REMLA Phishing Detection – Group 5

Tim den Blanken, Jurriaan Buitenweg, Saga Sunnevudóttir, Rado Todorov

1 INTRODUCTION

We have created an application that can be used to predict whether URLs are phishing or not. All source code can be found in the repository[1], to which we will reference in this report. The purpose of this project was not creating the application itself, but the infrastructure surrounding it, with many extensions such as monitoring and continuous experimentation. This report serves as documentation for the setup of the project and contains explanations of the design decisions that we made. Moreover, we will cover the release pipelines (section 2), the deployment (section 3), a proposed extension (section 4), the additional use case and experimental setup through Istio (section 5 and section 6 resp.) and lastly the ML pipeline and ML testing (section 7 and section 8 resp.).

2 RELEASE PIPELINE

This section contains an overview of release pipelines for software packages (lib-ml and lib-version) and container images (app and model-service). It will cover their steps, purpose, implementation and dataflow.

2.1 Release pipeline for software packages

This project contains two software packages. The lib-ml package contains all pre-processing logic for data that is used for training or queries. This means that both the model-service and model-training repositories use the lib-ml package. The lib-version package is a version-aware library that can be asked for its version. Both packages are versioned and released through a GitHub workflow, which is triggered when a push is made to the main branch. The release workflow is visualized in Figure 1, where each step is explained in detail below. The numbers in the figure correspond to the numbers in the list below.

(1) **Checkout:** First the repository is checked-out under $GITHUB_WORKSPACE, such that the workflow can access the codebase[13].
(2) **Configure git credentials:** Next git is configured with a user name and email for the GitHub Actions bot. This way it is clear where the commits from the workflow came from.
(3) **Parse version information from tag:** This step retrieves the latest tag, increments that patch version and sets the new tag in the GitHub Actions environment.
(4) **Setup Python:** Sets up a Python 3.9 environment using actions/setup-python[16], which is a version that supports all used packages.
(5) **Install and configure Poetry:** Next the workflow installs a specific Poetry version which is used for dependency management within the project, this is done using snok/install-poetry[15].
(6) **Update poetry package version:** Then the package version in the pyproject.toml is updated to the new tag, ensuring consistency for the versions.
(7) **Commit, push and release new version:** The final step first commits the new pyproject.toml and then creates a new tag. Afterwards, the new version is pushed and released.

After these steps, the package can be included, e.g. through a requirements.txt file, by adding the following line package-name @ git+https://github.com/REMLA24-Team-5/package-name@v*.*.*.

2.2 Release pipeline for container images

Next to the software package, the project also contains two container images. One for the app, which contains both the frontend and backend of the app which then queries the model, which is all contained in the model-service image. Both these container images are versioned and released to the ghcr.io container registry using a GitHub workflow. This workflow is triggered when a new tag (v.*.*.*) is pushed, ensuring it only runs for new versions. All steps are visualized in Figure 2, where each step is also explained in detail below. The numbers in the figure correspond to the numbers in the list below.

(1) **Checkout:** The repository is checked out under $GITHUB_WORKSPACE so that the workflow can access the codebase.
(2) **Parse version information from tag:** This step retrieves the latest tag and parses the version into major, minor, and patch components, setting them in the GitHub Actions environment.
(3) **Login to GitHub Container Registry:** Authenticates Docker to push images to the GitHub Container Registry (ghcr.io) using GitHub's credentials using docker/login-action[14]. The GITHUB_TOKEN is stored as a secret.
(4) **Replace uppercase with lowercase in repository name:** Converts the repository name to lowercase as this is required by container image name conventions.
(5) **Build and Push Docker Image:** Builds the Docker image for the app or model-service and tags it with the new version. Also versions using latest are released, such that other components in the project can refer to the latest version. The image is then pushed to the GitHub Container Registry.
(6) **Release to GitHub:** Creates a new release on GitHub associated with the new tag, making the new container image version publicly available.

After these steps, the container image is available in the GitHub Container Registry and can be pulled from the following location: ghcr.io/remla24-team-5/image-name:latest.

3 DEPLOYMENT

In this section, we detail our application's final deployment, focusing on the various tools used as well as the components and data flow within our system. For provisioning and configuration we use Vagrant [20] and Ansible [30]. The whole system is deployed using Kubernetes [10] with Istio [5] to manage traffic routing. Additionally, Prometheus [11] and Grafana [28] are utilized for monitoring

