Project Report – Group 7

Melle Koper, Alex Ivanov, Arjun Vilakathara, Kevin Tran

1 RELEASE PIPELINE

This section documents two release pipelines in our project. The release pipeline in `lib-ml` will serve as an example for the release of a software package. The `model-service` will serve as an example for releasing a container image in our project. The workflows introduced in this section are single-purpose release workflows that do not contain any code testing. The goal of this section is to inform and educate potential contributors on our release structure.

1.1 General Tools

We use Poetry for package and dependency management, version tracking, and version bumping. To reference our own Poetry projects, we use version-tagged Git dependencies, leveraging the semantic versioning scheme built into Poetry. Python with Poetry has excellent support for Git dependencies with the added benefit that package names are consistent with the `org/repo` structure. GitHub releases and Git tags are based on the Poetry version rather than allocated; our workflows fail gracefully when attempting to release a version that already exists.

1.2 ML Library Release Pipeline

The release and pre-release workflows for `lib-ml` are relatively straight forward through the use of the tools mentioned in section 1.1. Pushing to the main branch automatically triggers the following release workflow:

(1) Checkout code.
(2) Setup Python.
(3) Install Poetry (only for version management)..
(4) Extract the project version and create a GitHub release and a corresponding Git tag (`v1.2.3`).
(5) If successful, push a patch-level (`1.2.4`) version bump commit on the current branch as a GitHub bot.

Pre-releasing involves one adjustment to this pipeline: we *first* push a prerelease-level (`1.2.3a1`) version bump commit on the current branch, and *then* create a GitHub pre-release and a corresponding Git tag (`v1.2.3a1-<branch>`).

The main consideration given these choices is that developers should always make sure to merge the version from the main branch into theirs before finalizing pull requests. Moreover, if you prerelease your feature branch, you should make sure to pull the automatic version bumps locally afterwards.

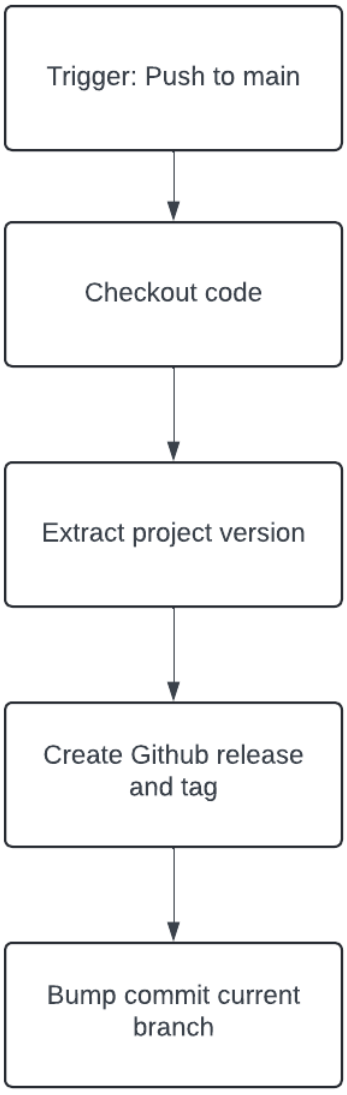


Figure 1: Flowchart for the `lib-ml` release pipeline

1.3 Model Service Container Image Release Pipeline

Releasing the `model-service` extends the software package release pipeline by introducing three more steps to publish the container image that require information flow:

(1) Checkout code.
(2) Setup Python.
(3) Install Poetry (only for version management).
(4) Extract the project version and create a GitHub release and a corresponding Git tag (`v1.2.3`).
(5) The released version is saved in the workflow step outputs upon successful GitHub release. Read the saved version to configure image metadata such as name, tags, and labels.
(6) Log in to GitHub Container Registry using a job token with permissions to create packages.

(7) Build and push the image given the metadata (obtained from the workflow step outputs). This produces the artifact `ghcr.io/remla24-team7/model-service:1.2.3`.
(8) Finally, push a patch-level version bump commit on the current branch as a GitHub bot.

2 DEPLOYMENT

The deployment structure of our project is displayed in Figure 3. The application is deployed in a multi-node Kubernetes cluster using K3S. The cluster contains three nodes: the `controller`, worker `node1`, and worker `node2`.

Controller The `controller` node, also known as the control-plane node, is the *K3S server node* and is responsible for managing the overall state of the Kubernetes cluster. It runs the key components in the kube-system namespace. Unlike worker nodes, the `controller` node does not run application pods but focuses on maintaining the health and desired state of the cluster. The `controller` node also hosts Prometheus, Grafana and Kiali for monitoring, further explained in subsection 2.2.

Node1 Worker `node1` is a *K3S agent node* and is responsible for running the app pod. This node is receives tasks from the `controller` node. The application pod deployed here handles incoming REST requests from external clients and communicates with the `model-service` pod on worker `node2` via REST to query the model. This inter-node communication allows for the distribution of workloads and efficient resource utilization across the cluster.

Node2 Worker `node2` is a *K3S agent node* and hosts the model-service pod. This node operates similarly to worker `node1` but is dedicated to running the model-service. When the application pod on worker `node1` sends a REST request, the `model-service` pod on worker `node2` processes the request and queries the model. The worker nodes purposely have separate functions to allow for more efficient management of resources, ensuring that the `model-service` can handle requests independently of the application logic.

