CropSegmentation: M6

1. M5 Improvements

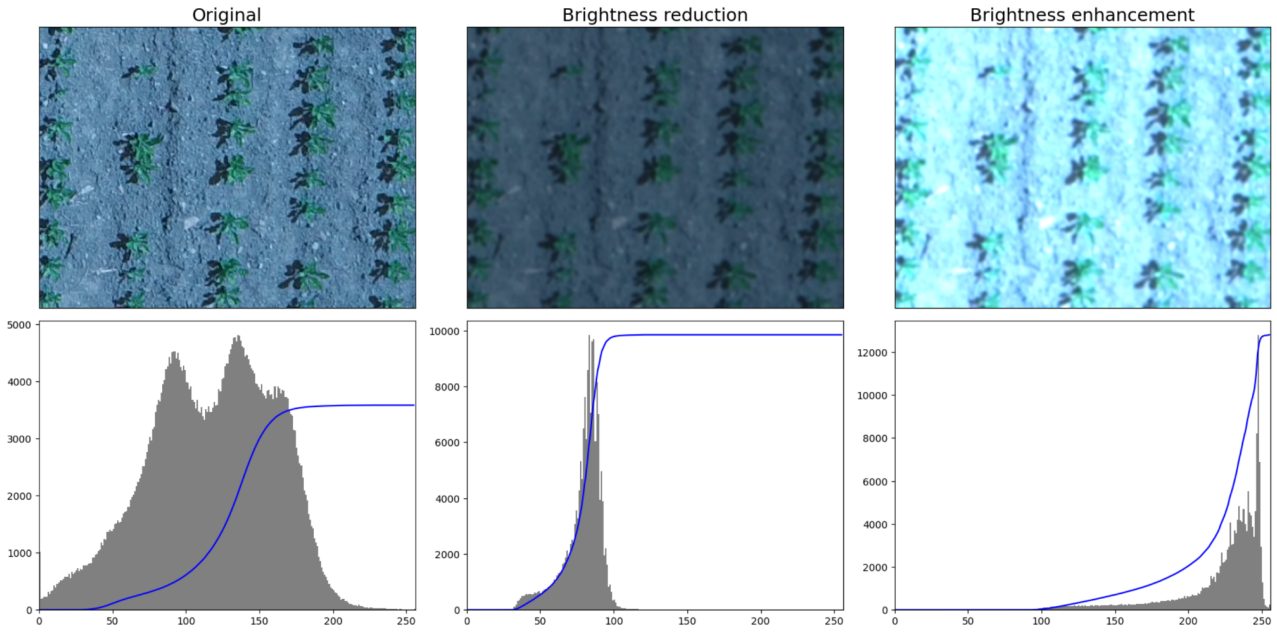
As recommended during the review of the previous milestone, branch merging prevention was applied in the case of Github Action failure for all the workflows implemented during the previous milestone. This was done by using a branch protection rule for the *main* branch of the repository.

2. Drift Analysis with Alibi Detect

Alibi Detect is a Python library that can be used for drift detection, which involves identifying when the distribution of the input data changes over time, which can negatively impact the performance of a machine learning model. In our project, we utilized Alibi Detect to analyze data drift, specifically focusing on scenarios where a change in the detection sensor, sensor performance degradation, or changes in lighting conditions could lead to variations between training and testing data. We considered two specific cases: one where there was a reduction in brightness, and another where there was an increase in brightness.