[1] https://github.com/REMLA24-Team-5/operation

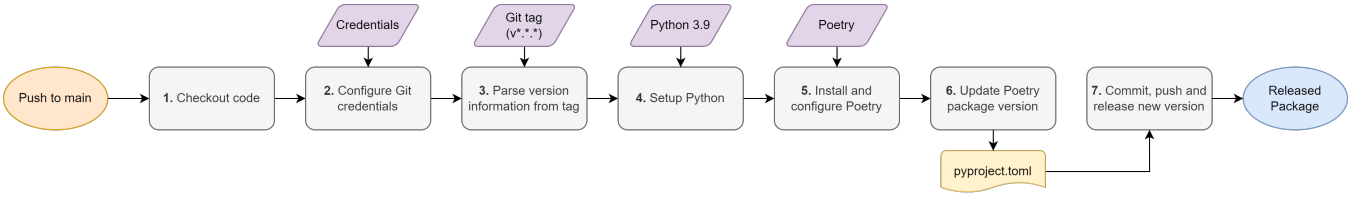

Figure 1: Release pipeline for a software package

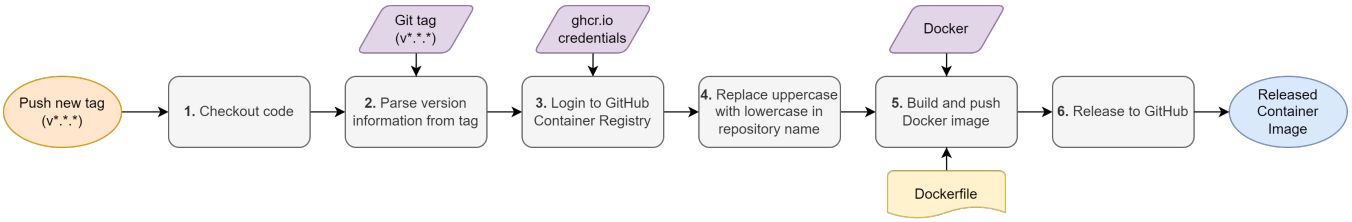

Figure 2: Release pipeline for a container image

the application for experimentation. The section also delves into the various components of the system, detailing the connections between them and illustrating the data flow across the system.

3.1 Provisioning

The deployment starts with setting up virtual machines using Vagrant, which simplifies virtualized development management through declarative configuration files. Vagrant uses Ruby syntax, enabling dynamic setup of multiple nodes and flexible network configurations. It can also automate the provisioning of VMs, including software installation, configuration, and setup through provisioners like Ansible [21]. We create both a controller node virtual machine and two worker node virtual machines.

To achieve idempotent provisioning and set up the virtual machines, we use Ansible, a versatile automation tool known for its agentless architecture and YAML-based playbooks, facilitating streamlined configuration management and deployment tasks [1]. We setup the following with Ansible playbooks:

- **Controller node:** Configured with Kubernetes using k3s, Helm for package management, and Istio for service mesh installation.
- **Worker nodes:** Joined to the Kubernetes cluster using k3s agents, enabling distributed workload management and scaling for the application and model-service images.

This setup ensures easy deployment and management of our Kubernetes-based infrastructure.

3.2 Kubernetes

Initially, our deployment was a simple Docker Compose setup [2], which created and started the app and model-services images to run and expose a web service that interacts with the model-service.

To make our setup more decoupled and scalable, we opted to use K3s to deploy our application in a lightweight Kubernetes environment[3]. K3s is a certified Kubernetes distribution designed for resource-constrained environments, making it ideal for our project. It simplifies the setup process and reduces the overhead associated with standard Kubernetes clusters [24]. The cluster is deployed to the VMs mentioned above with one controller node and two worker nodes. The control-plane divides the deployed pods among the worker nodes. The following sections explain the different entities of the Kubernetes setup in more detail and how they interact.

3.2.1 Deployments . A Kubernetes Deployment is a resource that ensures the availability and management of application pods. Both the application and model-service images are deployed using Kubernetes Deployment resources, which create and manage Pods for each deployment. Deployments are essential for maintaining the desired state of Pods, allowing for updates, rollbacks, and scaling [25]. The app Deployment uses an environment variable for the model-service URL, which is specified in a Kubernetes ConfigMap. This setup enables seamless updates to the model-service endpoint without the need to redeploy the application by simply updating the ConfigMap.

3.2.2 Services . A Kubernetes Service exposes an application running on a set of Pods as a network service, ensuring stable connections even if Pods are replaced or scaled. It handles traffic routing and load balancing, distributing incoming traffic across the Pods [27]. In our case, our app uses a Service to maintain a single access point for the user through Ingress and to the model-service as well for the app to query it, regardless of the number of Pod replicas. This simplifies network management and enhances application reliability and scalability.

3.2.3 Ingress . A Kubernetes Ingress is a resource that manages external access to services within a Kubernetes cluster. It provides routing rules to distribute traffic to different services based on the

[2]https://github.com/REMLA24-Team-5/operation/blob/main/compose.yml

[3]https://github.com/REMLA24-Team-5/operation/blob/main/playbooks/setup_k8s_cluster.yml

request path or host [26]. Our application uses an Ingress to direct incoming traffic to the app Service.

3.3 Istio

Istio is a powerful open-source service mesh that provides advanced traffic management, security, and observability for microservices. It is an excellent tool for implementing various deployment strategies, ensuring smooth and reliable operations [22]. In our system, we utilized Istio's traffic management capabilities to create a canary release of the application and to Shadow Launch a new version of the model which are explained in more detail in later sections. The following sections explain the different resources of Istio and how they are used in our experimental setups.

3.3.1 Gateway . An Istio Gateway manages incoming network traffic into the service mesh, supporting HTTP, HTTPS, and TCP protocols. It acts as a gateway for external traffic entering the mesh and routes it to appropriate services. In our canary release setup we need an Istio Gateway to route external traffic from the app Ingress to the Istio mesh.

3.3.2 VirtualService . An Istio VirtualService controls how requests are routed within the service mesh. It defines rules for HTTP request routing, load balancing between service versions, timeouts, retries, and fault injections. In our canary release setup, we leverage VirtualService to direct users more frequently to the original app version and to a smaller subset to the canary version. For our Shadow Launch, we utilize VirtualService's mirroring functionality to duplicate all traffic from one version of the model-service to another.

3.3.3 DestinationRule . An Istio DestinationRule sets policies for traffic to a service after routing by a VirtualService. It manages load balancing, connection pools, outlier detection, and retry policies. DestinationRules enable fine-grained control over service interactions, supporting canary releases, circuit breaking, and more. In our setup, DestinationRules are essential for directing traffic within the Istio Service mesh to specific versions of both the app and model-service, using defined subsets for differentiating between versions (v1 and v2) of the app and model Services.

