Project Report – Group 10

Michael Chan, Rémi Lejeune, Jan van der Meulen, Shayan Ramezani

1 PIPELINE DOCUMENTATION

In this section, all the pipelines are documented. The purpose of the pipelines in the `lib-ml`, `lib-version`, `model-service` and app repositories is to release the software. The purpose of the pipeline of the `model-training` repository is testing. The goal is to help new team members understand the pipeline steps, the tools used, and the flow of data and artifacts throughout the process.

1.1 Release Pipeline Documentation for `lib-ml` Python Package

For an illustrative overview, see Figure 1

1.1.1 Pipeline Overview. The release pipeline is triggered when a pull request (PR) is closed. It consists of two main jobs:

(1) **Test**: Runs tests on multiple environments to ensure the code is stable and functional.
(2) **Bump Version and Publish**: Bumps the package version, updates files, and publishes the package to PyPI.

1.1.2 Pipeline Steps.

Testing (`test` job). The purpose of the testing job is to ensure the package works correctly in different environments and Python versions.

Implementation.

- **Triggered**: When a pull request is merged.
- **Runs on**: Multiple OS and Python versions specified in a matrix.
- **Timeout**: 10 minutes.

Steps.

(1) **Checkout code**
  - Uses `actions/checkout@v4`.
  - Fetches the code from the merged pull request.
(2) **Set up Python**
  - Uses `actions/setup-python@v5`.
  - Sets up the specified Python version from the matrix.
(3) **Install Poetry**
  - Installs Poetry using `pipx install poetry` or `pip install poetry`.
(4) **Install dependencies**
  - Runs `poetry install –with dev` to install development dependencies.
(5) **Run tests**
  - Executes `poetry run pytest` to run the test suite.

Bump Version and Publish (`bump_version_and_publish` job). The purpose of this step is to bump the package version, update the version in relevant files, and publish the package to PyPI.

Implementation.

- **Triggered**: After the successful completion of the `test` job.
- **Runs on**: `ubuntu-latest`.

Steps. Steps (1) to (4) of the test job are repeated before moving on to the next steps, which are:

(1) **Bump version and push tag**
  - Uses `anothrNick/GitHub-tag-action@1.67.0`.
  - Bumps the version and pushes a new tag to the repository.
  - Environment variables used:
    - `GitHub_TOKEN`: Token for accessing GitHub.
    - `DEFAULT_BUMP`: Default version bump type (`patch`).
    - `TAG_CONTEXT`: Context for tagging (`branch`).
    - `WITH_V`: Whether to include 'v' in the version tag (false).
    - `PRERELEASE`: Pre-release flag (false).
(2) **Update files with new version**
  - Runs `poetry run bump-my-version replace –config-file pyproject.toml –new-version $(git describe –tags –abbrev=0)`.
  - Updates the version in the `pyproject.toml` file.
(3) **Build and publish to PyPI**
  - Runs `poetry publish –build -u __token__ -p $ secrets.PYPI_API_KEY` to build and publish the package to PyPI.
  - Uses `PYPI_API_KEY` secret for authentication.

1.1.3 Artifacts and Data Flow.

(1) **Source Code**: Checked out from the merged pull request.
(2) **Python Environment**: Set up using specified versions from the matrix.
(3) **Dependencies**: Installed using Poetry.
(4) **Test Results**: Determines if the publish job should proceed.
(5) **Version Tag**: Created and pushed to the repository.
(6) **Updated `pyproject.toml`**: Contains the new version.
(7) **Published Package**: The final artifact, published to PyPI.
(8) **Matrix**: a matrix specifying different operating systems and their versions.

1.1.4 Tools Used. Due to most tools being used in multiple pipelines, all tools are described in Section 1.6.

1.2 Release Pipeline Documentation for `model-service` Container Image

For an illustrative overview, see Figure 2.

1.2.1 Pipeline Overview. The release pipeline is triggered when a new tag matching the pattern `v[0-9]+.[0-9]+.[0-9]+` is pushed to the repository. It consists of a single job:

(1) **Build**: Builds the Docker image and pushes it to the GitHub Container Registry (GHCR).

1.2.2 Pipeline Steps.

Build (`build` job). The purpose of building the Docker image for the `model-service` and push it to the GitHub Container Registry (GHCR).