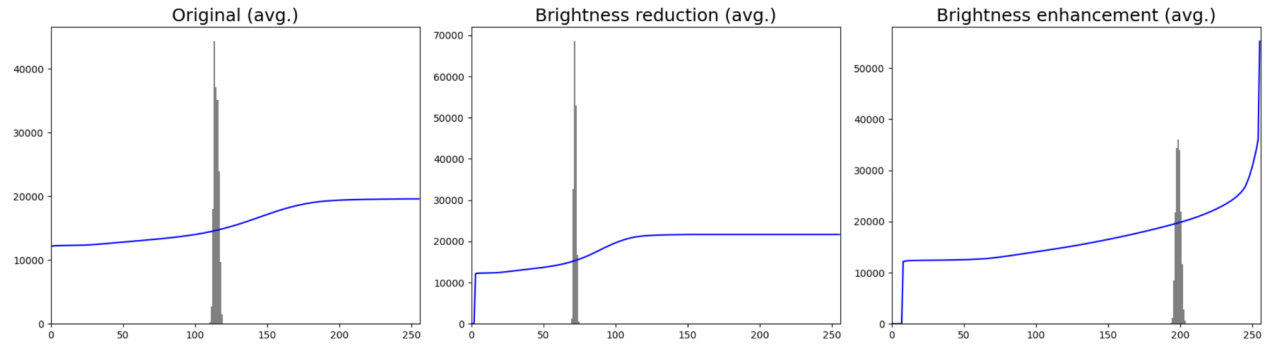
We randomly selected a subset of images from our training data and converted them to grayscale. For each image, we computed a histogram representing the tonal distribution of the image. We then created two sets of artificially distorted images, one with reduced brightness and one with increased brightness, and computed their grayscale histograms. The figure below shows an original image from the training data, randomly selected, and compared with those where an artificial distortion of brightness has been applied. The histogram as a numerical feature of the image is considered, representing the tonal distribution of a digital image by plotting the number of pixels per tonal value. The horizontal

axis of the graph represents tonal variations, while the vertical axis represents the number of pixels of that particular tone. Additionally, in this figure, we also present images showing brightness reduction and enhancement to visually represent these distortions alongside their respective histograms and cumulative distribution functions for a comprehensive analysis.



It is evident that a reduction in light intensity shifts the tonal spectrum towards lower values (closer to zero), while an increase pushes the distribution towards higher values (closer to 255).

We then applied the Kolmogorov-Smirnov test to detect data drift, using the average histogram values obtained from the entire selected set composed of 7100 images. The distribution shown in the figure below.



The Kolmogorov-Smirnov test produced the following values related to distance and p-value:

Property	Low images	High images
Test name	KSDrift	KSDrift
Threshold	0.01	0.01
Drift	Yes	Yes
Distance	0.3984375	0.3828125
P value	7.6135265e-19	2.2198778e-17

This analysis demonstrates the effectiveness of Alibi Detect in identifying and quantifying data drift in our project, providing valuable insights into the stability and reliability of our machine learning model under varying conditions. The visualizations and statistical tests allowed us to quantify the impact of brightness changes on the tonal distribution of our images, which is crucial for understanding how such changes could affect the performance of our model.

3. Drift Detection with Deepchecks

We used the checks provided by Deepchecks to measure the drift between training and test data. Namely, we measured how much the distribution of production data deviates from the training data and how this impacts the performance of the model. As done for Alibi Detect, a subset of images was altered by introducing Gaussian blur and randomly increasing or decreasing the brightness. Specifically, we performed the following checks on both the original data and the synthetic production data:

- **Prediction Drift**: This check detects prediction drift by using univariate measures on the prediction properties (samples per class, segment area, number of classes per image). Prediction drift is when drift occurs in the prediction itself. Calculating prediction drift is especially useful in cases in which labels are not available for the test dataset, and so a drift in the predictions is a direct indication that a change that happened in the data has affected the model's predictions. For the calculation of the drift score, the Kolmogorov-Smirnov test and Cramer's V were used. The **K–S test** determines whether or not an empirical distribution conforms to a theoretical distribution, or if there is a significant difference between data distributions. The K-S test checks whether the null hypothesis (which is that the two samples come from the same distribution) is true. So, this metric calculates how much the distributions of two data sets differ, together with a confidence score. If the confidence score is less than 0.05, the null hypothesis is rejected, indicating a model drift. The **Cramer's V** is a measure of association between two nominal variables, giving a value between [0,1], where 0 corresponds to no association between the variables and 1 to complete association. In the table below, it is possible to observe how the drift score increases up to 10 times on the production data.

	Segment Area (K-S)	#Classes per Image (K-S)	Samples per Class (Cramer's V)
Original	0.078	0.052	0.0
Production	0.468	0.56	0.165

- **Image Dataset Drift**: This check detects multivariate drift leveraging images properties (such as brightness, aspect ratio, RGB Relative Intensity). Image dataset drift is a drift that occurs in more than one image property at a time, and may even affect the relationships between those properties, which are undetectable by univariate drift methods. While no results are shown for the original data, implying that

no drift was detected, the drift score reached 0.366 on the production data. This indicates a significant shift in the underlying data distribution between the training and production datasets. The features that contributed most to the drift are shown: in our case, the brightness is at the base of the differences between the training and production datasets. This suggests that the alterations made to the images, such as the introduction of Gaussian blur and random changes in brightness, have had a substantial impact on the model's ability to make accurate predictions. The model was trained on a specific data distribution, and these alterations have caused the production data to deviate from this original distribution.

