REMLA Project Report – Group 14

Dani Rogmans, Justin Luu, Nadine Kuo, Yang Li

In this report, we document the overall setup and design decisions made for our URL Fishing project (TU Delft course Release Engineering for ML Applications, CS4295), for which a high-level architecture is visualized in Figure 1. As can be seen, this project is composed of several independent components which have their dedicated repositories.

The `model-service`[1] represents a wrapper service for the release ML model and offers a REST API to expose the trained model to other components. Both our `model-service` and `model-training`[2] artifacts depend on our `lib-ml` library[3] through PyPi.

Moreover, our `app`[4] contains logic for querying the `model-service` through REST requests and allowing users to enter URL input for phishing detection. It depends on our version-aware `lib-version`[5] library, allowing for querying and displaying its version.

Both the `app` and `model-service` are released as two separate container images with their respective dependencies, meaning they can be deployed separately from each other.

1 RELEASE PIPELINE DOCUMENTATION

The release workflows for our various components ensure a streamlined continuous delivery and deployment process using GitHub Actions as build server. This section focuses on our `app` and `lib-ml` components, thereby outlining each step in the pipeline, detailing its purpose, implementation, and the flow of data and artifacts through the process.

1.1 `App`: Container Image Release Pipeline

The main purpose of this release workflow[6] is to ensure that every change meeting the release criteria is automatically versioned, built, tagged, and made available as Docker images stored in GitHub's Container Registry (GHCR). This automation allows us to streamline the continuous delivery process and maintaining a high standard of version control and traceability. It is triggered by either:

(1) Closing pull requests to the main branch
(2) Pushes of tags matching `v[0-9]+.[0-9]+.[0-9]+`

The latter ensures that releases are systematically handled through semantic versioning in the format `MAJOR.MINOR.PATCH`, allowing for unique and meaningful version numbers.

The workflow begins with the *"Checkout code"* step, which retrieves the latest code from the repository. Using the `actions/checkout@v2`-action, the codebase from the specified branch or tag is cloned into the runner environment, providing the source code required for subsequent steps. Note that in case (1), we also need to reference the specific pull request merge commit SHA in order to make support for pre-releases possible.

Next, the *"Extract version"* step determines the release version from the tag in case (2). By giving this step a unique ID, it becomes trivial in later steps to access this version again using `steps.<VERSION_STEP_ID>.outputs.version`.

Following this, the *"Log into GitHub Container Registry"* step authenticates with GitHub Container Registry (GHCR) to facilitate the pushing of Docker images. The Docker login command is used with credentials from GitHub secrets, ensuring secure handling of authentication credentials.

Only in the case of PRs being merged to main (1), we automatically bump the current app version and push a new tag to the repository (also in format MAJOR.MINOR.PATCH). It uses *anothrNick/github-tag-action@v1* to determine the type of version bump—major, minor, patch, or none—based on the merge commit message, defaulting to a patch bump if no specific tag is found. We have also enabled support for multiple pre-releases of the same version (e.g. *"1.2.3.beta-1"*) - for all those branches other than main.

The *"Build Docker image"* step involves building the Docker image for the application, based on the `Dockerfile`. This image forms the core artifact of our release process.

Next, the *"Parse semantic version from tag"* step takes the created tag (either the manual or automatic one) and extracts the major, minor and patch components. These are set to output variables for use in the next steps.

In the *"Tag Docker image with version"* step, the built Docker image is tagged with distinct versions:

(1) *vMAJOR.MINOR.PATCH*
(2) *vMAJOR.MINOR.latest*
(3) *vMAJOR.latest*
(4) *latest*

These ensure the image is properly versioned and easily identifiable.

Subsequently, the *"Push Docker image to GitHub Container Registry"* step uploads the tagged Docker images to GHCR, under the repository `ghcr.io/remla2024-team14/app`, making them available for distribution.

Finally, the *"Trigger release"* step creates a release in GitHub based on the new version tag. The `actions/create-release@v1` action is utilized to create a release, with the GitHub token used for authentication. The release includes the version tag, release name, and a descriptive body, making it available under the GitHub repository's *"Releases"* section.

1.2 `Lib-ml`: Software Package Release Pipeline

This release workflow[7] automates the process of building distribution packages and uploading our `lib-ml` library (containing pre-processing logic for training our ML model e.g.) to PyPI, a package registry dedicated to Python. This pipeline has similar triggers as the one described in previous section - and again begins with the *"Checkout code"* step.