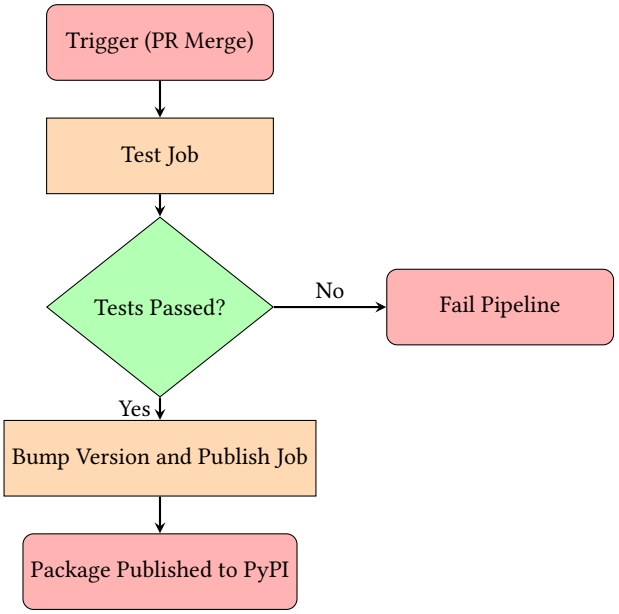

Figure 1: Flowchart of `lib-ml` Python Package Release Pipeline

Implementation.

- **Triggered**: When a tag matching the pattern `{v[0-9]+.[0-9]+.[0-9]+}` is pushed.
- **Runs on**: `ubuntu-22.04`.

Steps.

(1) **Checkout code**
   - Uses `actions/checkout@v4`.
   - Checks out the code from the repository.

(2) **Parse version info from tag**
   - Runs a shell script to parse the version information from the tag.
   - Extracts the major, minor, and patch version numbers.
   - Sets the parsed version numbers as environment variables.

(3) **Registry Login (ghcr.io)**
   - Logs into the GitHub Container Registry (GHCR) using the `GH_TOKEN` secret.
   - Uses the GitHub Actions context for authentication.

(4) **Build and Push Docker Image**
   - Builds the Docker image using the Dockerfile in the repository.
   - Tags the image with:
     - Full version (e.g., `v1.2.3`).
     - Major and minor version with `.latest` suffix (e.g., `1.2.latest`).
     - Major version with `.latest` suffix (e.g., `1.latest`).
     - `latest` tag.
   - Pushes all tagged images to the GitHub Container Registry.

1.2.3 Artifacts and Data Flow.

(1) **Source Code**: Checked out from the repository.
(2) **Docker Image**: Built from the source code.
(3) **Version Tags**: Parsed from the pushed tag and used to tag the Docker image.
(4) **Published Image**: The final artifact, pushed to the GitHub Container Registry.

1.2.4 Tools Used. Due to most tools being used in multiple pipelines, all tools are described in Section 1.6.

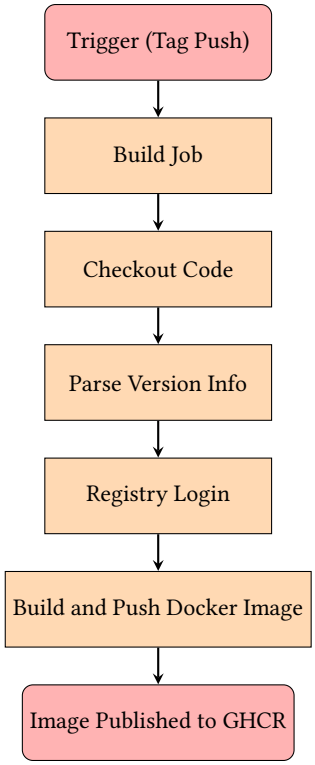

Figure 2: Flowchart of `model-service` Container Image Release Pipeline

1.3 Release Pipeline Documentation for `App` Container Image

As the release pipeline of the app container image is the same as the model-service release pipeline, please refer to Section 1.2 for a detailed specification.

1.4 Release Pipeline Documentation for `Lib-Version` Container Image

1.4.1 Pipeline Overview. The release pipeline is triggered when a new tag matching a specific pattern is pushed to the repository. The pipeline then runs a single job.

(1) **Get Version from Metadata and Publish**: the poetry-dynamic-versioning package uses the git metadata to retrieve the latest git tag. Afterward, it uses this tag to push to PyPi.

1.4.2 Pipeline Steps.

Implementation.

(1) **Triggered**: when a new tag matching the pattern `v[0-9]+.v[0-9]+.v[0-9]+` is pushed to the repository.
(2) **Runs on**: latest version of Ubuntu available to GitHub.

Steps. These steps are visualized in Figure 3.