3.4 Monitoring

Prometheus and Grafana are tools which are used for monitoring the deployments of our system. Both are installed using Helm for simplicity [9]. Prometheus collects and queries custom made metrics from various components within our Kubernetes cluster. It supports metric types such as Count, Gauge, Summary, and Histogram [29]. A ServiceMonitor custom resource is used to define how Prometheus should scrape metrics from services. It specifies the services to monitor, which are in our case the app and model-service Services, the endpoints to scrape, and the interval for scraping. Additionally, an Alertmanager was configured to raise alerts if the web app experiences high request rates.

Grafana is a powerful open-source analytics and monitoring platform that integrates with Prometheus and other data sources. It provides customizable dashboards and visualizations for metrics, enabling users to analyze and interpret system performance in real-time [18]. We created custom grafana dashboards to showcase and interpret the Prometheus metrics scraped in both the experimental setup described in section 6 and the additional use case we implemented described in section 5.

3.5 Entities and their connections

The main entities of the system are the web application and the model-service. These are deployed to our Kubernetes cluster as Pods and are managed with Services for networking within the cluster. Istio is used for external traffic management, handling routing and load balancing. The Kubernetes cluster is deployed on VMs where each entity is automatically distributed to the worker nodes. The app interacts with the model-service via REST to query the service, which utilizes a machine-learning model to predict phishing URLs. Details on how the model is trained and deployed for the model-service to use can be found in section 7. Both the app and model-service rely on libraries, lib-ml and lib-version, mentioned in sections above. The interactions between the users, web app, and model-service are collected as metrics by Prometheus and visualized with Grafana to gain insight into the performance and behavior of the system during canary releases and shadow launches.

3.5.1 Dataflow . The dataflow within our deployment begins with user interactions to the web application. Users inputs a URL they want to be predicted as a legitimate or phishing. Before the request is made, the user himself is asked to make a prediction if the URL is phishing or innocent. Requests from users are managed by Istio, which handles external traffic routing, ensuring efficient and reliable communication. The web application sends REST requests to the model-service, querying it to predict phishing URLs using a machine-learning model. The model-service processes these requests, leveraging the trained model to generate predictions. Responses are then sent back to the web application, which presents the results to the user. Throughout this process, interactions and metrics are collected by Prometheus, providing valuable insights into system behavior. Grafana visualizes these metrics, enabling real-time monitoring and analysis. This comprehensive dataflow supports the seamless operation of our main system as well as canary releases and shadow launches, facilitating continuous updates and testing without service disruption.

3.5.2 Visualisation . An overview of how the Kubernetes cluster is set up and how the entities connect can be seen in Figure 3. The diagram also shows the client making a request and how the data flow is through the system.

4 AUTO-SCALING EXTENSION

Looking into the future, we expect our project to attract more users seeking help with phishing attacks. This increasing demand would be a problem to our current setup as we would need to manually increase the number of virtual nodes and instances inside the cluster. However, this can be problematic and expensive, since usage spikes can be unpredictable. We would ideally want to expand the usage of resources only when necessary in order to utilize it for other tasks in the meantime, like model training, data processing etc. To ensure efficient resource utilization and improve the reliability of the application, we propose an extension to the project which involves utilizing Kubernetes auto-scaling.

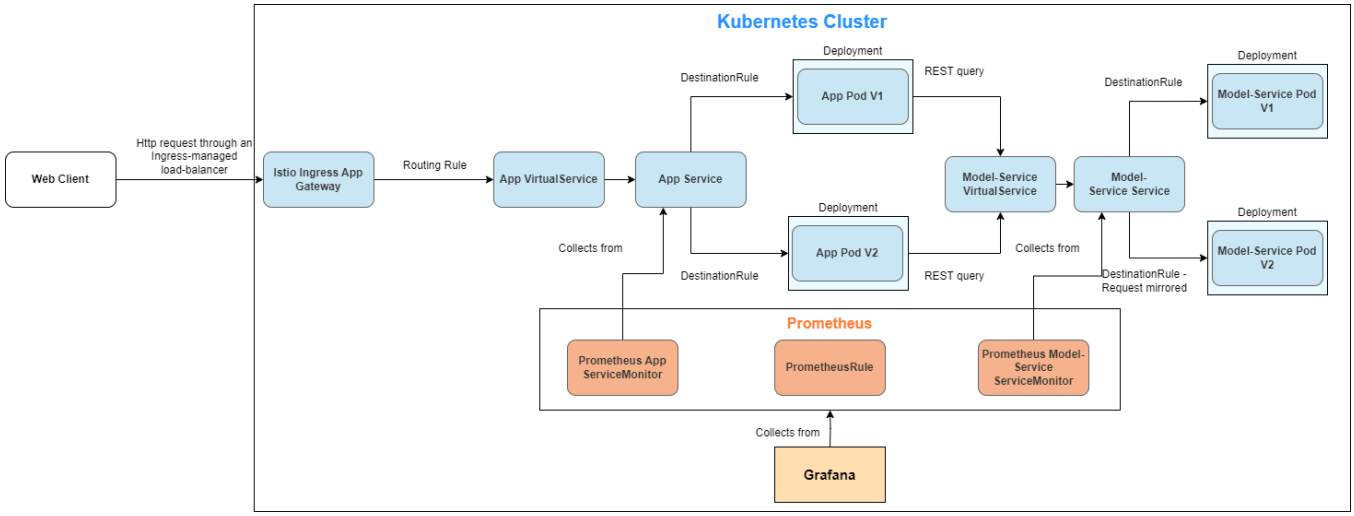

Figure 3: Deployment diagram of the Kubernetes cluster with information about the dataflow through the system.

