A6 REMLA Project Report – Group 4

Smruti Kshirsagar, Lucian Negru
Razvan Popescu, Ming Da Yang

1 INTRODUCTION

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

We are documenting the release pipeline for the Python package $lib-ml$. It hosts a library responsible for handling the preprocessing 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 or pull request 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 ($v*$) is created and pushed, 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 unit and metamorphic 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.
(10) **Clean Up:** Clean up unnecessary files and resources.

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 Python Package Index (PyPI) using twine.
(8) **Clean Up:** Clean up unnecessary files and resources.

2.1.2 Purpose & Implementation. The first job starts by retrieving the latest version of the codebase from a version control repository to ensure that the most recent updates are available for subsequent stages. This is done using $actions/checkout@v4$. Following this, the Python environment is configured using $actions/setup-python@v4$, with Python version 3.x. This ensures that the correct Python version is consistently utilized during the pipeline execution. Python dependency management is streamlined through $Poetry$, a tool that simplifies package installation and versioning, which can be obtained using $snok/install-poetry@v1$. Once dependencies are installed, the environment is prepared for both development and testing purposes. To uphold code quality, Python files undergo a thorough analysis for errors and adherence to coding standards using $pylint$. Similarly, code is analyzed against style conventions to ensure uniform formatting and readability with $flake8$. Test coverage reports are generated using $coveragepy$ to evaluate how thoroughly the code is tested using unit and metamorphic testing. A coverage threshold of 100% is set to ensure that every executable line of code is exercised by the test suite. Upon completion of testing, artifacts are created using $actions/upload-artifact@v2$ to document comprehensive test coverage results, facilitating complete review and analysis. Following that, unnecessary files and resources are removed to maintain an efficient and clutter-free development environment.

Regarding the second job in the pipeline, the primary differences lie in steps 5, 6, and 7 when compared to the previous job. Step 5 determines the version number for the library. It uses the $update\_version.sh$ script to fetch the latest Git tag version and

adjust the version number in the project's configuration file *pyproject.toml*. This step ensures that the library's version follows semantic versioning conventions and reflects the latest changes. The next step compiles the source code and packages it into distributable formats (e.g., wheel, source distribution) using *Poetry*. It generates the final artifacts that can be published to the package registry. The following step uses the built artifacts and publishes them to *PyPI* using the *twine* utility tool. This makes the library available for installation and use by other Python developers and projects.

2.1.3 Pipeline Data Flow. For the first job, the dataflow starts with the retrieval of the codebase, followed by the configuration of the Python environment. Project dependencies are installed next, creating a virtual environment with the necessary libraries. The code is then analyzed for errors and coding standards violations, producing linting reports. Testing generates test results and coverage data, which are subsequently evaluated against a predefined coverage threshold. If the threshold is met, HTML coverage report artifacts are uploaded. The process concludes with the cleanup of unnecessary files and resources.

For the second job, the initial part of the data flow remains the same as in the previous job. Then, if a new Git tag has been pushed, the library version is updated accordingly. The source code is then compiled and packaged into distributable formats. These packages, representing the main artifact, are subsequently published to the package manager. Finally, unnecessary files and resources are cleaned up.

2.2 Container Image Release Pipeline

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. The service exposes the model via a REST API, ensuring scalability and accessibility to other components. This release pipeline is also divided into multiple 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, while the second one focuses on testing and coverage reports. The last job automates the process of building and deploying container images for the model service. Triggered by new version tags ($v*$), 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. 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.

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 (GHCR) with GitHub actor and token.
(3) **Build & push image:** Build the Docker image, tag it with the Git version, and push it to the GHCR.
(4) **Clean Up:** Clean up unnecessary files and resources.

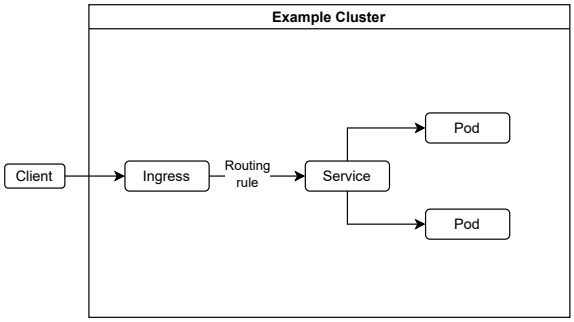

Figure 1: Example diagram illustrating the structure and flow of the Kubernetes deployment for A6

2.2.2 Purpose & Implementation. The job starts by retrieving the codebase, as explained in the software package release pipeline. This is crucial as it allows the workflow to work with the most recent changes and updates in the repository. The following step involves logging into the GHCR using the GitHub actor and token, accomplished through $docker/login - action@v1$. This authentication process is necessary to gain the appropriate permissions to push Docker images to the registry, ensuring secure and authorized access. Finally, the Docker image is built and tagged with the Git version, providing a unique and traceable identifier for the image. Once the image artifact is built, it is pushed to GHCR, making it available for deployment and use in various environments. After this, a cleanup step is performed to remove unnecessary files and resources.

2.2.3 Pipeline Data Flow. The workflow begins by retrieving the codebase, ensuring the latest version of the source code is available for the build process. It then logs into the GHCR for secure access. Following this, the Docker image is built, tagged with the Git version, and pushed to GHCR, creating a uniquely identifiable artifact. Finally, the workflow performs a cleanup, removing unnecessary files and resources.

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 1 you can see the structure and the data flow of the deployment.

Using th

4 EXTENSION PROPOSAL

In this section, we critically reflect on the current stage of the project. Identifying any shortcomings that are critical, annoying or error-prone. Lastly, state any improvements that can be made.

