# Project Report – Group 7

Melle Koper, Alex Ivanov, Arjun Vilakathara, Kevin Tran

## 1 RELEASE PIPELINE

We will use `lib-ml` as the example of a software package release and `model-service` as one of container image release.

We use Poetry for package and dependency management, version tracking, and version bumping. To reference our own Poetry projects, we use version-tagged Git dependencies, leveraging the semantic versioning scheme built into Poetry. Python with Poetry has excellent support for Git dependencies with the added benefit that package names are consistent with the `org/repo` structure. GitHub releases and Git tags are based on the Poetry version rather than allocated; our workflows fail gracefully when attempting to release a version that already exists.

The release and pre-release workflows for `lib-ml` are therefore straightforward. Pushing to the main branch automatically triggers the following release workflow:

(1) Checkout code.
(2) Setup Python.
(3) Install Poetry (only for version management)..
(4) Extract the project version and create a GitHub release and a corresponding Git tag (`v1.2.3`).
(5) If successful, push a patch-level (`1.2.4`) version bump commit on the current branch as a GitHub bot.

Pre-releasing involves one adjustment to this pipeline: we *first* push a prerelease-level (`1.2.3a1`) version bump commit on the current branch, and *then* create a GitHub pre-release and a corresponding Git tag (`v1.2.3a1-<branch>`).

The main consideration given these choices is that developers should always make sure to merge the version from the main branch into theirs before finalizing pull requests. Moreover, if you prerelease your feature branch, you should make sure to pull the automatic version bumps locally afterwards.

Releasing the `model-service` extends the software package release pipeline by introducing three more steps to publish the container image that require information flow:

(1) Checkout code.
(2) Setup Python.
(3) Install Poetry (only for version management).
(4) Extract the project version and create a GitHub release and a corresponding Git tag (`v1.2.3`).
(5) The released version is saved in the workflow step outputs upon successful GitHub release. Read the saved version to configure image metadata such as name, tags, and labels.
(6) Log in to GitHub Container Registry using a job token with permissions to create packages.
(7) Build and push the image given the metadata (obtained from the workflow step outputs). This produces the artifact `ghcr.io/remla24-team7/model-service:1.2.3`.
(8) Finally, push a patch-level version bump commit on the current branch as a GitHub bot.

These are single-purpose release workflows without code tests.

## 2 DEPLOYMENT

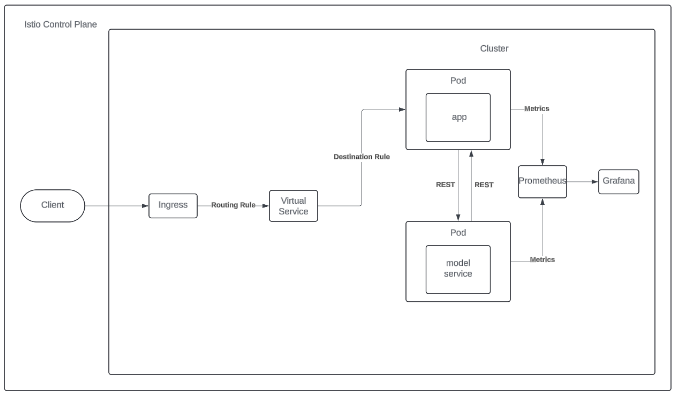


**Figure 1: Deployment diagram**

The diagram above details the deployment. The description of the components is as follows.

### 2.1 User Request Flow:

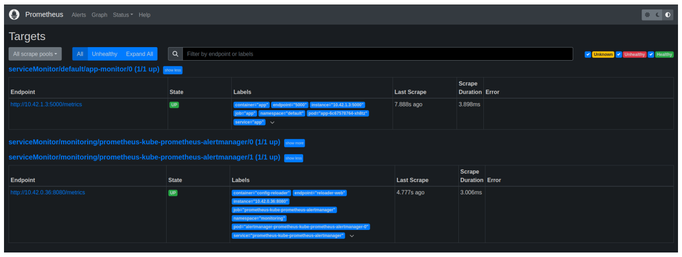
(1) Client Interaction: A user initiates a request from their device, accessing the web application through a web browser.
(2) Ingress Gateway: The request enters the Kubernetes cluster via the Istio-managed Ingress Gateway, which serves as the main entry point for all external traffic.
(3) Routing: The Ingress Gateway forwards the request to the appropriate version of the service based on rules defined in the Virtual Service. The Virtual Service routes the request to the app.
(4) Frontend to Backend: The Frontend Application processes the request and forwards it to the model Service.
(5) Backend to Frontend: The model service uses the pre-trained model to process the request and send back the result to the frontend. Once the Model Service processes the request and returns the predictions, the App presents the results to the user.

