REMLA Project Report – Group 14

Dani Rogmans, Justin Luu, Nadine Kuo, Yang Li
Mon 10 June 2024

All: Please refer back to the assignments, lecture slides etc. when writing and include relevant resources

Also see the rubrics under A5 and A6 - note the report is 30% – page limit: 10 excl. references

Use texttt for code-style and refer to repo files if needed

In this report, we document the overall setup and design decisions made for our URL Fishing project (TU Delft course Release Engineering for ML Applications, CS4295), for which a high-level architecture is visualized in Figure 1. As can be seen, this project is composed of several independent components which have their dedicated repositories.

The `model-service`[1] represents a wrapper service for the release ML model and offers a REST API to expose the trained model to other components. Both our `model-service` and `model-training`[2] artifacts depend on our `lib-ml` library[3] through PyPi.

Moreover, our `app`[4] contains logic for querying the `model-service` through REST requests and allowing users to enter URL input for phishing detection. It depends on our version-aware `lib-version`[5] library, allowing for querying and displaying its version.

Both the `app` and `model-service` are released as two separate container images with their respective dependencies, meaning they can be deployed separately from each other.

1 RELEASE PIPELINE DOCUMENTATION

The release workflows for our various components ensure a streamlined continuous delivery and deployment process using GitHub Actions as build server. This section focuses on our `app` and `lib-ml` components, thereby outlining each step in the pipeline, detailing its purpose, implementation, and the flow of data and artifacts through the process.

1.1 `App`: Container Image Release Pipeline

The main purpose of this release workflow[6] is to ensure that every change meeting the release criteria is automatically built, tagged, and made available as Docker images stored in GitHub's Container Registry (GHCR). This automation allows us to streamline the continuous delivery process and maintaining a high standard of version control and traceability.

It is triggered by pull requests to the main branch or pushes of tags matching `v*`, ensuring that releases are systematically handled through versioning and tagging. We support semantic versioning in the format `MAJOR.MINOR.PATCH`, allowing for unique and meaningful version numbers.

The workflow begins with the *"Checkout code"* step, which retrieves the latest code from the repository. Using the `actions/checkout@v2`-action, the codebase from the specified branch or tag is cloned into the runner environment, providing the source code required for subsequent steps.

Next, the *"Extract version"* step determines the release version from the tag. The version adheres to our semantic versioning strategy, ensuring consistent and meaningful version numbers.

Following this, the *"Log into GitHub Container Registry"* step authenticates with GitHub Container Registry (GHCR) to facilitate the pushing of Docker images. The Docker login command is used with credentials from GitHub secrets, ensuring secure handling of authentication credentials.

The *"Build Docker image"* step involves building the Docker image for the application, based on the `Dockerfile`. This image forms the core artifact of our release process.

In the *"Tag Docker image with version"* step, the built Docker image is tagged with both the extracted version and *"latest"* tags. These ensure the image is properly versioned and easily identifiable.

Subsequently, the *"Push Docker image to GitHub Container Registry"* step uploads the tagged Docker images to GHCR, under the repository `ghcr.io/remla2024-team14/app`, making them available for distribution.

Finally, the *"Trigger release"* step creates a release in GitHub based on the new version tag. The `actions/create-release@v1` action is utilized to create a release, with the GitHub token used for authentication. The release includes the version tag, release name, and a descriptive body, making it available under the GitHub repository's *"Releases"* section.

1.2 `Lib-ml`: Software Package Release Pipeline

This release workflow[7] automates the process of building distribution packages and uploading our `lib-ml` library (containing pre-processing logic for training our ML model e.g.) to PyPI. The pipeline is triggered by pushes to the main branch or tags that match the pattern `v*`, which again supports semantic versioning.

The process begins with the *"Checkout code"* step, similarly to the previous workflow discussed. Next, the *"Set up Python"* step sets up the Python environment. By using the `actions/setup-python@v2`-action, Python 3.8 is installed and configured on the runner. This ensures that the environment is prepared for running Python scripts and installing dependencies.

Following this, the *"Install dependencies"* step installs the required dependencies for building and publishing the package. The `python -m pip install –upgrade pip` command ensures that pip is updated to the latest version, and `pip install setuptools wheel twine` installs the necessary tools for packaging and distributing the software.