(1) **Checkout Repository**: uses `actions/checkout@v4` to load the code from the repository.
(2) **Set up Python**: uses `actions/setup-python@v2` to set up python.
(3) **Install Poetry and Dependencies**: uses pip through a shell command to install poetry. Afterward, the following dependencies are installed: setuptools, setuptools_scm, wheel, twine and poetry-dynamic-versioning.
(4) **Install Python Dependencies**: poetry is used to install Python dependencies.
(5) **Build Package**: Poetry is used to build the package.
(6) **Publish Package to PyPi**: Lastly, twine is used to upload the package to PyPi.

1.4.3 Artifacts and Dataflow.

(1) **Source Code**: checked out from the repository.
(2) **Source Distribution**: the `.tar.gz` pushed to PyPi.
(3) **Built Distribution**: the `.whl` created by Poetry.

1.4.4 Tools Used. Due to most tools being used in multiple pipelines, all tools are described in Section 1.6.

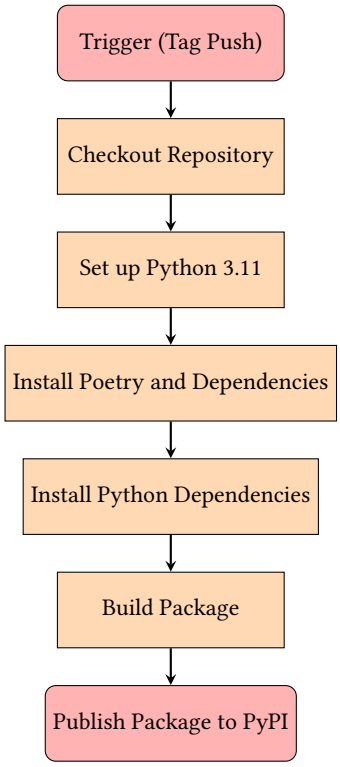

Figure 3: Flowchart of `lib-version` Release Pipeline

1.5 Pipeline Documentation for `Model-Training`

1.5.1 Pipeline Overview. This testing pipeline is triggered on any push to any branch, and when a pull request to the main branch is opened. On these triggers, the following jobs are executed:

(1) **Unit Tests**: runs all the unit tests on macOS, Windows and a Linux system. Afterward, the test results are uploaded.
(2) **Integration Tests**: this job is only executed whenever a pull request to main is made. This job runs all the integration tests on different operating systems.
(3) **Publish Test Results**: this job downloads the test results and creates a badge to display the results.

1.5.2 Pipeline Steps.

Implementation.

(1) **Triggered**: on push and PR.
(2) **Runs on**: `ubuntu-latest`, `windows-latest` and `macos-latest`.

Steps - testing jobs. These steps are visualized in Figure 4. As the steps for both testing pipelines are very similar, they are only described once. The flag in the testing command decides which tests are executed, and the file locations are slightly different.

(1) **Checkout Repository and Setup Python**: uses `actions/checkout@v4` to check out the repository code. It references the specific push request by making use of the merge commit SHA. Then, based on which OS the job is executed on, `actions/setup-python@v5` sets up the correct version of python. To find the correct version, the `matrix.python` environment variable is used.
(2) **Install Poetry and Dependencies**: shell commands are used to interface with pip and poetry. Firstly, pip installs poetry, then, poetry installs the dependencies.
(3) **Create and Set Permissions for Test Directory** this step uses shell commands to set up the testing directory.
   (a) `chmod 777 ./tests` gives read-write permissions to all users.
   (b) `mkdir -p tests-results/unit-test/matrix.name` create a directory to store the results. The file is stored in the `tests-results/integration-test/matrix.name` folder in the integration tests.
   (c) `chmod 777 tests-results/unit-test/matrix.name` gives read-write permissions to all users for this folder.
(4) **Run Tests**: all tests are executed with the `poetry run pytest` command. Some flags are set, 1) code coverage for the src directory is measured, 2) output location of coverage report is set to: `--cov-report=xml:./tests/unittests.xml`, and 3) output format of test results is set to junit-xml and location is based on `matrix.name` environment variable.
(5) **Upload Coverage to Codecov**: the code coverage is upload to Codecov using: `codecov/codecov-action@v4` with the token `secrets.CODECOV_TOKEN`.
(6) **Upload Test Results** the test results are uploaded to GitHub using the action `actions/upload-artifact@v4`.

Steps - publish results job. These steps are visualized in Figure 5.

(1) **Download Artifacts** uses `actions/download-artifact@v4` to download the results of the tests.