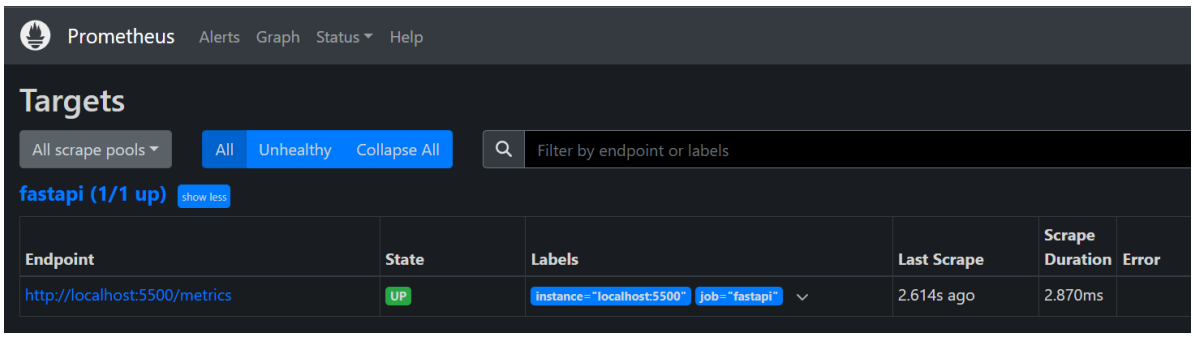
- **Image Property Drift**: This check detects image property drift by using univariate measures on each image property (brightness, aspect ratio, RGB Relative Intensity, etc.) separately. In this case, image drift is a data drift that occurs in images in the dataset. Deepchecks shows us that, while no drift is detected in original data, when it comes to production data, two drifted properties stand out: brightness, as we expected, with a drift score of 0.387, and RMS contrast (contrast of the image calculated by standard deviation of pixels), with a drift score of 0.312. Other image properties remained stable. This outcome, as expected, predictably reflected the brightness alterations made to the production images.

  In conclusion, the use of Deepchecks allowed us to quantify and understand the impact of data drift on our model's performance. This understanding is vital for maintaining the accuracy and reliability of our model in a production environment. It highlights the importance of continuous monitoring and adjustment in response to changes in the data distribution. This is particularly relevant in dynamic environments where data can change over time.

4. Performance Monitoring with Prometheus

Prometheus is an open-source systems monitoring and alerting toolkit and integrating it into our project allowed us to ensure the reliability and performance of our application; this tool provided insights into the performance and behavior of the application, allowing for thorough analysis. Below, an outline of the steps taken to integrate Prometheus monitoring into our project is provided.

1. **Prometheus Configuration**: The configuration file, *prometheus.yml*, serves as the starting point for establishing the monitoring framework. It sets the global scrape interval and external labels, laying the groundwork for the subsequent collection of metrics. The scrape_configs section defines the job for Prometheus, specifying the target as the FastAPI application running on localhost:5500. This configuration establishes the foundation for collecting metrics at regular intervals.
2. **Monitoring Instrumentation:** The *monitoring.py* file uses the prometheus_fastapi_instrumentator library to instrument the FastAPI application. The Instrumentator class is configured with options such as grouping status codes, ignoring untemplated paths, and instrumenting in-progress requests. Various metrics, including request and response sizes, latency, and total requests, are added using the metrics module. Additionally, a custom metric, segmentation_result, is introduced to track the outcome of image segmentation. This metric is associated with the /predict handler and records success and failure labels based on the X-image-segmentation-result header in the response.
3. **Integration in server.py**: The *server.py* file was updated to import the instrumentator from monitoring.py and incorporate it into the API. This was done by using the instrumentator.instrument(app) method, and the monitoring was excluded from the /metrics endpoint to prevent interference with the monitoring itself.



5. Data Visualization with Grafana

Grafana is a multi-platform open-source analytics and interactive visualization web application. In our project, it was integrated with Prometheus in order to improve the visualization of different monitoring metrics implemented with Prometheus. Following the installation guidelines provided by Grafana, we successfully established access to the Grafana web interface. One of the key features of Grafana is its ability to directly integrate with data sources, and in this case, Prometheus is configured as the primary data source.

