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

1.1.1 Pipeline Overview. There are two release pipeline:

(1) **Test**: Runs tests on multiple environments to ensure the code is stable and functional. This is triggered for any push. See Figure 1.
(2) **Bump Version and Publish**: Bumps the package version, updates files, and publishes the package to PyPI. This is triggered when a Pull Request is closed. See Figure 2.

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**: After a push.
- **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`, if not possible `pip install poetry` is used.
(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.

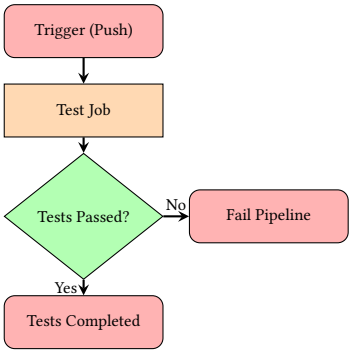

Figure 1: Flowchart of `lib-ml` Python Package Test Pipeline

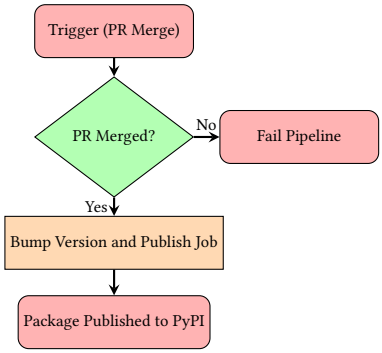

Figure 2: Flowchart of `lib-ml` Python Package Publish Pipeline

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

For an illustrative overview, see Figure 3.

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).

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.

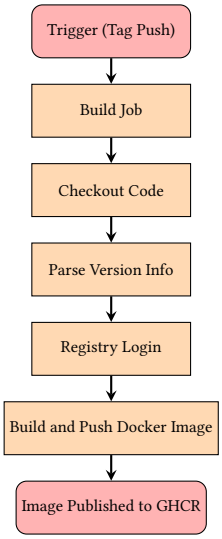

Figure 3: Flowchart of `model-service` Container Image Release Pipeline

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 4.

(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.

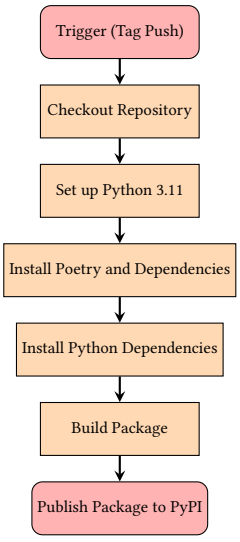

Figure 4: Flowchart of `lib-version` Release Pipeline

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) **Lint Code**: This job checks the code for styling issues using pre-commit hooks and automatically formats the code if necessary.
(2) **Unit Tests**: Runs all the unit tests on macOS, Windows, and a Linux system. Afterward, the test results are uploaded.
(3) **Integration Tests**: This job is only executed whenever a pull request to the main branch is made. This job runs all the integration tests on different operating systems.
(4) **Publish Test Results**: This job downloads the test results and creates a badge to display the results.

1.5.2 *Pipeline Steps.*

Implementation.

- **Triggered**: On push to any branch and pull request to the main branch.
- **Runs on**: `ubuntu-latest`, `windows-latest`, and `macos-latest`.

Steps - Lint Code Job.

(1) **Checkout Code**: Uses `actions/checkout@v4` to check out the repository code.
(2) **Set up Python**: Uses `actions/setup-python@v5` to set up Python 3.11.
(3) **Install Poetry**: Installs Poetry using `pipx install poetry` or `pip install poetry`.
(4) **Install Dependencies**: Runs `poetry install` to install dependencies.
(5) **Install mypy Stubs**: Installs mypy stubs using `poetry run mypy --install-types --non-interactive`.
(6) **Install pre-commit**: Installs pre-commit hooks using `poetry run pre-commit install`.
(7) **Run pre-commit Hooks**: Runs pre-commit hooks using `poetry run pre-commit run --all-files`.
(8) **Commit and Push Changes if any**: Checks for changes and commits them if necessary.

