A6 REMLA Project Report – Group 4

Smruti Kshirsagar, Lucian Negru
Razvan Popescu, Ming Da Yang

1 INTRODUCTION

This report documents the project of Group 4 for REMLA 2024. Phishing represents a cyber attack where malicious users impersonate legitimate organizations or individuals through misleading communications, typically messages, emails, and websites, to trick victims into revealing sensitive information such as passwords, banking information, or personal identification details. Moreover, URL phishing is a specific type of phishing attack where attackers create malicious websites that mimic legitimate ones to deceive users into providing such sensitive information. These attacks involve the use of modified URLs that closely resemble those of trusted organizations, often using slight misspellings or variations in domain names to trick users. This project explores the use of Convolutional Neural Networks (CNNs) in detecting this type of phishing URLs.

2 RELEASE PIPELINE DOCUMENTATION

In this section, we document the process of making a new or updated version of a software package and a container image.

2.1 Software Package Release Pipeline

This type of release pipeline deals with the process of automating the stages of building, testing, and deploying software to ensure that new versions are delivered reliably. More exactly, we are documenting the release pipeline for $lib - ml$. This repository hosts a library responsible for handling the pre-processing logic applied to data utilized in model training, tokenization, and queries. The pipeline is divided into two jobs to maintain a clear separation of concerns and facilitate better organization and management of the release workflow. The first job focuses on code quality assurance through linting, testing, and coverage analysis, and it is triggered on every push to the *main* branch and on pull requests targeting the *main* branch. The purpose of the second job is to handle versioning, building, and publishing of the library, and it is triggered only when a new version tag is created, ensuring that the release process is initiated only for new version releases.

2.1.1 *Pipeline Steps.*

The pipeline steps of the quality assurance job are the following:

(1) **Checkout code:** Retrieves the codebase.
(2) **Set up Python:** Configures the Python environment using Python version $3.x$.
(3) **Install Poetry:** Install Poetry for Python dependency management.
(4) **Install dependencies:** Installs project dependencies using Poetry.
(5) **Lint with pylint:** Analyzes Python files for errors and coding standards violations using pylint.
(6) **Lint with flake8:** Checks Python code against coding style conventions using flake8.
(7) **Run tests with coverage:** Executes tests with pytest and generates coverage reports using coveragepy.
(8) **Check coverage threshold:** Ensures test coverage meets a defined threshold (100% coverage in this case).
(9) **Upload coverage report:** Uploads HTML coverage report artifacts.

When it comes to the building and publishing job, these steps are used:

(1) **Checkout code:** Retrieves the codebase.
(2) **Set up Python:** Configures the Python environment using Python version $3.x$.
(3) **Install Poetry:** Installs Poetry for Python dependency management.
(4) **Install dependencies:** Installs project dependencies using Poetry.
(5) **Update lib version:** uses *update_version.sh* script to fetch the latest Git tag version and adjust the version in the $pyproject.toml$ file accordingly.
(6) **Build lib:** Compiles the source code and packages it into distributable formats using Poetry.
(7) **Publish lib:** Publishes the built library to a package registry (PyPI) using twine.

2.1.2 *Pipeline Data Flow.*

In the first pipeline data flow, the input originates from the codebase stored in the repository. This input undergoes processing, in which code analysis is conducted to identify errors, violations, and assess test coverage. Following processing, the output includes linting reports and coverage data, essential for assessing code quality and test coverage. These artifacts offer valuable insights into the overall quality and robustness of the codebase.

In the second pipeline data flow, the input stems from the codebase stored in the repository, initiating a process of versioning, building, and packaging of the library. During processing, the source code is compiled and packaged into distributable formats. The output includes packaged distribution files and the repository with a tagged version, serving as indicators of the release status and version history. Versioned library files enable version control and distribution, ensuring the availability of stable and updated releases to users. The tagged version in the repository provides visibility into the release status and versioning history, enhancing clarity and organization within the codebase.

2.2 Container Image Release Pipeline

The release pipeline for the container image streamlines the process of packaging, testing, and deploying the software application within a containerized environment. We are documenting the release pipeline for $model - service$. This repository automates the process of building and deploying a containerized wrapper service for our pre-trained ML model. This service exposes the model via