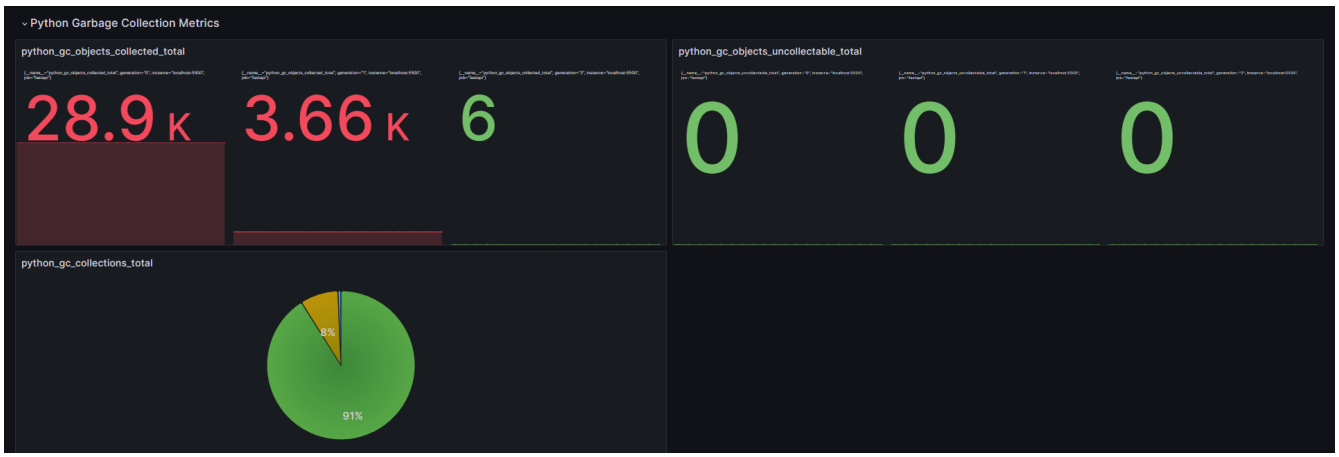
The URL and access settings were defined to establish a connection between Grafana and the Prometheus server to retrieve the metrics. All components, including our API (running at localhost:5500), Prometheus (running at localhost:9090), and Grafana (running at

localhost:3000), were deployed locally. Following this, we created the dashboard by adding panels that define visualizations of specific metrics to it in order to provide a clear and informative representation of the data. Subsequently, Prometheus queries were executed to fetch relevant metrics. In order to enhance the coherence and aesthetics of the dashboard, we organized the panels into three rows.

Below, we detail the specific Prometheus queries used to fetch relevant metrics for our Grafana dashboard, organized into three categories: Python Garbage Collection Metrics, Scrape Metrics, and Image Segmentation Metrics.

Queries related to the Python garbage Collection Metrics row:

- python_gc_objects_collected_total: total number of objects collected by the Python Garbage Collector;
- python_gc_objects_uncollectable_total: total number of uncollectable objects identified by the Python Garbage Collector;
- python_gc_collections_total: total number of garbage collection collections performed by the Python Garbage Collector.



Queries related to the Scrape Metrics row:

- scrape_duration_seconds: time spent by Prometheus to collect metrics from the target;

- scrape_samples_post_metric_relabeling: number of samples obtained after metric relabeling during a Prometheus scrape;
- scrape_samples_scraped: total number of samples scraped by Prometheus during scrape operation;
- scrape_series_added: number of time series added by Prometheus during a scrape.

The dashboard was saved as a json file and uploaded to the *dashboards* folder in the project repository.

6. Cloud Deployment on Okteto

For the deployment on Okteto, we used the same Docker Compose configuration as in the previous milestone. Okteto is capable of recognizing Docker Compose configurations and converting them into a Kubernetes configuration file, which it uses behind the scenes for the creation and deployment of the necessary services. To accomplish this, it was sufficient to log in via the command line to an enabled account and launch the command ~ *okteto up*. It is also possible to directly import the repository via the Okteto dashboard, provided the necessary permissions are granted. In our case, two pods were created, one for the frontend application and one for the backend application. These were published through the Okteto SaaS service at the following addresses:

- https://app-albertovalerio.cloud.okteto.net/
- https://api-albertovalerio.cloud.okteto.net/

Furthermore, to optimize storage (limited to 5GB in the free version), a *.stignore* file was defined. In this file, only the paths strictly necessary for the operation of the services were enabled, not the entire volume of images managed by the model during preprocessing and training.

