Project Report – Group 13

Maarten van Bijsterveldt [5295505]
Vanessa Timmer [4728033]
Nick Dubbeldam [5647703]

1 RELEASE PIPELINE DOCUMENTATION

1.1 lib-version Library

The lib-version library is the version-aware library. A version of the package is released to PyPI everytime a tag of the format v[#major].[#minor].[#patch] is pushed to the repository. To keep track of its version, it has a VERSION file in REMLA_Test_Lib_version, which can be polled through the library functions. The workflow, which can be seen in Figure 1, is ran with the publishing URL in the environment and goes through the following steps:

Checkout code First, as is standard in workflows, the repository contents get fetched using the checkout action.

Update version file Then, the version file is updated. This is done by retrieving the current tag using a GitHub environment variable. As the tag is also the version number, this gets stored in the aforementioned VERSION file.

Create distribution After this, a distribution is created by running setup.py using sdist. The setup.py file uses the contents of the VERSION file edited in the previous step to publish with the proper version.

Publish top PyPI Lastly, the distribution is published to PyPI. This is done with the gh-action-pypi-publish release action by pypa (Python Packaging Authority).

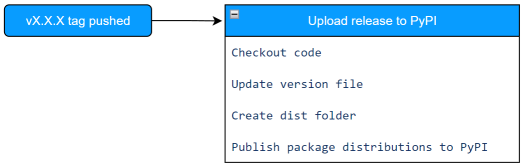

Figure 1: The release workflow of lib-version

1.2 model-service Image

The model-service image is responsible for handling requests to the model. It does not store the model in the image; instead, it is downloaded when the model is ran. A version of the image is released to the GitHub Container Registry (GHCR) everytime a tag with the format v[#major].[#minor].[#patch] is pushed to the repository. The workflow, as seen in Figure 2, goes through the following steps:

Parse version from tag First, after fetching the repository contents using the checkout action, the version is parsed. This is done by using the GITHUB_REF environment variable, which contains the tag in this case, as that is what triggered the workflow. Because the actual value given by GITHUB_REF is refs/tags/vX.X.X, the first 11 characters need to be ignored to end up with just the numbers. Then, the major, minor, and patch numbers are extracted by using the "cut" command with "." as the delimiter, and then retrieving the first, second, and third field respectively. These values are then stored for later by storing a new variable in the GitHub environment.

Login to GHCR The workflow then needs to login to the GHCR. It does this by logging in to ghcr.io through docker with the user that triggered the workflow as username and the automatically generated GitHub Actions token as password.

Convert repository name to lowercase Because of GHCR restrictions, the name of the image needs to be in lower case. Thus, we retrieve the repository name through the *github* context, convert it to lowercase, and store it in another environment variable.

Build, tag, and push image Finally, the releasing contains a couple of steps as well. First, the path of the image in GHCR is set using the lowercase repository name stored before. Then, the image is built, using all version tags: the full version, x.x.latest, x.latest, and latest. Finally, the image is pushed to the image path specified before.

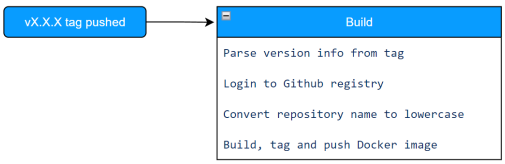

Figure 2: The release workflow of model-service

2 DEPLOYMENT DOCUMENTATION

The code consists of 6 repositories. These are split up into a frontend, backend, a model-service and ellipsis as seen in Figure 3 The frontend, backend and the model service connect to each other over the REST protocol. The model gets downloaded from ellipsis and the model training gets done once locally and then uploaded to ellipsis. These repositories can be executed via the operation repository. From here there are 3 different ways to launch all the software.

2.1 Docker compose

The first option is to use Docker Compose. When using the command "compose up" in the operation folder, it will deploy all the docker containers needed to run the program. these are:

- serve-model
- backend

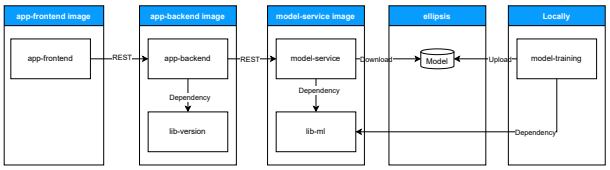

Figure 3: the deployment overview

- frontend

As seen in Figure 3.

