REMLA Project Report – Group 15

Marianna Louizidou, Mitali Patil, Shantnav Agarwal

1 INTRODUCTION

In this report, we explain the design and engineering of the URL phishing detection project.

2 RELEASE PIPELINE

In this section, we present a detailed overview of the release workflow for both the software packages (the ‘lib-ml‘ and ‘lib-version‘ Python libraries) and the container images for the Flask app and model service.

2.1 Releasing a Software Package

The ‘lib-ml‘ and ‘lib-version‘ repositories contain the code for two distinct Python packages. The ‘lib-ml‘ package is designed for preprocessing both training data and incoming queries for the REST-API call, while the ‘lib-version‘ package implements a version-aware library capable of returning the library version. Both packages are uploaded to the PyPI package index via a GitHub Actions workflow, triggered when a new tag starting with "v" is pushed, followed by the (semantic) version of the package to be published. The steps in the "Publish package to PyPI" workflow are as follows:

- **Checkout Code** : This step uses the ‘actions/checkout@v2‘ to checkout the latest version of the repository.
- **Set Up Python** : This step uses ‘actions/setup-python@v2‘ to ensure python version 3.10 is set up in the working environment.
- **Extract Version from Tag** : Extracts the version number from the tag name and sets it as an output variable VERSION. This is used for versioning the package.
- **Update version.py** : This step updates the version.py file with the extracted version number from the pervious step if the current reference is a tag. This file contains the version of the package. The cat command outputs the contents of ‘version.py‘ to the log for verification.
- **Install Poetry** : Since we used Poetry[2] for our dependency and package management across the project, we installed poetry for the required dependencies.
- **Update pyproject.toml Version** : Updates the version in .pyproject.toml using the extracted version number to be used by Poetry to install the dependencies in the .toml file.
- **Lint with Flake8** : Runs flake8 to perform linting on the code in the ‘lib_version_URLPhishing‘ directory. Linting helps us identify and fix potential issues and code style violations if the pipeline breaks.
- **Commit and pull latest changes** : The updated versions are committed and the latest changes are pulled. This step is crucial to make sure the build uses the most recent updates.
- **Build and Publish to PyPI** : This step builds the package using poetry build, and publishes the built package to PyPI using the provided credentials (stored as secrets in the repository).
- **Install and generate documentation**: Pdoc is installed for generating documentation for this published package. It is also published on GitHub pages for viewing online.

ADD PYPI PUBLISHED PACKAGES

2.2 Releasing Container Images

The app repository contains a Flask app which is the front-end service that exposes our phishing URL model to users. The ‘app‘ relies on ‘model-service‘ to preprocess and predict the queries using REST API call and ‘lib-version‘ to make the version visible on the front-end. We release two container images for the app and the model-service respectively. For the model prediction, we do not include the model in the model-service’s image , instead, it is fetched from the model-training repository into the docker container. Using a GitHub Actions workflow, the container images for these repositories are built and uploaded to the GitHub Container Registry, allowing us to easily start instances of the app and model-service using Docker or Kubernetes. Just like the workflows for the packages, the workflow is triggered when a new tag starting with v is pushed, which should be followed by the (semantic) version of the package that is to be published. The different steps to Build and Publish Docker Image workflow are as follows:

- **Checkout Code** : This step uses the ‘actions/checkout@v2‘ to checkout the latest version of the repository.
- **Parse Version Info from Tag**: Extracts version information from the tag (e.g., v2.3.5), splitting it into major, minor, and patch components, and saves them as environment variables.
- **Login to GitHub Registry**: We use the GitHub token to log into the Container Registry(ghcr.io). This authentication allows us to push images to the container.
- **Build and Push Docker Image** : This step builds the Docker image and tags it with the semantic git tag and latest which is then pushed to the container registry.