### 2.2 Monitoring:

(1) Metrics Collection: Prometheus continuously scrapes the defined metrics from all services, the Prometheus dashboard can be seen in figure 2.
(2) Grafana, connected to Prometheus, uses these metrics to generate real-time visualizations and dashboards that can be used to monitor the application and can be found in figure 3.

## 3 EXTENSION PROPOSAL

Our current deployment process for the app and model-service involves several manual steps, such as modifying the '/etc/hosts'



Targets

serviceMonitor/default/app-monitor/0 (1/1 up)

| Endpoint | State | Labels | Last Scrape | Scrape Duration | Error |
|---|---|---|---|---|---|
| http://10.42.1.3:5000/metrics | UP | container="app" endpoint="5000" instance="10.42.1.3:5000" job="app" namespace="default" pod="app-6c67578764-xh8tz" service="app" | 7.888s ago | 3.898ms | |

serviceMonitor/monitoring/prometheus-kube-prometheus-alertmanager/0 (1/1 up)

serviceMonitor/monitoring/prometheus-kube-prometheus-alertmanager/1 (1/1 up)

| Endpoint | State | Labels | Last Scrape | Scrape Duration | Error |
|---|---|---|---|---|---|
| http://10.42.0.36:8080/metrics | UP | container="config-reloader" endpoint="reloader-web" instance="10.42.0.36:8080" job="prometheus-kube-prometheus-alertmanager" namespace="monitoring" pod="alertmanager-prometheus-kube-prometheus-alertmanager-0" service="prometheus-kube-prometheus-alertmanager" | 4.777s ago | 3.006ms | |

**Figure 2: The targets scraped by Prometheus can be found in the Prometheus Dashboard. The service monitor 'app-monitor' is configured to scrape our ML app every thirty seconds**

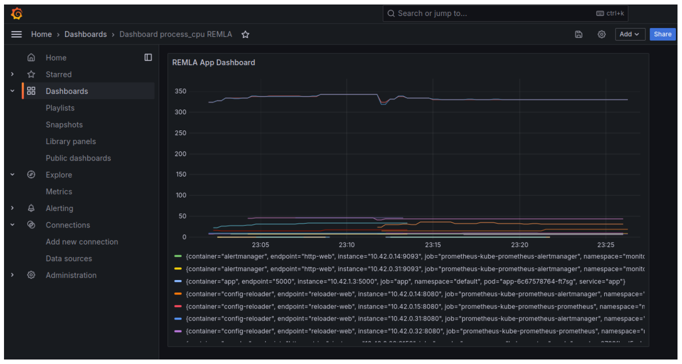

Search or jump to...
ctrl+k
Home › Dashboards › Dashboard process_cpu REMLA
Add
Share
Home
Starred
Dashboards
Playlists
Snapshots
Library panels
Public dashboards
Explore
Metrics
Alerting
Connections
Add new connection
Data sources
Administration
REMLA App Dashboard
350
300
250
200
150
100
50
0
23:05
23:10
23:15
23:20
23:25
{container="alertmanager", endpoint="http-web", instance="10.42.0.14:9093", job="prometheus-kube-prometheus-alertmanager", namespace="monit
{container="alertmanager", endpoint="http-web", instance="10.42.0.31:9093", job="prometheus-kube-prometheus-alertmanager", namespace="monit
{container="app", endpoint="5000", instance="10.42.1.3:5000", job="app", namespace="default", pod="app-6c67578764-ft7sg", service="app"}
{container="config-reloader", endpoint="reloader-web", instance="10.42.0.14:8080", job="prometheus-kube-prometheus-alertmanager", namespace="
{container="config-reloader", endpoint="reloader-web", instance="10.42.0.15:8080", job="prometheus-kube-prometheus-prometheus", namespace="n
{container="config-reloader", endpoint="reloader-web", instance="10.42.0.31:8080", job="prometheus-kube-prometheus-alertmanager", namespace="
{container="config-reloader", endpoint="reloader-web", instance="10.42.0.32:8080", job="prometheus-kube-prometheus-prometheus", namespace="

