REMLA24 – Team 8

Tip ten Brink (4927192), Felipe Bononi Bello (5069238), Dielof van Loon (5346894)

1 RELEASE PIPELINE DOCUMENTATION

In this section two different release pipelines will be documented. The first of which releases the software package containing the preprocessing logic that is used for both training and new queries. The second one will highlight the release pipeline of the model-service container image. This container image represents a wrapper service for the released ML model. For both release pipelines the different steps will be elaborated.

1.1 Software Package: *lib-ml*

The source code of the released software package can be found on Github [1]. This repository holds the logic regarding managing of the tokenizer, publishing of new releases and the continuous integration and deployment (CI/CD) of the package.

1.2 Pipeline steps: *lib-ml*

The release pipeline is triggered when a new tag is created with the correct pattern `v[0-9]+.[0-9]+.[0-9]+` on the main branch. This triggers the following pipeline steps consisting of two jobs: `build-test` and `publish`. Where `build-test` runs the latest version of Ubuntu to ensure the code is functional and `publish` releases the package on Python Package Index (PyPI) .

Implementation.

- **Triggered**: When a new tag following the correct pattern is created on to main.
- **Environment**: Ubuntu-latest and Python 3.12.

Build-test.

(1) **Checkout code**: Fetches the code from the created tag with `actions/checkout@v4`.
(2) **Poetry install**: Installs `poetry==1.8.3` for dependency management using `pipx`.
(3) **Setup Python**: Setup-py with `actions/setup-python@v5`.
(4) **Install dependencies**: Installs dependencies using Poetry and the `pyproject.toml` file.
(5) **Run tests**: Perform checks on the functionality and versioning using `pytest` and `semver`.
(6) **Build**: Run `poetry build` building it for distribution.
(7) **Upload artifact**: Uploads the artifact to the dist folder of the GitHub workspace with `actions/upload-artifact@v4`.

Publish.

(1) **Download artifact**: Uses `actions/download-artifact@v4` from dist folder.
(2) **PyPI release**: Puts the package on Python Package Index (PyPI) with `pypa/gh-action-pypi-publish@release/v1`.

[1] https://github.com/remla24-team8/lib-ml

1.3 Data Flow and artifacts: *lib-ml*

(1) **Input data**: Source code from the main branch of the tagged version.
(2) **Environment**: Set up Ubuntu and Python environment.
(3) **Dependencies**: Install using Poetry.
(4) **Testing**: Checks if release can continue.
(5) **Build**: Builds the package.
(6) **Publish**: Releases the package on PyPI.

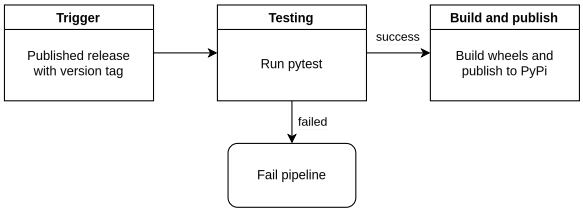

Figure 1: *lib-ml* Python Package Release Pipeline

1.4 Container Image: *model-service*

The source code for the mode-service container image is hosted on GitHub [2]. This repository includes all configurations needed for efficient deployment and management of the service, and handles interaction with the machine learning model through a REST API. The repository also manages the continuous integration and deployment (CI/CD) of the container image to GitHub Container Registry (ghcr), ensuring timely updates and releases.

1.5 Pipeline Steps: *model-service*

Implementation.

- **Triggered**: When a tag following the semantic versioning is created on main.
- **Environment**: Ubuntu-latest.

(1) **Checkout code**: Fetches the code from the repository with `actions/checkout@v4`
(2) **Login to ghcr**: Authenticates to allow pushing images using `docker/login-action@v3` with Personal Access Token (PAT).
(3) **Setup Docker Buildx**: Prepares the Docker build environment with `docker/setup-buildx-action@v3`.
(4) **Determine image tags**: Depending on the event type (release or non-release), sets appropriate Docker tags.
(5) **Build and push**: Builds the Docker image and pushes it to ghcr with `docker/build-push-action@v3`, using caching mechanisms to optimize build time and resource usage.