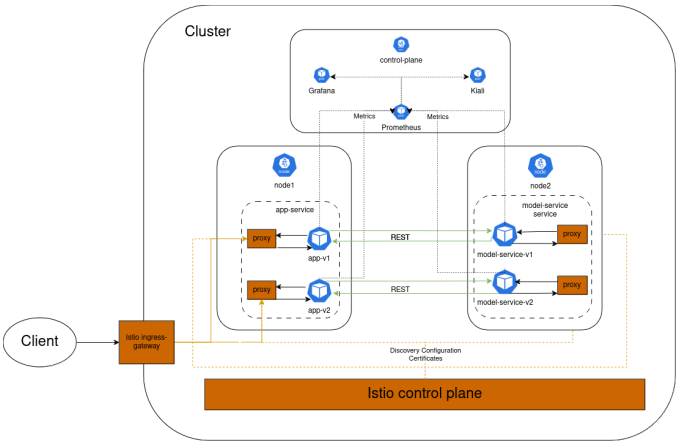


Figure 3: Deployment diagram

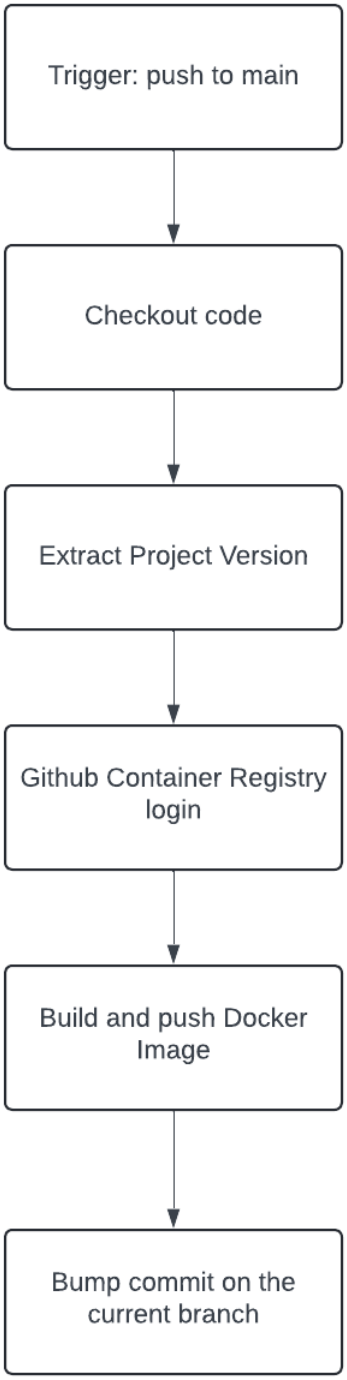


Figure 2: Flowchart for the Model-service release pipeline

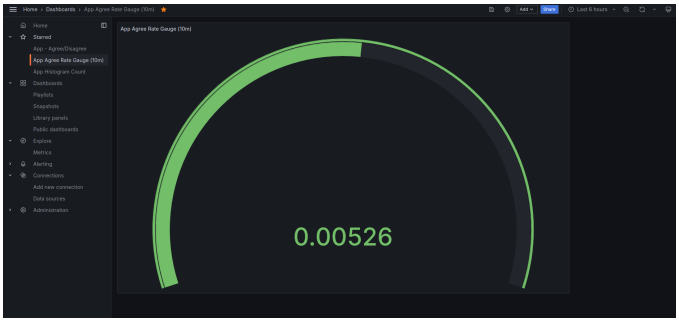


Figure 4: Gauge metric for agree rate within a time interval of 10 minutes

Figure 5: Counter metric for number of times user agrees with the prediction vs. number of times user disagrees with the prediction

2.1 User Request Flow:

(1) Client Interaction: A user initiates a request from their device, accessing the web application through a web browser.
(2) Ingress Gateway: The request enters the Kubernetes cluster via the Istio-managed Ingress Gateway, which serves as the main entry point for all external traffic.
(3) Routing: The Ingress Gateway forwards the request to the appropriate version of the service based on rules defined in the Virtual Service. The Virtual Service routes the request to the app.
(4) Frontend to Backend: The Frontend Application processes the request and forwards it to the model Service.
(5) Backend to Frontend: The model service uses the pre-trained model to process the request and send back the result to the frontend. Once the Model Service processes the request and returns the predictions, the App presents the results to the user.

2.2 Monitoring:

(1) Metrics Collection: Prometheus continuously scrapes the defined metrics from all services which are then used by Grafana for further visualization.
(2) Grafana, connected to Prometheus, uses these metrics to generate real-time visualizations and dashboards that can be used to monitor the application and can be found in figure 4, figure 5 and figure 6.

Figure 6: Histogram metric for time taken per prediction in seconds.

2.3 Traffic Management

Effective traffic management is crucial for maintaining the reliability and security of applications deployed within a Kubernetes cluster. Traffic management involves controlling the flow of traffic between services, ensuring that requests are routed optimally and that the system can handle different scenarios such as traffic spikes, service failures, and version upgrades.

We utilize the Istio service mesh to manage traffic within the K3S cluster. Istio provides a powerful suite of tools for fine-grained control over service-to-service communication. The two main features that we implemented are:

- **Traffic Routing:** Istio allows us to define rules for routing traffic between different versions of services. This allows for continuous experimentation where a small percentage of traffic is directed to a new version for testing before full deployment.
- **Rate limiting:** Istio provides rate limiting of traffic to prevent overcrowding of an application, this is further explained in section 4

In our implementation, we have deployed a second version of the ‘app‘ and ‘model-service‘ and configured Istio to route 90% of the traffic to the original version and 10% to the new version. This setup is managed through Istio’s VirtualService and DestinationRule resources, which define the routing logic based on service labels and other criteria.