[1] https://github.com/remla2024-team14/model-service
[2] https://github.com/remla2024-team14/model-training
[3] https://github.com/remla2024-team14/lib-ml
[4] https://github.com/remla2024-team14/app
[5] https://github.com/remla2024-team14/lib-version
[6] https://github.com/remla2024-team14/app/blob/main/.github/workflows/release.yml
[7] https://github.com/remla2024-team14/lib-ml/blob/main/.github/workflows/release.yml

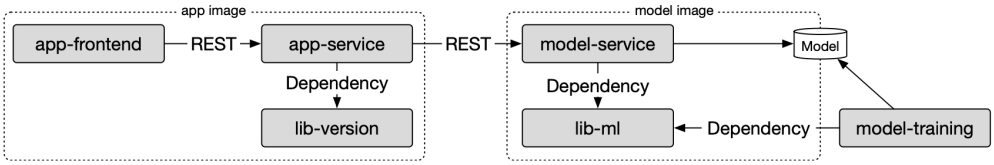

Figure 1: High-level project architecture (source: REMLA Assignment 2)

In the case of PRs being merged to main (1), we automatically bump the current library version and push a new tag to the repository. It again uses *anothrNick/github-tag-action@v1* to determine the type of version bump—major, minor, patch, or none—based on the merge commit message, defaulting to a patch bump if no specific tag is found. We have also enabled support for multiple pre-releases of the same version (e.g. *"1.2.3.beta-1"*) - for all those branches other than main.

Next, the *"Set up Python"* step sets up the Python environment. By using the `actions/setup-python@v2`-action, Python 3.8 is installed and configured on the runner. This ensures that the environment is prepared for running Python scripts and installing dependencies. Following this, the *"Install dependencies"* step installs the required dependencies for building and publishing the package. The `python -m pip install –upgrade pip` command ensures that pip is updated to the latest version, and `pip install setuptools wheel twine` installs the necessary tools for packaging and distributing the software.

The *"Build Distribution"* step creates the distribution packages for the Python project, making it easy to send and unwrap the library. The command `python setup.py sdist bdist_wheel` generates both source distribution (`sdist`) and built distribution (`bdist_wheel`) packages. Essentially, `python setup.py sdist` creates a *.tar.gz* or `.zip` file in the *dist/* directory that contains all the source code of our library. On the other hand, `python setup.py bdist_wheel` creates a *.whl* file in the *dist/* directory that can be instantly installed using tools such as *pip*. These packages are the primary artifacts of our release process.

Subsequently, the *"Publish to PyPI"* step uploads the built distribution packages to the Python Package Index (PyPI). The `twine upload dist/*` command uploads all files in the `dist` directory to PyPI, using authentication credentials stored in GitHub secrets. *Twine* is a commonly used tool for uploading packages because it handles authentication securely and supports modern Python packaging standards.

2 K8S DEPLOYMENT DOCUMENTATION

Kubernetes is essential for our application deployment due to its robust features that help us enhance scalability, reliability, and security, enabling us to focus on development rather than infrastructure management. It offers service discovery and load balancing, ensuring efficient communication and traffic distribution by exposing containers using DNS or IP address. Storage orchestration allows automatic mounting of storage systems, such as local storages or public cloud providers. Automated rollouts and rollbacks ensure smooth updates with minimal disruptions, as K8s will keep the actual state in sync with the configured desired state, at a controlled rate. Automatic bin packing optimizes resource usage, and self-healing capabilities restart or replace failed containers to maintain application health. Additionally, Kubernetes manages secrets and configurations securely, allowing easy updates without rebuilding images or exposing secrets in our configuration.

In this section, we detail the final deployment structure for our URL fishing application (see Figure 2), designed to run and expose a web service that interacts with a model service, ensuring a streamlined flow of data and interactions. Initially in this project we had a `docker-compose` setup[8], which was mere a simple setup to illustrate that app can run in a containerized form independent from host OS. Here, however, we will elaborate on our Kubernetes (K8s) deployment[9], which increases the decoupling further by allowing for multiple hosts per cluster (Docker on the other hand only supports one daemon per host OS).

2.1 Ingress and Services

The data flow begins when incoming HTTP requests from users enter the cluster through the *Ingress* resource, which routes these requests to a single `fishing-web-serv` Service - upon matching the domain or request path. As an Ingress itself does not have a port, the relevant port (of the Service) is determined by the *Ingress Controller*.

Our *Services* are introduced to prevent internal *Pods* from being accessed publicly. We have one for both `model-service` and the web application `fishing-web`. Note that in Figure 2, our `model-service` is left out for simplicity, but it follows a similar structure (except for the fact that the *Ingress* plays no role).

Our `fishing-web-serv` Service (ClusterIP by default) listens on a specified port and forwards requests to the corresponding port on the `fishing-web` container within the *Deployment Pods*. Note that the Service IP address does not change as long as the Service exists, even if the underlying Pods (which have dynamic IPs) change.

If the web service needs to call the model service, it uses the `MODEL_HOST` environment variable, which points to the (configurable) URL of `model-service`. We pass these configuration options into containers via environment variables - specifically `ConfigMap` (Key-Value pairs). This enables us to keep (non-confidential) configuration data separated from the application.