The workflow also generates and deploys documentation to GitHub Pages. The respective images are available in the GitHub container registry of the Organisation under the packages section. ADD IMAGE REGISTRY LINKS

ADD IMAGES FOR THE PIPELINE

3 PROVISIONING AND OPERATING A WEB APP

3.1 Provisioning

A vagrant file is used to provision the virtual machines. Two types of machines have been defined - controller, worker#. The worker nodes are provisioned within a loop; therefore the number of worker nodes deployed can be indicated in the vagrant file. Two ansible playbooks are written, one for each type of machine. On the worker machines, K3S and Docker are installed. On the controller machine, helm charts and the required repos are also installed in addition to K3S and Docker.

All machines are setup as part of a K3S cluster. The controller machine acts as the K3S server while the worker machines register as K3S agent nodes. The pods for Prometheus, Grafana and the URL App are deployed as helm charts in the controller machine/ node. The cluster automatically deploys these pods to any of the nodes.

3.2 Kubernetes

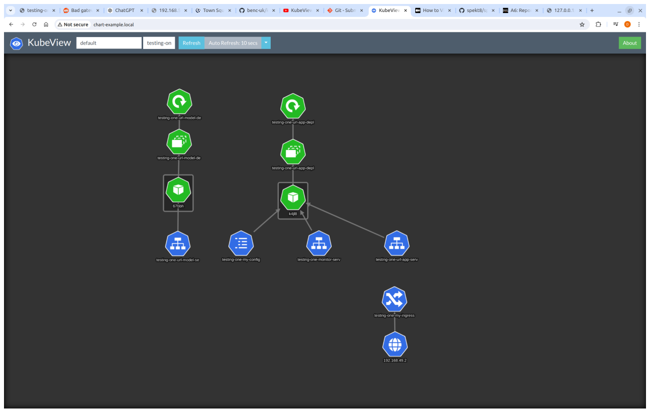


Figure 1: Nodes deployed for the URL-App. Shown here as a helm chart deployment with name *testing-one.* Visualised using kubeview

The URL App consists of two main deployments: the URL-app -> this serves the app front-end; and the URL-model -> this serves the ML model in the back end. An ingress *my-ingress* receives the request from the *load-balancer* and forwards it to the service *URL-app-serv*. This service forwards the request to the deployment. The *URL-app-depl* knows the address of the *URL-model-serv* from the *my-config* configuration. The query with the URL to be tested is forwarded to this address for inference from the ML model. A service monitor is defined with the same release label as the name of the prometheus-kube chart installation. This helps the URL app to be discovered by prometheus.

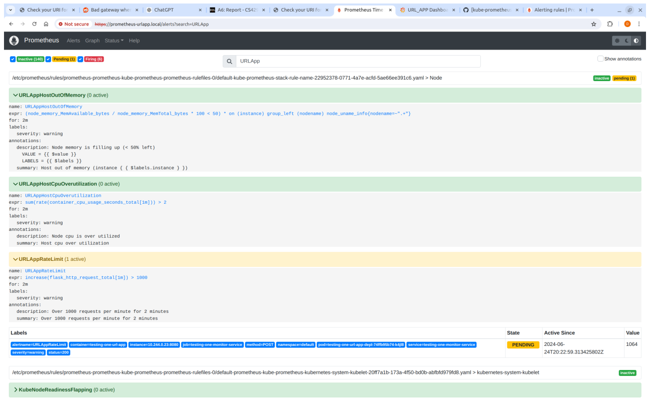
A helm chart of this deployment has been created. Using a value config, container image path, number of replica sets and resource limits (cpu, memory) can be defined. Multiple installations of this helm chart can be run at the same time.

3.3 Monitoring

The Prometheus & Grafana are launched using the prometheus-kube-stack helm chart. The values file has been modified to configure the ingress for Prometheus and Grafana. This value file also configures the alerts for Prometheus. The Grafana dashboard is defined as a *ConfigMap* and applied using kubectl.

The URL app publishes the metrics at the endpoint */metrics* about various things such as request duration, count and exceptions. These metrics are reported separately for each endpoint. This implementation has been done using the prometheus$_{f}$*lask* python library.