(2) **Publish Test Results**: the downloaded test results are published using `EnricoMi/publish-unit-test-result-action@v2`. This is a mature GitHub Action that publishes the results nicely in pull requests.
(3) **Set Badge Color**: a shell command is used to retrieve the test results and set the badge color. The badge will be green, if all tests pass.
(4) **Create Badge**: uses the GitHub Action `emibcn/badge-action@`. to create the badge.
(5) **Upload Badge to Gist**: uploads the created badge to Gist using the GitHub Action `andymckay/append-gist-action@`.

1.5.3 *Artifacts and Dataflow.*

(1) **Code coverage**: the percentage of lines in the `src` directory covered by the tests.
(2) **Test Results**: the results of the unit and integration (e.g. pass/fail/crash) tests on the different operating systems.
(3) **GitHub Badge**: the badge created from the test results (Figure 12).
(4) **Matrix**: a matrix specifying different operating systems and their versions.

1.6 Tools Used

(1) **GitHub Actions**: CI/CD platform for running workflows.
(2) **Pip**: The default package installer for Python, enabling users to install and manage software packages from the Python Package Index (PyPI).
(3) **Poetry**: A Python dependency management tool that simplifies project setup, dependency resolution, and packaging.
(4) **PyTest**: A mature Python testing framework.
(5) **Codecov**: A code coverage analysis tool that integrates with CI/CD workflows to provide reports and insights on test coverage.
(6) **GitHub Badge**: A visual indicator that provides real-time updates to indicate project status.
(7) **Twine**: a utility for publishing Python packages to the Python Package Index (PyPI).
(8) **Poetry dynamic versioning package**: a tool that does dynamic package versioning.
(9) **anothrNick/GitHub-tag-action**: Action for tagging releases.
(10) **emibcn/badge-action** an action for generating a badge.
(11) **EnricoMi/publish-unit-test-result-action** publishes test results to GitHub repository.
(12) **codecov/codecov-action** report code coverage.
(13) **andymckay/append-gist-action** appends results to gist.
(14) **Basic GitHub Actions**: *checkout*, *setup-python*, *download-artifact* and *upload-artifact*.