The *"Build Distribution"* step creates the distribution packages for the Python project. The command `python setup.py sdist`

[1] https://github.com/remla2024-team14/model-service
[2] https://github.com/remla2024-team14/model-training
[3] https://github.com/remla2024-team14/lib-ml
[4] https://github.com/remla2024-team14/app
[5] https://github.com/remla2024-team14/lib-version
[6] https://github.com/remla2024-team14/app/blob/main/.github/workflows/release.yml
[7] https://github.com/remla2024-team14/lib-ml/blob/main/.github/workflows/release.yml

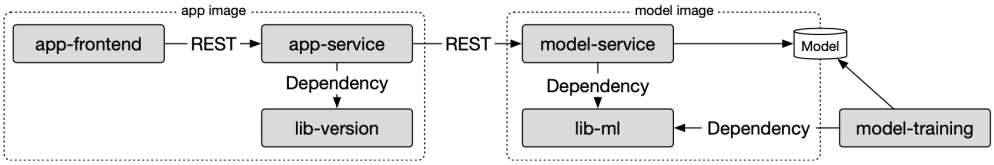

Figure 1: High-level project architecture (source: REMLA Assignment 2)

`bdist_wheel` generates both source distribution (`sdist`) and built distribution (`bdist_wheel`) packages. These packages are the primary artifacts of our release process.

Subsequently, the *"Publish to PyPI"* step uploads the built distribution packages to the Python Package Index (PyPI). The `twine upload dist/*` command uploads all files in the `dist` directory to PyPI, using authentication credentials stored in GitHub secrets.

2 K8S DEPLOYMENT DOCUMENTATION

In this section, we detail the final deployment structure for our URL fishing application (see Figure 2), designed to run and expose a web service that interacts with a model service, ensuring a streamlined flow of data and interactions. Initially in this project we had a `docker-compose` setup[8], which was mere a simple setup to illustrate that app can run in a containerized form independent from host OS. Here, however, we will elaborate on our Kubernetes (K8s) deployment[9], which increases the decoupling further. In fact, we automatically provision a functional K8s cluster on dedicated VMs using Vagrant and Ansible. Below is a detailed explanation of each resource and its role within the K8s cluster.

2.1 Ingress and Services

The data flow begins when incoming HTTP requests from users enter the cluster through the *Ingress* resource, which routes these requests to the `fishing-web-serv` Service.

Our *Services* are introduced to prevent internal *Pods* from being accessed publicly. We have one for both `model-service` and the web application `fishing-web-`. Note that in Figure 2, our `model-service` is left out for simplicity, but it follows a similar structure (except for the fact that the *Ingress* plays no role).

Our `fishing-web-serv` Service (ClusterIP) listens on a specified port and forwards requests to the corresponding port on the `fishing-web` container within the *Deployment Pods*.

If the web service needs to call the model service, it uses the `MODEL_HOST` environment variable, which points to the (configurable) URL of `model-service`. To this end, we use a `ConfigMap` to store configuration data required by the web service.

The `fishing-model-serv` Service (ClusterIP) facilitates this communication by directing the requests to the `fishing-model-depl` deployment, where the `fishing-model` containers process the model service requests and return the results to the web container.

2.2 Deployments and Pods

Deployments are controller-like resources that allow us to ensure that pod(s) remain alive, apply or revert updates or use to scale up and down replicas of Pods. For instance, in case our URL fishing web app is actually getting popular, but there is not feasible to handle the increasing load with only one *Service*, we could add more in single command:
`kubectl scale deployment fishing-web-depl –replicas=10`.

Our `fishing-web-depl` *Deployment* is responsible for managing the web service by maintaining a set of replicas (as specified in our `YML` configuration) of the `fishing-web` container. We use `imagePullSecrets` as authorization token that stores Docker credentials used for accessing images in GHCR. A similar setup is adopted for our `fishing-model-depl` *Deployment*.

2.3 Helm Charts

In our K8s deployment, the additional Helm charts[10] streamline the process by encapsulating our configurations for the `fishing-model` and `fishing-web` services, along with their respective deployments and secrets. This ensures that all resources are versioned, templatized, and easily configurable, enabling smooth deployments and rollbacks - as these charts can be installed more than once.