3.3.1 Prometheus. The prometheus service scraps the metrics endpoint of the URL app and collects information about the number of requests to each endpoint, time taken to complete these requests etc. Based on information from the kubernetes cluster and the URL App, 3 alerts have been configured.



/etc/prometheus/rules/prometheus-prometheus-kube-prometheus-prometheus-rulefiles-0/default-kube-prometheus-stack-rule-name-22952378-0771-4a7e-acfd-5ae66ee391c6.yaml > Node

URLAppHostOutOfMemory (0 active)

```
name: URLAppHostOutOfMemory
expr: (node_memory_MemAvailable_bytes / node_memory_MemTotal_bytes * 100 < 50) * on (instance) group_left (nodename) node_uname_info{nodename=~".+"}
for: 2m
labels:
  severity: warning
annotations:
  description: Node memory is filling up (< 50% left)
    VALUE = {{ $value }}
    LABELS = {{ $labels }}
  summary: Host out of memory (instance { { $labels.instance } })
```

URLAppHostCpuOverutilization (0 active)

```
name: URLAppHostCpuOverutilization
expr: sum(rate(container_cpu_usage_seconds_total[1m])) > 2
for: 2m
labels:
  severity: warning
annotations:
  description: Node cpu is over utilized
  summary: Host cpu over utilization
```

URLAppRateLimit (1 active)

```
name: URLAppRateLimit
expr: increase(flask_http_request_total[1m]) > 1000
for: 2m
labels:
  severity: warning
annotations:
  description: Over 1000 requests per minute for 2 minutes
  summary: Over 1000 requests per minute for 2 minutes
```

Labels	State	Active Since	Value
alertname=URLAppRateLimit container=testing-one-url-app instance=10.244.0.23:8080 job=testing-one-monitor-service method=POST namespace=default pod=testing-one-url-app-depl-74ffb95b74-k4jl8 service=testing-one-monitor-service severity=warning status=200	PENDING	2024-06-24T20:22:59.313425802Z	1064

/etc/prometheus/rules/prometheus-prometheus-kube-prometheus-prometheus-rulefiles-0/default-prometheus-kube-prometheus-kubernetes-system-kubelet-20ff7a1b-173a-4f50-bd0b-abfbfd979fd8.yaml > kubernetes-system-kubelet inactive

KubeNodeReadinessFlapping (0 active)

Figure 2: Alerts show up based on CPU usage, memory consumption and request rate (triggered).

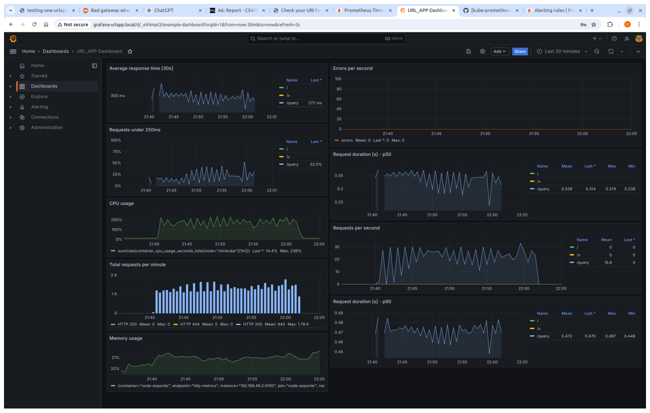

Average response time [30s]
Errors per second
Requests under 250ms
Request duration [s] - p50
CPU usage
Requests per second
Total requests per minute
Request duration [s] - p90
Memory usage

Figure 3: Visualisations shown using the grafana dashboard. This dashboard show the high rate of requests which triggered the alert in Figure 2.

3.3.2 Grafana. Grafana is used to continuously monitor the services using multiple visualisations. Where required, metrics such as rate aggregations, averages are also shown. Please refer Figure 3.