1.6 Artifacts and Data Flow: *model-service*

(1) **Source Code**: Pulled from the specific release tags.

[2] https://github.com/remla24-team8/model-service

(2) **Environment Setup**: Uses Ubuntu as the base operating environment along with Docker.
(3) **Container Image**: Built and tagged according to the git commit SHA or release tag.
(4) **Deployment**: Images are pushed to the GitHub Container Registry and can be used by orchestration tools like Kubernetes or Docker Compose.

2 DEPLOYMENT DOCUMENTATION

Our deployment runs on three virtual machines, set up using Vagrant with VirtualBox VMs. We have a single server node and two agent nodes running k3s, a lightweight Kubernetes distribution. The fact that each of these nodes are on different IP addresses is abstracted away by Kubernetes.

To allow external traffic to reach the cluster, we use Istio Gateways. For these to work, they require an Ingress Controller to have an external IP assigned by a load balancer. This controller is included in Istio. Since we are not on the cloud, but run locally (in a sort of "bare metal" environment with the three VMs), we use MetalLB for that. We configure it with a pool of IP addresses, of which it gives out the first to the Istio LoadBalancer service. This is the IP we can access our cluster at. Since we defined our range of addresses to be 10.10.10.0 to 10.10.10.99, it will always assign 10.10.10.0.

Since our setup is still quite simple, we have just a single gateway. That gateway is also associated with just a single VirtualService, which is used to set up all the routing rules for requests from all hosts. It is this VirtualService that determines the routing of our application.

A VirtualService consists of a number of HTTPRoutes, each with certain match rules. These all then forward to a specific service, optionally defined with a certain subset. DestinationRules then ensure a subset corresponds to a specific label, so that they reach the right pod. The rough flow of a request through the cluster can be seen in figure 2. Furthermore, the logical structure of our application can be seen in figure 3.

It is important to note that our frontend and backend are fully decoupled and that our frontend is deployed as a static HTML application on a simple fileserver. This means that any request made must flow from the browser directly to the backend. This requires also exposing our backend and ensuring that the version between the frontend and the backend match. Furthermore, the frontend HTML has script tags that load in specific JavaScript and CSS files. These must also match the same version. As it is not possible to add any headers to script tag file requests, we must ensure that the information about which version we request is stored in the path. So the HTML script tag requests /frontend/<version>/index.js. We then have a rule defined in the VirtualService that redirects this to the <version> fileserver pod. The fileserver also uses an environment variable that takes in the URL to the backend, so that different deployments can decide this and it is not baked into the build.

The actual request for the index.html of the frontend is what decides which version is requested. Using random weights, it either sends it to the destination with the "canary" version or the mainline version. Since it includes headers that say which version is requested, the fileserver uses these headers to set another variable in the HTML that the JavaScript can then also include when it sends requests to the backend. The VirtualService then contains further rules to ensure that they are routed to the correct backend based on these headers (by sending them to a specific subset, which corresponds to a label based on a DestinationRule).

The backend then communicates directly with the model service, of which we have just one.

The backend is also what handles the monitoring, handled by Prometheus, which collects specific metrics. Grafana then aggregates this information and presents it in a dashboard. Grafana and Prometheus are available simply at /grafana and /prometheus, respectively. In a real setup, they would be protected and at a different domain, so that host matching could be applied, making the routing much easier as now complicated configuration had to be applied to change the base URL's of both these services.

The data flow of a user loading the frontend and then running a single predict, which we just explained, is visualized for a specific version in figure 4.