3 EXTENSION PROPOSAL: ROLLBACK STRATEGY

We believe that the current troubled point is that the current project lacks an effective rollback option. This means that when deploying a new version, if a problem or error occurs, there is no way to quickly revert to the previous stable version. Consequently, when a problem is discovered after deploying a new version, the lack of a rollback mechanism significantly increases the time to fix it, leading to longer service outages.

The proposed project refactoring and extension aim to address this issue by improving the repository structure and incorporating automated rollback strategies. The visualization below outlines the new repository structure and introduces scripts and configuration files designed to streamline deployment and rollback processes.

[8] https://github.com/remla2024-team14/operation/blob/main/docker-compose.yml
[9] https://github.com/remla2024-team14/operation/blob/main/kubernetes/k8s-deployment.yml
[10] https://github.com/remla2024-team14/operation/tree/main/kubernetes/urlfishing_chart

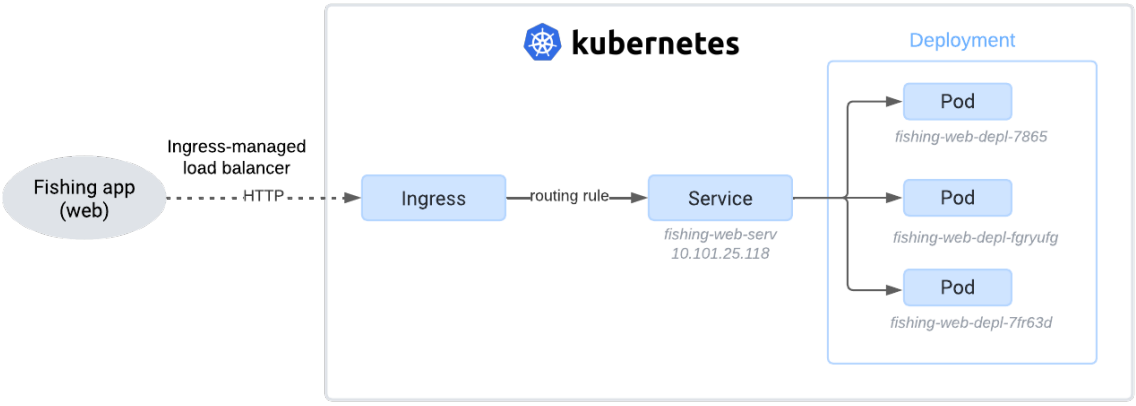

Figure 2: Kubernetes Deployment Overview

3.0.1 Enhancing Project Structure through Refactoring and Extension.

Current Structure.

```
remla2024-team14
 app
    src
    tests
    Dockerfile
 model-service
    src
    tests
    Dockerfile
 model-training
    src
    tests
    Dockerfile
 operation
    src
    tests
    Dockerfile
 lib-ml
     src
     tests
     Dockerfile
```

Improved Structure.

```
remla2024-team14
 backend
    model-service
       src
       tests
       Dockerfile
    rollback-scripts
         rollback.sh
           deploy.sh
    model-training
       src
       tests
       Dockerfile
    lib-ml
        src
        tests
        Dockerfile
 frontend-app
    src
    tests
    Dockerfile
 docs
    architecture.md
    api-docs.md
    user-guide.md
 tests
     unit
     integration
     e2e
```

The rollback scripts are integral to our proposed solutions in Section 3.2. They automate the process of reverting to a previous version, significantly reducing the time needed to respond to deployment failures.

3.1 Main Benefit of Rollback Extension

3.1.1 Improved System Stability. The rollback mechanism is an important tool for ensuring application stability. After deploying a new version, if problems or errors are found, it is possible to quickly revert to the previous stable version, reducing service interruption time and ensuring system stability and reliability

3.1.2 Reduced deployment risk. Every time a new version is deployed there is a certain amount of risk. Rollback mechanisms can be used as a last resort protection measure to quickly restore services

3.2 Rollback Strategy Suggestions

3.2.1 Configuring Automated Rollback with CI/CD Tools. Automated rollback is key to ensuring that you can quickly revert to a previous stable version in the event of a deployment failure.