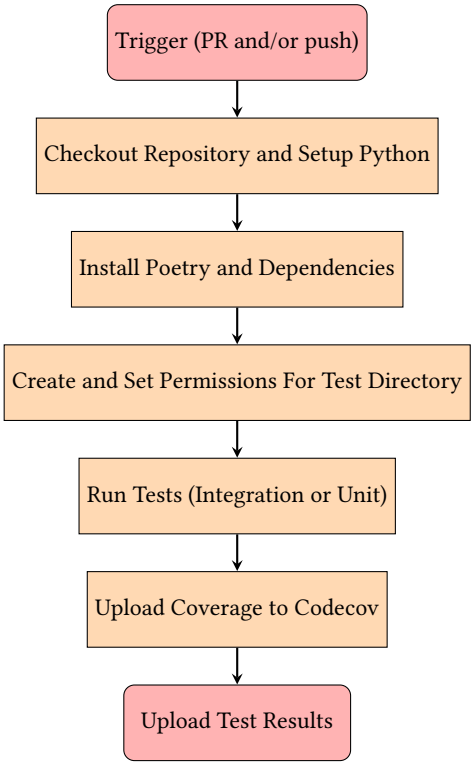

Figure 4: Flowchart of `model-training` Testing Jobs

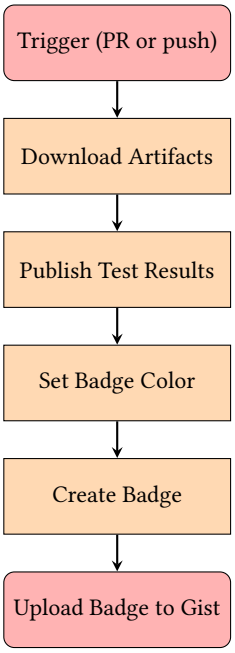

Figure 5: Flowchart of `model-training` Test Publishing Job

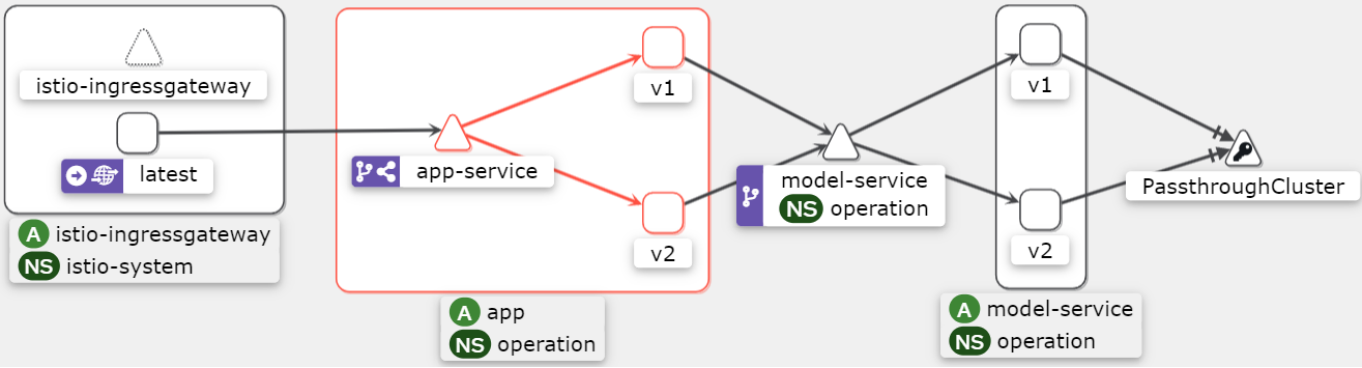

Figure 6: Deployment of operation

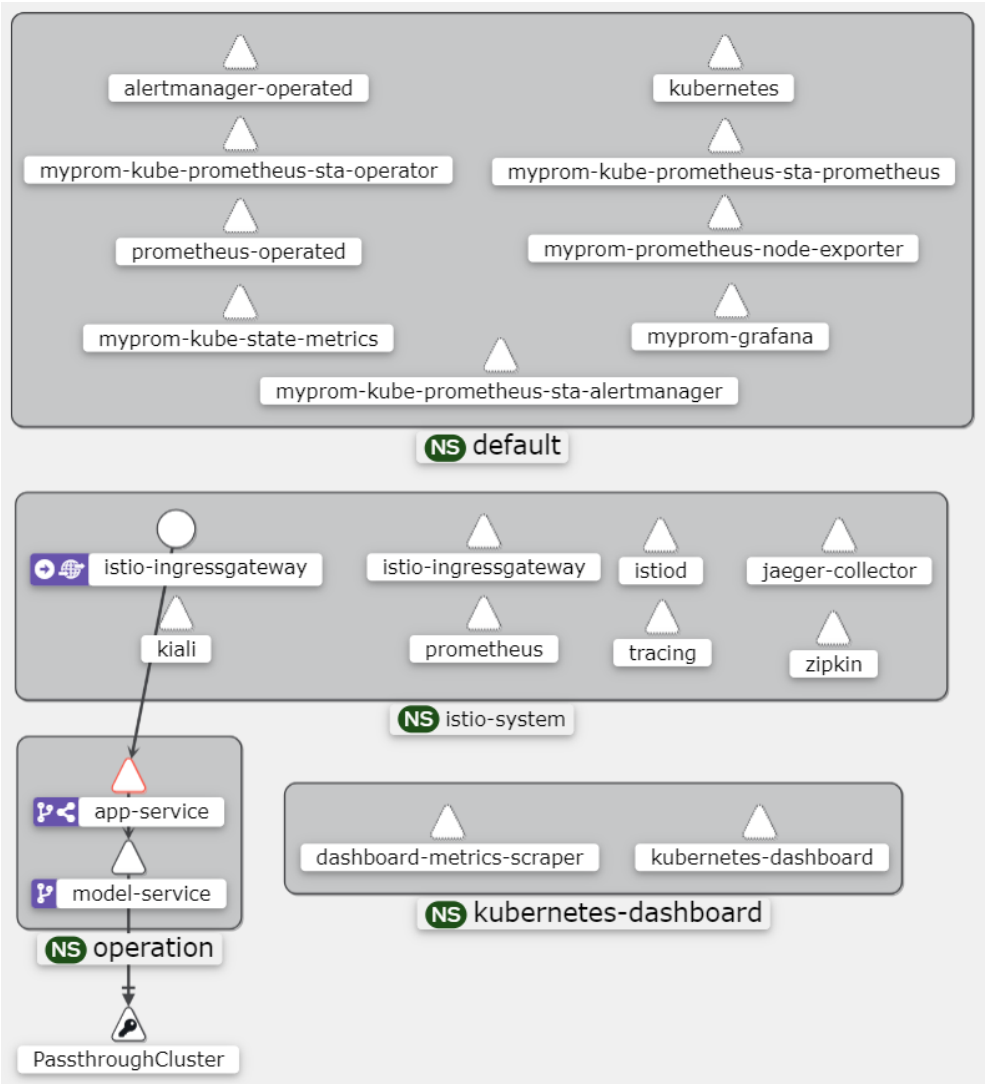

Figure 7: Entire deployment cluster

2 DEPLOYMENT DOCUMENTATION

This section covers the deployment and data flow of the project. The project is deployed on a minikube cluster, this allows for scaling of the project as well as stability through backup deployment replicas.