Horizontal Pod Autoscaler is a technique for adding additional pods when faced with high demand. The idea is quite simple, we select a desired metric value like CPU or memory usage and the autoscaler dynamically adds or removes pods in order to preserve the following equality.

$$\text{desiredReplicas} = \left\lceil \text{currentReplicas} \times \left(\frac{\text{currentMetricValue}}{\text{desiredMetricValue}} \right) \right\rceil$$

A downfall of the standard Kubernetes autoscaler is the fact that the procedure is executed at a specified time period. This implies, that we always react to user traffic. An improvement proposed by Zhou et al. [31] utilizes a predictive model, which makes use of past data in order to aid in the responsiveness of the autoscaler.

Vertical scaling in Kubernetes[4] is yet another technique which could be utilized in our goal of automation, instead of employing more pods, we adjust the resources that a pod utilizes, independent of others. Auto-scaling can also be scheduled using a tool like Cron [4] allowing us to scale down at night or expected periods of low usage rates.

To verify that our changes successfully tackle the problem of automatic adjustments, we can load test the cluster, by using tools like JMeter [6] that allow us to send thousands of requests at the api endpoint. The provided monitoring of the proposed tools enable us to judge whether the system is able to adapt to the sudden spike in traffic effectively. Moreover, we can monitor the CPU or memory usage over a period of time with and without the extension in order to verify its efficiency.

5 ADDITIONAL USE CASE

We implemented two additional use cases. We were able to showcase the traffic mirroring capabilities of Istio as well as limit the rate at which users interact with the system.

[4]https://github.com/kubernetes/autoscaler/tree/master/vertical-pod-autoscaler

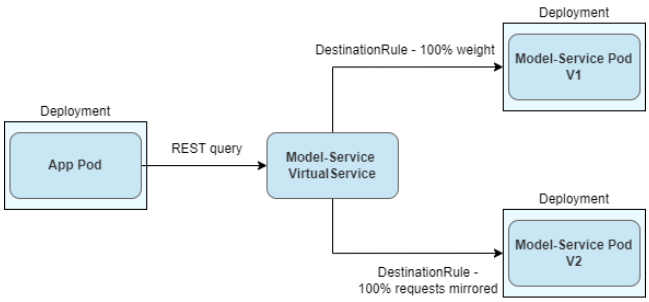

Figure 4: Diagram of the entities that make up the Istio shadow launch in the Kubernetes cluster

5.1 Shadow Launching

Traffic mirroring, also known as shadowing, is a powerful technique that enables developers to deploy changes to production with minimal risk. It works by sending a duplicate of live traffic to a mirrored service, allowing the new service to process real traffic without affecting the deployed service [23]. In our system, we decided to shadow launch a new version of the model service that uses a newer model to predict phishing URLs. To achieve this, we deployed a second instance of the model service with the updated model. The updated model is trained on more epochs and thus should perform better. The deployment mirrors existing traffic, submitting it to both the original and the new service. By scraping custom metrics of both services, we can evaluate the new model's performance without exposing it to users. This approach allows us to thoroughly assess the new model in a real-world environment while ensuring the user experience remains unaffected. Figure 4 shows a simplified diagram of the traffic mirroring to the new model-service version.

5.2 Rate Limiting

Limiting the number of requests users can perform on our system per time period is a standard way of preventing DDoS attacks.

```
vagrant@controller:~$ for i in $(seq 1 12); do curl -H "Authorization: Bob"
-s -o /dev/null -w "%{http_code}\n" http://192.168.56.10:80/; done
200
200
200
200
200
200
200
200
200
429
429
vagrant@controller:~$ for i in $(seq 1 12); do curl -H "Authorization: Nick"
 -s -o /dev/null -w "%{http_code}\n" http://192.168.56.10:80/; done
200
200
200
200
200
200
200
200
200
200
429
429
```

Figure 5: Users Bob and Nick exceed their request limits

We have implemented two types of limits - user-based and global. This helps us manage the total number of requests the system can handle and prevent malicious users from overloading the system. The service is implemented by applying a filter to the requests which pass through the gateway. Using an additional service for counting user requests, we are able to enforce restrictions which track specific users. Currently, this feature is supported through having an additional header tag when calling the gateway, but in the future can be incorporated in a form of a token or cookies within the application service. An example of limiting individual users is shown in figure 5. Here, user *Bob* and *Nick* execute their requests within the same minute, but are only *blocked* after exceeding their own limit.

5.3 Changes to Base Deployment Design

The changes made to the base deployment involve deploying two Kubernetes Deployments of the model-service, resulting in two model pods, one for each version of the model service. To enable traffic mirroring, a VirtualService was configured with a DestinationRule that mirrors all traffic from version one of the model service to version two. Rate limiting was enabled by implementing a redis service, used to store request counters as well as EnvoyFilters, used to enforce the rate limit configuration. These modifications are applied to the Kubernetes cluster using an Ansible playbook, as mentioned above.

5.4 Prometheus Metrics

To evaluate the new model, custom metrics are scraped from both model-services using Prometheus. The metrics collected include the number of requests to each model, the number of URLs predicted as phishing by each model, and the phishing URL rate, which is the percentage of URLs detected as phishing out of the total URLs checked. This data collection is done by deploying a ServiceMonitor to the Kubernetes cluster to scrape metrics from the model-service instances. By accessing the Prometheus dashboard, we can observe that metrics from both model services are being collected. Figure 6 shows the Prometheus dashboard after querying the metric of total model requests for each model version. These should show to be the same, every request is mirrored to the shadow-launched model-service.

Figure 6: Screenshot of the Prometheus dashboard after querying the metric of total model requests for each model version

To determine if the new model is better than the original, we compare the phishing URL rate of the new model to that of the older model. If the new model shows a higher rate of phishing detection, it may be considered better, as it is crucial to minimize false negatives (failing to detect a phishing URL) even if it results in more false positives (incorrectly labeling a safe URL as phishing). This approach prioritizes security by ensuring that fewer phishing URLs are missed, which can be more critical than having a few incorrect phishing detections.