7. Monitoring with Better Uptime

Better Uptime is a monitoring platform that provides real-time alerts and status pages, ensuring that teams are always informed about the health and performance of their systems. We incorporated monitoring with Better Uptime into our project by creating an account on Better Uptime's platform and entering our website URL into the Better Uptime dashboard to initiate the monitoring process. This allowed Better Uptime to start tracking the uptime and performance of our website.

The following features of Better Uptime have been utilized in this project:

- **Uptime Monitoring:** Better Uptime checks the availability of our website every minute from multiple locations around the globe. This helps us ensure that our website is accessible to users worldwide.
- **Alerts:** We have configured Better Uptime to send alerts via email whenever the website is down. This allows us to respond quickly to any issues.
- **Status Page:** We have set up a public status page with Better Uptime and added it to the project's README page on github. This page provides our users with real-time information about the operational status of our website.

In particular, the Status Page serves as a comprehensive platform for users to stay informed about the website's status. It features three tabs:

1. **Status:** Provides real-time updates about the operational status of the website.
2. **Maintenance:** Informs users about any ongoing or scheduled maintenance activities.
3. **Previous Incidents:** Offers a historical record of past incidents, providing transparency about the website's performance and uptime.

Overall, the integration with Better Uptime is a significant aspect of this project as it offers essential tools for monitoring the website's uptime and performance, contributing to the overall user experience.

8. Load Testing with Locust

Locust is an open-source load testing tool used to evaluate the performance of web applications under stress. It simulates users interacting with the application to identify potential bottlenecks and performance issues, ensuring that the system remains responsive and stable under heavy load.

Five different tasks were implemented:

1. **Main Endpoint Test:** Verifies the server's ability to handle requests to the main endpoint.
2. **Get Samples Test:** Assesses the functionality of fetching a limited number of images.
3. **Upload Image Test:** Tests the server's capacity to handle image uploads.
4. **Predict Image Test:** Focuses on the prediction functionality of the application.
5. **Get Metrics Test:** Evaluates the server's performance in computing and returning metrics.

Different weights were assigned to each task. Assigning weight arguments to tasks in a Locust load test script is an important step for realistically simulating user behavior, since the weight determines how often a particular task is picked relative to other tasks.

- The **main_endpoint** task was assigned a weight of **1**. The main endpoint (/) is typically the entry point of an application and is likely to be hit frequently. However, it might not be as frequently accessed as specific functionality endpoints once a user is actively engaged with the application. Assigning a lower weight makes sense because, after initial access, users will likely navigate to more specific parts of the application.
- The **get_samples** task was assigned a weight of **3**. This task fetches images from the server and can be considered a common action if users regularly view different images. The higher weight compared to the main endpoint reflects the assumption that once users are in the application, browsing images is a more frequent activity.
- The **upload_image** task was assigned a weight of **2**. Image upload is a critical but potentially less frequent operation than viewing images. Users might spend more time

browsing or analyzing images than uploading new ones. Thus, this task should have a moderate weight, indicating it's a regular but not the most common activity.

- The **predict_image** task was assigned a weight of **2**. Making a prediction is a core functionality of the application, but might not be used as often as simply viewing images. It's reasonable to assume that for every few images viewed, a prediction might be made on one of them. A weight of 2 strikes a balance between frequent and occasional use.
- The **get_metrics** task was assigned a weight of **1**. Requesting metrics might be less frequent compared to other tasks like viewing or uploading images. It's often an action taken after several other interactions (like uploading or predicting images). Therefore, a lower weight is justified as it's likely an action taken occasionally, perhaps for detailed analysis or review of a prediction.

The implemented locustfile also has the following characteristics:

- **Error Handling:** The locustfile was designed with robust error handling, ensuring that any non-200 status codes or unexpected responses from the server triggered explicit error messages. This feature is critical for quickly identifying and addressing issues during the testing phase.
- **Random Image Selection in Tasks:** To enhance the robustness of our load tests, some tasks in the locustfile were designed to select images randomly. This approach prevented the caching mechanism from skewing the test results, providing a more accurate representation of the server's performance under varied and unpredictable conditions.