Reference: [11]

3.2.2 Blue-Green Deployment. Blue-Green Deployment is a seamless rollback strategy that retains the current version (blue) and the new version (green) of the environment and switches traffic to the new version when it passes validation. If there is a problem with the new version, you can immediately switch back to the old version.

Reference: [12]

3.3 Evaluation of Extension

3.3.1 Testing of the automated Rollback with CI/CD Tools. First, set up a test environment that closely resembles the production environment, ensuring that CI/CD tools (e.g., GitHub Actions) are properly integrated. This step is crucial for simulating real-world conditions and validating the rollback mechanism accurately.

Next, deploy a new version in the test environment through the CI/CD pipeline. Introduce deliberate bugs in this version to ensure that it triggers a rollback condition. This helps in testing the effectiveness of the automated rollback mechanism under controlled failure scenarios.

[11] https://docs.github.com/en/actions
[12] https://adamtheautomator.com/kubernetes-blue-green

During the deployment process, closely monitor the progress to confirm that the automatic rollback mechanism is activated correctly if the new version fails to deploy. The system should revert to the last stable version without manual intervention.

After the rollback is triggered, validate the results by checking the rollback logs and system status. Ensure that the rollback was successful and no additional issues have arisen. Automated test scripts can be employed to verify that the system functionality has returned to normal.

Finally, use monitoring tools such as Prometheus and Grafana to analyze metrics during the deployment and rollback process. This helps in assessing whether performance and availability were significantly impacted during the rollback, ensuring that the system remains stable and reliable.

3.3.2 Testing for Blue-Green Deployments. Setting up the Blue-Green environment involves configuring the blue-green deployment policy in the test environment to ensure that both blue (current stable release) and green (new release) environments exist. Once the environments are configured, the next step is deploying a new version to the green environment. After deployment, perform basic functional tests to ensure that the new version operates normally in the isolated environment.

With the new version verified in the green environment, traffic switching can commence. Start by redirecting a portion of user traffic to the green environment and monitor the performance and any errors associated with the new version.

If any issues are detected in the green environment, promptly switch the traffic back to the blue environment to ensure user impact is minimized. Investigate and resolve the problems identified in the green environment.

If the green environment passes all tests without issues, proceed with a full release by switching all user traffic to the green environment. Continue monitoring for a period to ensure the new release remains stable.

Finally, verify the results by checking the deployment logs and system status for both blue and green environments. Ensure the traffic switchover and any necessary rollback processes occurred smoothly, maintaining system stability and a positive user experience.

4 ADDITIONAL USE CASE (ISTIO)

In *Istio*, the expansion function of *Egress* is used to manage and control the traffic of external services in the management and control service grid. *Egress* allows the access strategy of external services, including but not limited to the aspects covered below.

4.1 Description of Egress extension

In *Istio*, the *Egress* extension is used to manage and control traffic to external services in the service grid. The *Egress* configuration allows you to specify which external services can be accessed and how they can be accessed, including the URLs to access them, the protocols and ports used, etc. This allows us to effectively manage and control the traffic to external services in the service grid. This allows us to effectively control access to external services.

Secondly, the *Egress* configuration can force all access to external services through security channels (such as TLS) and support identity certification for external services, thereby enhancing security and certification mechanisms.

In addition, as we are implemented in the project now, *Egress* can also monitor external traffic, collect traffic data, and manage the flow limit, load balancing and other management operations to ensure the stability and performance of the service. These functions allow *Istio* to provide comprehensive external traffic management and control capabilities, and improve the security, observation and reliability of the service grid.

4.2 Changes to the Base Deployment

In our specific Kubernetes (K8s) deployment for the URL fishing application, the usual deployment steps include the creation and configuration of namespaces, deployment of application services (such as the fishing-web and fishing-model services), injection of *Envoy* agents, and basic service verification. During the implementation of the Rate Limiting function of *Istio*, we have made the following changes and extensions in the deployment part of the basic design:

4.2.1 Increase ENVOYFILTER configuration. On the basis of basic deployment configuration, a dedicated *EnvoyFilter* resource allocation has been added to define the current limit strategy. This includes detailed rules configured Local Rate Limit, such as the token bucket parameters, Filter configuration, etc.