2.1 Deployment structure

The deployment structure is visualized in Figure 6. It consists of two main services with each having two deployment versions for a potential canary release:

(1) **app-service**: App-service handles the front-end of the application and serves the page that the user directly interacts with. This is done through two flask deployments each running a different version. Each of these deployments consist of only 1 replica and therefore do not provide redundancy out of the box. This can be scaled up to improve availability.
(2) **model-service**: Model-service handles the back-end of the application and provides the "predict" endpoint for the app to interact with, which returns the prediction result given an url. This is done through two flask deployments hosting the model each running a different version. Each of these deployments also consist of only 1 replica.

2.2 Data flow

The Ingress gateway is provided by Istio and serves as the entry point to the application. The request is then handled by the app-service which forwards the request to one of the two deployment versions with equal probability (50/50). The user is then able to make a prediction on the app. The app then sends a post request to the model-service which forwards it to the corresponding model-service deployment version. The prediction is then directly returned through the passthrough cluster.

2.3 Full cluster deployment

The project furthermore employs various monitoring tools on the cluster, these include Prometheus and Grafana. Additionally various other dashboards can be added to the cluster such as the Minikube, Jaeger and Kiali dashboard. The full deployment of all clusters can be found in Figure 7. Prometheus scrapes the "/metrics" endpoint of the model-service deployments to keep track of various metrics. Grafana can then be connected to prometheus to provide a intuitive dashboard of the various metrics.

3 EXTENSION PROPOSAL

During the project, we experienced issues with setting up the software environment using Ansible and Vagrant. These issues became apparent as we had to do a more complex task by connecting the virtual machines and setting up the Kubernetes cluster for the virtual machines. The goal of Ansible is to provision a local software environment together with Vagrant, which enables the software to run locally on virtual machines. This should remove the *it runs on my machine* argument by providing a stable and reproducible environment. As a result, software development should speed up. However, during the development process, we noticed that using Ansible instead slowed down development due to several issues we encountered. As solutions depend on the goal of the system, we have chosen the following goal: *deploying and maintaining an in-house cluster*. After a discussion of the issues, we will propose an extensions based on this use-case.

3.1 Deployment of an In-House Cluster

This extension would change the architecture of the system and the deployment process by using a different provisioning technology. Vagrant and Chef would be used to provision this server to allow for easy scaling and more versatility in continuous deployment. Docker-compose would be used for easy to set up and quick local development. This new architecture is visualized in Figure 8. It would tackle some issues that we had with Ansible, which are described in the next paragraph.

Issues With Ansible. In this paragraph, we will discuss some issues that we experienced personally. These issues are not irresolvable, but we have seen them reflected in multiple posts and blogs online. During development, Ansible sometimes breaks despite no apparent changes in the local environment. Different inventory.cfg files might be required depending on the developer's operating system, which hurts reproducibility. Command outputs have limited verbosity or the verbosity can be needlessly complex, and playbook execution can have very long wait times[3]. Additionally, some commands that run successfully when executed directly via SSH do not work in the playbook. Configuration setup options are also limited and sometimes difficult to implement, due to Ansible using a different thread for every command. As discussed in this blog by Eric Hu, Ansible seems better for setting up applications than for configuration management (e.g. setting up a Kubernetes cluster)[2]. This was something we experienced ourselves as well, as Ansible creates a new shell for every command. As a result, the inexperienced user can lose environment and/or configuration settings when executing commands sequentially[4]. Even though it is certainly possible to configure a complex environment with Ansible, it might not be the best tool for our specific job.

A comparison of Ansible and Chef. [1][5] Chef is generally considered to have better continuous deployment features. Furthermore, Chef is a more mature technology and has better configuration management and better deployment features. However, Ansible offers stronger security features, primarily leveraging SSH for secure communication. This makes it easy to set up and maintain a secure environment. Furthermore, Ansible is usually considered easier to set up.

Applicability of Chef in Use-Case. In this case Chef is likely the better option. The benefits of Ansible, strong security, and easy set-up are less important in a secure environment and with a team that works on the project for a longer time. In this scenario, the deployment features and versatile configuration management likely provide a lot of value.