5.5 Grafana Dashboard

A custom Grafana dashboard was created to gain insight into the metrics scraped by Prometheus. It includes a graph displaying the request rate for each model, which should be identical since the traffic is mirrored. This helps ensure that the mirroring is functioning correctly. Additionally, the dashboard has a graph showing the rate of which each model detects phishing URLs. There are also gauges indicating the percentage of URLs detected as phishing out of all URLs checked by each model, as well as the average phishing URL detection rate per model. The following image shows the dashboard after querying the models a few times. Figure 7 shows the custom Grafana dashboard made for the Shadow-Launch. In this demonstration, the models' detection rates for phishing URLs are identical. However, in a real-world scenario, this dashboard could reveal more valuable insights to determine if the new model version outperforms the original.

6 EXPERIMENTAL SETUP

In our system, we utilized Istio's capabilities to create a canary release. A canary release is an experimental setup where a new version of a deployment is tested on a small subset of users while the majority continue to use the stable version. This method is beneficial for minimizing risks and ensuring a smooth transition by providing real-world usage data and feedback on the new deployment version while maintaining a stable user experience for the majority. This approach also allows us to carefully monitor the new version's performance and catch any potential issues before deployment of the new version. Figure 8 shows a simplified diagram of the traffic mirroring to the new model-service version.

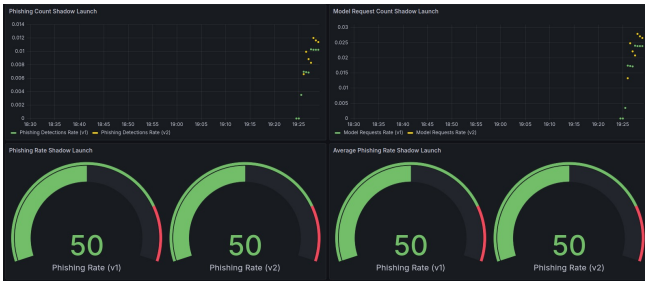

Figure 7: Screenshot of the custom Grafana dashboard with metrics scraped with Prometheus from the original model-service (v1) and the shadow-launched one (v2).

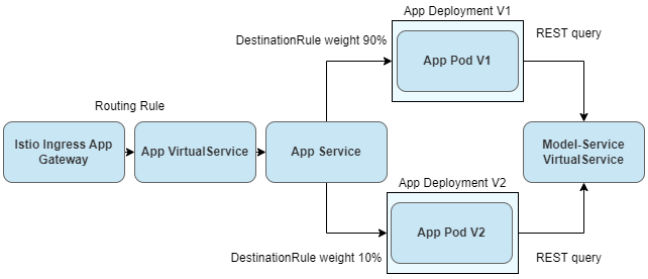

Figure 8: Diagram of the entities that make up the Istio canary release in the Kubernetes cluster

6.1 Description of Experiment

In our canary release experiment, we identified a usability issue in the original application related to user interaction when determining whether a URL is phishing or not. We observed that the button colors did not intuitively align with their intended actions, causing potential confusion. Originally, the buttons were green for marking a URL as phishing and red for indicating it as legitimate, which could lead users to misinterpret their actions. Therefore, in the new version, we switched the button colors to red for phishing URLs and green for legitimate ones, ensuring that the visual cues align more closely with user expectations and actions.

6.2 Changes to Base Deployment Design

To implement this, we deployed two application instances with Kubernetes, resulting in two pods running simultaneously, one with the original version of the app and one with the canary release. Additionally, we configured Istio with a Gateway and a Virtual Service to make our web apps accessible through the IngressGateway. We further used a DestinationRule to direct a small fraction (10%) of the traffic to the new version. To stabilize the subset of requests redirected to the new service, we utilized header information to ensure that once a user is selected for the experiment, they consistently see the new version upon reloading.

6.3 Hypothesis

The hypothesis tested in this experiment is that aligning button colors with their respective actions, red for phishing URLs and green for legitimate URLs, will improve clarity for the user and thus minimize confusion during URL assessments. This may then result in closer alignment with the assessment of the model.

Figure 9: Screenshot of the Prometheus dashboard after querying the metric of total page requests for each app version

6.4 Prometheus Metrics

To evaluate the new app version, we implemented a comprehensive metric collection system using Prometheus. We monitor various custom metrics from both app instances, including the number of page and model requests, the count of URLs identified as phishing by each model, and the phishing detection rate, which indicates the percentage of URLs flagged as phishing out of the total evaluated. Additionally, we track user interactions, such as the number of phishing guesses made, how often these guesses match the model's predictions, and the agreement rate between user guesses and model results. We also summarize and create histograms for model request durations to gain insights into performance. These metrics are collected by deploying a ServiceMonitor within our Kubernetes cluster. By accessing the Prometheus dashboard, we can see that metrics from both apps are being collected. Figure 9 shows the Prometheus dashboard after querying the number of page requests to each app version.

6.5 Grafana Dashboard

A custom Grafana dashboard was created to gain insight into the metrics scraped by Prometheus. The dashboard includes multiple visualizations to ensure comprehensive monitoring and analysis.

Phishing-related metrics are covered extensively: graphs show the rate of phishing detections and user phishing guesses for each version. A gauge indicates the phishing rate, showing the percentage of URLs detected as phishing. Another gauge displays the user phishing guess rate, highlighting how often users predict a URL as phishing. The dashboard also includes a gauge for the user guess agreement status, which tracks if user guesses align with model predictions.

To further analyze performance, there are graphs for the user same guess count and gauges for the average phishing rate, average user phishing guess rate, and average user same guess rate per app. The dashboard also presents a summary and histograms of model request durations, providing insights into the performance and efficiency of each app version.