4.2.2 Enable and verify the flow limit function. After the basic service deployment, a series of operations were performed to verify whether the current limit strategy took effect. This includes a large number of requests through the *POD* (such as Sleep) inside the cluster to observe the actual effect of the current limit strategy.

4.2.3 Monitoring and log analysis. Increase the logs and statistical information of the *Envoy* agent to verify and monitor the implementation of the current restriction strategy. By viewing *Envoy* statistics (such as Ratelimit -related counter) and agency logs, you can confirm whether the current limit rule is triggered.

4.2.4 Adjustment and optimization strategy. After the basic deployment configuration is completed, the parameters of the current limit strategy may need to be adjusted according to the actual test results, such as maxtokens, tokensperfill, etc. These adjustments are to ensure that the current limit strategy can effectively protect services from excessive use, and at the same time does not affect normal service requests.

5 EXPERIMENTAL SETUP (ISTIO)

The goal of the experimental setup is to allow us to easily deploy and evaluate two different versions of the web-app, which is the frot-end component of the application that allows users to make calls to the inference server and receive predictions regarding whether or not a certain link is marked as a phishing URL. This section describes the way our repositories - specifically the *app-frontend* and *app-service* components - were configured to make it possible to easily create and test hypotheses related to new additions to the web-app. This was done by configuring an Istio service mesh, changing different image building and pushing steps in our existing Github workflows, and using Grafana and Prometheus to retrieve and display metrics related to the web-app and the inference server.

In order to **qualitatively** compare the old version of the web-app with a newer, experimental version, the Istio service mesh deploys separate versions as part of the same deployment, each using a differently versioned image tag that is pushed by the Github workflow. In order to make a **quantitative** comparison, a Grafana dashboard allows the person conducting the experiment to look at different metrics that are retrieved by Prometheus.

5.1 General Description of the Experiment

Although our Istio service mesh allows one to test any hypothesis they have about potential improvements in the metrics of a web-app, we tested a specific hypothesis to make an illustrative example for the experiment. The hypothesis we test is as follows: *Given a version of the web-app that returns binary predictions (valid/phishing) to the user and a version that returns probabilities instead, the user will prefer the version of the web-app.* After getting predictions from the server, the user is prompted to indicate whether or not they are satisfied with the prediction. The ratio and number of "Yes" and "No" given by the user are collected by Prometheus and displayed by Grafana, and our hypothesis asserts that there will be more "Yes" feedback given in the version of the web-app that returns probabilities.

5.2 Changes in the User Interface

To test our hypothesis, we had to extend our current version of the web-app to display probabilities instead of binary predictions. The two versions of the web-app can be side-by-side in Figure 3 (binary predictions) and Figure 4 (probability predictions). These two versions of the web-app will be deployed in parallel by our Istio deployment mesh, simulating a canary deployment in which some users get access to the older version and some users get access to the newer version.

5.3 Changes in the Github Pipeline

The original Github workflow, which submits upon opening a Pull Request, was not appropriate for such an experiment and also had to be extended to accommodate for this new need. The original workflow built the Docker image and tagged it with the 'latest' tag, which was then pushed to the Github Container Registry. This would override the previous image, and therefore only the latest built image would be available to be deployed.

The build stage of the Github pipeline now operates as follows: at any given moment, the GHCR stores an image for the previous version of the web-app and the new version of the web-app. Upon opening a Pull Request, the previous "latest" image keeps its version tag (for example: image with tag v2 keeps the tag v2), and the previous "old" image gets tagged with the new version tag (for example: image with tag v1 gets tagged with v3). This allows us to have two different image tags that will be deployed under the same mesh and easily compared to confirm or reject a hypothesis.

5.4 Prometheus Metrics

The *app-service* component of our application is a Flask server that is triggered by the *app-frontend* and makes REST calls to the *model-service* inference server to receive and display predictions. Prometheus offers the *prometheus_client* Python package that decorates Flask calls to get Counter, Gauge and Histogram metrics (note: I will also add Summaries later). The metrics collected by Prometheus are as follows:

- Counter: the number of "Yes" amd the number of "No"s given by the user when prompted for feedback about predictions.
- Gauge: the number of current in-progress requests.
- Histogram: the binned values for the total inference time in milliseconds per call, which could be used to experiment and see whether or not longer URLs come with higher inference times.