The `fishing-model-serv` Service (ClusterIP) facilitates this communication by directing the requests to the `fishing-model-depl`

[8] https://github.com/remla2024-team14/operation/blob/main/docker-compose.yml
[9] https://github.com/remla2024-team14/operation/blob/main/kubernetes/k8s-deployment.yml

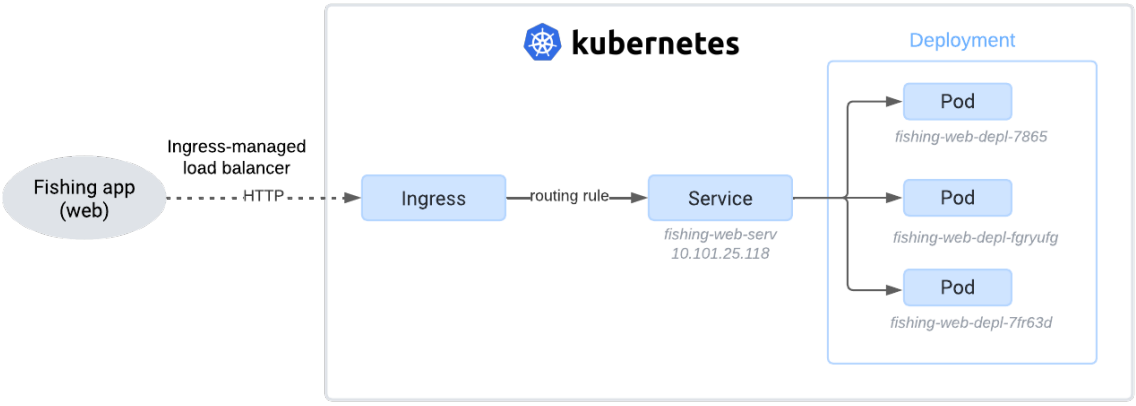

Figure 2: Kubernetes Deployment Overview

Deployment, where the `fishing-model` containers process the model service requests and return the results to the web container.

2.2 Deployments and Pods

Deployments are controller-like resources that allow us to declare a desired state of Pods - the smallest deployable computing units which group one or more containers. This specifically entails ensuring that *Pods* remain alive, apply or revert updates or use to scale up and down replicas of *Pods*. A *Service* essentially acts as load balancer that alternates between the different instances. Furthermore, it is the *Deployment Controller* that reacts to updates and moves from the actual to desired state at a controlled rate.

Our specific `fishing-web-depl` *Deployment* is responsible for managing the web service by maintaining a set of replicas of `fishing-web` Pods to be spawned simultaneously. This exact replication factor is set in our YML configuration. A similar setup is adopted for our `fishing-model-depl` *Deployment*.

In case our URL fishing web app is actually getting popular, but there is not feasible to handle the increasing load with only one *Service*, we could add more in single command: `kubectl scale deployment fishing-web-depl –replicas=10`. As *Pods* are disposable entities (used to run stateless applications), these can be destroyed or created at the will of K8s. Moreover, whenever a Pod goes down, K8s will automatically schedule the creation of a new container to get back to the desired state.

Finally, we use `imagePullSecrets` as authorization token that stores Docker credentials used for accessing our *app* and *model-service* images in GHCR. Note that in this default deployment setting, we always pull the latest versions, whereas our experimental setting (Istio deployment) considers two specific versions - as will be elaborated on in Section 3.5.

2.3 Helm Charts

In order to install complex deployments in our K8s cluster with a single command, we have leveraged Helm Charts [10] i.e. a collection of files describing a related set of K8s resources. These facilitate our deployment process by encapsulating our configurations for the `fishing-model` and `fishing-web` services, along with their respective deployments and secrets. This ultimately ensures that all resources are versioned, templatized, and easily configurable, enabling smooth deployments and rollbacks - as these charts can be installed more than once. Moreover, it is possible to simultaneously run multiple installations of the same chart, as long as the chart supports it.

[10] https://github.com/remla2024-team14/operation/tree/main/kubernetes/helm_charts

3 EXPERIMENTAL SETUP

The goal of the experimental setup is to allow us to easily deploy and evaluate two different versions of the app, which is the frontend component of the application that allows users to make calls to the inference server and receive predictions regarding whether or not a certain link is marked as a phishing URL. This section describes the way our repositories - specifically the `appfrontend` and app-service components - were configured to make it possible to easily create and test hypotheses related to new additions to the app. This was done by configuring an Istio service mesh, changing different image building and pushing steps in our existing Github workflows, and using Grafana and Prometheus to scrape and display metrics related to the web-app and the inference server.

In order to **qualitatively** compare the old version of the web-app with a newer, experimental version, the Istio service mesh deploys separate versions as part of the same deployment, each using a differently versioned image tag that is pushed by the Github workflow. In order to make a **quantitative** comparison, a Grafana dashboard allows the person conducting the experiment to look at different metrics that are retrieved by Prometheus.

3.1 General Description of the Experiment

Although our Istio service mesh allows one to test any hypothesis they have about potential improvements in the metrics of a web-app, we tested a specific hypothesis to make an illustrative example for the experiment. The hypothesis we test is as follows: *Given a version of the web-app that returns binary predictions (valid/phishing)*

to the user and a version that returns probabilities instead, the user will prefer the version of the web-app that returns binary predictions. After getting predictions from the server, the user is prompted to indicate whether or not they are satisfied with the prediction. The ratio and number of "Yes" and "No" given by the user are collected by Prometheus and displayed by Grafana, and our hypothesis is accepted when there is more "Yes" feedback given in the version of the web-app that returns probabilities.

Naturally, we did not have the time to test the experiment realistically by deploying different versions of the web-app to real-life users and getting their feedback for each version. Therefore, we simulated the feedback given by the users as the aim is to show that our experiment setup allows for a hypothesis to be easily be devised and tested.