We devised four different load tests in order to test the system under different conditions:

1. **Baseline Test:** small number of users (5) with a low spawn rate (1 user per second) for a moderate duration (10 minutes). This test is useful to ensure that the system is working correctly and can handle a minimal amount of traffic. It can help identify any glaring issues before starting more intensive testing.

2. **Moderate Load Test:** moderate number of users (20) with a moderate spawn rate (2 users per second) for a longer duration (30 minutes). This test is designed to simulate a hypothesized typical load on the system. It can help understand how the system performs under normal conditions and identify any performance issues that might affect the user experience.
3. **High Load Test:** high number of users (50) with a high spawn rate (5 users per second) for a longer duration (30 minutes). This test is designed to stress the system and see how it performs under heavy load. It can help identify any performance bottlenecks and understand how the system behaves when it's close to its capacity.
4. **Longevity Test:** moderate number of users (20) with a moderate spawn rate (2 users per second) for a very long duration (2 hours). This test is designed to check for issues that might only become apparent over time. It can help ensure that the system is stable and can handle a sustained load for a long period.

The different load testing configurations are summarized in the table below:

Test	Number of users	Spawn rate	Duration
Baseline Test	5	1	10 minutes
Moderate Load Test	20	2	30 minutes
High Load Test	50	5	30 minutes
Longevity Test	20	2	2 hours

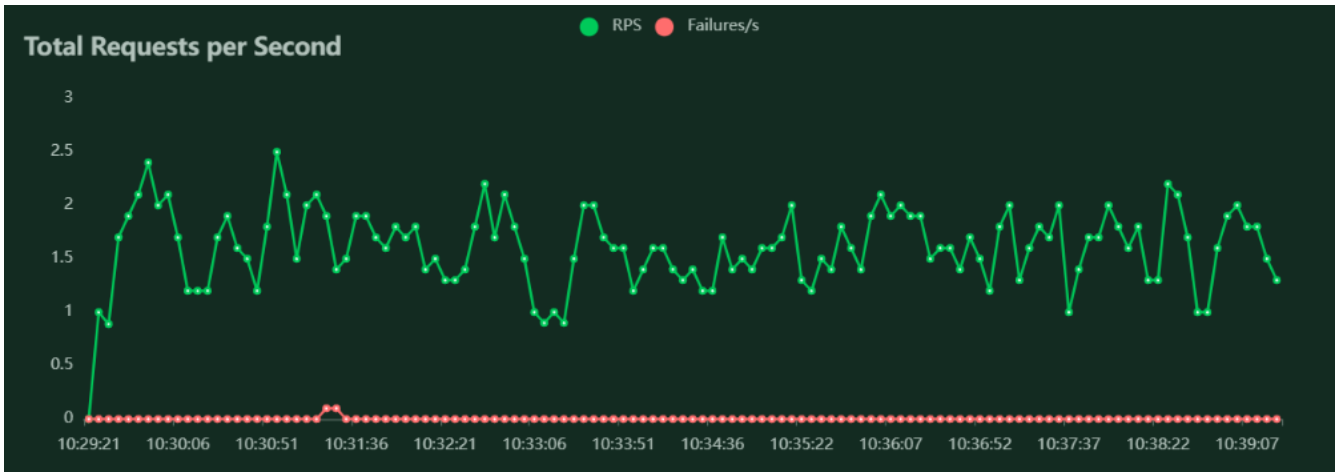
During the first run of the baseline test, we encountered a high failure rate. After analyzing the failures encountered by Locust, we decided to partially refactor our server code in order to improve the server's efficiency and its ability to handle multiple users. Two main improvements were made:

- **Image Saving Optimization:** In the previous version of the server code, all images were read into memory and then saved to the TEMP_PATH. In the new version, each

