REMLA Project Report – Group 02

Thijs Penning, Rodin Haker, Nada Mouman, Arjan Hasami

1 INTRODUCTION

In this document, we describe design choices for the phishing URL detection project.

2 RELEASE PIPELINE DOCUMENTATION

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

2.1 Releasing a Software Package

The `lib-ml` and `lib-version` repositories contain the code for two different Python packages. The `lib-ml` package is used for pre-processing both the training data and the incoming queries. The `lib-version` package implements a version-aware library, which can be asked for the version of the library. These packages are both uploaded to the PyPI package index using a GitHub Actions workflow. This workflow is executed 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 in the *Publish package to PyPI* workflow are as follows:

- **Checkout code**: Using the `actions/checkout@v3` action, we clone the latest version of the repository. This ensures the workflow will publish the most recent version of the code.
- **Set up Python**: Using the `actions/setup-python@v4` action, we setup our specified version of Python (3.12).
- **Install Poetry**: Across our project, we use Poetry as a dependency management tool. In this case, we will use it as a packaging tool. We use a shell command to install Poetry and add it to the system path.
- **Extract version from Git tag**: As mentioned previously, we wish for the version of the published package to be the same as the version specified in the Git tag that starts the workflow. In this step, we use a shell command to extract the version from the Git tag, and set is as an environment variable.
- **Set project version**: Using the environment variable created in the previous step, we use the command `poetry version` to set the version of the package.
- **Build and publish the package**: The final step builds the package using `poetry build`, and publishes it to PyPI with `poetry publish` using the API key stored in the repository's GitHub secrets.

The final packages are published on PyPI as `lib_ml_remla24_team02` and `lib_version_remla24_team02`.

2.2 Releasing a Container Image

The `app` repository contains a Django app which exposes our phishing URL model to users. It relies on the `model-service` to provide a FastAPI based REST API for model predictions. 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 executed 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. It should be noted that some of the versions for the app end with *-canary*. Versions of the app that should be released as a canary release should be tagged like this.

The different steps in the *Build and Publish Docker Image* workflow are as follows:

- **Checkout code**: Using the `actions/checkout@v3` action, we clone the latest version of the repository. This ensures the workflow will publish the most recent version of the code.
- **Registry login (ghcr.io)**: Using the GitHub token of the user that triggered the workflow, we log into the GitHub container registry. By piping the previously mentioned token to the `docker login` command, we authenticate the Docker CLI to allow pushing images to the registry.
- **Make envfile**: Using the `SpicyPizza/create-envfile@v2.0` action, we create an environment file (`.env`) with the necessary environment variables, which are retrieved from the organizations GitHub secrets.
- **Build and Push Docker Image**: Using the `docker/build-push-action@v2`, we build the Docker image and push it to the GitHub Container Registry. The `context` and `file` parameters specify the build context and Dockerfile location, respectively. The image is tagged with both the current Git tag (containing the semantic version) and `latest`.

As mentioned, the images are then available in the organizations GitHub Container Registry, found under the *packages* section on the GitHub organization home page.

3 ML PIPELINE

In the `model-training` repository, we have configured a machine learning training pipeline using Data Version Control (DVC) together with Amazon S3 buckets to manage data and model artifacts. This section gives an overview of the pipeline and discusses the setup and decisions made.

The DVC pipeline consists of several stages, each automating specific tasks essential for the machine learning workflow:

- **Get data**: This stage fetches the raw datasets from the S3 bucket and stores them in designated directories.
- **Preprocess data**: This stage pre-processes the raw data using the `lib-ml` package. It is dependent on the raw datasets, and generates pre-processed training, testing, and validation datasets, along with a character index file used for feature encoding. These are stored as `.joblib` files.

- **Define model**: This stage defines the architecture of our machine learning model. The resulting structure is saved as a `.joblib` file.
- **Train model**: This stage trains the saved model structure using the pre-processed training data. The validation data is used to tune and evaluate the model during training. The resulting model is saved as a `.joblib` file.
- **Evaluate model**: This stage evaluates the performance of the trained model using the pre-processed test dataset. Metrics and plots are generated to assess accuracy, precision, recall and other performance indicators.

Note: this section is not finished and will be expanded

4 DEPLOYMENT DOCUMENTATION

This chapter discusses the final deployment of our application, highlighting the structure, components, and data flow within the Kubernetes[1] cluster. The deployment uses Vagrant[7] for provisioning, Ansible[8] for configuration management, Istio[6] for traffic management, and a suite of observability tools including Kiali[9], Prometheus[3], Grafana[11], and the Kubernetes dashboard[2]. Figure 1 visually represents the deployment architecture, showing the connections between the various components and the data flow for incoming requests. Each section in this chapter describes the components, explaining their roles and how they contribute to the functionality and reliability of the system.