3.2 Changes in the User Interface

To test our hypothesis, we had to extend our current version of the web-app to display probabilities instead of binary predictions. The two versions of the web-app can be side-by-side in Figures 3 (binary predictions) and 4 (probability predictions). These two versions of the web-app will be deployed in parallel by our Istio deployment mesh, simulating a canary deployment in which some users get access to the older version and some users get access to the newer version.

3.3 Prometheus Metrics

The `app-service` component of our application is a Flask server that is triggered by the `app-frontend` and makes REST calls to the `model-service` inference server to receive and display predictions. Prometheus offers the *prometheus_client* Python package that decorates Flask calls to get Counter, Gauge and Histogram metrics. The metrics scraped by Prometheus are as follows:

- Counter: the number of "Yes" and the number of "No"s given by the user when prompted for feedback about predictions.
- Gauge: the number of current in-progress requests. This is useful to see if the inference server is "stuck" or if it is handling too many concurrent requests.
- Histogram: the binned values for the total inference time in milliseconds per call, which could be used to experiment and see whether or different models or different input URLs lead to higher inference times.
- Summary: using the Python library *prometheus-summary*, we were able to add a summary of inference times to be scraped.

Although the dashboard shows the above metrics that were scraped from the Flask server, the addition of Prometheus to the Istio service mesh also means that the dashboard can easily be configured to add information about the state of the deployment and the internal communication of the cluster, using Kiali, Jaeger and Prometheus metrics.

3.4 Alert rules

In some extreme cases, instead of just monitoring calls to our application and our inference server, we also want to be alerted in the case of extreme events. For example, we could want to be alerted about the fact that too many calls were made in a short period of time, which could overload the server or cost too much in terms of resource usage. For cases like these, we used the AlertManager to define alert rules in the *alert_rules.yml* [11] file in the *app* repository. The default cases sends an alert if there are more than 10 calls made in 2 minutes, but this can easily be changed. In order to send such alerts, an SMTP server needs to be configured; these settings reside in the *alertmanager.yml* [12] file in the *app* repository.

3.5 Istio Service Mesh Setup

The Istio Service Mesh was configured and used to allow for both versions of the web-app and app-service to be deployed in parallel, and for each version to be fed to a different set of users. Essentially, this is an extension to our base deployment as described in Section 3.3 - now allowing for continuous experimentation with two versions. We call these two versions app-v1 and app-v2, and in the Istio Service Mesh deployment, they both occupy the same port. The app-v1 and app-v2 Pods each use two different, consecutive tags of the Docker images that were pushed to the Github Container Registy.

Each Pod that runs its version of the app is also decorated with the *prometheus.io* annotation. This allows Prometheus to collect the app-specific Gauge, Histogram, Summary and Counter metrics that were defined in the Flask server, that will later be displayed by the Grafana dashboard.

The command `istioctl dashboard prometheus` launches the Prometheus dashboard, which can be used to display the custom metrics we defined but also access a lot of Istio metrics that help monitor the state of the mesh and the health of services. For example, Istio defines metrics like *istio_request_duration_seconds* to return the duration of requests or *istio_response_bytes* to get the size of responses in bytes.

As part of a typical Canary release, a certain, low percentage of users should be presented during the experimentation phase with the new version of the web-app. In order to do this, a VirtualService handles all incoming requests from the Gateway and selects app versions based on a *weight* parameter that sums to 100. In the case of a 90/10 split, 90 percent of users are shown the previous version of the web-app, and 10 percent of users are shown the newer version. In fact, this custom routing is not random - repeated requests from the same origin have a stable routing. We achieved this through using specific header information in the HTTP request.

VirtualService in an Istio Service Mesh also wraps the service into a combination of a proxy and the service instance. This is referred to as the "Sidecar" pattern and comes with many benefits. For example, the proxy handles all incoming and outgoing traffic for the app-v1 and app-v2 service, which includes load balancing. The proxy also collects metrics that can be used to evaluate the health of a sidecar service; for example, *envoy_http_downstream_rq_total* is a metric that collects the total number of requests handled by the Envoy proxy.

3.6 Grafana Dashboard

The Grafana dashboard uses the Prometheus metrics that are scraped from the Flask server that resides in the *app* repository and that

[11] https://github.com/remla2024-team14/app/blob/main/alert_rules.yml
[12] https://github.com/remla2024-team14/app/blob/main/alertmanager.yml

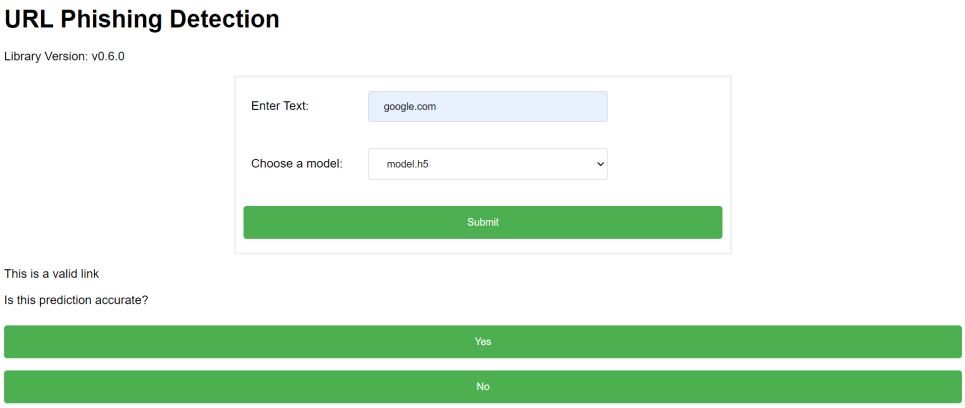

Figure 3: The original web-app, which returns binary predictions.