4.1 Shortcomings

The biggest shortcomings of the current project is the missing multi-VM Kubernetes deployment using Vagrant. Due to time constraint and making sure other parts of the project was working we had decided to move forth on the development on other aspects. The development and deployment of the application was done on a local kubernetes cluster using Minikube.

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

This section outlines the setup and configuration of our ML pipeline, including the tools chosen, design considerations, and automated tasks. We describe the pipeline's steps and the artifacts it generates, highlighting the decisions made to optimize project configuration. Specifically, we are documenting the pipeline for $model - training$. This repository handles the processes of training, predicting, and evaluating the model, ensuring they occur reproducibly while also versioning the generated artifacts and trained models.

7.1 Technology Stack

We use a *Cookiecutter* template to structure our project, ensuring a consistent and standardized project setup that promotes best practices and simplifies initialization. Our pipeline operates on Python version >= 3.10 and < 3.11, benefiting from its extensive library support and compatibility with the most recent tools and technologies. Furthermore, we use *Poetry* to manage dependencies and package our project. *Poetry* simplifies dependency management by providing a single tool for handling package installation, virtual environments, and project packaging. It ensures reproducibility by locking dependencies in a *poetry.lock* file, enabling consistent builds across different environments.

We also employ *Pylint* and *Flake*8 for code quality and style checks in our development process. *Pylint* analyzes Python code for errors, potential bugs, and adheres to coding standards, providing detailed feedback to ensure code consistency and quality. On the other hand, *Flake*8 focuses on enforcing coding style guidelines, to maintain clean and readable code. In addition, we utilize *Pytest* and *Coveragepy* for testing and code coverage analysis in our pipeline. *Pytest* is a powerful testing framework for Python that allows us to write simple and scalable tests. It provides a wide range of features for test discovery, fixtures, and assertions, making it highly flexible and easy to use. *Coveragepy*, on the other hand, measures the extent to which our codebase is covered by tests. It generates reports that show which parts of our code have been tested and which have not. This helps us identify areas that require additional testing and ensures that our tests effectively validate the functionality of our code.

Lastly, we use Data Version Control (DVC) for managing and versioning our data and models in the development lifecycle. DVC helps us automate the pipeline and track changes to datasets, models, and experiments, facilitating reproducibility. DVC is advantageous because it separates data and model management from code, enabling efficient versioning of datasets and models without bloating version control systems like Git. Moreover, it provides robust support for remote storage via *Google Drive*, allowing us to store and access large datasets and models efficiently. By utilizing DVC, we make sure that our project maintains traceable and reproducible workflows, necessary for collaboration and scalability.

7.2 Design Decisions

- to be scalable - to be reproducible - to store large data - etc.

7.3 Pipeline Stages

Our pipeline consists of five steps, each playing a crucial role in the overall workflow. Initial data and other interim results are fetched from and stored on Google Drive using DVC.

(1) **Data pre-processing:** This stage prepares the raw data (URLs and corresponding labels) for the subsequent stages of the pipeline, using the $lib - ml$ library. This involves tokenizing, padding, and encoding the data to ensure it is in the correct format for training and evaluation. The process includes loading the raw text data, splitting it into training, testing, and validation sets, fitting the tokenizer on it, and then applying tokenization and padding. This stage outputs the pre-processed data and the trained tokenizer together with the character index.
(2) **Model creation:** This stage involves defining the architecture of the ML model for URL phishing detection, using the *Keras* library. The process begins by loading the character index, a mapping of characters to their corresponding integer indices, which is essential for the embedding layer. The model is then constructed by incorporating several layers such as Embedding, Conv1D, MaxPooling1D, Dropout, Flatten, and Dense layers with a sigmoid activation function. This stage outputs the defined model architecture.
(3) **Model training:** This stage begins by compiling the model with specified parameters including binary_crossentropy as loss function, adam optimizer, and evaluation metrics such as accuracy. After compilation, the model is trained for 30 epochs using a batch size of 5000 and validation data to monitor performance. This stage outputs the trained model.

(4) **Model inference:** This stage uses the trained model to generate predictions on the test dataset. After prediction, the probabilities are converted into binary labels using a threshold of 0.5, and the true labels are reshaped for consistency. This stage outputs the reshaped true labels and the binary predictions.
(5) **Model evaluation:** Using the reshaped true labels and predicted binary labels, this stage calculates a classification report detailing precision, recall, F1-score, and support for each class. Additionally, it computes a confusion matrix to visualize true positive/negative and false positive/negative predictions, displaying accuracy as well. This stage outputs the accuracy, ROC AUC score, and F1-score in a JSON file for further analysis.

7.4 Automated Tasks

Automation plays an important role in our ML pipeline through both DVC and Git workflow. DVC automates data management and versioning across stages such as data pre-processing, model creation, training, inference, and evaluation. It tracks datasets, model architectures, hyperparameters, and evaluation metrics, ensuring reproducibility and allowing comparisons between model iterations. Simultaneously, the Git workflow automates code quality through linting tools such as *Pylint* and *Flake*8, enforcing coding standards and enhancing code readability. Automated testing frameworks such as *Pytest* validate the functionality of our pipeline, ensuring that changes do not compromise existing features. This is done through unit and metamorphic testing. Additionally, coverage reports generated by *Coveragepy* are automated within the workflow, providing insights into the codebase's test coverage. Together, these automated processes streamline development, improve reliability, and support efficient iteration in our ML project.

7.5 Artifacts

The ML pipeline generates essential artifacts across its stages to support development, evaluation, and deployment. These include raw data, pre-processed datasets, tokenizers, character indices, model architecture, trained models, binary predictions, and evaluation metrics (accuracy, F1-score, ROC-AUC). These artifacts, along with coverage reports from automated tests, ensure transparency, reproducibility, and efficiency throughout the pipeline's lifecycle.

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/