Measuring the Success. To measure the success, firstly the quality of the baseline will have to be measured (the current Vagrant + Ansible system). Some useful metrics would be: average time taken by an engineer to close a configuration issue, time to deploy a machine, cluster uptime and cost of maintaining the cluster. Afterward, the new cluster can be deployed, and the same metrics can be measured.

Based on these metrics, some level of objectivity can be achieved. It is important to note that this solution depends on the proficiency of the engineers with Chef and Ansible. Therefore, some time might be needed for everyone to get proficient with the new technologies.

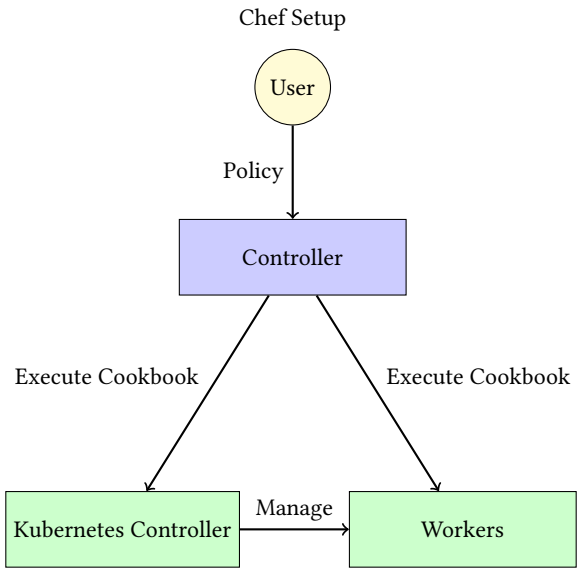

Figure 8: Example Chef Infrastructure

4 ADDITIONAL USE CASE

In addition to the canary release the Istio service mesh is used to limit the rate of local requests. For this an EnvoyFilter is deployed which limits the number of requests allowed for each instance of app-deployment. It defines a bucket with a maximum capacity of 20 requests. This bucket is completely refilled every minute. This ensures that the services will not be overrun and that they will be able to keep up with the workload. However since it is random which deployment service will be assigned to a user it is possible that a user will be rate limited on one deployment while not on the other which results in inconsistent page availability as the user is not rerouted to the available service.

5 EXPERIMENTAL SETUP

The current experiment is still being discussed, however the infrastructure allows us to test two different phishing detection models and then utilize metrics such as the average rate of phishing detected and the average phishing probability returned by the model. The current model-service endpoints does not yet support predictions with known feature-label pairs so it is not yet possible to determine metrics such as accuracy. An example hypothesis that can currently be tested could be: *A model trained on fewer epochs is more likely to predict "legitimate" and therefore has a lower average phishing rate.*

6 ML-PIPELINE

This section covers the pipeline for training the phishing detection model as well as the different tools used. Furthermore it explains some of the shortcomings of the pipeline.

6.1 Pipeline

The pipeline is managed by DVC and consists of following three stages: preprocessing, training, and testing. The pipeline was designed such that after each of these stages intermediate output files could be created which can serve as checkpoints for the next stage. An overview of the pipeline can be found in Figure 9.

6.2 Tools

The project relies on DVC for both pipeline management and data version control. This allows for great reproducibility as well as efficient data management. Furthermore google drive was used for remote storage. One google drive folder manages all the DVC artifacts, while a seperate google drive folder contains the models and related files specifically for deployment. These files are downloaded during runtime of model-service. This allows for swapping of the models without creating a new image.

6.3 Shortcomings

A major shortcoming of the pipeline is that it is only partially automated as some artifacts still need to be manually uploaded to the separate drive folder that was created specifically for deployment. This can be very error prone as this can not only introduce human error but it also means that the manually uploaded version for deployment is not versioned. This makes it not immediately clear which version of the model is currently deployed.

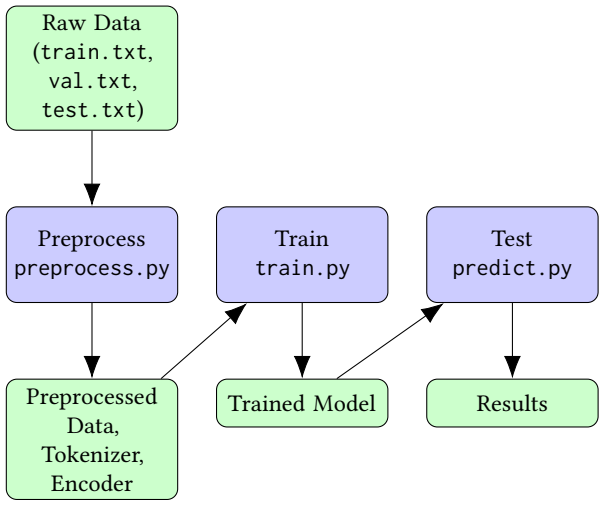

Figure 9: ML Pipeline

7 ML TESTING DESIGN

This section will first describe and explain the tests implemented, it will then talk about the limitations and which additional tests could

be added. Finally, it will explain what was done for continuous training.