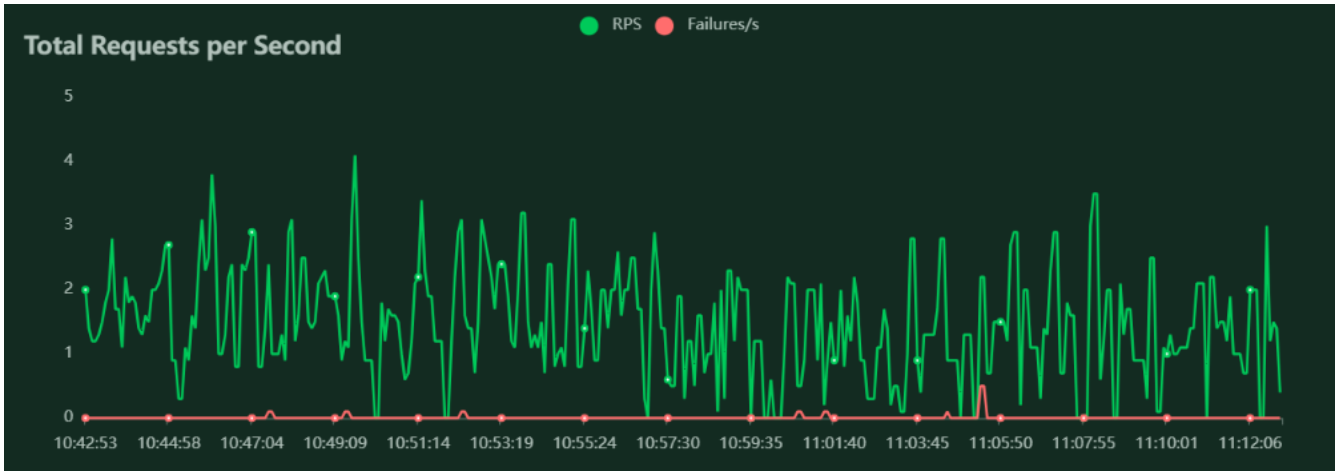
image is checked if it already exists in the TEMP_PATH. If it doesn't, only then is it read into memory and saved. This reduces unnecessary I/O operations and memory usage, especially when dealing with a large number of images.

- **File Upload Improvement:** In the previous version, every uploaded file was saved with the same name 'upload.jpg', which could cause issues when multiple users are uploading files simultaneously. In the new version, a counter is added to the filename if a file with the same name already exists in the TEMP_PATH. This ensures that every uploaded file has a unique name, thus preventing any potential file conflicts.

We were able to verify that implementing these optimizations was beneficial when we re-ran the baseline test after these changes and obtained a rounded 0% failure rate, with only a single failure out of 981 requests.

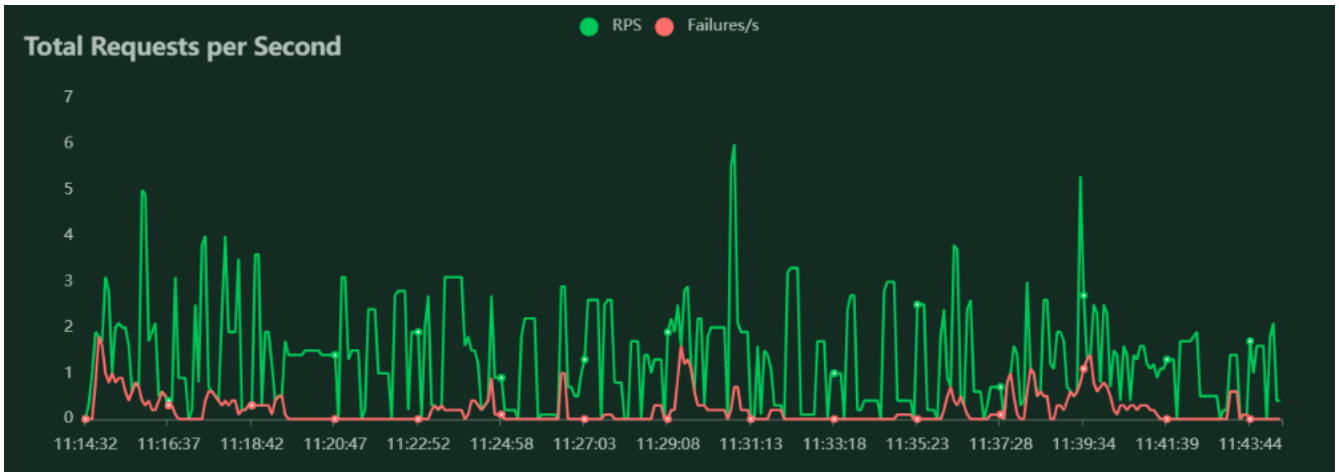


Following this, we performed the moderate load test, which resulted in a rounded failure rate of 0%, with 11 failures out of a total of 2873 requests.