3 EXTENSION PROPOSAL

In the current deployment of our project there are two main pain points: the quality of the model and no shutdown procedure. As the performance of the model was not the focus of this project we will keep that for what it is. More importantly, having no robust shutdown procedure can cause residual artifacts to interfere with future deployments.

Currently there is no clear way to delete all components of the deployment upon shutdown. This can lead to residual configurations, data, or services persisting in the environment, which can cause conflicts or errors in subsequent deployments.

To counteract this problem a robust cleanup process is proposed as an extension to the current deployment. This extension would include implementing automated checks, forced deletion options and post-deletion verification for all parts of the project.

This includes a graceful shutdown of the Grafana environment, the vagrant Virtual Machines (VMs), the Kubernetes pods and clusters. The Grafana environment including the dashboard and alert rules must be deleted using the Grafana API [1]. The stability of the Kubernetes pods can be improved by implementing a graceful shutdown as posed by Polencic [3]. Where utilizing preStop hooks and adjusting the grace period of the termination ensures that the connections are properly closed. Subsequently, these procedures facilitate the smooth decommissioning of the entire Kubernetes cluster, maintaining data integrity and operational continuity. The vagrant virtual machines can be then be destroyed leaving a clean state behind [4].

To evaluate whether the extension proposal effectively resolves the issue of incomplete deletions different test cases must be implemented. Simulating various shutdown scenarios to ensure the new shutdown procedure works as expected. Subsequent to this, automated scripts could be employed to verify that no remnants remain active or consume resources unexpectedly.

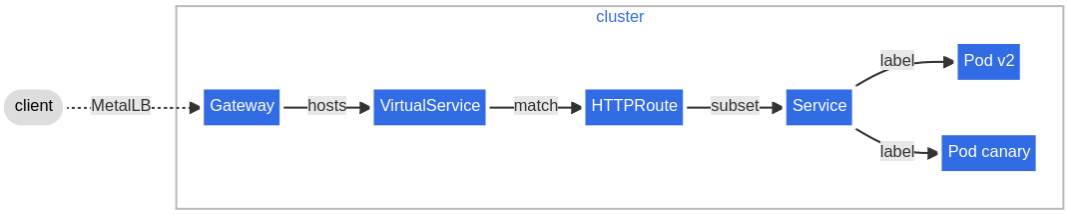

Figure 2: Routing of a single request. "hosts" indicates that the hosts determine the matching with a VirtualService, while certain matching rules "match" to a HTTPRoute, which then sends the request to a specific destination (with optionally a subset) defined, namely a service. This service matches specific deployments based on their labels, which allows certain requests to go to the v2 mainline Pod, while others go to the canary Pod.

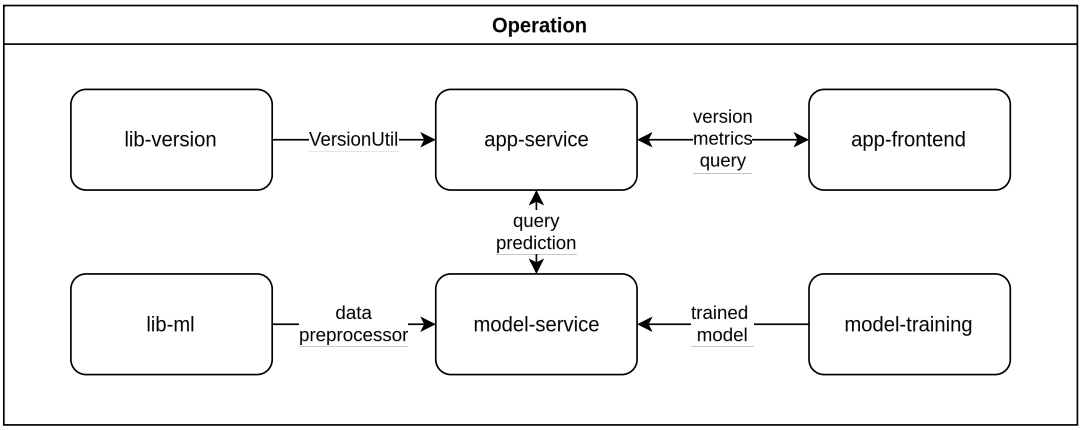

Figure 3: Logical application structure. The logical relationships between all components of our application are shown. "model-service", "app-service" and "app-frontend" are actual applications that run in the clusters, the rest are dependencies or provide some input.