4.1 Vagrant

The deployment process starts with Vagrant. Vagrant is used to create three lightweight and reproducible environments (nodes) running Ubuntu. These nodes consist of a controller node and two worker nodes.

Controller Node. The controller node is the central point of control in the Kubernetes cluster. It maintains a record of all Kubernetes objects and runs the control plane components such as the scheduler and controller manager. It ensures the cluster remains in the desired state and makes decisions on the scheduling and scaling of the applications.

Worker Nodes. The two worker nodes are the nodes where the actual application workloads are run. They host the containers and provide the required computing, memory, and storage resources for the applications.

Provisioning. Vagrant simplifies provisioning these nodes by using a Vagrantfile that specifies the configuration for each node. The Vagrantfile ensures that each node uses the correct operating system, network configuration, and resources. Once the Vagrantfile is executed, Vagrant automatically creates and configures the virtual machines, and sets up the networking to ensure that the nodes can communicate.

4.2 Ansible

Ansible automates the setup and configuration of the environments, ensuring a consistent and efficient deployment process. Using Ansible playbooks, the installation, and initialization of MicroK8s (Kubernetes)[12] are automated on all nodes. This includes setting up the control plane on the controller node and securely joining the worker nodes to the cluster. Ansible also handles the deployment and configuration of the Istio service mesh, which includes setting up the Istio Ingress Gateway for managing incoming traffic and implementing canary deployments to safely test new features.

Additionally, Ansible deploys the core application and model services, ensuring that both stable and canary versions are correctly configured. It also automates the installation of the various observability tools.

4.3 Istio

Istio is used as a service mesh to manage the microservices within the Kubernetes cluster. The Istio Ingress Gateway is the entry point for all incoming traffic and routes the requests to the appropriate services. The application and model service have been set up to use a canary deployment strategy. In this setup, 10% of users are routed to the new version of the application and/or the model service. This allows for the safe testing of new features in production. Sticky sessions are maintained using cookies, ensuring the subsequent requests from a user are consistently routed to the same version of the services.

4.4 Application and Model Service

At the core of the deployment lies the application and model service. The application handles the main business logic and user interactions, while the model service provides a way of querying a machine-learning model to detect phishing URLs. Both services are configured to support rolling updates and canary deployments, allowing for seamless updates and testing of new features without disruption. The services communicate internally with the cluster, with Istio managing traffic routing, load balancing, and service discovery.

4.5 Monitoring and Observability Tools

To ensure the health and performance of the deployment, several observability tools have been integrated:

- **Kiali:** Provides a detailed visualisation of the service mesh. This helps in understanding the topology and health of the microservices. Figure 1 is a Kiali visualisation.
- **Prometheus:** Collects and stores metrics from the services. It monitors the performance and resource utilization of the deployment and triggers alerts based on predefined rules. For example, when more than 15 requests per minute are made over 2 minutes an alert is triggered.
- **Grafana:** Visualises the metrics collected by Prometheus and Istio.
- **Kubernetes Dashboard:** Offers a UI to manage and monitor the Kubernetes cluster.

5 ML TESTING DESIGN

In order to guarantee the robustness and reliability of a machine learning (ML) model, it is important to have a testing strategy. ML testing is assessing and validating the performance of ML models [16] to ensure various aspects, such as their accuracy, robustness, reliability and feature and data integrity. Therefore, we devised a

