You pushed to main just now

Merge branch 'development' of gitlab.cs.mcgill.ca:comp585_2023f/team-4 into development
Varun Shiri authored 7 minutes ago

M2-report.md 31.28 KiB

Offline evaluation

Evaluation of the performance of the model

To evaluate the performance of the collaborative filtering model, we divided the dataset containing the ratings into two parts:

- A test dataset that corresponds to 20% of all ratings we have collected.
- A train dataset that corresponds to 80% of all ratings we have collected. The model is then trained only on the train dataset. Then, we predict a rating with the model for each of the (user_id, movie_id) pairs present in the test dataset, and we calculate the Root Mean Squared Error between the list of predictions of the model and the list of real ratings present in the test dataset. Because the model was not trained on the data from the test dataset, we are know that RMSE is not biased by overfitting the test data.

The Root Mean Squared Error measures the average difference between values predicted by a model and the actual values. We choose this performance indicators because it permits to know the average error that is made on predictions in general, which seemed to us to be the most relevant indicator. For example, this value can then be compared to the standard deviation of the rating dataset, which allows us to compare our predictor to a predictor which would always predict the average of the ratings.

With our ratings dataset we have a Root Mean Squared Error of: 0.7215937657284225

Unit tests for offline evaluation

We also coded unit tests allowing us to see if the recommendations seemed at least coherent and which also allows us to test the other filters. For these unit tests, we created our own datasets, much smaller and simpler than the real datasets. Creating these small datasets allows us to have a better understanding of what is happening and to be able to more easily test simple cases.

In these unit tests, we test:

- If a person under the age of 18, no film marked as "for adults" has been recommended to them (test for the adult filter)
- Verify that the recommendation includes 20 movies
- Verify that movies already rated by a user are not recommended to him/her again.
- Verify that a new user have also a list of 20 movies when he asks for recommendation.

We also test some functions used for our model in the unit tests:

- Verify that the model saves in the correct file.
- Verify that our helping function that print the results of the recommendation prints the result correctly.
- Verify that we load the entire dataset by checking the size.

Links to the implementation:

Separation of the dataset into two parts and testing: https://gitlab.cs.mcgill.ca/comp585_2023f/team-4/-/blob/development/app/model/movie_rec.py Unit tests and print the RMSE: https://gitlab.cs.mcgill.ca/comp585_2023f/team-4/-/blob/development/app/tests/model/test_model.py

Online Evaluation

We will determine the performance of our system while it is deployed based on the following factors:

- online evaluation metric (which we will define and explain below)
- response time of the API

Online evaluation metric

Usually, online evaluation involves A/B tests which means there should be another recommendation model to compare our recommendation with (as we do not have access to ground truth). But in our case, instead of contrasting and comparing our recommendations with another model, we instead looked at the logs and the ratings from the user in real time. Our rationale was something along the lines of the phenomenon observed while watching youtube. Say, the user is watching some video and he/she gets a bunch of recommendations and they may also get asked "what do you think about this recommendation?". Borrowing from that idea we thought, what if we look at the logs in real time to check for the ratings and latest watch history of the user.

So our online evaluation works like this: We look at the latest ratings provided by a user. We also collect the data on what proportion of the movie is watched. Then we compare them with what our recommendation is and calculate a score for each recommendation. Lastly, we record the response time to get this recommendation and record this to track the performance across a random sample of users.

This approach makes sure that we make use of the telemetry information provided from the logs (in real systems, we presume there will more details available in the telemetry data including user behavior on the website, which is not present in the available logs. But we do have information on watchtime and ratings which we use.)

Telemetry collection system: How did we work with real time data?