4 ADDITIONAL USE CASE

For the additonal use case, we chose a global rate limiting setup. This uses the low-level EnvoyFilter combined with a Redis key-value database to ensure fast operation. It acts globally and is configured for exposition purposes to trigger after 10 requests in 1 minute, it then returns 429 Too Many Requests to ensure the service does not get overloaded.

This was relatively simple to set up with Istio, showcasing its extensibility.

5 EXPERIMENTAL SETUP

As part of the Istio deployment an experiment has been set up to showcase the usefulness of Prometheus and Grafana to visualize metrics. The experiment consists of adding functionality to the frontend. Where in the experimental version the query can consist of multiple urls separated by a comma in contrast with only one for the base design. There is also a change in the description and background colour to showcase when you are in the base version or the experimental version. The hypothesis is that when you are able to query the model with multiple links at one time, the amount of requests should go down while the time to query the model should increase. To be able to verify this hypothesis a number of metrics are being gathered and showcased in a Grafana dashboard.

The metrics used for this experiment include a counter for the total number of queries and a histogram for the prediction latency. We are particularly interested in the time it takes to return a prediction; therefore, we will focus on the highest latency quantiles. These metrics are integrated into a Grafana dashboard to showcase the number of requests over time, the distribution of prediction latencies, specifically, the 95th and 99th quantiles of the latency.

The users have a 70 percent chance of landing on the base version and a 30 percent chance of landing on the experimental version. This seperation must be taken into account when analysing the metrics. After being in deployment for a while we can create a distribution of the amount of queries per deployment and analyse the differences. The hypothesis would be accepted if the amount of queries is significantly less for the experimental case.

We can simulate the deployment by taking a list of urls testing them seven times in the base case for every three in the experimental case. In the base case the urls are tested individually while in the