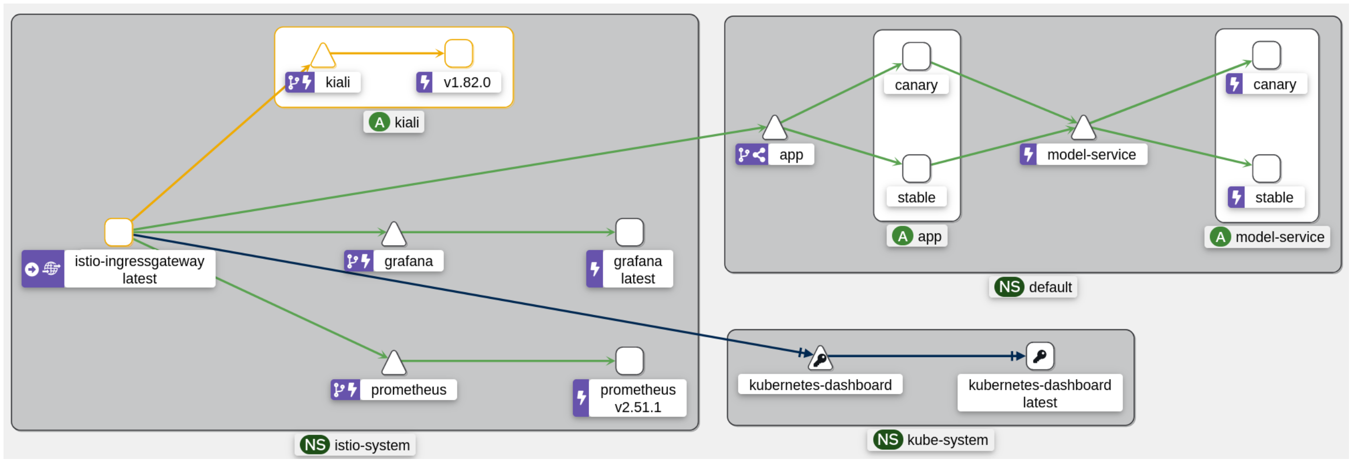


Figure 1: Deployment Visualisation

testing strategy covering the following angles: Feature and Data, Model Development, ML infrastructure and monitoring tests.

The feature and data tests ensure data integrity and feature quality. Our tests check whether the ratio between legitimate and phishing websites was fair for each dataset used for the model: training, validation and test. Furthermore, we have a non-functional test that checks whether the size of the trained model is below 10 MB. This is important because the larger the model is, the longer it takes to load it into memory for further operations.

For the model development angle, we have one test that checks the model performance across different data slices to identify potential biases. The test dataset has been divided into three data slices depending on the lengths of the URLs of the websites. We have also tested the non-determinism robustness of the model to ensure the latter produces consistent results across multiple runs with different seeds but the same data and parameters.

It is crucial to have tests that monitor the ML model, thus we have one test which checks the staleness of the trained model. In our test, a model is considered stale if it is older than 60 days. Checking the age of a model is essential because a model trained with outdated data may produce less accurate or invalid predictions on new test data.

Currently, we are missing a mutamorphic test in our testing framework. Mutamorphic testing is a procedure for evaluating the performance of ML models beyond validation using information from real data. The intention is to validate the model performance under different transformations of the real data. It would be great to incorporate a mutamorphic test in the future. This test would mutate various words in the URLs from the test dataset.

6 ADDITIONAL USE CASE (ISTIO)

In modern applications, it is essential to prioritize reliability, performance, and security due to limited available memory. Therefore, limiting the number of requests to an application is important.

Rate limiting is a technique for managing network resources by limiting the number of requests made to a server or system [4]. It is often used to prevent a single user from monopolizing resources and bot attacks such as DoS attacks and brute force attacks. The limit also ensures the system resources are fairly distributed among the users, improving the user experience since it protects the system's performance.

One of Istio's features for traffic management is rate limiting, which limits the number of a user can make in the given time frame. We defined a EnvoyFilter to enable local rate limit to the app service. Envoy, an L7 proxy, is specifically made for cloud-native apps and can be used as a sidecar in pods [13]. Additionally, they have built-in filter chains that allow for the integration of traffic management features, like rate limiting. We configured the rate limit to allow a maximum of 10 requests/min and to add an `x-local-rate-limit` response header to requests that are blocked. As shown in figure (TO ADD), the user receives a 429 response after 10 requests within a minute.

7 EXPERIMENTAL SETUP (ISTIO)

Note: this section will be added later

8 EXTENSION PROPOSAL

Current shortcoming. Many actions throughout the repositories are automated to speed up the different production phases. One of these actions is machine learning testing, explained more in Section 5. This action is used to verify some properties of the model and its training process. If changes are made to these components these tests can be run to check if their results are still correct either manually or automatically on push or pull requests. However, these tests require information about the model performance and other metrics only available from training a model which is expensive. The result is an expensive-to-run test set that is used for all changes even minor insignificant ones. The (multi-)hour-long duration of these tests can hinder the development process and annoy in situations where they are not relevant. Additionally, following the principles of sustainable software engineering, a rapidly growing discipline, we can not justify running these long tests unnecessarily [10, 14]. Running individual tests would solve this, however, this requires more manual input which is not desired. An easy (semi-)automated control over what size of tests to run would solve the majority of these problems.

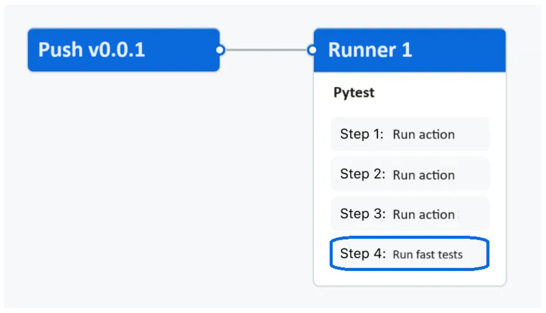
Proposed solution. Currently, both the instructions in the README and the GitHub Pytest workflow are written to execute all tests in the "tests" directory. We propose to use Pytest's built-in functionality to skip tests via commands [15] combined with GitHub's status check expressions [5]. As in the example provided by [15] the expensive tests can be marked as slow. With command arguments, these can then easily be skipped or included. The README can contain the different commands for manual usage, for automatic usage the GitHub workflows have to be adjusted. Some criteria have to be chosen to decide whether to skip or include these slow tests as documented in [5]. We suggest using a version tag for this, skipping the expensive tests for patches and non-versioned commits, and including them for minor or major updates.