Steps - Unit and Integration Testing Jobs. These steps are visualized in Figure 5. The steps for both testing pipelines are very similar, with differences primarily in the flags used during testing and file locations for results.

(1) **Checkout Repository and Setup Python**: Uses `actions/checkout@v4` to check out the repository code, referencing the specific push request by the merge commit SHA. `actions/setup-python@v5` sets up the correct version of Python using `matrix.python`.
(2) **Install Poetry and Dependencies**: Uses shell commands to install Poetry and dependencies.
(3) **Create and Set Permissions for Test Directory**: Uses shell commands to set up the testing directory with read-write permissions.
    - `chmod 777 ./tests`: Gives read-write permissions to all users.
    - `mkdir -p tests-results/unit-tests/matrix.name`: Creates a directory to store the results. For integration tests, results are stored in `tests-results/integration-tests/matrix.name`.

- `chmod 777 tests-results/unit-tests/matrix.name`: Gives read-write permissions to all users for this folder.

(4) **Run Tests**: Executes tests with `poetry run pytest`, setting flags for code coverage, coverage report output location, and test result format.
(5) **Upload Coverage to Codecov**: Uses `codecov/codecov-action@v4` to upload code coverage to Codecov using the token `secrets.CODECOV_TOKEN`.
(6) **Upload Test Results**: Uses `actions/upload-artifact@v4` to upload test results to GitHub.

Steps - Publish Results Job. These steps are visualized in Figure 6.

(1) **Download Artifacts**: Uses `actions/download-artifact@v4` to download the test results.
(2) **Publish Test Results**: Uses `EnricoMi/publish-unit-test-result-action@v2` to publish test results in pull requests.
(3) **Set Badge Color**: Uses a shell command to retrieve the test results and set the badge color. The badge is green if all tests pass.
(4) **Create Badge**: Uses `emibcn/badge-action@` to create the badge.
(5) **Upload Badge to Gist**: Uses `andymckay/append-gist-action@` to upload the created badge to Gist.

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 tests (e.g., pass/fail/crash) on the different operating systems.
(3) **GitHub Badge**: The badge created from the test results (Figure 15).
(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.

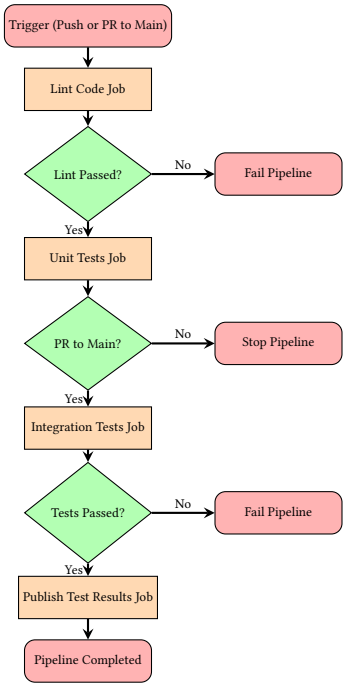

Figure 5: Flowchart of `Model-Training` Testing Pipeline

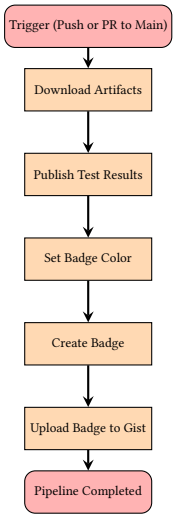

Figure 6: Flowchart of `Model-Training` Publish Results Pipeline

(13) **andymckay/append-gist-action** appends results to gist.
(14) **Basic GitHub Actions**: *checkout*, *setup-python*, *download-artifact* and *upload-artifact*.

2 LINTERS FOR PRE-COMMIT

The `Model-Training` repository utilizes several linters integrated with pre-commit hooks to ensure code quality and consistency.

The configuration for these linters is defined in the `.pre-commit-config.yaml` file. Below is a brief overview of the linters used, why they were chosen, and the rationale behind their specific settings.