The following diagram (Figure 7) illustrates how traffic is managed and routed within our service mesh. The Istio Ingress Gateway serves as the main entry point for external traffic, directing requests to the appropriate service version based on the defined routing rules. This setup ensures that we can deploy updates with minimal risk and closely monitor the performance and stability of new service versions.

3 EXTENSION PROPOSAL

Our current deployment process for the app and model service involves several manual steps, such as modifying the `/etc/hosts` file and manually ensuring the model files are placed in the correct repository. Also, to deploy our app, model-service and Service Monitors, multiple Kubernetes configurations need to be applied with `kubectl`. This method is prone to human error and goes against the idea of automated release engineering. It might also lead to inconsistencies between different environments due to unnecessary complexity of manual commands. To improve on the deployment,

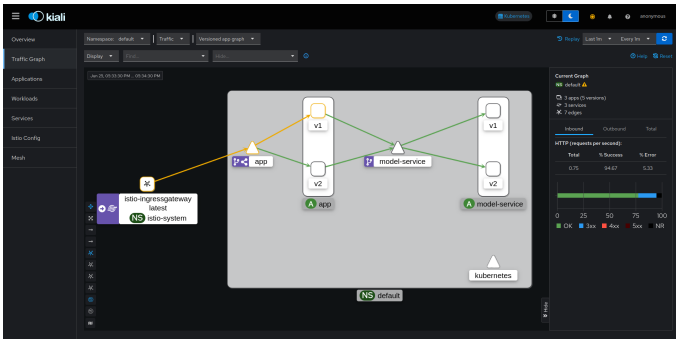


Figure 7: Istio Traffic Management visualized in Kiali dashboard. HTTP requests sent to v1 of the app will also strictly be sent to v1 of the model-service, this also applies for v2 of the app and model-service.

in the future we could make use of Helm charts for complete deployments that can be installed multiple times into the same cluster. The same Helm chat can be reused across different projects, which improves reusability of the application. The 'How to deploy application on Kubernetes with Helm' by [5] provides a clear instructions on how to refactor our deployment from a multitude of YAML files into a single Helm chart.

An additional useful plugin for Helm that can be used for secret management is 'helm-secrets'. The helm-secrets plugin stores the encrypted secrets in Helm charts such that any sensitive data is kept secret. In our case this would be mostly relevant for the admin authentication bearer token for the Kubernetes Dashboard as well as the admin password for Grafana. The following article 'How to Handle Secrets in Helm' [3] gives a clear step-by-step process on handling secrets in Helm charts and can be relevant for our refactoring.

To test the effectiveness of the Helm chart deployment, we will:

(1) Deploy in a Staging Environment: Initially deploy the application using Helm in a staging environment such as a separate Minikube cluster with the same configuration and resources to verify the deployment.
(2) Automated Tests: Run automated tests to ensure that all components are correctly deployed and functional, these tests should include smoke tests, unit tests and integration tests as mentioned by Ellingwood as well as in the lecture [2] and perhaps system testing in case we want to deploy multiple applications and clusters.
Smoke tests include verifying that all critical resources are running and that all expected resources are available (e.g. pods, services, service monitors, endpoints) are all running and healthy. Integration tests includes hooking up multiple components and ensuring they can communicate and work as intended, such as making sure that the control-plane can communicate with all worker nodes and that the bridging has been set up correctly.

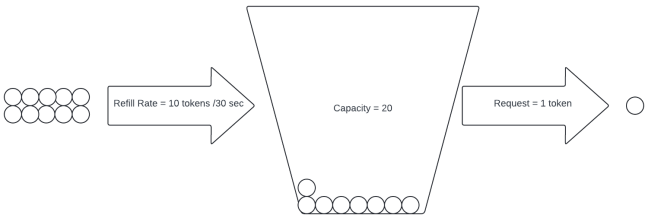


Figure 8: Figure displaying the mechanics of the token bucket algorithm.

4 ADDITIONAL USE CASE (ISTIO)

As an additional use case, we chose to implement rate limiting using the Istio service mesh. The goal of rate limiting is to limit the number of requests a client can send to a server. Naturally, this is important for our service as we do not want the application to break down due to overcrowding or in the worst case, bad actors.

Through the use of an EnvoyFilter, we implemented a local rate limiter. We chose to implement the local rate limiter instead of a global one, because it is not reliant on a rate limiting service. The EnvoyFilter implemented works through a token bucket algorithm. Each client has a bucket with a limited capacity. As displayed in figure 8, The bucket gets refilled at a constant rate, however every requests costs the client one token. If the client's bucket is empty, their request gets denied. In our application, a client has a bucket capacity of 20, that gets refilled with 10 tokens every 30 seconds. Meaning, if a client sends 21 requests to the app within 30 seconds, their 21st request is denied. Our implementation was aided by the tutorial published by Peter Jausuvec [4].

5 EXPERIMENTAL SETUP (ISTIO)

Continuous experimentation is a core component of modern software development, particularly for web applications where user engagement and system performance are critical metrics for success. This section of the report discusses the setup and implementation of the experiment designed to evaluate two versions of our phishing prediction application. The aim is to determine whether the changes made in the newer version improve user engagement in a measurable metric.