2.2 Vagrant

Vagrant is a good option for creating and managing portable development environments. It can be run by using the command: "vagrant up" in the operation folder.

2.3 Kubernetes

For an easier upscaling of the program, we can use Kubernetes. This runs the same software, but split up in different notes. It can be run by using the command: "kubectl apply -f kubernetes.yml" in the operation folder.

2.4 Usage

To use the program, go to this link: localhost:5000 and use the website to check if the link is a spam link.

3 EXTENSION PROPOSAL

-

4 ADDITIONAL USE CASE

Our additional use case involves rate limiting for our application. Rate limiting prevents our server from being overloaded by excessive requests. By setting a limit on the number of requests per limit that can be made to the application, we ensure that any requests exceeding this limit are denied. Without rate limiting, our server could become overwhelmed by a large volume of requests. This could lead to degraded performance or downtime, which negatively affect user experience. Rate limiting allows us to maintain the stability and performance of our application, especially under high traffic conditions, and provides protection against malicious actors trying to launch a DDoS attack.

To implement the additional use case, we introduced a new YAML file that defines the necessary rules, leveraging Istio [1] and Envoy [2] to control the number of requests to the application. More specifically, we implemented global rate limiting, which applies the limit to the entire service mesh rather than the individual services. The YAML file consists of these components in the given order:

(1) A *ConfigMap* to put a ceiling to the number of requests per minute
(2) A global rate limit service which implements Envoy's rate limit service protocol
(3) An *EnvoyFilter* applied to the *ingressgateway* to enable global rate limiting with Envoy's global rate limit filter
(4) Another *EnvoyFilter* applied to the *ingressgateway* to define the route configuration on which rate limiting should be applied

5 EXPERIMENTAL SETUP

In this experiment, we aim to evaluate the impact of different frontend designs on user interaction. We deploy two versions of the frontend: the existing design (Design A) and a new, alternative design (Design B). By directing a portion of users to each version, we can measure and compare user interaction between the two designs. This setup allows us to gain insights into user preferences and behavior.

Our base design is the current frontend (Design A) that we deploy as per Section 2. To implement the experiment, we introduce Design B and utilize Istio's [1] traffic routing. In this setup, 90% of the users will continue to see the base design (Design A), while the remaining 10% will see Design B. Apart from the frontend changes, all other aspects of the deployment, including backend services, remain the same.

The hypothesis that we test with our experiment is that Design B will result in higher user interaction compared to Design A. We expect that the changes introduced in Design B will lead to an increase in user interaction. To assess the impact of the frontend changes, we will measure the number of incoming requests **(TBD: add more metrics)**. We achieve this by using Prometheus [8] for monitoring and collecting metrics. By analysing the number of requests each design receives, we can quantify the differences in user engagement between the two designs. The number of requests for each frontend version will be visualized using a Grafana dashboard [3] to provide a clear representation of the metrics and support the decison making process.

The decision criteria for accepting or rejecting the experiment will be based on predefined thresholds for user interaction metrics. For instance, if Design B shows at least a 10% increase in the number of requests compared to Design A, we will accept the hypothesis. Conversely, if the increase is marginal or nonexistent, we will keep the existing design and either continue refining design B or discard it.

6 ML PIPELINE

The biggest pipeline in the project is DVC, it creates a new model. The second one is creating an image, as discussed in section 1. For creating an image we need all the correct dependencies. This gets done with Poetry. The third pipeline is pylint which enforce a coding standard.

6.1 DVC Repro

DVC is a tool for data science that takes advantage of existing software engineering toolset. It helps machine learning teams manage large datasets, make projects reproducible, and collaborate better. [9] In this project it gets used to do the steps shown in Figure 4 and track the versions. These steps are:

Dataset retrieval The first step is to get the data to train the model. The data gets downloaded from the Kaggle website. This is automated by using the Kaggle API.

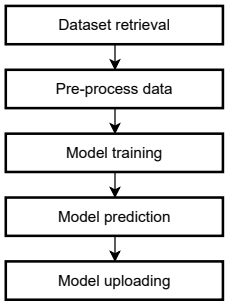

Figure 4: DVC pipeline steps

Preprocessing data The second step is to preprocess the data. This is done by using the lib-ml library.

Model training The third step is to create the model and train it based on the dataset.

Model prediction The fourth step is to test the model on a never-seen part of the dataset to verify that the model

Model uploading The last step is to upload the new model so it can be used by all the clients.