Our team initially considered the use of a standard database along with Python scripts to collect, process and store data from the Kafka stream in different formats. However, while exploring tools for our monitoring infrastructure, we came across Grafana Loki, an efficient log aggregation system inspired by Prometheus. Quoting from their description, "It is designed to be very cost effective and easy to operate. It does not index the contents of the logs, but rather a set of labels for each log stream." [1] Considering the volume of data and the experiences of our peers shared during the M1 presentation, we decided to use Loki as our logs database. Loki seemed promising as it would help us evade both the problem of quickly running out space on our server with a local database or running out of limited free storage on a cloud database. We used the default Loki storage option, where logs are compressed and stored in chunks in the file system. The logs are rotated weekly.

To send the logs from our Kafka stream to Grafana Loki, we used the agent Promtail. We also parsed and processed the logs with the help of Promtail. We defined four categories of logs: recommendation (successful recommendation requests with status 200), error (failed recommendation requests with status 0), history (records of the users' watch history, i.e. the GET /data endpoint) and rating (records of users' ratings for different movies, i.e. the GET /rate endpoint). Each log line was matched to a regular expression to identify its category. Named groups pertaining to important details were extracted from the log line and processed to create the final log to be sent to Loki in different formats depending on the category. (For example, *timestamp, userid, movied, minute_watched* for history and *timestamp, userid, error, error_msg, response_time* for error). An example can be found here. This categorization of logs made it easy for us to fetch only the logs of interest during online evaluation and monitoring. Initially, we were using these tools for log collection and monitoring but we discovered that they can be exposed over an API to gain access to the log data so that it can be used as dataset that would be updated in real time from the Kafka stream. So, we wrote a wrapper for the above mentioned tools to send API requests with the queries we wanted to obtain the results for our online processing

Calculation involved?

Scoring recommendation list

- If the sample-movie lies in the top 5: score = 10/10
- If the sample-movie lies in the top 6-10: score = 8/10
- If the sample-movie lies in the top 11-15: score = 5/10
- If the sample-movie lies in the top 16-20: score = 3/10
- If the sample-movie does nit lie in the recommended list: score = 0.01/10

Scoring ratings

The users would have given some rating to the movie: (score/5)

Scoring runtime

Based on how much of the movie the user has watched.

```
[0-20%):    1/5
[20-50%):   3/5
[50-70%):   4/5
[70-100%]:  5/5
```

Final calculation

We wanted to capture how the user is perceiving our recommendations: i) Depending on how much of the movie was watched: Recommended_list_score x runtime_score ii) what rating was given: Recommended_list_score x Ratings_score

(i+ii): predicted movie scaled score to tell us appropriate the recommendation was.

What does the metric mean? How to interpret it?

Care was taken to ensure that we do not end up with zero values while multiplying the scores. We capture the following trends in our metric:

- How highly ranked was the sample-movie in the list of recommended movies or if the sample-movie was even part of the recommendation at all?
- What did the user feel about the movie? (captured by the rating they gave to the movie)
- Apart from feeling, how did the user actually act for that movie? (captured based on how much of the movie did they actually end up watching)
- How much time did we take to give our recommendation?

Special note

Originally we spent quite a bit of time to calculate the score for online evaluations as follows:

We looked at the latest ratings provided by the user. For that user and the movie combination, we looked at how much of the movie was watched by the user. Then we compared it with our recommendation. If the movie which the user rated high and watched a lot of was indeed part of our recommendation, then our recommendation was good.

This would have given us a single scaled score as follows: Recommended_list_score x Ratings_score x runtime_score = predicted movie scaled score to tell us appropriiate the recommendation was.

But **the problem** we faced with this approach was that we were limited by the LOKI API to get the result set for our queries to ratings and history view. This meant we could not access more than 5000 records at a time through the request in our code but we had the access to full history through directly getting the values in the grafana dashboard. The problem was that a common movie and user combination did not exist in the ratings and history view for the 5000 records we had access to in a single API request. We did not have the time to write a selenium script to bypass their restrictive limit. It is possible that since we are using free which is why we had this limitation. To overcome this, we changed our evaluation strategy such that it could work with rolling logs as described above.