An example of the proposed solution is provided in Figure 2. This example demonstrates how the GitHub runner runs a different selection of tests based on the semantic version split criteria we suggested. In Figure 2a a patch version is pushed which generally are small changes and do not require expensive tests. In the figure, it can be seen that only the fast tests are run. In Figure 2b a minor version that might significantly alter the model is pushed. For this push, the script will run all tests. This simple visualization demonstrates how our proposed solution could be implemented to save running expensive tests on small changes.

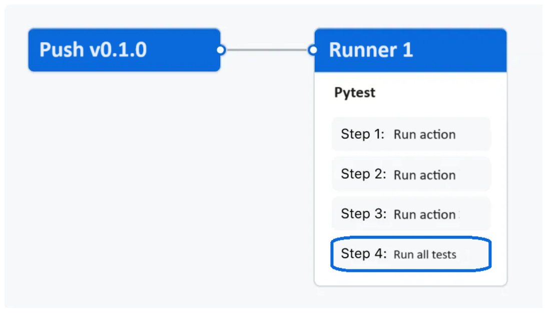
Extension testing. The proposed extension should be easily testable for both its manual and automatic applications. To test the manual use cases, the commands described in [15] should be added in the README if this extension is implemented. Then running these commands should show a difference in the number of tests that pass and are skipped as well as the runtime. Testing the changes to the automatic testing script requires a slightly different approach. The script is activated by pushing semantic version-tagged commits. With this, the script is testable by pushing commits with different semantic versions, one to trigger each set of tests. Depending on the semantic version, either the full or partial test set should be run. Again, the results can be observed in the number of passing/skipped tests and the runtime.

REFERENCES

[1] The Kubernetes Authors. [n.d.]. Kubernetes. https://kubernetes.io/.
[2] The Kubernetes Authors. [n.d.]. Kubernetes Dashboard. https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/.
[3] The Prometheus Authors. [n.d.]. Prometheus. https://prometheus.io/.
[4] Bijit Ghosh. 2023. Everything you need to know about rate limiting for APIs. (6 2023). https://medium.com/@bijit211987/everything-you-need-to-know-about-rate-limiting-for-apis-f236d2adcfff
[5] Github. 2024. Expressions - GitHub Docs — docs.github.com. https://docs.github.com/en/actions/learn-github-actions/expressions#status-check-functions. [Accessed 08-06-2024].
[6] Lyft Google, IBM. [n.d.]. Istio. https://istio.io/.
[7] HashiCorp. [n.d.]. Vagrant. https://www.vagrantup.com/.
[8] Red Hat. [n.d.]. Ansible. https://www.ansible.com/.
[9] Red Hat. [n.d.]. Kiali. https://kiali.io/.
[10] Avita Katal, Susheela Dahiya, and Tanupriya Choudhury. 2023. Energy efficiency in cloud computing data centers: a survey on software technologies. *Cluster Computing* 26, 3 (2023), 1845–1875.
[11] Grafana Labs. [n.d.]. Grafana. https://grafana.com/.
[12] Canonical Ltd. [n.d.]. MicroK8s. https://microk8s.io/.
[13] Ishujeet Panjeta. 2023. Mastering ISTIO rate limiting for efficient traffic management. (11 2023). https://medium.com/@ishujeetpanjeta/mastering-istio-rate-limiting-for-efficient-traffic-management-ff5acbcde7b3
[14] Giuseppe Procaccianti, Héctor Fernández, and Patricia Lago. 2016. Empirical evaluation of two best practices for energy-efficient software development. *Journal of Systems and Software* 117 (2016), 185–198.
[15] Pytest. 2024. Basic patterns and examples - pytest documentation — docs.pytest.org. https://docs.pytest.org/en/stable/example/simple.html#control-skipping-of-tests-according-to-command-line-option. [Accessed 08-06-2024].
[16] Sitemanager and Sitemanager. 2024. Comprehensive Guide to ML Model Testing and Evaluation. https://www.testingxperts.com/blog/ml-testing



(a)



(b)

Figure 2: An example of how pushes with different semantic versions trigger different steps in the GitHub runner changing the amount of tests to run. (a) is only a patch version and therefore only needs to run the fast tests while (b) is a minor version and requires all tests.