Experiment Setup

As part of the experiment setup, we have two versions of the application:

- **Version 1 (Control):** This version presents the core functionality without any advanced styling or user guidance. Shown in image 9.
- **Version 2 (Test):** Improvements include the addition of CSS for improved overall aesthetics and user instructions to aid in navigation and usability. Shown in image 10.

The experiment hypothesizes that Version 2, with its user interface enhancements and additional instructions, will lead to increased user engagement and improved operational metrics. Specifically, we expect Version 2 to show higher interaction through increased views, requests, and user feedback actions (agree/disagree).

Figure 9: Control version of the website.

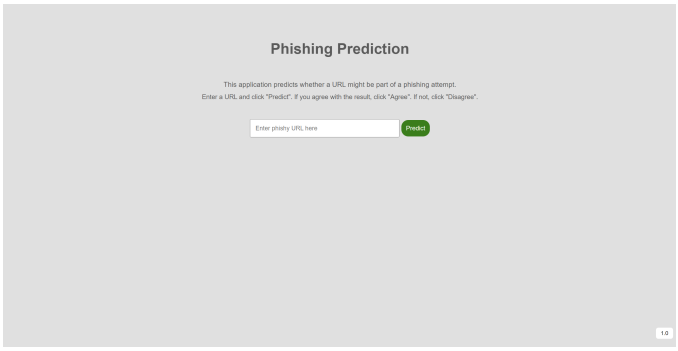


Figure 10: Test version of the website.

Metrics

To objectively evaluate the impact of the changes between versions, the following app-specific Prometheus metrics were selected:

- **Views Counter:** Counts how often the website has been loaded, indicating user interest.
- **Requests Counter:** Tracks the number of predictions users request, reflecting user interaction.
- **Agree/Disagree Counters:** Measures the frequency of user responses to predictions, providing insight into user agreement and engagement, and their interest in the application beyond just prediction.

Both versions of the application are deployed, and we then control the incoming traffic to divert users to either version of the application through a determined weighting. Prometheus is then configured to scrape metrics from both deployments at specified intervals, ensuring that data collection is consistent and reliable. For this part of the report, experimentation is done using the Docker deployments, to allow for easier explanation.

Grafana Dashboards

Grafana dashboards are configured to visualize metrics from both versions simultaneously, enabling real-time monitoring and comparison. Each metric is displayed in separate panels, with filters allowing us to view data from either version or both together for direct comparison. For the metrics defined above:

- **Views Counter Dashboard:** This metric counts how often the website has been loaded, indicating user interest. The values can be shown in a Time Series or Bar Chart. To compare we create a time series graph that plots the views_counter for both Version 1 (Control) and Version 2 (Treatment) over time. By comparing the trends and total counts, you can determine which version gets more views and engagement. A consistently higher line or taller bars for Version 2 would suggest that its UI enhancements are effectively attracting more users.
- **Requests counter dashboard:** This metric tracks the number of prediction requests made by users, reflecting their active engagement with the application's core functionality. The values can be visualized using a Time Series or Bar Chart. We create a time series graph to display the requests_counter for both Version 1 and Version 2 across the same time period. By analyzing the graph, you can evaluate which version experiences more user interaction. A consistently higher line or taller bars for Version 2 would indicate that the improvements in UI and usability are effectively encouraging more user interaction with the predictive features of the application.
- **Agree/Disagree counter dashboard:** The Agree/Disagree Counters measure the frequency of user responses to predictions, providing insights into user agreement, engagement, and their interest in the application beyond just predictions. Visualization with Stacked Bar Charts or Multi-Bar Charts is effective for these metrics. Setting up separate bar charts for each version to display the agree_counter and disagree_counter metrics allows a clear comparison of user feedback. An increase in 'agree' responses, especially for Version 2, might suggest that its enhancements not only enhance user satisfaction but also increase trust in the application's predictions.

6 ML PIPELINE

The ML pipeline is located in our `model-training` repository, where we demonstrate a continuous training GitHub workflow using DVC. Our DVC pipeline consists of three main stages and two additional stages for mutamorphic testing:

(1) Dataset preprocessing.
(2) Training the model.
(3) Evaluating the model on test data.
(4) Generating a mutamorphic version of the training data.
(5) Evaluating the model on the mutamorphic training data.

The Jupyter notebook has been refactored into independent scripts and their configurables extracted into `params.yaml`, which is read by each script via the DVC `params` API. The preprocessing stage depends the existence of the dataset folder, which is managed by DVC; the script itself is just a wrapper around a `lib-ml` method. Additionally, each stage depends on a corresponding section of the parameter configuration, thus reproducing the pipeline also saves the parameter values to the lock file. Stages with no changes to their parameter configuration are therefore not rerun – the saved outputs are used instead.

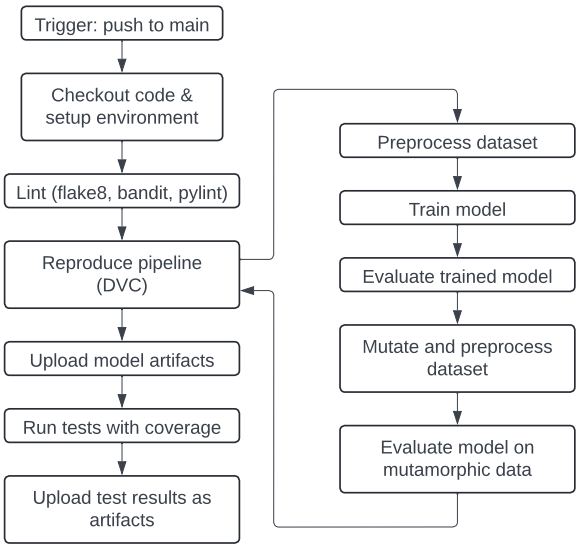


Figure 11: Flowchart for the continuous training pipeline

The GitHub workflow automates the reproduction and testing of this pipeline as shown in Figure 11. It sets up the Python environment, installs dependencies via Poetry, and runs linting checks using `flake8`, `bandit`, and `pylint`. The workflow then executes the DVC reproduction pipeline and reports metrics. Afterwards, the model artifacts (tokenizer, encoder, and trained model) are bundled and uploaded as attachments, making them accessible for deployment or further analysis. Lastly, it runs our test suite with coverage analysis and uploads the results thereof.