**Figure 3: The relevant metrics for the app are visualized in the Grafana dashboard**

file and manually ensuring the model.h5 file is placed in the correct repository. Also, to deploy our app, model-service and Service Monitors, multiple Kubernetes configurations need to be applied with 'kubectl'. This method is prone to human error and goes against the idea of automated release engineering. It might also lead to inconsistencies between different environments due to unnecessary complexity of manual commands. To improve on the deployment, in the future we could make use of Helm charts for complete deployments that can be installed multiple times into the same cluster. The same Helm chat can be reused across different projects, which improves reusability of the application. The 'How to deploy application on Kubernetes with Helm' by [3] provides a clear instructions on how to refactor our deployment from a multitude of yaml files into a single Helm chart.

An additional useful plugin for Helm that can be used for secret management is 'helm-secrets'. The helm-secrets plugin stores the encrypted secrets in Helm charts such that any sensitive data is kept secret. In our case this would be mostly relevant for the admin authentication bearer token for the Kubernetes Dashboard as well as the admin password for Grafana. The following article 'How to Handle Secrets in Helm' [2] gives a clear step-by-step process on handling secrets in Helm charts and can be relevant for our refactoring.

To test the effectiveness of the Helm chart deployment, we will:

(1) Deploy in a Staging Environment: Initially deploy the application using Helm in a staging environment such as a separate Minikube cluster with the same configuration and resources to verify the deployment.
(2) Automated Tests: Run automated tests to ensure that all components are correctly deployed and functional, these tests should include smoke tests, unit tests and integration tests as mentioned by Ellingwood as well as ini the lecture [1] and perhaps system testing in case we want to deploy multiple applications and clusters.
    Smoke tests include verifying that all critical resources are running and that all expected resources are available (e.g. pods, services, service monitors, endpoints) are all running and healthy. Integration tests includes hooking up multiple components and ensuring they can communicate and work as intended, such as making sure that the control-plane can communicate with all worker nodes and that the bridging has been set up correctly.

## 4 ADDITIONAL USE CASE (ISTIO)

## 5 EXPERIMENTAL SETUP (ISTIO)

## 6 ML PIPELINE

The ML pipeline is located in our `model-training` repository, where we demonstrate a continuous training GitHub workflow using DVC – though we never tried training the model in an action, reproducing the stages with the lock file completes successfully.

Our DVC pipeline consists of three main stages and two additional stages for mutamorphic testing:

(1) Dataset preprocessing.
(2) Training the model.
(3) Evaluating the model on test data.
(4) Generating a mutamorphic version of the training data.
(5) Evaluating the model on the mutamorphic training data.

The code from the Jupyter notebook has been refactored into independent scripts and their configurables extracted into `params.yml`, which is read by each script via the DVC `params` API. The preprocessing stage depends the existence of the dataset folder, which is managed by DVC and therefore `dvc repro --pull` downloads the dataset as *stage 0* as needed. Each stage depends on a corresponding section of the parameter configuration, thus reproducing the pipeline also saves the parameter values to the lock file.

(...) The last stage of the GitHub workflow runs our testing framework via `pytest`, which we elaborate on next.

## 7 ML TESTING DESIGN

In this section, we will discuss the testing approach taken to validate the machine learning aspect of our application. First we will cover the testing strategy. Then we will elaborate on the tests implemented to execute the testing strategy. After which, we will elaborate on the testing results of our pipeline. Finally, we cover the limitations in our strategy, the future opportunities to improve the testing coverage, and the lessons learned from our implementation.

## 7.1 Testing Strategy

In order to ensure the consistency, robustness, and accuracy of our ML application, we have created the follow testing strategy:

#### 7.1.1 *Testing Framework.*

The main testing framework consists of various automated tests, the advantages of which are discussed in 7.1.2. When a change is pushed to the main branch of the Model-Training branch of the application repository, a testing pipeline is triggered. First the workflow builds the project, installs all necessary dependencies, and pulls all necessary data from the DVC Google Drive. Once all this is complete, the workflow executes all automated tests implemented in the model-training repository.

