer separation and more detailed isolation in tests are necessary to ensure that individual features and model evaluations can be independently verified. This approach will allow for pinpointing specific components or processes within the model that may be underperforming or causing errors, facilitating targeted improvements and enhancing the overall robustness of the model.

9 CONCLUSION

Our report offers an overview of the release and deployment processes, underlining the crucial role of security and systematic testing in system reliability. It points out opportunities for advancements in monitoring and security testing to effectively address new challenges and technological shifts. We have successfully met all specified requirements to at least a sufficient level.

For further improvements, for the Istio use case, it is necessary to switch from echoService to our app-service to protect our application from flow attacks, ensuring a more secure and resilient operational environment. Additionally, conducting experiments with Istio to compare the performance of our v1 and v2 versions could provide valuable insights into system efficacy and areas for further improvement.

REFERENCES

[1] Leonhard Applis, Annibale Panichella, and Arie van Deursen. 2021. Assessing Robustness of ML-Based Program Analysis Tools using Metamorphic Program Transformations. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1377–1381. https://doi.org/10.1109/ASE51524.2021.9678706

[2] Invicti. 2024. How to choose a DAST solution: An 8-step evaluation checklist. https://www.invicti.com/blog/web-security/dast-solution-8-step-evaluation-checklist/ Accessed: 2024-06-16.

[3] Kaixuan Li, Sen Chen, Lingling Fan, Ruitao Feng, Han Liu, Chengwei Liu, Yang Liu, and Yixiang Chen. 2023. Comparison and Evaluation on Static Application Security Testing (SAST) Tools for Java. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 921–933.

[4] Cynthia C. S. Liem and Annibale Panichella. 2020. Run, Forest, Run? On Randomization and Reproducibility in Predictive Software Engineering. *CoRR* abs/2012.08387 (2020). arXiv:2012.08387 https://arxiv.org/abs/2012.08387