Even though our continuous training pipeline enables the deployment of models trained in a workflow by downloading the artifacts from GitHub, our project does not automate this step; instead, we maintain `model.dvc` fixtures pointing to specific DVC-managed artifacts in our `operation` and `model-service` repositories. Releasing versioned/labeled model artifacts as part of the pipeline as well as consuming them elsewhere is left as future work.

7 ML CODE QUALITY

Pylint is a great tool to provide some metrics on the code quality of a project. However, although a lot of the configurations are valid, some of them need to be modified to fit an ML context. This section will go through what changes were made.

Original Limitations and Improvements Through Changes:

(1) **Whitelisting of ML-centric C extensions:**
- **Original Limitation:** Pylint's default settings do not specifically accommodate the loading of C extensions common in ML libraries, which can lead to false positives during code analysis.
- **Improvement:** By explicitly whitelisting extensions for popular ML libraries like `numpy`, `scipy`, and `pandas` (although we don't use all of them), Pylint can better handle and support these libraries, reducing false positives related to external module usage.

(2) **Relaxing variable naming conventions:**
- **Original Limitation:** Strict restrictions on naming styles (e.g., `snake_case` for variables as default by pylint) do not align with common ML practices where short and specific names like X for features and y for targets are commonly used.
- **Improvement:** Relaxing naming rules to accept commonly used identifiers in ML improves the readability of code and decreases the false positives within the ML context. It makes the linting more inline with the domain-specific conventions in ML development.

(3) **Increased thresholds for complexity metrics:**
- **Original Limitation:** Default complexity metrics in Pylint (e.g., maximum function arguments or branches) are potentially too restrictive for ML code, which could have very complex functions for model definition and data processing.
- **Improvement:** Adjusting these metrics to allow for higher complexity reflects the practical needs of ML codebases and experimentation as it is possible to make very complicated functions. This change helps avoid unnecessary warnings and enables developers to focus on genuine code quality issues during development and experimentation rather than trying to accommodate standard practices into ML contexts when it makes work more complicated.

(4) **Ignore code duplication:**
- **Original Limitation:** By default, Pylint flags duplicate code blocks as warnings or errors. While this is generally a good practice to maintain code cleanliness and extension, it does not suit the iterative and experimental nature of ML development which can often include rewriting the same code for a different model such that the developer can experiment with tuning parameters or functions.
- **Improvement:** Disabling the duplicate-code check can be useful in ML projects where experimentation can lead to repetitive chunks of code, especially during the testing of different model configurations, hyperparameters, or data subsets.

Addition of New Rules for ML:

(1) **Signature Mutators for ML Frameworks:**
- **Original Limitation:** Pylint, by default, does not accurately handle the dynamically modified function signatures typical in ML frameworks like TensorFlow and PyTorch, leading to incorrect linting.
- **Improvement:** Introducing 'signature-mutators' for these frameworks ensures that Pylint recognizes and correctly handles changes made by decorators and other dynamic features used in ML. This adaptation would reduce false positives related to function signature issues.

These changes make Pylint a more effective tool for ML projects by aligning its functionality with the unique patterns and practices found in ML development.

8 ML TESTING DESIGN

In this section, we will discuss the testing approach taken to validate the machine learning aspect of our application. First we will cover the testing strategy. Then we will elaborate on the tests implemented to execute the testing strategy. After which, we will elaborate on the testing results of our pipeline. Finally, we cover the limitations in our strategy, the future opportunities to improve the testing coverage, and the lessons learned from our implementation.

8.1 Testing Strategy

In order to ensure the consistency, robustness, and accuracy of our ML application, we have created the follow testing strategy:

8.1.1 Testing Framework. The main testing framework consists of various automated tests, the advantages of which are discussed in 8.1.2. When a change is pushed to the main branch of the Model-Training branch of the application repository, a testing pipeline is triggered. First the workflow builds the project, installs all necessary dependencies, and pulls all necessary data from the DVC Google Drive. Once all this is complete, the workflow executes all automated tests implemented in the model-training repository.

8.1.2 Automated Tests. The goal of our testing strategy is to ensure the continuous performance and health of our Machine Learning Application we have implemented Automated tests. Automated testing brings with it various significant advantages. They ensure the reliability and robustness of the model by consistently verifying that the application and its services function as expected, this is also done through the development life cycle. The continuous validation helps to identify and address any potential issues or bugs early when introducing changes to the code base, which in turn reduces the risk of deploying a faulty model. In addition, automated tests enhance the reproducibility of any experiments run in the future, as they can run the same tests under different conditions, while ensuring the results are consistent. Finally, automated tests are more efficient when compared to manual testing, and will allow developers working on the project to commit more resources to further develop the application rather than spending resources verifying past implementations.