7.1 Automated Tests

The tests created are there to ensure the functionality of the training pipeline, therefore they were implemented in the model-training repository.

Here's a list of the different tests that were done:

- Data quality tests: These tests are here to ensure the quality of the data, this is done by verifying the uniqueness of data samples, the data should at least be 99% unique.
- Integration tests: These tests ensure that the complete pipeline can be run together without causing any issues, they follow the following pattern: preprocess the data, build the model, train the model, test the model, and plot
- Model definition test: Verify that the model is defined properly
- Model development tests:
  - Capability test: Ensure that if the data (URL) starts with HTTP or HTTPS it yields similar results.
  - Non-determinism test: Ensure that the training phase is nondeterministic and that the difference accuracy between 2 trained models should be minimal
- Test monitoring: Ensure that it doesn't use too much RAM <4GB
- Test train: Ensure that the training phase returns a trained model
- Test Preprocess: Ensure data exists and that it has the right shape

To improve the testing multiple things can be done, first, add more unit tests to increase the test coverage. Currently, none of the tests are using the DVC data, therefore it would be useful to add some. Finally, some metamorphic tests could be added.

7.2 Continuous training

To ensure the quality of our code, a testing pipeline was set up using GitHub workflows.

This pipeline will run the tests in each OS (Windows, MacOS, Linux) to ensure no dependency issues. Then, it will upload the results to Codecov as seen in Figure 10.

Then a test report will be created as shown in Figure 11, it will show up as a comment in each merge request.

Finally, a badge is displayed as shown in Figure 12 on the README with the results, this badge is updated every time we merge to the main branch.

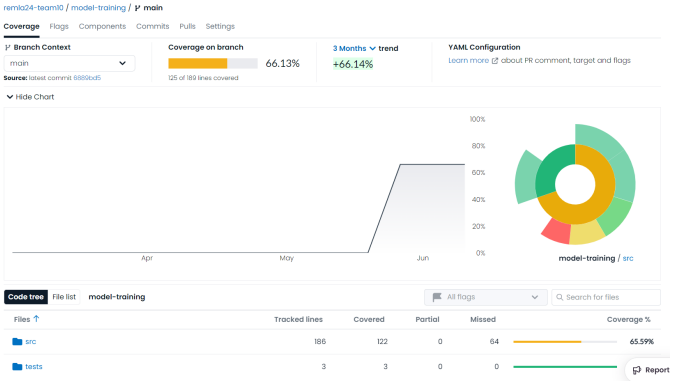

Figure 10: Screenshot of Codecov dashboard of model-training

Figure 11: Screenshot of a test report from the workflow

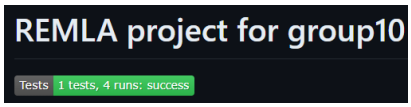

Figure 12: Screenshot of the badge

REFERENCES

[1] Shannon Flynn. 2022. *Ansible vs Chef: Compare DevOps Tools.* https://www.techrepublic.com/article/ansible-vs-chef/ Accessed: 2024-06-10.

[2] Eric Hu. 2024. *Chef vs. Puppet vs. Ansible: a side-by-side comparison for 2024.* https://betterstack.com/community/comparisons/chef-vs-puppet-vs-ansible/#7-configuration-management-chef-and-puppet-wins Accessed: 2024-06-10.

[3] HubertNNN. 2022. *Why is ansible slow with simple tasks.* https://stackoverflow.com/questions/71565392/why-is-ansible-slow-with-simple-tasks Accessed: 2024-06-10.

[4] Pldimitrov. 2023. *Not possible to source .bashrc with Ansible.* https://stackoverflow.com/questions/22256884/not-possible-to-source-bashrc-with-ansible Accessed: 2024-06-10.

[5] Kaushik Sen. 2024. *Ansible vs Chef Updated for 2024.* https://www.upguard.com/blog/ansible-vs-chef#toc-4 Accessed: 2024-06-10.