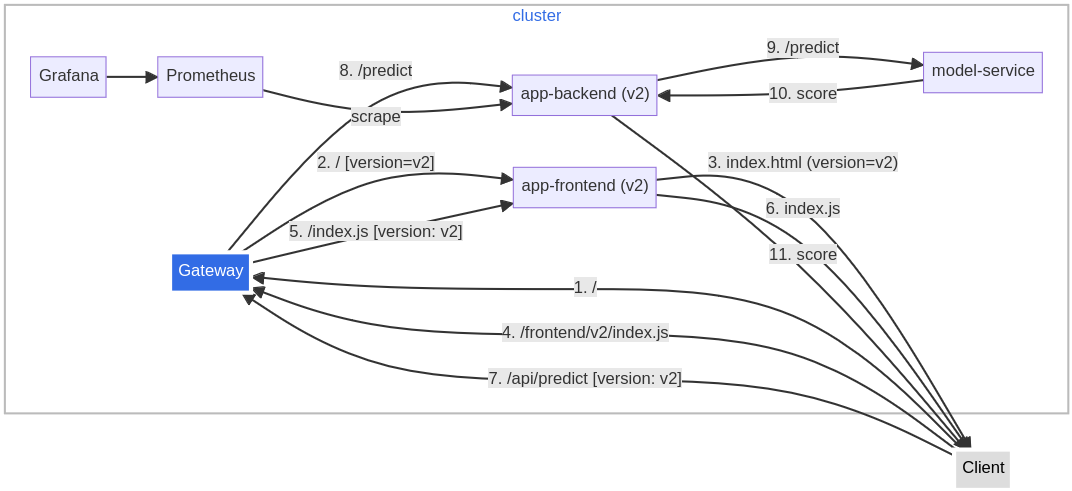

Figure 4: Data flow of a full page load of the frontend and a predict request, showing all components and how requests flow between them. "[version=v2]" indicates that headers corresponding to the v2 subset have been added to the request, while "index.html (version=v2)" means it is the HTML file with the version information baked into it, allowing it to later call the correct backend. Note that responses to the gateway would also go through a specific boundary, but are shown to route directly to the client for simplification. The connections from Grafana and Prometheus to the client and Gateway have also been omitted for simplicity. The matching, the service and other mechanism that does the actual routing has been simplified to be handled by the Gateway only. The request and response for the CSS files has also been omitted for simplicity, and the exact header names and request paths can differ slightly.

experiment the urls are passed on as a group separated by commas. This leads to the graphs of the metrics as shown in Figure 5.

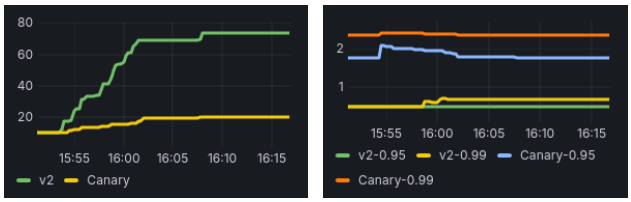

Figure 5: Amount of Prediction Requests (left) and Prediction Latency Quantiles 0.95-0.99 (right)

Before starting the simulation both versions had 10 individual queries made. At the end the base (v2) had 73 queries for 19 of the experimental (canary) version this is by design. The more interesting part is the difference in latency between the two versions, where the experimental version has a 95 and 99 quantile latency of around 2 seconds compared to around 0.5 seconds for the base version.

6 ML PIPELINE

The Machine Learning (ML) Pipeline is a crucial component of our project, encompassing the entire process from data acquisition to model deployment. This section outlines the key steps and design choices that form the backbone of our ML workflow, ensuring efficiency, reproducibility, and maintainability of our model development process.

6.1 Pipeline steps

The model training pipeline steps can be seen in Figure 6.

- **get_data.py** downloads the data CSV files from Google Drive and splits them into x, y and train, test, val sets. These sets are saved as lists in a joblib format, similar to pickled files, and passed on to the next stage.
- **process_data.py** then uses `lib-ml` and its tokenizer to tokenize and encode the data. The `char_index` is then saved to replicate the tokenizer when using the model. The final output of this process thus includes the data ready for training and the saved tokenizer. These artifacts are also uploaded to the DVC repository, also saved in Google Drive.
- **train.py** starts the model training. It first loads the data and fits the model. It also generates a `metrics.json` for use in DVC. This only contains loss and accuracy though. After the model file has been saved, a Google OAuth2 process pops up, allowing the trainer to authorize the program to upload the model and its corresponding tokenizer to its public spot. This allows it to be used by the web app.
- **test_mutamorphic.py** is then triggered to immediately test the new model. It uses the model saved in Google Drive which makes it able to be ran in GitHub Action, making it part of the CI/CD testing pipeline. It can also trigger `gen_mutamorphic_data.py`, which generates a new testing dataset for mutamorphic tests using the data described in subsubsection 7.1.2.