8.1.3 Continuous Training. Through Github Actions we have implemented continuous training. This is crucial in any machine learning application to maintain accuracy, relevance and performance over time. As a domain changes and evolves, models tend to drift. This means the models predictive power becomes less due to changes in underlying patterns. This is especially relevant in detecting phishing links, as bad actors will continuously change their approach to remain undetected. With continuous training, it is necessary to test the newly trained model to ensure it performs in the appropriate manner.

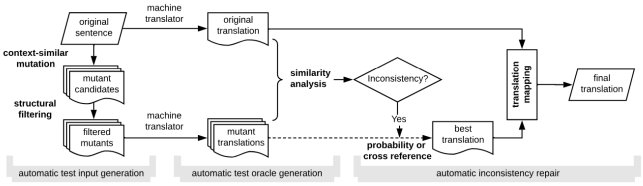


Figure 12: Graphic representing generation of mutated data, graphic comes from Automatic Testing and Improvement of Machine Translation [6]

8.2 Implemented Tests

To execute our testing strategy, we implemented various types of tests based on the ML Test Score guidelines [1]. these test are meant to evaluate the following:

(1) The models capabilities
(2) The features and the Data
(3) The model development
(4) The ML infrastructure
(5) Monitoring
(6) testing Non-Determinism
(7) Testing with data slices

8.2.1 Testing Model Capabilities. To test the capabilities of the model, we evaluate it on various aspects such as accuracy and robustness. To test the performance of the model, we evaluate the model on test data, which the model should not have seen during it's training phase. The testing pipeline returns a classification report generated by sklearn, as well as a confusion matrix, roc curve, and roc auc score.

We test the robustness of the model through mutamorphic testing. Our implementation of mutamorphic testing creates a new data set from the original test set.

This "data creation" works as follows, from each entry in the original set we generate various mutants. These mutants are then vetted through a sanity check, each sample is checked to see if it remains a structurally valid url. Then all remaining mutants are converted to machine code through the use of the tokenizer. These converted urls are compared to the tokenized original sample, and only the most similar mutant is kept. Similarity is measured through cosine similarity. This strategy has been inspired by the paper Automatic Testing and Improvement of Machine Translation [6].

Once the mutamorphic data set is created, we evaluate the model with this new data in the same way as the original test set, and we expect the model to perform the same.

8.2.2 Testing Features and Data. The validation of the data is also essential for the proper training and development of the model. Our pipeline confirms the validity of the data through two relatively simple tests. We ensure that each sample is properly labelled through checking whether the labels are legitimate. We then check to see that all entries in the dataset is non-empty.

8.2.3 Testing for Non-Determinism. To ensure model stability and reliability, non-deterministic testing checks for consistent outcomes

across multiple training iterations despite inherent randomness in initial weights or batch shuffling. We train the model two times on the same training data set, recording accuracy each time, and ensuring that variations in testing accuracy among them on a testing dataset do not exceed a pre-defined threshold. This method highlights potential variability in training.

8.2.4 Testing with Data Slices. Data slice testing evaluates model performance across various dataset segments, potentially affected by underlying biases or missing characteristics in the broader dataset. The dataset is segmented into two equal size segments such that one has a 50-50 split of legitimate and phishsing links and the other has a 70-30 split. The model is assessed on these slices to ensure a baseline accuracy is achieved. Passing this test would allow us to be confident in the effectiveness of the trained model regardless of the split between labels in the training dataset.

8.2.5 Testing Model Development. To validate the proper development of the model, we have implemented the following tests:

There is a test verifying the proper loading of the train and test datasets. We then test whether the training of the model is properly executed. This is tested through training the model and analyzing whether the training of the model produced a loss, and whether the training process produced any accuracy metrics.

8.2.6 Testing ML Infrastructure. The current implemented test of the ML infrastructure is testing whether or not the model can be properly built and initialized. This is done through loading the model from the DVC version control and building it.

8.2.7 Monitoring Tests. In our pipeline we have implemented monitoring tests that test for the time to train the model, and the RAM usage of the model. Testing the training time is essential for ensuring the model has an efficient training time and therefore less resource intensive when the model needs to be updated. The RAM usage of the model is important to test to ensure the model does not take up unnecessary resources and maintain efficiency.

8.3 Integration of ML Test Score Guidelines in Our Testing Strategy

Our approach to testing the machine learning components of our application is guided by the outlines in the ML Test Score guidelines[1]. By structuring our tests around these guidelines, we aim to achieve a high ML Test Score, indicating strong automated testing and monitoring. In the development of our testing framework, we focused on several key areas recommended by the ML Test Score guidelines for example:

(1) **Model Specifications and Code Review:** All our model specifications, and our code are regularly code-reviewed by one another during pull requests and are maintained in a version-controlled environment.
(2) **Test model quality on important data slices:** Our use of data slice testing allows us to ensure that our models perform adequately across different segments of data, addressing any concerns about underlying biases or variance in model performance.
(3) **Integration test the full ML pipeline:** Following the ML Test Score guidelines, our automated testing framework is a core component of our ML pipeline. When changes are pushed to our model-training repository, our system automatically builds the project and runs a series of tests, ensuring continuous integration and delivery.
(4) **Test for dramatic or slow-leak regressions in training speed, serving latency, throughput, or RAM usage:** We have tests that check for the resource usage of training the model and ensure that they are within acceptable bounds.
(5) **Test the reproducibility of training:** Our strategy for testing non-determinism involves running multiple training cycles and comparing outcomes to ensure stability, addressing potential variability in model performance as highlighted in the ML Test Score guidelines.