2.1 Linters Used

(1) **isort**:
- **Repository**: https://github.com/pycqa/isort
- **Version**: 5.13.2
- **Purpose**: isort is used to sort imports in Python files according to the specified style guide.
- **Settings**: The `–profile black` argument is used to ensure compatibility with the Black formatter.

(2) **black**:
- **Repository**: https://github.com/psf/black-pre-commit-mirror
- **Version**: 24.4.2
- **Purpose**: Black is an opinionated code formatter that ensures code style consistency.
- **Settings**:
  - `–line-length 88`: Sets the maximum line length to 88 characters.
  - `exclude: tests/`: Excludes the `tests/` directory from formatting.

(3) **pylint**:
- **Repository**: Local repository.
- **Purpose**: Pylint is used for code analysis to detect bugs and enforce coding standards.
- **Settings**:
  - `entry: pylint`: Specifies the pylint command.
  - `types: [python]`: Ensures it runs on Python files.
  - `files: src/`: Limits pylint checks to the `src/` directory.

(4) **mypy**:
- **Repository**: https://github.com/pre-commit/mirrors-mypy
- **Version**: v1.10.0
- **Purpose**: Mypy performs static type checking for Python code.
- **Settings**:
  - `–install-types –non-interactive`: Automatically installs missing type stubs.
  - `additional_dependencies: [numpy, matplotlib]`: Ensures that numpy and matplotlib type stubs are available.
  - `exclude: 'tests/'`: Excludes the `tests/` directory from type checking.

(5) **bandit**:
- **Repository**: https://github.com/PyCQA/bandit
- **Version**: 1.7.8
- **Purpose**: Bandit is used for security analysis to identify common security issues in Python code.
- **Settings**:
  - `–skip B106`: Skips the check for hardcoded passwords.
  - `–exclude tests`: Excludes the `tests/` directory from analysis.
  - `files: src/`: Limits Bandit checks to the `src/` directory.

2.2 Rationale for Linter Selection and Settings

The chosen linters each play a specific role in maintaining code quality and security:

- **isort** and **black** ensure that the codebase follows consistent style guidelines, which makes the code more readable and maintainable. The settings for isort ensure compatibility with Black's formatting style, while Black's settings enforce a standard line length and exclude test files which do not need styling.
- **pylint** is used for comprehensive code analysis to catch potential errors and enforce coding standards. It is configured to focus on the source directory to ensure the main codebase adheres to these standards.
- **mypy** enhances code reliability through static type checking, catching type errors before runtime. The settings enable automatic installation of type stubs and focus on the main codebase, excluding test files where type checking is less critical.
- **bandit** provides security checks to identify vulnerabilities in the code. The settings skip checks that are not relevant (like hardcoded passwords) and focus on the main codebase to ensure it is secure.
- Excluding the `tests/` directory in several linters prevents unnecessary checks and formatting on test files, which are less critical for these types of analysis.

3 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.

3.1 Deployment structure

The deployment structure is visualized in Figure 7. 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.

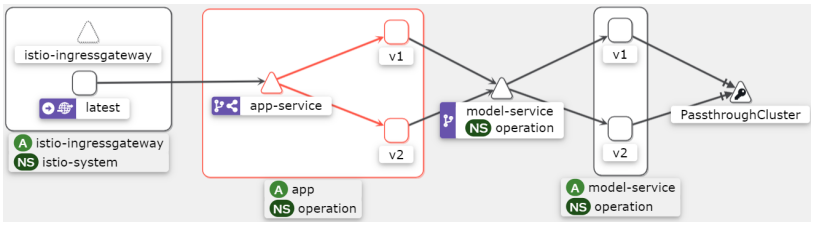

Figure 7: Deployment of operation

3.2 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 a 90/10 probability ratio. 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.

3.3 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 8. 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.

4 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.