5.5 Grafana Dashboard

5.6 Istio Service Mesh Setup

5.7 Left TODO

- Description of decision process: which data will be available (**add Grafana screenshot) and how it will be used to derive a decision/answer regarding the hypothesis**

Ideas:

- AlertManager implementation
- `PrometheusRules` defined to warn developers through a channel of their choice
- Grafana dashboards: visualizations, query functions, manual imports, ...
- Continuous experimentation setup: infrastructure config (e.g. `annotations` under `Pods` in our `istio-deployment.yml`) to enable collection of app-specific metrics
- Istio Service Mesh: for traffic management and continuous experimentation

6 ML PIPELINE

In the following sections, we will explain how we made use of specific tools and practices, like Data Version Control (DVC) and Poetry, to manage the complexities of ML projects. We will detail the configuration of our project and explain why these decisions lead to more efficient and effective ML workflows.

6.1 Overview of ML Training Pipeline

Our ML project is structured into distinct folders and files representing key pipeline stages: data retrieval, preprocessing and model training. A visualization of this process is shown in figure [**?**] This approach enhances maintainability and clarity, allowing team members to focus on specific aspects of the workflow without conflicts.

We selected Data Version Control (DVC) to manage our ML pipeline due to its integration with existing Git workflows, enabling seamless collaboration among developers. DVC automates the end-to-end process of data fetching, preprocessing, and training ensuring reproducibility and traceability. With DVC we do not have to run the whole pipeline every time we make a change. It will know which stages have changed and only run those. It also supports remote storage, in our case we use an AWS S3 bucket [NOTE: need dani's bucket here as a footnote], allowing for efficient data management and scalability. Our AWS S3 bucket currently stores our ML model and training data.

To handle dependencies, Poetry was chosen for its intuitive handling of Python packages. This tool simplifies our environment

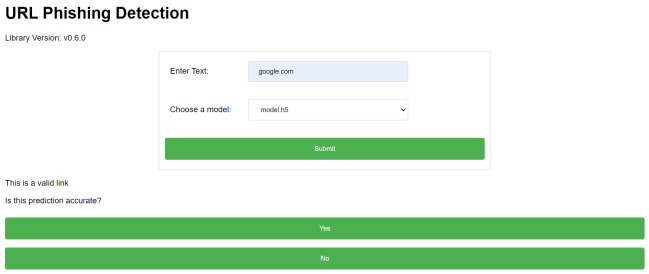

Figure 3: The original web-app, which returns binary predictions.

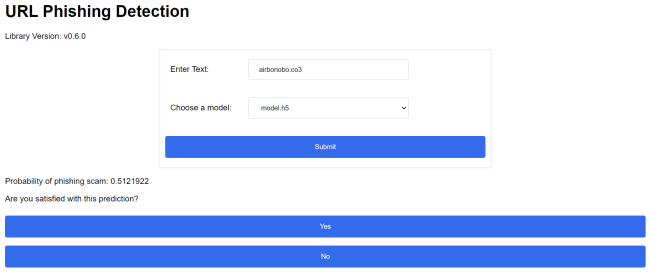

Figure 4: The new version of the web-app, which returns probability predictions.

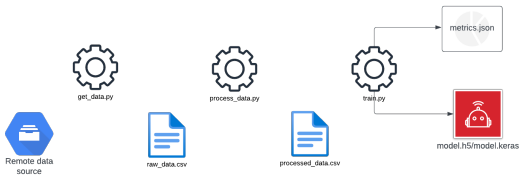

Figure 5: DVC repro pipeline.

management by storing every dependency in a `poetry.lock` file, enabling reproducible builds and consistent environments across all stages of development and deployment.

6.2 Artifacts

Throughout the pipeline, several artifacts are generated: trained models, raw and preprocessed data are stored, versioned, and managed via DVC. This way we can easily compare the performance of two different models to each other. Performance metrics are generated after model training to evaluate the effectiveness of the model against predefined benchmarks.