To determine if the new app is better than the original, we compare the metrics and graphs mentioned above. Most importantly, we can assume that if the users' guesses align more closely with

Figure 10: Screenshot of a part of the custom Grafana dashboard with metrics scraped with Prometheus from the original app (v1) and the canary-release (v2).

the model's predictions, then the users are less confused by the user interface colors. This alignment indicates that the new color scheme enhances clarity and reduces miscommunication, leading to more accurate user assessments. By validating these improvements through the collected data, we can confirm the effectiveness of the canary release.

Figure 10 shows the custom Grafana dashboard made for the canary-release. In this demonstration, the user guess rates for phishing URLs (User Phishing Guess Rate) is not more in line with the models predictions (Phishing Rate) in the canary-release (v2) compared to the original (v1), indicating that the hypothesis was wrong. However, in a real-world scenario, this dashboard could reveal more valuable insights to determine if the new app version shows improvements compared to the original.

7 ML PIPELINE

The training pipeline used for the Phishing Detection model was set up following best practices and tools. This section explains how the pipeline was set up and decisions that were taken for this along the way.

7.1 Setup

The first and one of the most important decisions to make is the structure of the project. Having a good structure creates an easy entrypoint for outsiders, and ensures a logical pipeline from separated files. We have chosen to use the Cookie Cutter data science (ccds) template [17]. Since data science code quality is about correctness and reproducibility, ccds helps ensure this.

However, the project structure is just the first step in this process. Adequate version control is needed, as this allows developers to move fast and software teams to preserve efficiency and agility [2]. The use of git is an obvious one, as this captures all iterations and developments of the code itself. For this project however, we have also used Data Version Control (DVC), which takes care of saving iterations of datasets and models. On their website DVC is described as: "DVC helps machine learning teams manage large datasets, make projects reproducible, and collaborate better". From here it is apparent that the use of DVC in data science projects is very useful and important. We have setup a pipeline consisting of four steps, each are described in the coming sections and in Figure 11.

1. Get data: In order to 'start' the pipeline, the training (and testing and validation) data should be retrieved. Since the data largely determines the model performance, it is important to keep track of this data, such that poor (or good) model performance can be traced back to changes in training data, if that is the case. For this reason the data is part of the pipeline and managed by DVC. The `get_data.py` script is simply to retrieve the data and store it in a proper format. The dependencies and outputs for this step are listed in Table 1.

Dependencies	Outputs
- src/data/get_data.py - data/train.txt - data/test.txt - data/val.txt	- output/raw_x_train.joblib - output/raw_y_train.joblib - output/raw_x_test.joblib - output/raw_y_test.joblib - output/raw_x_val.joblib - output/raw_y_val.joblib

Table 1: Dependencies and outputs for step 1.

2. Preprocess: The next step is to preprocess the data, such that it is in a format that the model accepts. Once again, the preprocessing methods have influence on the model performance, and for that reason one should keep track of the changes in order to relate it back to changing model performance. This is why preprocessing is also a separate step in the DVC pipeline. The preprocessing itself is imported through the `lib-ml` package. Sticking to a specific version of this package will ensure that the preprocessing does not change, and if it does it will be captured by DVC. The `process_data.py` script imports the preprocessing module and applies it to the data files. The dependencies and outputs for this step are listed in Table 2.

Dependencies	Outputs
- src/features/process_data.py	- output/char_index.joblib
- output/raw_x_train.joblib	- output/x_train.joblib
- output/raw_y_train.joblib	- output/y_train.joblib
- output/raw_x_test.joblib	- output/x_test.joblib
- output/raw_y_test.joblib	- output/y_test.joblib
- output/raw_x_val.joblib	- output/x_val.joblib
- output/raw_y_val.joblib	- output/y_val.joblib

Table 2: Dependencies and outputs for step 2.

3. Train: With the preprocessed data from the previous step, we can now move to training the model. For this we need to define a model architecture, which is done in the `model_definition.py`. The model is then created, after which it is trained on the data, in `train.py`. Obviously the trained model is saved and changes in this model are monitored by DVC. This will allow us to keep track of model iterations, which is very important, for example in case the team wants to revert back to a model that was performing better. The dependencies and outputs for this step are listed in Table 3.

4. Predict: The final step uses the trained model to perform inference and test the performance. This is done in the `predict.py`. Once again it is important that the data we test on is always the same, allowing for a fair comparison between models. DVC helps

Dependencies	Outputs
- src/models/model_definition.py - src/models/train.py - output/char_index.joblib - output/x_train.joblib - output/y_train.joblib - output/x_val.joblib - output/y_val.joblib	- output/model.h5

Table 3: Dependencies and outputs for step 3.

us keeping track if this is actually the case. This is also the step where metrics are collected, such as the accuracy, confusion matrix and other performance scores. These values are saved to a metrics.json, which is then also monitored by DVC. This allows to compare metrics between different experiments. The dependencies and outputs for this step are listed in Table 4.

Dependencies	Outputs
- src/models/predict.py - output/model.h5 - output/x_test.joblib - output/y_test.joblib	- metrics

Table 4: Dependencies and outputs for step 4.

A question that may have arisen by now is: where is the pipeline stored? The team has chosen to use Google Drive for this, for its easy implementation with DVC. This means that when new users pull the pipeline, it downloads the data from Google Drive.

Another very important aspect of any project is dependency management. Nearly every project makes use of other libraries to make code cleaner and more efficient. However the use of other libraries can lead to dependency conflicts. For this reason it is very useful to use a tool that can resolve these conflicts and store all possible versions of libraries that can work together. We have chosen to use Poetry [7] for dependency management, as it is proven and effective software.