4.1 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 9. 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

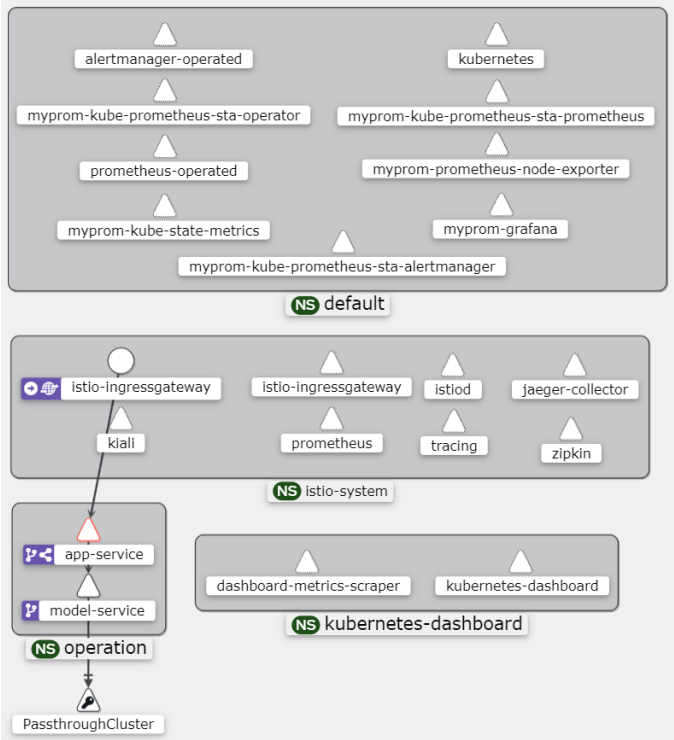

Figure 8: Entire deployment cluster

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.

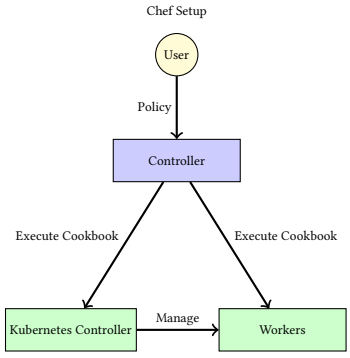

Figure 9: Example Chef Infrastructure

5 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.

6 EXPERIMENTAL SETUP

The experiment is to use the capability of canary releases in order to test 2 different models. The first model was trained for 2 epochs while the other model is trained for 5 epochs. The hypothesis is that the model trained on more epochs has a higher accuracy score. In order to evaluate this the model_accuracy metric is collected. To do this experiment we release two different versions of the model-service image. The v1 model-service deployment is set to serve the 2 epoch model while the v2 model-service deployment serves the 5 epoch model. To evaluate the result Prometheus can be queried for model_accuracy or the Grafana dashboard can be used instead as seen in Figure 10.

7 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.

7.1 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

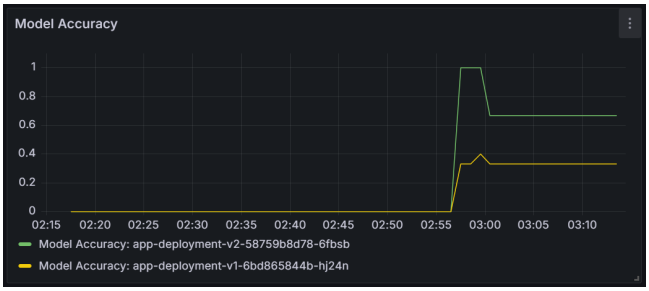

Figure 10: Grafana dashboard for model accuracy

could be created which can serve as checkpoints for the next stage. An overview of the pipeline can be found in Figure 11.

7.2 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.

7.3 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.

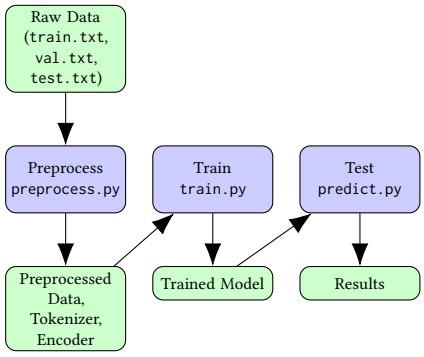

Figure 11: ML Pipeline

8 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.