6.2 Poetry

Poetry is a tool for Python that simplifies the process of creating, managing, and publishing Python projects. [5] The model-training package has a poetry.lock to easily update all the dependencies.

6.3 Pylint

Pylint is a tool that checks for errors in Python code, tries to enforce a coding standard and looks for bad code smells. [7] The source code gets checked at every push to the repository by a GitHub Action. As a result, it will show what is wrong. This can also be checked locally before uploading.

7 ML TESTING DESIGN

In our project, we implemented a comprehensive testing strategy using pytest [6] to ensure the robustness and reliability of our ML system. Our strategy consisted of implementing at least one test for each category outlined by Breck et al. [4]. This section details the various tests we conducted and mentions future improvements.

7.1 Tests for Model Development

In order to verify the dependability of the ML system, we deviced the following tests:

Test for Nondeterminism Robustness We want the model's performance to remain consistent despite variations in the random seed during training. We test this by training two models with identical configurations except for the random seed value and comparing their prediction accuracies. The model accuracies should not differ by more than 2.5%.

Test on Data Slice Given the high stakes of phishing detection (e.g. financial loss or theft of personal information) it is crucial that the model accurately identifies phishing links, with false negatives being particularly important to minimize. We therefore chose to evaluate performance of the model on phishing links by taking a selection of illegitimate URLs from the complete test dataset as a data slice. We then measure the model's performance on this slice, requiring the model to correctly identify at least 80% of the URLs.

Code Review for Model Specifications Each model specification undergoes a code review by at least one additional team member. This helps identify potential bugs or mistakes that might have been overlooked by the original author. The code review is facilitated by pushing the code to a repository each team member has access to.

7.2 Tests for ML Infrastructure

We also tested various components of the ML pipeline to ensure they function correctly:

Test Loss Decrease This test seeks to confirm that the model's loss decreases during training. We compare the loss after the first epoch with the loss after the final epoch. The expectation is that the final loss should be lower, indicating that the model has learned effectively.

Deploying New Model with a Canary Process Istio enables us to deploy a new model to 10% of users to monitor its performance and compatibility with the existing serving system. This gradual rollout helps identify any peformance-breaking changes before a full deployment, allowing us to safely deploy new models without disrupting the user experience.

7.3 Monitoring Tests for ML

Our model serves predictions on demand, it is therefore crucial to ensure the accuracy and validity of these predictions.

Test Validity of Prediction Values The prediction values should fall within a valid range, otherwise we cannot guarantee that the model produces sensible outputs. We test this by verifying that the model's predictions are between 0 and 1 and checking for invalid values such as NaN or infinity.

7.4 Tests for Features and Data

The data that a model learns during training determines the behaviour of the ML system, which is why it is important to make sure that the input data has the structure we expect it to have.

Test URL Processing Our system takes URLs as input, we should thus ensure that these are correctly preprocessed otherwise the model might not behave as intended. After preprocessing, each URL should be represented by a dictionary of tokens, with the tokens padded to a length of 200.

7.5 Limitations and Future Improvements

While our current tests cover many aspects of the ML system, there are areas for improvement. We currently do not test for model

staleness, so we might use an outdated model without being aware of it. As we continue to gather new data, such as new phishing techniques, it is important to periodically retrain the model. A valuable addition to our test cases would be to implement tests that verify the prediction accuracy of the retrained model and measure its impact on user interaction with the service. These tests would help maintain or improve the model's performance over time.

REFERENCES

[1] [n.d.]. https://istio.io/latest/
[2] [n.d.]. https://www.envoyproxy.io/docs/envoy/latest/
[3] [n.d.]. https://grafana.com/
[4] Eric Breck, Shanqing Cai, Eric Nielsen, Michael Salib, and D Sculley. 2016. What's your ML test score? A rubric for ML production systems. (2016).
[5] Sébastien Eustace. 2024. Poetry. https://python-poetry.org/. Accessed: 2024-06-10.
[6] Holger Krekel, Bruno Oliveira, Ronny Pfannschmidt, Floris Bruynooghe, Brianna Laugher, and Florian Bruhin. 2004. pytest. https://github.com/pytest-dev/pytest
[7] Logilab. 2024. Pylint. https://pylint.org/. Accessed: 2024-06-10.
[8] Bjorn Rabenstein and Julius Volz. 2015. Prometheus: A Next-Generation Monitoring System (Talk). USENIX Association, Dublin.
[9] DVC Team. 2024. DVC Documentation. https://dvc.org/doc