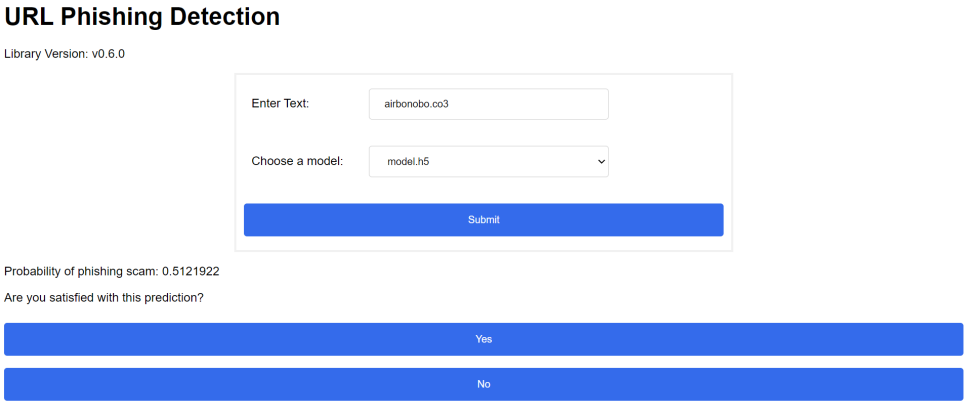

Figure 4: The new version of the web-app, which returns probability predictions.

Figure 5: A Grafana dashboard showing different metrics that were scraped from both versions of the deployed web-app. Each panel shows the metrics for the first version and for the second version of the app.

acts as a connection between the front-end and the inference server. As the list of scraped metrics has been covered in Section 3.3, this section goes over the use of Grafana to visualize these metrics and how the dashboard allows us to validate or reject the previously mentioned hypothesis.

Figure 5 shows an example Grafana dashboard displaying metrics that were scraped from both deployments of the app-frontend and app-service. Here, the metrics are displayed using Gauges (number of Yes and No feedback given), time series (Yes and No feedback given over time) and a histogram (binned inference time in milliseconds). Our previously established hypothesis stated that users would prefer the first version of the app that returned binary predictions over the second verion, and this hypothesis can be accepted as the Grafana dashboard shows the first version receiving one "Yes" feedback point and one "No" feedback point (top left), while the second version received zero "Yes" feedback points and three "No" feedback points (top right).

Although our hypothesis is concerned with the feedback that is given by users, the list of metrics scraped by Prometheus allows for many more different hypotheses types to be tested. This includes but is not limited to: the number of calls, the inference time per call, and the hardware load on each server.

4 ADDITIONAL USE CASE (ISTIO)

In *Istio*, the expansion function of *Egress* is used to manage and control the traffic of external services in the management and control service grid. *Egress* allows the access strategy of external services, including but not limited to the aspects covered below.

4.1 Description of Egress extension

In *Istio*, the *Egress* extension is used to manage and control traffic to external services in the service grid. The *Egress* configuration allows you to specify which external services can be accessed and how they can be accessed, including the URLs to access them, the protocols and ports used, etc. This allows us to effectively manage and control the traffic to external services in the service grid. This allows us to effectively control access to external services.

Secondly, the *Egress* configuration can force all access to external services through security channels (such as TLS) and support identity certification for external services, thereby enhancing security and certification mechanisms.

In addition, as we are implemented in the project now, *Egress* can also monitor external traffic, collect traffic data, and manage the flow limit, load balancing and other management operations to ensure the stability and performance of the service. These functions allow *Istio* to provide comprehensive external traffic management and control capabilities, and improve the security, observation and reliability of the service grid.

4.2 Changes to the Base Deployment

In our specific Kubernetes (K8s) deployment for the URL fishing application, the usual deployment steps include the creation and configuration of namespaces, deployment of application services (such as the fishing-web and fishing-model services), injection of *Envoy* agents, and basic service verification. During the implementation of the Rate Limiting function of *Istio*, we have made the following changes and extensions in the deployment part of the basic design:

4.2.1 Increase ENVOYFILTER configuration. On the basis of basic deployment configuration, a dedicated *EnvoyFilter* resource allocation has been added to define the current limit strategy. This includes detailed rules configured Local Rate Limit, such as the token bucket parameters, Filter configuration, etc.

4.2.2 Enable and verify the flow limit function. After the basic service deployment, a series of operations were performed to verify whether the current limit strategy took effect. This includes a large number of requests through the *POD* (such as Sleep) inside the cluster to observe the actual effect of the current limit strategy.