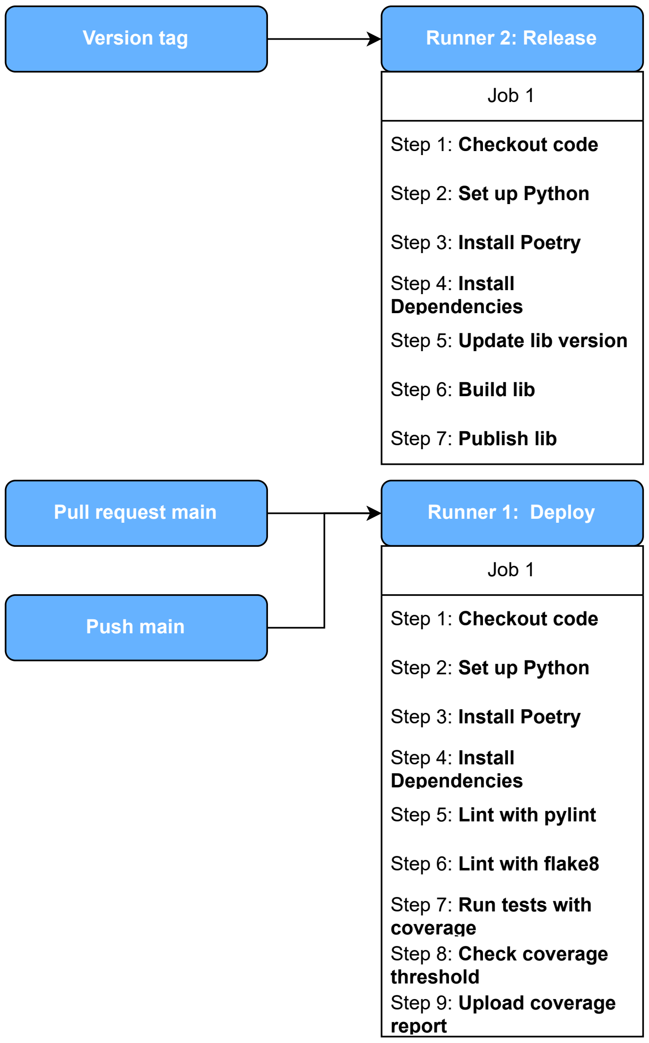

Figure 1: Github style visualization of the pipeline steps of quality assurance and, building and publishing jobs.

a REST API, ensuring scalability and accessibility to other components. This release pipeline is also divided into two distinct jobs to have a clear separation of concerns. The first job ensures code quality by analyzing Python files for errors and adherence to coding standards. On the other hand, the second job automates the process of building and deploying container images for the model service. Triggered by new version tags, it logs into the GitHub Container Registry (GHCR), builds the container image, and pushes it, facilitating the release of the model service. Together, these jobs ensure a streamlined and reliable release process for the containerized model service. Additionally, the repository includes a secondary pipeline that specifically focuses on testing the functionality of the model-service. Since the quality assurance and testing jobs follow similar steps as those in the $lib - ml$ pipeline, we will specifically focus on the image-building job in the following subsections.

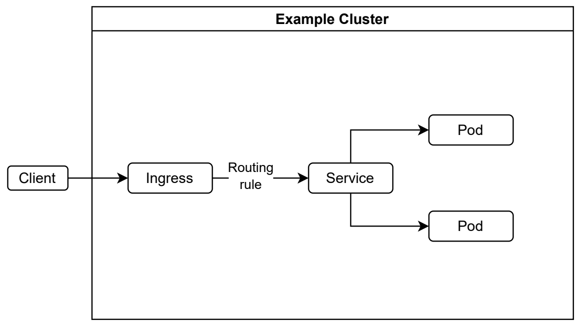

Figure 2: Example diagram illustrating the structure and flow of the Kubernetes deployment for A6

2.2.1 *Pipeline steps.*

The following steps are followed to build and push the container image:

1. **Checkout code:** Retrieves the codebase
2. **Login into GHCR:** Logs into the GitHub Container Registry with GitHub actor and token.
3. **Build & push image:** Build the Docker image, tag it with the Git version, and push it to GHCR.

2.2.2 *Pipeline Data Flow.*

In this pipeline, the data flow begins with the input of the codebase from the repository. During the build-and-push job, the code is processed by checking out the repository, logging into the GitHub Container Registry, and building the Docker image. The data output includes the Docker image tagged with the latest Git version. This image is then pushed to the GitHub Container Registry. The primary artifacts produced are the versioned Docker images, which facilitate version control and deployment. The tagged versions in the repository indicate the release status and version history, ensuring a streamlined and traceable release process.

3 DEPLOYMENT DOCUMENTATION

Deployment Documentation Document the final deployment of your application.

- Visualize the structure of the deployment (the entities and their connections).
- The diagram should illustrate the data flow for incoming requests.
- Include all resources that you have deployed in the cluster and their connections (not just the ones that are relevant for serving requests).
- Clarify the visualization through a concise elaboration

The current deployment of the application is done through a local Kubernetes cluster using Minikube. Currently a skeleton framework exists on the Github using vagrant and Ansible playbooks to setup a multi-machine Kubernetes network, however the setup for a multi-machine Kubernetes network was left out since it was not feasible within the timeframe.

In figure 2 you can see the structure and the data flow of the deployment.

4 EXTENSION PROPOSAL

Critically reflect on the current state of your project and identify the point that you find the most critical/annoying/error-prone. Equipped with the knowledge of the course ...

4.1 Shortcomings

4.2 Improvements

5 ADDITIONAL USE CASE

The Istio assignment asked you to adopt an additional Istio use case in your application. Excellent solutions document the use case in the report.

6 EXPERIMENTAL SETUP

The Istio assignment asked you to implement an experiment in your project infrastructure. Excellent solutions document the experiment in the report.

7 ML PIPELINE

As part of the assignment on ML Config Management, you had to configure your training pipeline using best practices and tools. Document how your pipeline is set up and the decisions you had to make to configure your project (tools, design, workarounds, and so on). This should come along with a description of your pipeline, its different steps, the tasks you have automated, which artefacts are being created, and so on.

8 ML TESTING DESIGN

Our project's testing strategy relies heavily on Unit testing, where we test each system component individually. The *model-training* repository tests mock DVC parameters and tests against the model's preprocessing, training, prediction and evaluation. Similarly, the **lib-ml** repository tests on consistent tokenization and encoding, including helper functions. *model-service* tests the **/predict** endpoint and the prediction. Finally, the *app* repository tests the rendered content on the type of URL provided for the phishing prediction.

All repositories include automated testing with reported coverage and a required 100% branch and method coverage. We chose unit tests because of their feasibility with our distributed system. The tests can be improved by applying Integration and Acceptance testing, which would require version-specific deployment of each repository to a test environment. Jenkins [1] is a powerful tool for this use case.

REFERENCES

[1]https://www.jenkins.io/