Links to artifacts

`Online evaluation based on history`

`Online evaluation based on ranking`

Data Quality

What makes data "high quality" in our context?

Common dimensions of data quality include accuracy, completeness, consistency, reliability, and timeliness.

Our pipeline for data was as follows:

- Save the logs to flat files.
- Preprocess the flat files to clean data collected. Inspection was done to identify any obvious issues with the data collected. We found missing values and spurious entries. We dropped all those records.
- Filter and segregate data collected according to the type of data files. Convert them to csv files.
- The user data was scrapped from the simulator API for 1 million users.
- Movie attributes were scrapped from the API for the items obtained from kafka logs.
    - As an enhancement, we loaded the movie attributes and imdb ratings from external sources too for M2. This was done using the "tmdb_id" attribute we had for each movie

    #### how can you use the external imdb files?

    The movieids from the simulator comprise of the name of the movie and the year separated by '+' symbol. Movie attributes returned by the simulator contain a mixture of data from imdb and tmdb datasets. There are unique ids corresponding to each of these datasets in the response. We want to use the imdb dataset as an external source and found that the name of the movie corresponds to original column of the imdb dataset and the year to endyear column. Through this we can obtain the imdb_id which can be used to join to other tables from imdb to get more information for the movies.

    - From the kafka logs, sometimes we would get made up movie details. Since it was mentioned in the requirements of the project that the simulator is using imdb movie database to generate logs, we went directly to the external source to avoid such spurious logs.
- Kafka logs are responsible for giving watch history, telemetry and ratings by each user (whether the liked the movie or not). We loaded the kafka logs to grafana monitoring. Further, we found that grafana can be used a database to serve the logs in response to API queries. So to retrieve real time information about the system for online evaluation, we added processing at the destination endpoint of those API queries to get filtered data outputs.

Data processing at different end points

One notable aspect was the handling of data at end points. Since we have parsed logs to loki and were accessing them through grafana, we made sure to take care of duplicates and spurious values when retrieving data through those end points.

Data quality reports

We generated reports for each of the data set used for training the model. In the report we tracked missing cells, duplicate rows, memory size, variable types, distinct values, data range and correlation among attributes. Following are the screenshots from one of our data reports:

Profiling Report: cleaned_rating.csv — Overview | Variables | Interactions | Correlations | Missing values | Sample

Overview

Overview | Reproduction

Dataset statistics

Number of variables	3
Number of observations	21398
Missing cells	0
Missing cells (%)	0.0%
Duplicate rows	0
Duplicate rows (%)	0.0%
Total size in memory	501.6 KiB
Average record size in memory	24.0 B

Variable types

Numeric	1
Text	1
Categorical	1

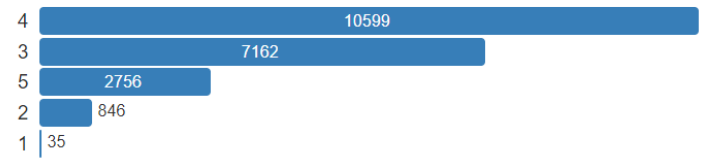
MovieName
Text

Distinct	10430
Distinct (%)	48.7%
Missing	0
Missing (%)	0.0%
Memory size	167.3 KiB

More details

Rating
Categorical

Distinct	5
Distinct (%)	< 0.1%
Missing	0
Missing (%)	0.0%
Memory size	167.3 KiB



More details

Overview | Categories | Words | Characters

Length

Max length	1
Median length	1
Mean length	1
Min length	1

Characters and Unicode

Total characters	21398
Distinct characters	5
Distinct categories	1
Distinct scripts	1
Distinct blocks	1

The Unicode Standard assigns character properties to each code point, which can be used to analyse textual variables.