6.3 Model Metrics

Our project employs a set of metrics—accuracy, loss, precision, and recall—to assess the performance of our models. These metrics help in fine-tuning model parameters and identifying areas for improvement.

6.4 Integration of Linters

To maintain high code quality, we integrated *pylint* and *flake8* into our development process. Pylint is configured to reflect the specific needs of our machine learning project. We have thoroughly analysed linter rules and made modifications accordingly to adapt it to this specific ML project. These modifications include:

- Allowing commonly used variable names in ML such as `X_train`, `Y_train`.
- Defining a set of bad names to discourage non-informative variable names such as *foo*, *baz* and *toto*.
- Extending the list of exceptions that will emit a warning with common errors in ML such as `ArithmeticError`, `BufferError` and `LookupError`.
- Ignoring files that are either auto-generated or do not contain Python code such as *git*, *build* and *dist*.
- Displaying only warnings with high confidence levels and those that lead to inference errors.

For *Flake8* the maximum allowed line length is set to 100, which is in line with Pylint, instead of 88 which is the default value.

These linters enforce coding standards and detect syntax errors and stylistic issues, helping to keep the codebase clean and readable.

6.5 Limitations and Opportunities

The dataset we use for this project is very simple and the model as well. As datasets and model complexity grow, the processing power and memory requirements can exceed the capabilities of our current infrastructure, leading to slower development cycles.

Automating more components of the ML pipeline, such as hyperparameter tuning and model validation for more complex models, could further streamline development and deployment processes.

We could use other tools to configure our ML pipeline such as *ML flow*. This tool provides more comprehensive management for ML projects, including experiment tracking, model serving, and a central model registry, which can enhance collaboration and model lifecycle management.

As an alternative to Poetry for package management we could use *Conda*. Conda offers robust environment management and support for non-Python dependencies, which could be beneficial for projects requiring a broader range of language support.

7 ML TESTING DESIGN

We utilized PyTest to automate our tests, focusing on various aspects from feature generation to model deployment. Given that our dataset initially does not contain explicit features, our testing strategy addresses this by simulating feature creation and evaluating these features as though they were part of the original dataset in order to demonstrate how tests for features could be implemented.

7.1 Features and Data Tests

Our tests cover the scripts that generate input features from the dataset, crucial for both training and serving phases. We developed tests to assess the computational and memory costs associated with each feature. We have also implemented scatter plots and distribution checks to ensure that each feature's distribution aligns with our expectations. This is vital for detecting anomalies in feature engineering which could lead to biased or incorrect model training.

7.2 Model Development Tests

Our tests include checks for non-determinism in model outputs by training the model with different seeds, ensuring consistent results across different runs under the same conditions. This is crucial for maintaining trust in model predictions.

We also utilize data slices to assess model performance across various segments of data. This helps in identifying any model biases or weaknesses in handling diverse data scenarios, thereby ensuring the model performs well across different feature-based groups.

7.3 ML Infrastructure Testing Tests

Before a model is deployed, we perform quality checks to verify its predictive accuracy and other performance metrics by checking if the model is making the right predictions on URLs we already know the outcome of. This ensures that only models meeting our quality standards are moved to production.

7.4 Monitoring Tests

We can monitor for any regressions in serving latency, or RAM usage of each test using `pytest-monitor`. These monitoring tests are vital for maintaining system health and ensuring that our ML infrastructure remains efficient and scalable over time.

7.5 Test Adequacy Metrics and CI Integration

To measure the effectiveness of our testing strategies, we utilize adequacy metrics that provide quantitative feedback on the coverage and completeness of our tests. These metrics include "statements" (The total number of statements in the package), "covered" (The percentage of statements that were executed during testing) and "missed" (The number of statements that were not executed during testing) which help us understand the extent to which our tests exercise the code under various scenarios.

7.6 Limitations and Opportunities

Our dataset is very simple at this point. Tests for non-determinism are robust, but they might not cover all sources of randomness within more complex models or training environments.

Currently, our feature tests are also limited to those we can engineer from the available data. This could miss potential issues in real-world scenarios where more complex or externally influenced features are involved.

The current monitoring setup might not be sensitive enough to catch subtle, slow-developing issues before they impact the system significantly.

REFERENCES