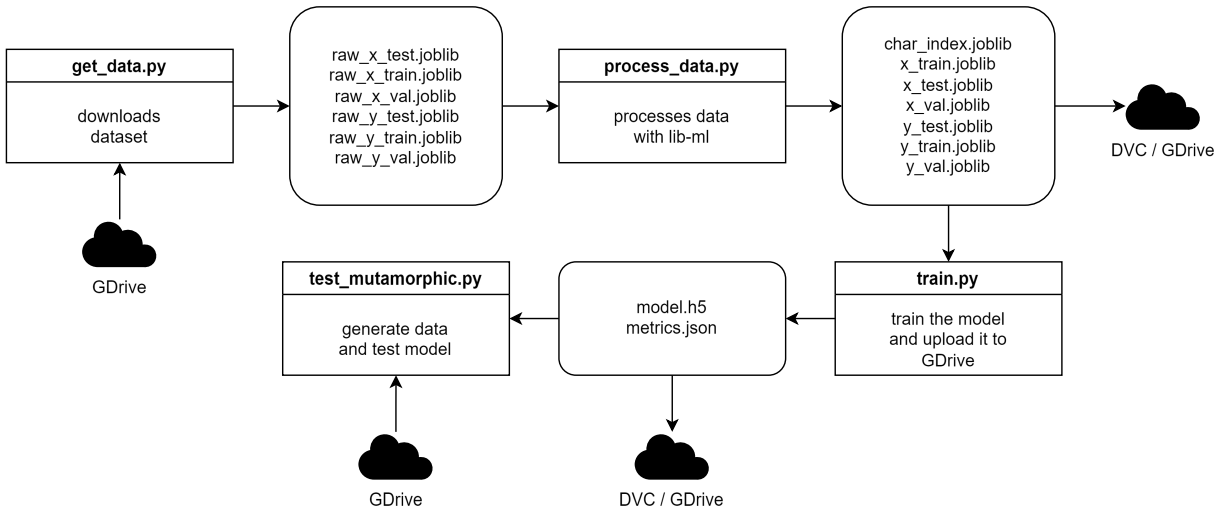

Figure 6: Flowchart of the data processing, model training, and testing pipeline, showing the interaction between various Python scripts and storage systems (DVC and Google Drive).

6.2 Design choices

In developing our ML pipeline, we made several strategic design decisions to optimize our workflow, enhance collaboration, and ensure the robustness of our model development process. The following subsections detail the key technologies and approaches we've adopted, explaining their benefits and implementation in our project.

6.2.1 Dependancy management. For every repository that is not packaged and build in a pipeline, we use UV package manager. This is a drop-in replacement of pip, but written in Rust and thus a lot faster. Usage is also very simple, add your requirements in `pyproject.toml` and UV will automatically generate a new `requirements.txt` with all its co-dependencies.

6.2.2 DVC. was used for pipeline and artifact management and continuous experimentation. A DVC remote in Google Drive was created to house all artifacts not suitable to be uploaded to GitHub. Using `dvc repro` would run the whole pipeline if all artifacts are missing and only run updated parts on an already run installation.

We also experimented in storing the model and tokenizer file in DVC for public access, but it unfortunately proved too difficult to get authentication working with DVC and Google Drive (via a service account) in different environments such as GitHub Actions and every local user. Using this method meant we needed to distribute an authentication key to every user and pipeline, without wanting to publish this key to our GitHub. DVC also stored its files garbled in MD5 hashes in Google Drive, making a direct download also impossible.

6.2.3 Google Drive. We however found another fitting solution to the publication of our model and tokenizer files. Normally Google Drive requires authentication when uploading and downloading files, but we found a scraper package, gdown, which did not require authentication for downloading files from Google Drive. This simplifies our process by a lot and also makes us able to run model testing in GitHub Actions without any secrets. Now we only have to find a solution for uploading our model and tokenizer.

We decided to use an OAuth2 flow for uploading the model and tokenizer files. This means every user that wants to train a model should first get authorization from us to the public files and second, be added as test user to our Google Cloud project, which will not be necessary once our app has been published. This also allows us to work without having to distribute private keys, only adding a public OAuth2 key in our repository.

6.2.4 Linters. For great code quality we decided to use the following linters in all of our repositories that contained Python code. All linters generate a report and are saved as artifacts in the GitHub Action.

- **pylint** is our first linter. Any run under a 10 will be failed.
- **black formatter** then tests if the code is formatted nicely.
- **bandit** will then check for any security flaws in our code.
- **ruff** also checks formatting and PEP-8 rules, but often detects other smells than pylint and is a lot faster.