TODO: ADD DIAGRAM

#### 7.1.2 *Automated Tests.*

The goal of our testing strategy is to ensure the continuous performance and health of our Machine Learning Application we have implemented Automated tests. Automated testing brings with it various significant advantages. They ensure the reliability and robustness of the model by consistently verifying that the application and its services function as expected, this is also done through the development life cycle. The continuous validation helps to identify and address any potential issues or bugs early when introducing changes to the code base, which in turn reduces the risk of deploying a faulty model. In addition, automated tests enhance the reproducibility of any experiments run in the future, as they can run the same tests under different conditions, while ensuring the results are consistent. Finally, automated tests are more efficient when compared to manual testing, and will allow developers working on the project to commit more resources to further develop the application rather than spending resources verifying past implementations.

#### 7.1.3 *Continuous Training.*

Through Github Actions we have implemented continuous training. This is crucial in any machine learning application to maintain accuracy, relevance and performance over time. As a domain changes and evolves, models tend to drift. This means the models predictive power becomes less due to changes in underlying patterns. This is especially relevant in detecting phishing links, as bad actors will continuously change their approach to remain undetected. With continuous training, it is necessary to test the newly trained model to ensure it performs in the appropriate manner.

## 7.2 Implemented Tests

To execute our testing strategy, we implemented various types of tests, these test are meant to evaluate the following:

(1) The models capabilities
(2) The features and the Data
(3) The model development
(4) The ML infrastructure
(5) Monitoring

#### 7.2.1 *Testing Model Capabilities.*

To test the capabilities of the model, we evaluate it on various aspects such as accuracy and robustness. To test the performance of the model, we evaluate the model on test data, which the model should not have seen during it's training phase. The testing pipeline returns a classification report generated by sklearn, as well as a confusion matrix, roc curve, and roc auc score.

We test the robustness of the model through mutamorphic testing. Our implimentation of mutamorphic testing creates a new data set from the original test set.

This "data creation" works as follows, from each entry in the original set we generate various mutants. These mutants are then vetted through a sanity check, each sample is checked to see if it remains a structurally valid url. Then all remaining mutants are converted to machine code through the use of the tokenizer. These converted urls are compared to the tokenized original sample, and only the most similar mutant is kept. Similarity is measured through cosine similarity.

Once the mutamorphic data set is created, we evaluate the model with this new data in the same way as the original test set, and we expect the model to perform the same.

TODO: add diagram for mutamorphic testing

#### 7.2.2 *Testing Features and Data.*

The validation of the data is also essential for the proper training and development of the model. Our pipeline confirms the validity of the data through two relatively simple tests. We ensure that each sample is properly labelled through checking whether the labels are legitimate. We then check to see that all entries in the dataset is non-empty.

#### 7.2.3 *Testing Model Development.*

To validate the proper development of the model, we have implemented the following tests:

There is a test verifying the proper loading of the train and test datasets. We then test whether the training of the model is properly executed. This is tested through training the model and analyzing whether the training of the model produced a loss, and whether the training process produced any accuracy metrics.

#### 7.2.4 *Testing ML Infrastructure.*

The current implemented test of the ML infrastructure is testing whether or not the model can be properly built and initialized. This is done through loading the model from the DVC version control and building it.

#### 7.2.5 *Monitoring Tests.*

Still need to be implemented

## 7.3 Testing Results

TODO: Include results

## 7.4 Limitations, Opportunities, and Lessons Learned

TODO: Include Limitations, opportuniteis and lessons learned

## REFERENCES

[1] Justin Ellingwood. 2022. An introduction to continuous integration, delivery, and deployment. https://www.digitalocean.com/community/tutorials/an-introduction-to-continuous-integration-delivery-and-deployment

[2] Guest Expert. 2024. How to handle secrets in helm. https://blog.gitguardian.com/how-to-handle-secrets-in-helm/

[3] Wojciech Krzywiec. 2020. How to deploy application on Kubernetes with helm. https://wkrzywiec.medium.com/how-to-deploy-application-on-kubernetes-with-helm-39f545ad33b8