8.1 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.

8.2 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 13.

Then a test report will be created as shown in Figure 14, it will show up as a comment in each merge request.

Finally, a badge is displayed as shown in Figure 15 on the README with the results, this badge is updated every time we merge to the main branch.

9 PROVISIONING

The deployment of a multi-node Kubernetes environment using Ansible presented unique challenges, primarily due to the selection and configuration of Kubernetes distributions. We have spent too much time on this assignment and sadly without satisfactory results. This time could have been spent on other assignments but we were hopefull that we could fix this. In this section we shortly describe the steps we took for an overview, as we attempted this assignment multiple times.

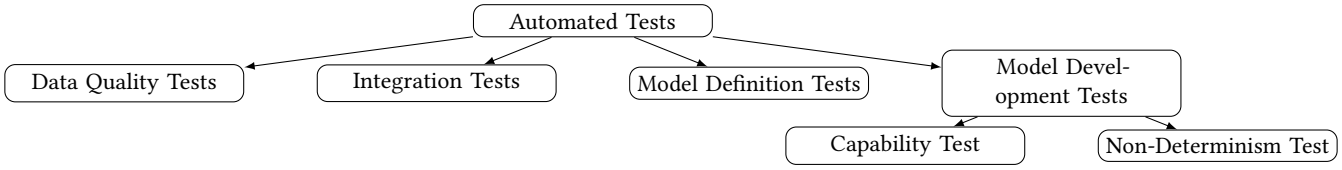

Figure 12: Visual Overview of ML Testing Design

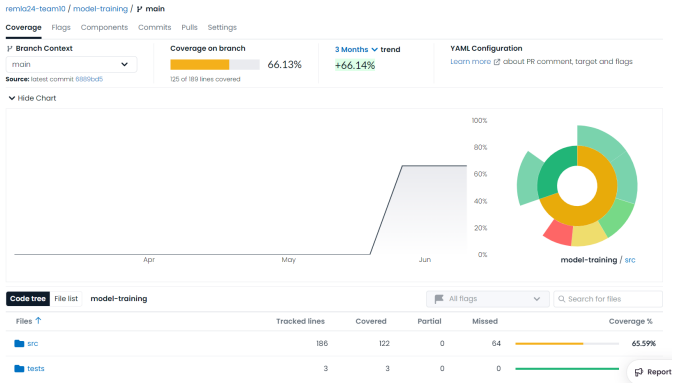

Figure 13: Screenshot of Codecov dashboard of model-training

Figure 14: Screenshot of a test report from the workflow

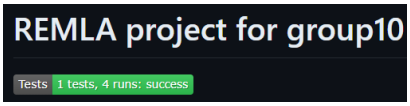

Figure 15: Screenshot of the badge

9.1 Minikube Limitations

Initially, Minikube was selected for its simplicity in setting up a Kubernetes environment. However, it was soon realized that Minikube is designed primarily for single-node setups and was unsuitable for our multi-node requirements.

9.2 Transition to k3d

K3d was considered next for its ease of use and lightweight nature, commonly employed within CI pipelines. Yet, k3d also fell short in fulfilling the multi-node orchestration needed for our project.

9.3 Attempt with k3s

Subsequently, k3s was employed for its lightweight structure and minimal resource demands. Despite successfully setting up the cluster, significant issues were encountered, particularly with making monitoring dashboards like Prometheus and Grafana accessible.

9.4 Persistent Dashboard Accessibility Issues

While k3s facilitated the cluster setup, dashboard accessibility remained a major hurdle. The inability to access critical monitoring interfaces due to networking and configuration errors highlighted the complexities involved in deploying operational Kubernetes environments.

9.5 Conclusion

This experience highlighted the critical importance of aligning Kubernetes distributions with project-specific requirements and the challenges in configuring complex environments with Ansible. These trials provided invaluable insights into the practical aspects of software deployment in real-world multi-node settings.

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.