7 ML TESTING DESIGN

Most of the testing is done in the model-training repository. This is because the model is the main testing subject and thus needs testing every time the model receives an update, which happens in the model-training branch. Some tests are currently not passing as we had corruption issues with the trained model, which required us to train a new model. We decided not to spend more time on training advanced new models, as that was not the focus of this course, thus the current model was very quick to train.

7.1 Current Machine Learning tests

To ensure the robustness and reliability of our phishing URL detection model, we have implemented a comprehensive testing strategy. This strategy consists of two main components: regular tests and mutamorphic tests. These tests are designed to evaluate the

model's performance across various scenarios and to identify potential weaknesses or inconsistencies in its classifications.

Our testing methodology comprises:

(1) **Regular Tests**: These tests evaluate the model's basic functionality, including its ability to correctly classify obvious good and bad URLs, handle different input types, and process international URLs.
(2) **Mutamorphic Tests**: These advanced tests assess the model's consistency and resilience when faced with subtle modifications to input URLs.

The following subsections provide detailed explanations of each testing component, including specific test cases, their implementations, and the rationale behind them.

7.1.1 Regular tests. The regular tests test obvious good and bad URLs and tests the output type. An overview can be seen in Table 1 and further details are described in the code comments. The international URLs test is currently failing, as the model supplied by our instructors scores the international URL lower in phishing then its legitimate counterpart. This is due to the training data not including these types of URLs.

As we would not like the test to fail the whole pytest pipeline, a method was devised to soft-fail. Unfortunately GitHub is one of the few Git platforms that does not support soft failing. This is why first the test is evaluated and if the test will fail, it will use the skip functionality of pytest to skip over this test. If the test will succeed, the test is performed. The PyTest result of this modification can be seen below, where the 's' stands for the skipped test.

```
tests/test_model.py ..s.                    [ 44%]
tests/test_mutamorphic.py .....             [100%]
```

Regular tests	Description
Predict types	This test enforces the model to return an numpy array for every query it gets, regardless if the input is a list or a string.
Correct URLs	This test checks if legitimate URLs return low scores
Incorrect URLs	This test checks if illegitimate URLs return high scores
International URLs	This test checks if URLs that use international characters made to look like English characters get a high score.

Table 1: Different regular PyTests in model-training

7.1.2 Mutamorphic testing. Five tests have been devised for mutamorphic testing of the machine learning model. As can be seen below, the different mutamorphic tests are described and if the change should be reflected in the score or that the score should remain the same. Two examples are also given of input data and how the mutamorphic test will change this URL. The data was generated from a URL dataset by Kaitholikkal et al.[2].

MR1: URL Length Invariance. Append a benign path that doesn't change the nature of the URL. Expectation: The classification should remain the same.

- **Original Input**: `http://example.com/login`
- **Output**: `http://example.com/login?extra=params`

MR2: HTTPS vs. HTTP. Change HTTP to HTTPS. Expectation: The classification should favour the HTTPS URL.

- **Original Input**: `http://example.com/login`
- **Output**: `https://example.com/login`

MR3: Subdomain Addition. Add a benign subdomain. Expectation: The classification should remain the same.

- **Original Input**: `http://example.com/login`
- **Output**: `http://safe.example.com/login`

MR4: Parameter Shuffling. Shuffle the parameters. Expectation: The classification should remain the same.

- **Original Input**: `http://example.com/login?user=test&session=123`
- **Output**: `http://example.com/login?session=123&user=test`

MR5: Case Variation in Path. Change the case of the path. Expectation: The classification should remain the same.

- **Original Input**: `http://example.com/login`
- **Output**: `http://example.com/LOGIN`

7.2 Non-Functional Requirement Testing

In addition to our functional tests, we implemented non-functional requirement testing to ensure our model meets performance and efficiency standards. This testing focuses on aspects such as resource utilization and response time, which are critical for the model's practical deployment and scalability.