We have now covered many tools which are part of the pipeline, ensuring reproducibility and a clear project structure in general. On a code-level, we should stick to these same high standards, since clean and readable code are very important. Readable code will allow outsiders to quickly pick up on what the code is supposed to do, streamlining the process. Of course, everyone in our team tries to write good code, adhering to code standards, but the only way to ensure good code quality and consistency is the use of linters. We have implemented both Pylint [8] with the dslinter [19] extension, which is specifically for machine learning development, and Bandit [3]. Bandit is used to find common security issues, this is very important, as it could be that the machine learning tools that one might be working with should not become public in any way. All the project code has been reduced to the main functionality and changed where necessary, such that both linters now output perfect scores. A more detailed look into the scores, together with a critical reflection on the (un)used linter rules can be found in the README of the model-training repository.

The final product from the pipeline that could actually be deployed is the trained model. However one should never replace the existing model with a newer version without (extensive) testing, either locally or through for example a shadow launch. This means that in this case, the deployment should not be directly linked to the output of the pipeline, but a validation step (either manually, but preferable automatically) should be in between. In section 3 we elaborate on how we deployed the model. Later in subsection 5.1, a method to test a new model was proposed, namely Shadow Launching.

7.2 Discussion and other considerations

The four-step process as described in subsection 7.1 covers the main aspects of the machine learning pipeline (i.e. getting data, preprocessing, training and testing). The team felt these steps explain the ML pipeline well, however some teammembers were having issues pulling the data from Google Drive through DVC, while for others it worked. This inconsistency is not desired and therefore it is probably advisable to switch to a different remote storage, such as Amazon S3 (AWS) or Microsoft Azure. These storage locations will also be more flexible in case of many iterations where the storage size increases a lot.

DVC also keeps track of performance metrics to compare the models. The metrics currently saved are adequate to compare the models, however the formatting in the .json file can be improved to allow for an easier comparison, especially when there would be many experiments.

Another important aspect is how the machine learning pipeline allows for testing as elaborated on in section 8. Initially the pipeline was not set up to include tests. We altered the operation repository structure so that it uses volumes. This requires the user to put the latest version of the model weight into a folder which is then loaded to the container through volumes. This requires an extra step but does make sure the latest version of the model is used. It also provides more flexibility to the user in picking model versions instead of fixing a version to a specific Google Drive file link.

8 ML TESTING

We tested our model extensively. We tested the following aspects of the project according to the ML test score [12]:

- Features & Data: Tests which verify that data is correctly gathered and features are processed in the right way.
- Model Development: Does the model perform like expected? And what if its only evaluated on parts of the data?
- Infrastructure: Does the complete pipeline function correctly?
- Monitoring: Do weight file sizes fit within set limits?

We wrote multiple types of tests. First, we wrote unit tests to test individual components of the system. Secondly, integration tests were written to ensure that the project pipelines functions as expected. Testing cleanups are executed at the end of each testing sequence. Fixtures are used so that identical data is not pulled from the internet multiple times to save time during model testing . Finally, testing is incorporated into the GitHub pipeline (Figure 12).

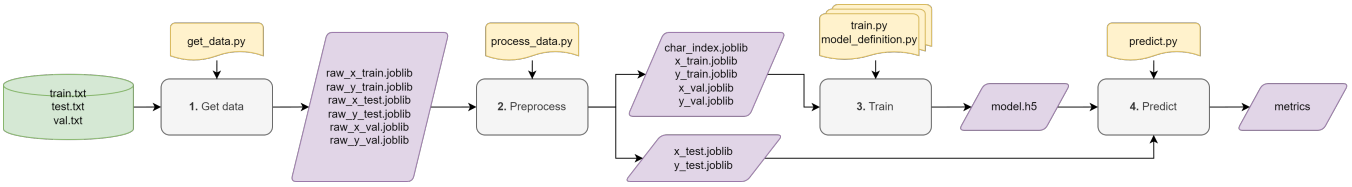

Figure 11: ML Pipeline implemented using DVC

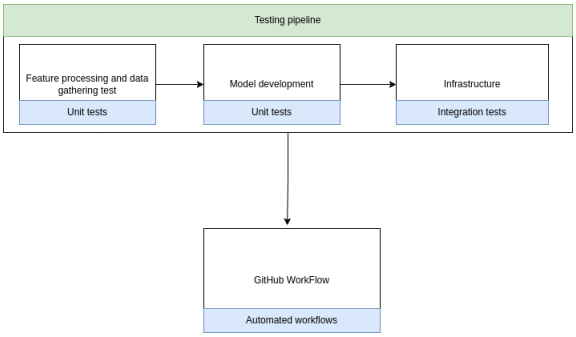

Figure 12: Our testing pipeline. First feature and data tests are done, then model development tests. Finally, through integration test the pipeline is tested.

```
---------- coverage: platform linux, python 3.10.12-final-0 ----------
Name                            Stmts   Miss  Cover   Missing
-------------------------------------------------------------
src/__init__.py                     0      0   100%
src/data/__init__.py                0      0   100%
src/data/get_data.py               28      1    96%   45
src/features/__init__.py            0      0   100%
src/features/process_data.py       28      1    96%   46
src/models/__init__.py              0      0   100%
src/models/model_definition.py     31      0   100%
src/models/predict.py              44      3    93%   75-76, 81
src/models/train.py                22      3    86%   34, 37, 41
-------------------------------------------------------------
TOTAL                             153      8    95%
```

Figure 13: Final testing results showing a test coverage of 95%

8.1 Features & Data

We implemented multiple unit tests, each testing a specific part of either data gathering or feature processing. First, a test was written to ensure that training, testing and validation sets are gathered correctly from Google Drive. Processing tests check if the data is correctly processed and exported to the corresponding joblib files.