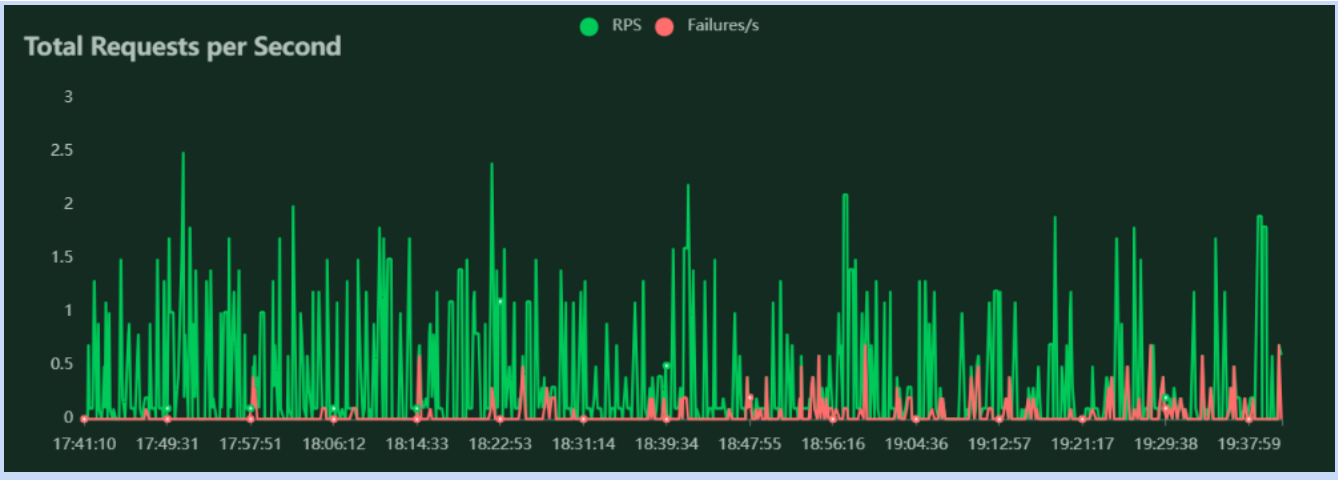


The number of failures is obviously higher than for the baseline test, but this is expected given the quadruple number of users, the double spawn rate and the triple testing time.

The most intensive test that we conducted was the high load test, with 50 users and a spawn rate of 5. During this test, a 14% failure rate was detected. This is much higher than the previous two tests, but is still a good result if we take into consideration the high number of users and the high spawn rate, in addition to the fact that the system is hosted using Okteto's free trial.



Lastly, the longevity test was conducted. During this test, a 6% failure rate was detected, which is to be expected given the long duration of the test and showcases generally satisfactory performance.



The results of the conducted tests are summarized in the table below:

Test	Users	Spawn rate	Duration	Failure rate
Baseline Test	5	1	10 minutes	0%
Moderate Load Test	20	2	30 minutes	0%
High Load Test	50	5	30 minutes	14%
Longevity Test	20	2	2 hours	6%

As can be seen from the results of the conducted tests, the system performed very well during the baseline test and the moderate load test and moderately well during the longevity test, signifying that it works correctly and that it can perform well under normal conditions without any significant performance issues for an extended period of time. The high load test, with its lower performance, was useful for understanding how the system behaves when it’s close to its capacity.

Performing load testing of our system helped us get an idea of our system’s capacity for handling concurrent users and determine the performance threshold under varying load

conditions. This information is important for optimizing our system for peak performance and ensuring a seamless user experience, even during periods of high traffic. It also aids in identifying potential bottlenecks and areas that require further improvement or scaling.
The results of the conducted tests were uploaded to the `reports/locust` folder of our repository, including the Locust HTML report and the CSV files reporting requests, failures, and exceptions for each test.

9. M6 Improvements

As recommended during the review of the work completed for this milestone, we implemented an automated workflow using GitHub Actions that periodically checks for data drift using the Alibi Detect library. The script first extracts a vector representation from the images (as described in point 2. Drift Analysis with Alibi), then saves this numerical representation to a timestamped file for comparison over time. In addition, a log file is generated at each execution, indicating whether the check was successful or if data drift was detected. This log file, in addition to being downloadable as a workflow artifact, is pushed into the repository to maintain a temporal trace of all data checks performed. Finally, if data drift is detected, the workflow is forced to terminate with an error (failure), to immediately highlight the presence of a problem that requires further attention and checks.