7.2.1 Workflow Telemetry in GitHub Actions. To facilitate this testing, we leveraged GitHub Actions to set up a workflow analyzer. This tool allows us to monitor and analyze key performance metrics during the model's execution:

- **Memory Usage**: We track the model's memory consumption to ensure it remains within acceptable limits, preventing potential out-of-memory errors in production environments.
- **CPU Utilization**: By measuring CPU usage, we can assess the computational efficiency of our model and identify any processing bottlenecks.
- **Network Usage**: We monitor network usage to understand any potential impacts on system resources during data fetching or model updates.

The workflow analyzer runs automatically with each push to our repository, providing consistent performance data across different versions of the model. This approach allows us to:

(1) Establish performance baselines
(2) Detect performance regressions early in the development process
(3) Optimize resource utilization based on concrete metrics
(4) Ensure the model meets non-functional requirements before deployment

7.3 Limitations and Future Work

The current method of testing still has some limitations which could be improved. It was decided to not test for non-determinism robustness. This test would comprise of retraining the model with different random seeds, which would mean a lot of time and computational power would be 'wasted'. This could however prove to be an interesting test to be performed in the future, but before that it might be a good idea to retrain the whole model solidly instead of the sub optimal replacement after our model corruption.

The limited internationalized URL phishing detection is also one to be addressed. This is by far the hardest phishing method for humans to detect and thus would have a much bigger impact on phishing safety if the model can detect this, as the current model runs on features humans can also detect easily.

The soft-fail workaround method for the international URL test, while practical, masks a fundamental issue with the model's performance. This should be addressed at the root of the issue as mentioned before, but this workaround offers a temporary fix to not hinder development.

These tests are the ones we could come up with and implement in a relative short period. This means that there could be a lot more useful tests out there.

7.3.1 Future work.

- **Expand training data:** Include a more diverse set of URLs, especially international ones, to improve the model's performance on non-English URLs.
- **Enhance mutamorphic testing:** Develop additional mutamorphic tests that cover a wider range of URL manipulations and phishing techniques.
- **Adversarial testing:** Implement adversarial testing to identify potential vulnerabilities in the model's decision-making process.
- **Cross-platform testing:** Extend testing to various platforms and browsers to ensure consistent performance across different environments.
- **Time-based testing:** Implement tests that evaluate the model's performance over time, considering the evolving nature of phishing techniques.
- **Performance optimization:** Use the insights from the non-functional requirement testing to optimize the model's resource usage and response time.
- **Automated model retraining:** Implement a system for automated model retraining based on new data and test results to continually improve performance.

8 CONCLUSION

This paper has detailed the development and deployment of a phishing URL detection system, showcasing modern DevOps practices and machine learning methodologies. The project demonstrates effective use of CI/CD pipelines, containerization, and Kubernetes orchestration. The machine learning pipeline incorporates robust data management and model training practices, while the testing strategy ensures model reliability through regular and mutamorphic tests.

Although successful in its current form, there's room for improvement, particularly in handling international URLs and expanding the testing suite. This project serves as a valuable example of integrating machine learning into a modern, containerized application environment, highlighting the importance of comprehensive testing, efficient deployment, and continuous monitoring in ML-powered applications.

REFERENCES

[1] Grafana. 2024. HTTP API reference. https://grafana.com/docs/grafana/latest/developers/http_api/
[2] JISHNU K S KAITHOLIKKAL. 2024. Phishing URL dataset. https://doi.org/10.17632/VFSZBJ9B36.1
[3] Daniele Polencic. 2024. Graceful shutdown in Kubernetes. https://learnk8s.io/graceful-shutdown
[4] Vagrant. 2024. Destroy Command: vagrant destroy [name|id]. https://developer.hashicorp.com/vagrant/docs/cli/destroy