4 ML PIPELINE

ML tasks involve various stages like data gathering, data preparation, training, and so on. We automate this workflow to be able to manage the code, reproduce, deploy at scale, maintain versions, and follow the best practices through pipeline orchestration. Each stage has its own code to process the data, and running the entire pipeline after every change is inefficient and resource-intensive. Hence, we used Data Version Control (DVC) [1] to configure an ML pipeline for this project.

Our DVC pipeline has 6 stages each automating specific tasks of the workflow:

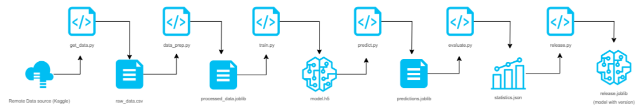


Figure 4: ML Config Pipeline.

- **Get Data** : This stage involves fetching and storing the raw text data from a remote repository. In our case we are fetching the train, test and evaluation text data from Kaggle [3].
- **Prepare Data** : This stage pre-processes the raw data. This involves cleaning the data, tokenizing, and padding to generate training, testing, and validation datasets, along with a character index file used for feature encoding. These are stored as .joblib files.
- **Train** : In this stage, the machine learning model is trained using the processed training data. The training script utilizes the specified parameters (like loss function, optimizer, batch size, epochs) from the params.yaml file to configure the training process. The resulting trained model is saved as an .h5 file.
- **Predict** : This stage uses the trained model to make predictions on the test dataset. The predictions are then saved as a .joblib file for further analysis.
- **Evaluate** : The evaluation stage assesses the performance of the model using the predictions made in the previous stage. Metrics such as accuracy, precision, recall, and F1-score are calculated and saved as a JSON file.
- **Release** : The release stage involves packaging the final model and any other necessary artifacts for deployment. This includes the trained model, character index, and the model version which helps in versioning the model releases, ensuring that the deployment process can track and manage different versions of the model.

DVC makes collaboration for ML tasks better and allows to work with remote files which is a significant advantage over utilities like Makefile. Moreover, it makes reproducibility, tracking, and saving of changes easier to handle allowing for better comparisons among various versioned experiments. A close alternative to DVC was considered , the Git LFS but it has drawbacks as to how much data can be uploaded per user per day. Considering our project data size and experiments, it was ruled out for data versioning.

We also used Poetry [2] to be able to manage our dependencies. Poetry has a significant advantage over using a plain 'requirements.txt' file to install packages. We do not have to maintain the dependency file every time we add a new package, unlike requirement.txt. Through poetry and a conda environment with the selected 3.10 Python version, we ensure that the dependencies are consistent across various systems and the builds are repeatable.

To ensure and maintain high quality code throughout our project, we have intergrated the use of two different linters, pylint and flake8 into our CI/CD pipeline. Pylint provides a detailed analysis of Python source code, checking for errors, enforcing a coding standard, and looking for code smells. The configurations have been customized to fit our project needs, an example of this is the addition of the dslinter plugin. On the other hand, flake8 focuses more on PEP 8 compliance and common programming errors. It complements Pylint by providing an additional layer of checks.

The code quality checks are implemented in our tag creation workflow. This ensures that if the code does not pass the linter checks, the workflow will break and no new tag will be created. Consequently, the release process is not triggered and no model is released unless the code quality meets our standards.

Automating the workflow reduced manual intervention and potential errors. Once the pipeline was set up, the build failures and linter issues could be easily handled to ensure code quality. Tags provide a clear reference to specific versions of the working code ensuring that experiments and results can be precisely reproduced. This practice enhanced the organization of our project by simplifying debugging and aiding the progress and performance of different iterations of changes every time.

5 ML TESTING

Our testing framework incorporates a variety of tests to ensure comprehensive coverage of the model's functionality and performance.