8.4 Testing Results

The results of the tests can be found as an artifact in the actions that is triggered by the push onto the branch.

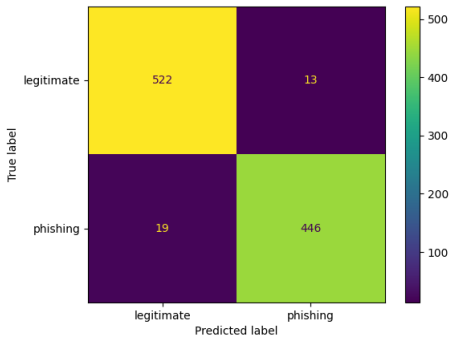


Figure 13: Confusion matrix illustrating the model's performance.

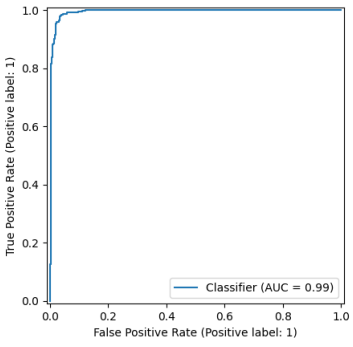


Figure 14: ROC curve illustrating the model's performance.

8.4.1 Performance Evaluation.

- **Confusion Matrix from Figure 13:**
  - True Positives (Legitimate as Legitimate): 522

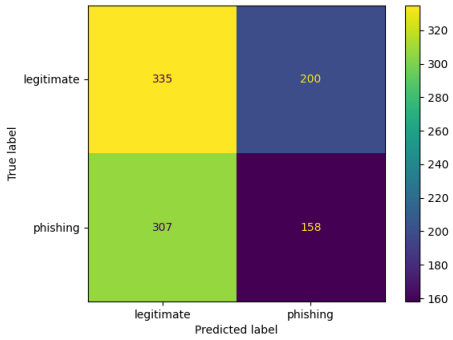


Figure 15: Confusion matrix illustrating the model's performance in mutamorphic testing.

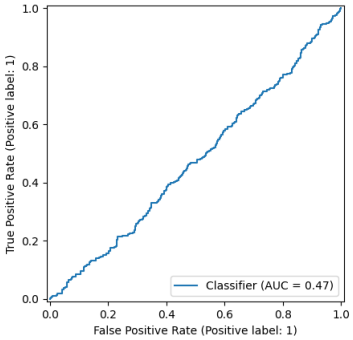


Figure 16: ROC curve illustrating the model's performance in mutamorphic testing.

  - True Negatives (Phishing as Phishing): 446
  - False Positives (Legitimate as Phishing): 13
  - False Negatives (Phishing as Legitimate): 19
- **ROC Curve and AUC from Figure 14:**
  - AUC Score: 0.99, indicating near-perfect performance in discrimination between phishing and legitimate classes.

8.4.2 Performance Evaluation with mutamorphic testing.

- **Confusion Matrix from Figure 15:**
  - True Positives (Legitimate as Legitimate): 335
  - True Negatives (Phishing as Phishing): 158
  - False Positives (Legitimate as Phishing): 200
  - False Negatives (Phishing as Legitimate): 307
- **ROC Curve and AUC from Figure 16:**
  - The AUC of 0.47 indicates a performance worse than random guessing, highlighting a major concern in the model's capability to distinguish between classes. This could suggest severe over fitting on training data or just an issue with the training data we mutate.

8.4.3 Testing Features and Data. Through testing the datasets we confirmed that each sample was correctly labeled and all entries were non-empty, ensuring data quality. This ensures the quality of the dataset used for training.

8.4.4 Testing for Non-Determinism. Consistent outcomes across multiple training sessions demonstrated the model's stability despite randomness in initial weights or batch shuffling. The difference in the accuracy of the two models is lower than 0.06. This suggests that a model trained on a dataset is reproducible.

8.4.5 Testing with Data Slices. Having tested with the two data segments, it was noticed that the accuracy's between the two were different to each other by less than 0.15, which makes sense as in once case there is equal data for both so it ends up training equally for both cases, but in the other case one class had more data than the other so it would make sense how it would perform worse on the class it has less data on (assuming testing done on the same dataset). But the accuracy's being this close and above 0.5 in both cases suggest some level of robustness against the split of class labels on training the model.

8.4.6 Testing Model Development. The code passes all tests we have that check for proper loading and execution of model training, with the successful generation of accuracy metrics and loss reduction. This means that the code we have is valid and can produce results as expected.

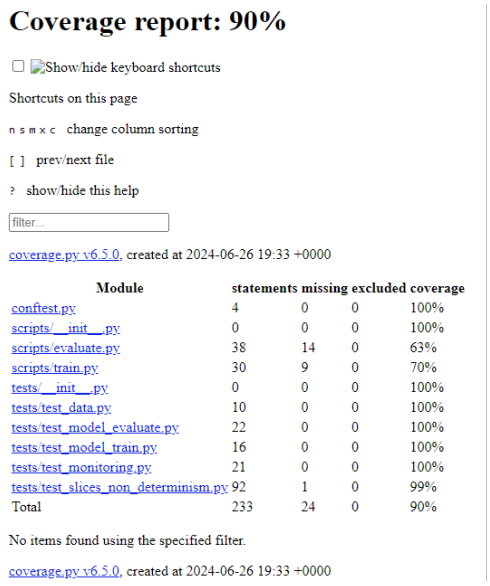
8.4.7 Testing ML Infrastructure. Confirmed that the model could be built and initialized properly.