4.2.3 Monitoring and log analysis. Increase the logs and statistical information of the *Envoy* agent to verify and monitor the implementation of the current restriction strategy. By viewing *Envoy* statistics (such as Ratelimit -related counter) and agency logs, you can confirm whether the current limit rule is triggered.

4.2.4 Adjustment and optimization strategy. After the basic deployment configuration is completed, the parameters of the current limit strategy may need to be adjusted according to the actual test results, such as maxtokens, tokensperfill, etc. These adjustments are to ensure that the current limit strategy can effectively protect services from excessive use, and at the same time does not affect normal service requests.

5 ML PIPELINE

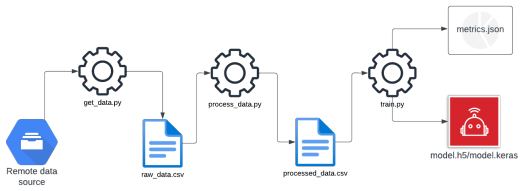

Figure 6: DVC repro pipeline.

In the following sections, we will explain how we made use of specific tools and practices, such as Data Version Control (DVC) and Poetry, to manage the complexities of ML projects. We will detail the configuration of our project and explain why these decisions lead to more efficient and effective ML workflows.

5.1 Overview of ML Training Pipeline

Our ML project is structured into distinct folders and files representing key pipeline stages: data retrieval, preprocessing and model training. A visualization of this process is shown in figure 6. During data retrieval the training, test and validation data is fetched from the database. Preprocessing ensures that this data is formatted in the desired way. Training the model happens as last when it has been ensures that the data is correct and the training configurations have been set. This approach enhances maintainability and clarity, allowing team members to focus on specific aspects of the workflow without conflicts.

We selected Data Version Control (DVC) to manage our ML pipeline due to its integration with existing Git workflows, enabling seamless collaboration among developers. DVC automates the end-to-end process of data fetching, preprocessing, and training ensuring reproducibility and traceability. With DVC we do not have to run the whole pipeline every time we make a change. It will know which stages have changed and only run those. It also supports remote storage, in our case we use an AWS S3 bucket [13], allowing for efficient data management and scalability. Our AWS S3 bucket currently stores our ML model and training data.

To handle dependencies, Poetry was chosen for its intuitive handling of Python packages. This tool simplifies our environment management by storing every dependency in a `poetry.lock` file, enabling reproducible builds and consistent environments across all stages of development and deployment.

5.2 Artifacts

Throughout the pipeline, several artifacts are generated: trained models, raw data and preprocessed data, which are all stored, versioned, and managed via DVC. Performance metrics are generated after model training to evaluate the effectiveness of the model against predefined benchmarks. This way we can easily compare the performance of two different models to each other.

5.3 Model Metrics

Our project employs a set of metrics—accuracy, loss, precision, and recall—to assess the performance of our models. These metrics help in fine-tuning model parameters and identifying areas for improvement. The performance of the model can be seen in table 1.

Table 1: Model Training and Validation Metrics

Metric	Value (%)
Accuracy	0.00022
Val Accuracy	0.000002
Precision	44.6082
Val Precision	44.6080
Recall	99.9593
Val Recall	100.0000
Loss	0.35034
Val Loss	0.35619

5.4 Integration of Linters

To maintain high code quality, we integrated *pylint* and *flake8* into our development process. Pylint is configured to reflect the specific needs of our machine learning project. We have thoroughly analysed linter rules and made modifications accordingly to adapt it to this specific ML project. These modifications include:

- Allowing commonly used variable names in ML such as `X_train`, `Y_train`.
- Defining a set of bad names to discourage non-informative variable names such as *foo*, *baz* and *toto*.
- Extending the list of exceptions that will emit a warning with common errors in ML such as `ArithmeticError`, `BufferError` and `LookupError`.
- Ignoring files that are either auto-generated or do not contain Python code such as *.git*, *build* and *dist*.
- Displaying only warnings with high confidence levels and those that lead to inference errors.

For *Flake8* the maximum allowed line length is set to 100, which is in line with Pylint, instead of 88 which is the default value.

Allowing commonly used variable names helps maintain consistency with common ML practices, making the code more readable. Encouraging clear variable names helps team members understand the codebase faster. By customizing the rules, we avoid unnecessary warnings and make the linter's output more relevant and actionable.

Handling a larger range of exceptions can prevent runtime failures, especially in production ML systems where such failures can be costly.

Ignoring irrelevant files improves the performance of the linting process, making it faster and more efficient, which is beneficial when you want to scale up your project.

By focusing on high-confidence warnings, we can focus on the most likely and impactful issues, improving code quality more effectively.

These linters enforce coding standards and detect syntax errors and stylistic issues, helping to keep the codebase clean and readable.

5.5 Limitations and Opportunities

The dataset we use for this project is very simple and the model as well. As datasets and model complexity grow, the processing power and memory requirements can exceed the capabilities of our current infrastructure, leading to slower development cycles.