Data Slices Test: The purpose of the data slices test is to evaluate the performance of the model on a representative subset of the data. This test was chosen to ensure that the model maintains consistent performance and generalizes well across different inputs that reflect the entire dataset. By creating a representative slice and comparing its performance to the entire dataset, we can confirm that the model does not favor specific segments and performs reliably on all data. We generate a random sample that preserves the original distribution of features and labels, and test whether the model's accuracy on this representative slice closely matches its accuracy on the full test set. This approach helps identify potential discrepancies and ensures that the model's performance is robust and unbiased.

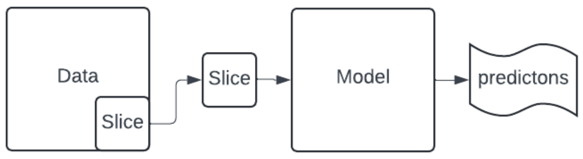


Figure 5: Data slices test

Features test: The feature test is designed to verify the integrity and correctness of the input features. This test was chosen to ensure that the data preprocessing steps were implemented correctly and that no sensitive information was included in the dataset. By confirming the correct tokenization and encoding of features, we prevent data leakage or incorrect feature extraction that could affect model performance. We test the tokenizer and encoder to ensure that they work as expected, and validate that sensitive patterns such as passwords or session tokens are appropriately masked.

Non-determinism robustness test: To ensure that model training is stable and produces consistent results, the non-determinism robustness test involves training the model multiple times with the same data and hyperparameters, and then comparing the weights and performance metrics. This test was chosen to identify any sources of randomness that could affect the reproducibility of the model, thereby ensuring consistent and reliable model training results. We trained our model multiple times with the same dataset and hyperparameters, and verified that the resulting models had nearly identical weights and performance metrics.

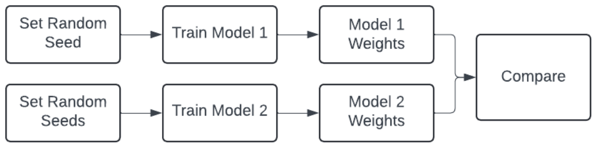


Figure 6: Non-determinism robustness test

Metamorphic Testing with Automatic Inconsistency Repair: Metamorphic testing detects inconsistencies in model predictions when inputs are slightly changed. This test was chosen to increase the robustness of the model by ensuring that it responds predictably to similar inputs. By generating mutated inputs and testing whether the model's predictions remain consistent with expectations, we can identify and repair any inconsistencies in the model's behavior. Specifically, we created new test inputs by replacing words in URLs with synonyms and observed whether the model's predictions for these altered inputs were consistent with the original predictions. When inconsistencies were detected, the system attempted to automatically repair the inconsistencies by adjusting the model's decision boundaries.

Cost of Features Test: The cost of features test evaluates the impact of different features on the model's performance. This test was chosen to determine which features are most important to the model, which helps with feature selection and dimensionality reduction. By focusing on the most important features, we can improve the efficiency of the model and potentially reduce the computational overhead. We conducted experiments to assess how including or excluding certain features, such as certain keywords or URL patterns, affected the model's performance metrics, such as accuracy and F1 score.

To ensure the reliability and completeness of our ML model, we integrated test coverage reporting into our CI pipeline using the pytest-cov plugin. Test coverage measures the proportion of our code base that is executed during testing, providing insight into which parts of the code are being tested and which are not. This metric is critical because it helps identify untested areas that are more prone to bugs. The coverage report is generated and stored in the reports directory, ensuring that we can continuously monitor and improve our coverage.

Lessons Learned: The process of implementing and running comprehensive tests, especially those involving large datasets and complex models, is time-consuming. This has highlighted the need for efficient testing strategies and the importance of planning for adequate testing time in the project timeline.

REFERENCES

[1] https://dvc.org/doc [n.d.]. DVC.
[2] https://python-poetry.org/ [n.d.]. Poetry.
[3] https://www.kaggle.com/datasets/aravindhannamalai/dl-dataset [n.d.]. Kaggle$_{data}$.