8.2 Model Development

Model development test are split into unit tests, each testing the performance of the model and if it performs well on a given set of input cases. Finally non-determinism and robustness checks are done by ensuring model performance is not significantly different when using different random seeds. Mutamorphic testing is used to ensure prediction for similar URLs do not differ significantly. This is done by applying random mutations to the input data while preserving the expected ground truth label.

8.3 Infrastructure

The complete infrastructure is tested using pipeline tests. These test the complete process of gathering data from the internet, processing this data, putting it through the model and exporting results.

8.4 Monitoring

Two monitoring tests are written. First, model staleness testing is applied to check if saved weight files do not exceed a certain time limit. This is to ensure that the model gets updated frequently. Secondly, test are written which ensure the size of the model file does not exceed limits.

8.5 Continuous Deployment

All tests are run every time new changes are pushed to the GitHub repository. This is to ensure that testing is performed before merging important changes. Merging only in test succession is enforced.

8.6 Metrics

Figure 13 shows our total test line coverage for all the files within the model training repository. Metrics are reported automatically by pytest after running the command.

8.7 Future testing

We have not yet implemented automatic metric generation within the pipeline. This is something which is highly recommended to do in the future since this adds insight into the ML test score and overall robustness of the model and its pipeline. In the future, more individual components (unit tests) could also be added to ensure more robustness. An example could be how individual data entries are processed individually as now, only chunks of data processing are tested. Another component that should be added in the future is merge blocking when test metric results are insufficient, for example in case of $< 70\%$ line coverage.

A difficult part was combining the required files for DVC with testing. Separating all the different stages into joblib files made it very difficult to test in an efficient way. Especially because the preprocessor needs the training, testing and validation files for initialization. This made it difficult to truly test the components of the pipeline individually. This is something that could be simplified in the future.

REFERENCES

[1] Ansible Community. 2024. *Ansible Documentation.* https://docs.ansible.com
[2] Atlassian. 2024. *What is version control?* https://www.atlassian.com/git/tutorials/what-is-version-control

[3] Bandit Authors. 2024. *Bandit.* https://bandit.readthedocs.io/en/latest/
[4] Cron Authors. 2024. *Cron.* https://keda.sh/docs/2.13/scalers/cron/
[5] Istio Authors. 2024. Istio. https://istio.io/.
[6] Jmeter Authors. 2024. *Jmeter.* https://jmeter.apache.org/
[7] Poetry Authors. 2024. *Poetry.* https://python-poetry.org/
[8] Pylint Authors. 2024. *Pylint.* https://pypi.org/project/pylint/
[9] The Helm Authors. 2024. *Helm - The Kubernetes Package Manager.* https://helm.sh/
[10] The Kubernetes Authors. 2024. Kubernetes. https://kubernetes.io/.
[11] The Prometheus Authors. 2024. Prometheus. https://prometheus.io/.
[12] Eric Breck, Shanqing Cai, Eric Nielsen, Michael Salib, and D. Sculley. 2017. The ML Test Score: A Rubric for ML Production Readiness and Technical Debt Reduction. In *Proceedings of IEEE Big Data.*
[13] GitHub Contributors. 2024. *Checkout.* https://github.com/actions/checkout
[14] GitHub Contributors. 2024. *Docker Login.* https://github.com/docker/login-action
[15] GitHub Contributors. 2024. *Install Poetry.* https://github.com/snok/install-poetry
[16] GitHub Contributors. 2024. *Setup Python.* https://github.com/actions/setup-python
[17] DrivenData. 2024. *Cookie Cutter Data Science.* https://cookiecutter-data-science.drivendata.org/
[18] Grafana Authors. 2024. *Grafana Documentation.* https://grafana.com/docs/
[19] Mark Haakman and Haiyin Zhang. 2024. *Dslinter.* https://pypi.org/project/dslinter/
[20] HashiCorp. 2024. Vagrant. https://www.vagrantup.com/.
[21] HashiCorp. 2024. *Vagrant Documentation.* https://developer.hashicorp.com/vagrant/docs
[22] Istio Authors. 2024. *Istio Documentation.* https://istio.io/docs/
[23] Istio Authors. 2024. *Istio Documentation: Traffic Management - Mirroring.* https://istio.io/latest/docs/tasks/traffic-management/mirroring/
[24] K3s Project Authors. 2024. *K3s Documentation.* https://k3s.io/docs
[25] Kubernetes Authors. 2024. *Kubernetes Documentation: Deployments.* https://kubernetes.io/docs/concepts/workloads/controllers/deployment/
[26] Kubernetes Authors. 2024. *Kubernetes Documentation: Ingress.* https://kubernetes.io/docs/concepts/services-networking/ingress/
[27] Kubernetes Authors. 2024. *Kubernetes Documentation: Services.* https://kubernetes.io/docs/concepts/services-networking/service/
[28] Grafana Labs. 2024. Grafana. https://grafana.com/.
[29] Prometheus Authors. 2024. *Prometheus Documentation.* https://prometheus.io/docs/
[30] Inc. Red Hat. 2024. Ansible. https://www.ansible.com/.
[31] Zhiqiang Zhou, Chaoli Zhang, Lingna Ma, Jing Gu, Huajie Qian, Qingsong Wen, Liang Sun, Peng Li, and Zhimin Tang. 2023. AHPA: Adaptive Horizontal Pod Autoscaling Systems on Alibaba Cloud Container Service for Kubernetes. *Proceedings of the AAAI Conference on Artificial Intelligence* 37, 13 (Sep. 2023), 15621–15629. https://doi.org/10.1609/aaai.v37i13.26852