Automating more components of the ML pipeline, such as hyperparameter tuning and model validation for more complex models, could further streamline development and deployment processes.

We could use other tools to configure our ML pipeline such as *ML flow*. This tool provides more comprehensive management for ML projects, including experiment tracking, model serving, and a central model registry, which can enhance collaboration and model lifecycle management.

As an alternative to Poetry for package management we could use *Conda*. *Conda* offers robust environment management and support for non-Python dependencies, which could be beneficial for projects requiring a broader range of language support.

6 ML TESTING DESIGN

We utilized PyTest to automate our tests, focusing on various aspects from feature generation to model deployment. Given that our dataset initially does not contain explicit features, our testing strategy addresses this by simulating feature creation and evaluating these features as though they were part of the original dataset in order to demonstrate how tests for features could be implemented.

6.1 Features and Data Tests

Our tests cover the scripts that generate input features from the dataset, crucial for both training and serving phases. We developed

[13] https://team14awsbucket.s3.amazonaws.com/

tests to assess the computational and memory costs associated with each feature. We have also implemented scatter plots and distribution checks to ensure that each feature's distribution aligns with our expectations e.g. class distribution in figure 7. This is vital for detecting anomalies in feature engineering which could lead to biased or incorrect model training.

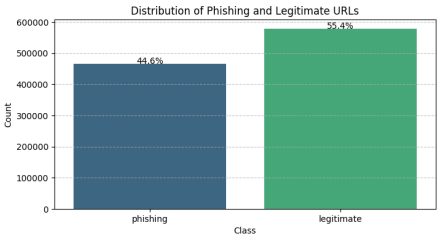

Figure 7: Class distribution of the data

6.2 Model Development Tests

Our tests include mutamorphic tests such as checks for non-deterministic behaviour by the ML model in the outputs with different seeds. We do this by training the model with two different seeds and then we assess that the model performance varies significantly with different random seeds.

We also utilize data slices to assess model performance across various segments of data. This helps in identifying any model biases or weaknesses in handling diverse data scenarios, thereby ensuring the model performs well across different feature-based groups.

6.3 ML Infrastructure Testing Tests

Before a model is deployed, we perform quality checks to verify its predictive accuracy and other performance metrics by checking if the model is making the right predictions on URLs we already know the outcome of. As mentioned before in section 5.3 this is done during training and testing of the ML model. This ensures that only models meeting our quality standards are moved to production.

6.4 Monitoring Tests

We can monitor for any regressions in serving latency, or RAM usage of each test using `pytest-monitor`. These monitoring tests are vital for maintaining system health and ensuring that our ML infrastructure remains efficient and scalable over time. A visualization of these test results can be seen in figure 8.

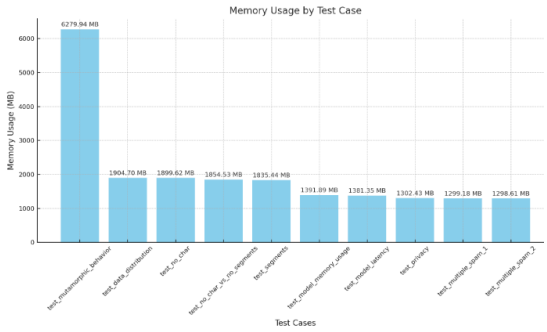

Figure 8: Ram usage test results

6.5 Test Adequacy Metrics and CI Integration

To measure the effectiveness of our testing strategies, we utilize adequacy metrics that provide quantitative feedback on the coverage and completeness of our tests. These metrics include "statements" (The total number of statements in the package), "covered" (The percentage of statements that were executed during testing) and "missed" (The number of statements that were not executed during testing) which help us understand the extent to which our tests exercise the code under various scenarios. The testing adequacy results we obtained are displayed in figure 9. It displays the percentage of statements that were covered by the test.

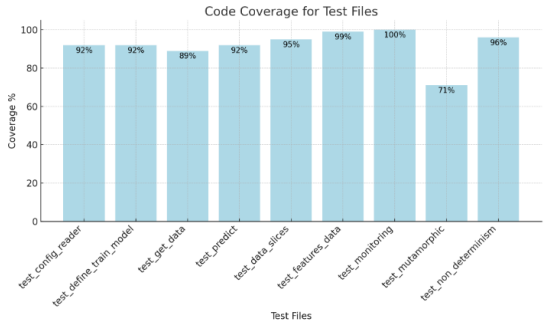

Figure 9: Test adequacy results

6.6 Limitations and Opportunities

Currently, our dataset is relatively simple. While this simplicity aids in initial testing phases, it inherently limits our ability to comprehensively test for non-determinism across more intricate models or variable training environments. The simplicity of the data might prevent us from uncovering anomalies that only emerge with more complex datasets. Expanding the complexity of our dataset by having more features could also enhance the robustness of our tests. By incorporating a wider array of data scenarios, we can better simulate real-world complexities and uncover deeper insights into model behavior across varied conditions.

Developing and integrating advanced validation methods to assess the training process can significantly strengthen the credibility of our model. Techniques such as cross-validation, ongoing training monitoring, and adaptive testing strategies could be helpful in detecting and rectifying training deficiencies.