8.4.8 Monitoring Tests. Monitoring revealed acceptable training times and RAM usage, ensuring the model's efficiency and resource management. The model completed training within the threshold of 60 seconds. The increase in RAM usage did not exceed 100 MB before and after the training on a smaller (10 items) data subset, which considering batch sizes and the size of out dataset is well within acceptable boundaries.

8.5 Code Coverage

Our project's code coverage is at 90%, as shown in Image 17. This metric is important to understand the breadth and depth of our tests. While the overall coverage is high, there are some concerns regarding the coverage of evaluate.py and train.py, which stand out with lower coverage rates of 63% and 70% respectively. In contrast, other files achieve 100% coverage.

The lower coverage in evaluate.py and train.py primarily results from our testing strategy, which focuses on the main functionalities within these files. Secondary functions like save_metrics() (utility functions in the file) and the main execution block (the __main__ block) are not directly tested. Instead, these functions are tested indirectly through duplication in our tests. This approach helps with easier test setup, such as configuring different hyperparameters. As seen in the figure, the test files have 100% coverage, therefore we can be confident in the functionalities of these helper functions.



Coverage report: 90%

Show/hide keyboard shortcuts

Shortcuts on this page

n s m x c change column sorting

[ ] prev/next file

? show/hide this help

filter...

coverage.py v6.5.0, created at 2024-06-26 19:33 +0000

Module	statements	missing	excluded	coverage
conftest.py	4	0	0	100%
scripts/__init__.py	0	0	0	100%
scripts/evaluate.py	38	14	0	63%
scripts/train.py	30	9	0	70%
tests/__init__.py	0	0	0	100%
tests/test_data.py	10	0	0	100%
tests/test_model_evaluate.py	22	0	0	100%
tests/test_model_train.py	16	0	0	100%
tests/test_monitoring.py	21	0	0	100%
tests/test_slices_non_determinism.py	92	1	0	99%
Total	233	24	0	90%

No items found using the specified filter.

coverage.py v6.5.0, created at 2024-06-26 19:33 +0000

Figure 17: Code coverage report.

8.6 Limitations, Opportunities, and Lessons Learned

Although our testing is based on comprehensive testing strategies and we have high coverage of our code with these tests, there are several limitations within our testing and code coverage that should be acknowledged.

(1) **Limited Scope of Automated Tests:** Our current set of automated tests, although extensive, does not cover every possible edge case or inputs. This could lead to unforeseen issues in production environments where unexpected inputs or conditions could occur.
(2) **Code Coverage Gaps:** Although as mentioned before that the gaps in the coverage are due to the code being duplicated into the testing file and that they are indeed tested, this approach to testing is not efficient as if a change is made in the actual code, it would also have to be made in the testing file, otherwise even if the new code is bad, the tests will still pass.
(3) **Dependency on External Tools:** Our testing infrastructure heavily relies on external tools and libraries, which can introduce their own limitations and bugs. This is unfortunately not something that we can address, and we trust that these libraries are well-tested.
(4) **Performance Under Load:** We have not fully tested the system's performance under high load conditions. This is crucial for real-world applications, where high traffic could significantly impact performance.
(5) **Testing of Mutamorphic Data:** The effectiveness of mutamorphic testing is limited by the quality of the mutated datasets generated. If these datasets do not adequately reflect realistic variations, the testing may not effectively measure the model's robustness to real-world data variations. We can see this in how the model trained on the mutated dataset performed poorly (with an AOC of 0.47).

Having discussed the limitations, we can also present some possible opportunities for improvement:

- **Increase the Scope of Automated Tests:** Create more comprehensive tests that explore more edge cases and input permutations. This would ensure code that is more robust to unforeseen issues in production environments.
- **Increase Code Coverage:** Refactor the code in such a way that the functions can be tested easily without having to duplicate the code. We had to duplicate code due to issues with setting hyperparameters also path variables. In the future, we should write our code such that we won't have these issues.
- **Incorporate Load Testing:** Conducting load testing could ensure the ability of the application to maintain its performance and stability under real-world usage conditions.
- **Advanced Mutamorphic Testing Methods:** Improving the generation algorithms for mutamorphic testing could improve the relevance and effectiveness of these tests, making them more reflective of the model's performance in a real application.

REFERENCES

[1] Eric Breck, Shanqing Cai, Eric Nielsen, Michael Salib, and D Sculley. 2016. What's your ML test score? A rubric for ML production systems. (2016).
[2] Justin Ellingwood. 2022. An introduction to continuous integration, delivery, and deployment. https://www.digitalocean.com/community/tutorials/an-introduction-to-continuous-integration-delivery-and-deployment
[3] Guest Expert. 2024. How to handle secrets in helm. https://blog.gitguardian.com/how-to-handle-secrets-in-helm/
[4] Peter Jausovec. 2023. How to configure rate limiter in Istio. YouTube. https://www.youtube.com/watch?v=xvoETrhso8A
[5] Wojciech Krzywiec. 2020. How to deploy application on Kubernetes with helm. https://wkrzywiec.medium.com/how-to-deploy-application-on-kubernetes-with-helm-39f545ad33b8
[6] Zeyu Sun, Jie M. Zhang, Mark Harman, Mike Papadakis, and Lu Zhang. 2019. Automatic Testing and Improvement of Machine Translation. arXiv:1910.02688 [cs.SE] https://arxiv.org/abs/1910.02688