The monitoring systems we have in place may not be adequately sensitive to detect subtle or slowly developing issues within the system. This could delay the identification of potential problems until they significantly impact the system's performance or stability.

7 EXTENSION PROPOSAL: ROLLBACK STRATEGY

We believe that the current troubled point is that the current project lacks an effective rollback option, outside of Kubernetes. This means that when releasing a new version to GitHub, if a problem or error occurs, there is no way to quickly revert to the previous stable version. Consequently, when a problem is discovered afterwards, the lack of a rollback mechanism significantly increases the time to fix it, leading to longer service outages.

The proposed project refactoring and extension aim to address this issue by improving the repository structure and incorporating automated rollback strategies. The visualization below outlines the new repository structure and introduces scripts and configuration files designed to streamline deployment and rollback processes.

Current Structure.

```
remla2024-team14
 app
    src
    tests
    Dockerfile
 model-service
    src
    tests
    Dockerfile
 model-training
    src
    tests
    Dockerfile
 operation
    src
    tests
    Dockerfile
 lib-ml
    src
    tests
    Dockerfile
 lib-version
     src
     tests
     Dockerfile
```

Improved Structure.

```
    remla2024-team14
 backend
    model-service
       src
       tests
       Dockerfile
       rollback-scripts
           rollback.sh
           deploy.sh
    model-training
       src
       tests
       Dockerfile
    lib-ml
       src
       tests
       Dockerfile
    lib-version
        src
        tests
        Dockerfile
 frontend-app
    src
    tests
    Dockerfile
 docs
    architecture.md
    api-docs.md
    user-guide.md
 tests
     unit
     integration
     e2e
```

In the restructuring of the project, we introduced the idea of grouping all backend related services and libraries into a separate backend directory, while placing the frontend applications under the frontend-app directory. This separation increases the modularity of the project, makes it easier to maintain and extend, and improves the efficiency of the build and deployment process. At the same time, keeping documentation and tests in the top-level directory supports common resource management and efficient collaboration across modules. This restructuring is intended to simplify the onboarding process for members unfamiliar with the project, as well as improve the implementation of automated deployment and continuous integration.

This proposed rollback mechanism is an important tool for ensuring application stability. After deploying a new version, if problems or errors are found, it is possible to quickly revert to the previous stable version, reducing service interruption time and ensuring system stability and reliability Moreover, every time a new version is deployed there is a certain amount of risk. Rollback mechanisms can be used as a last resort protection measure to quickly restore services

7.1 Rollback Strategy Suggestions

7.1.1 Configuring Automated Rollback with CI/CD Tools. Automated rollback is key to ensuring that you can quickly revert to a previous stable version in the event of a deployment failure.[14]

7.1.2 Blue-Green Deployment. Blue-Green Deployment is a seamless rollback strategy that retains the current version (blue) and the new version (green) of the environment and switches traffic to the new version when it passes validation. If there is a problem with the new version, you can immediately switch back to the old version.[15]

7.2 Evaluation of Extension

7.2.1 Testing of the automated Rollback with CI/CD Tools. First, set up a test environment that closely resembles the production environment, ensuring that CI/CD tools (e.g., GitHub Actions) are properly integrated. This step is crucial for simulating real-world conditions and validating the rollback mechanism accurately.

Next, deploy a new version in the test environment through the CI/CD pipeline. Introduce deliberate bugs in this version to ensure that it triggers a rollback condition. This helps in testing the effectiveness of the automated rollback mechanism under controlled failure scenarios.

During the deployment process, closely monitor the progress to confirm that the automatic rollback mechanism is activated correctly if the new version fails to deploy. The system should revert to the last stable version without manual intervention.

After the rollback is triggered, validate the results by checking the rollback logs and system status. Ensure that the rollback was successful and no additional issues have arisen. Automated test scripts can be employed to verify that the system functionality has returned to normal.

Finally, use monitoring tools such as Prometheus and Grafana to analyze metrics during the deployment and rollback process. This helps in assessing whether performance and availability were

[14] https://docs.github.com/en/actions
[15] https://adamtheautomator.com/kubernetes-blue-green

significantly impacted during the rollback, ensuring that the system remains stable and reliable.

7.2.2 Testing for Blue-Green Deployments. Setting up the Blue-Green environment involves configuring the blue-green deployment policy in the test environment to ensure that both blue (current stable release) and green (new release) environments exist. Once the environments are configured, the next step is deploying a new version to the green environment. After deployment, perform basic functional tests to ensure that the new version operates normally in the isolated environment.

With the new version verified in the green environment, traffic switching can commence. Start by redirecting a portion of user traffic to the green environment and monitor the performance and any errors associated with the new version.

If any issues are detected in the green environment, promptly switch the traffic back to the blue environment to ensure user impact is minimized. Investigate and resolve the problems identified in the green environment.

If the green environment passes all tests without issues, proceed with a full release by switching all user traffic to the green environment. Continue monitoring for a period to ensure the new release remains stable.

Finally, verify the results by checking the deployment logs and system status for both blue and green environments. Ensure the traffic switchover and any necessary rollback processes occurred smoothly, maintaining system stability and a positive user experience.