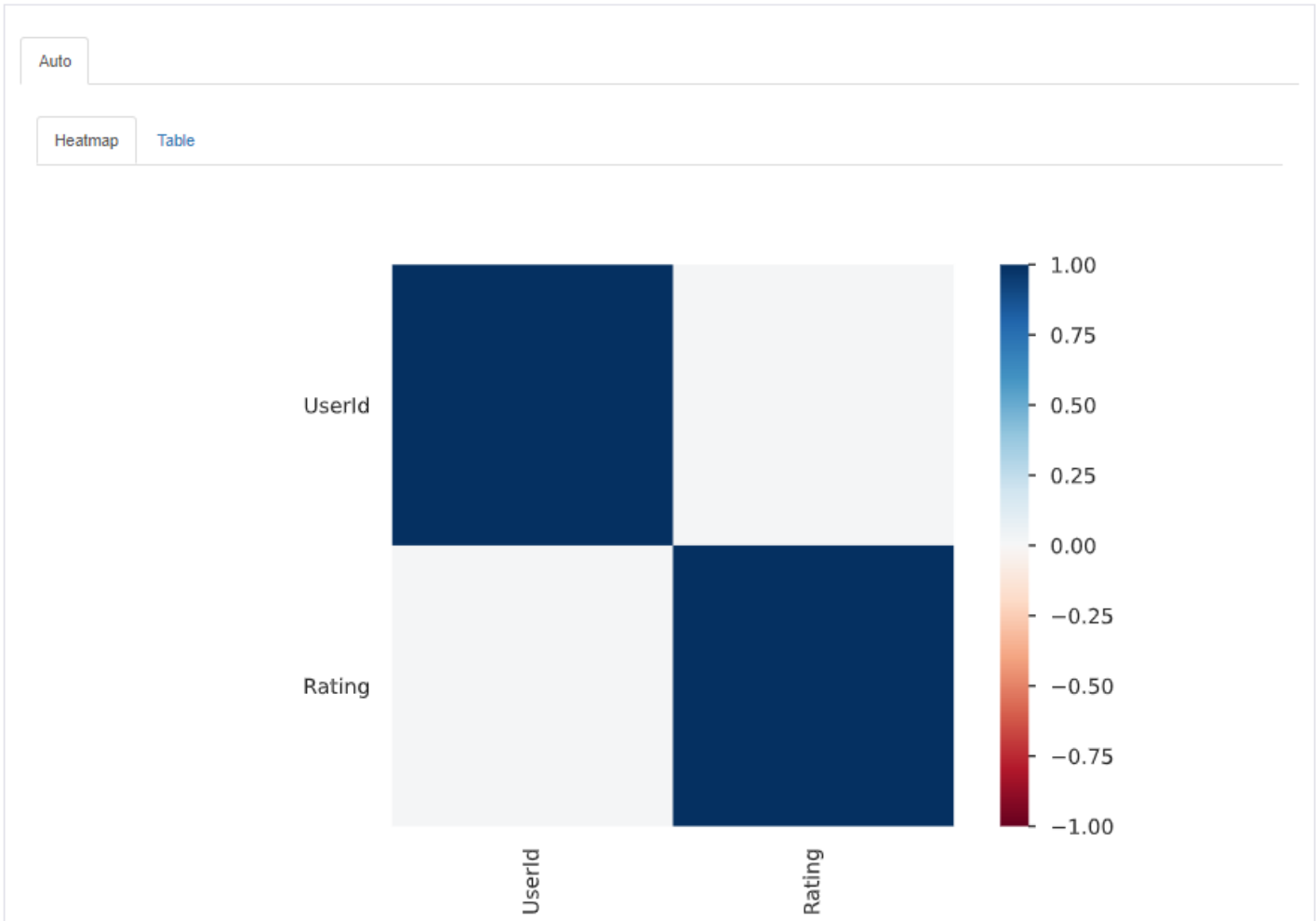
Unique

Unique	0
Unique (%)	0.0%

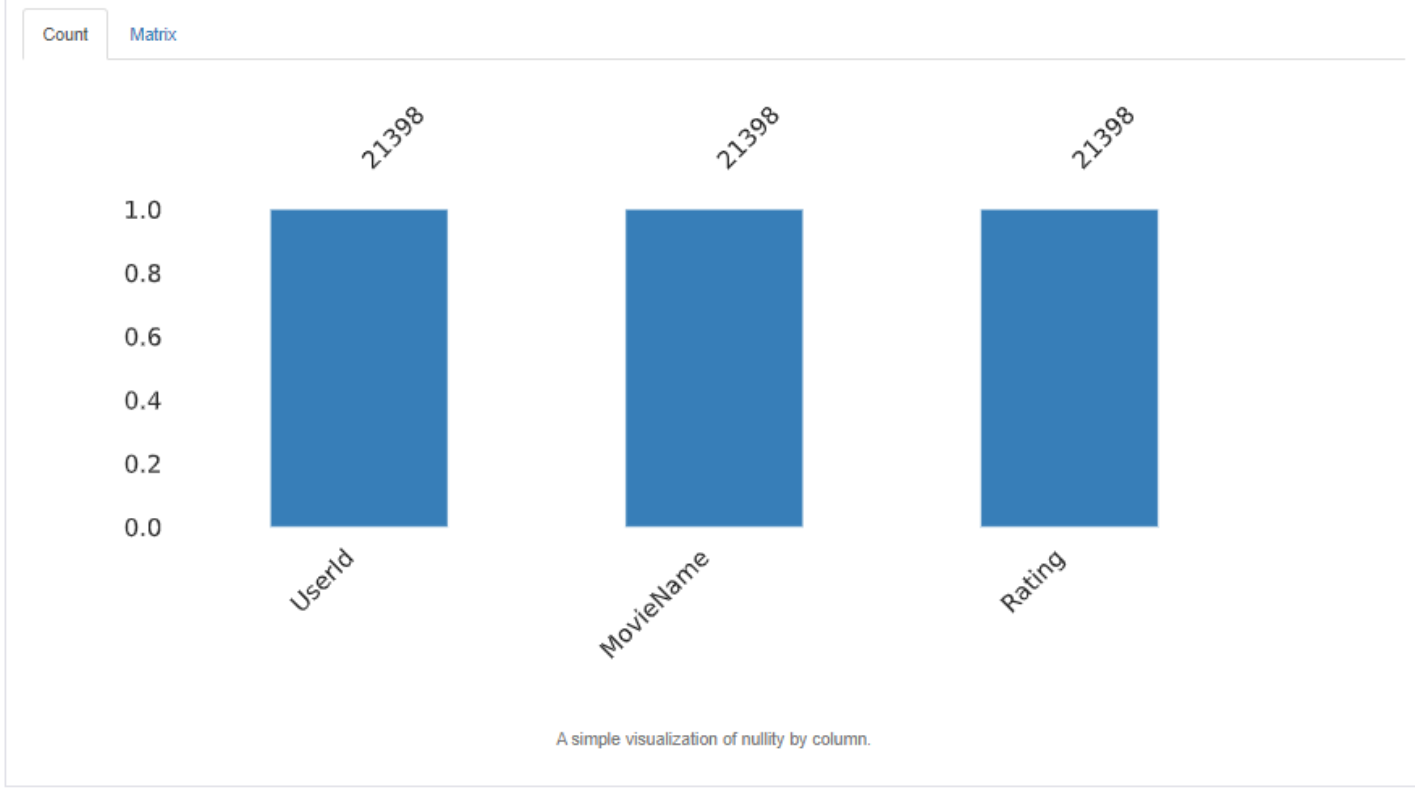
Sample

1st row	4
2nd row	4
3rd row	5
4th row	4
5th row	4

Correlations



Missing values



Links to artifacts

`loki scrapper remove duplication at end point`

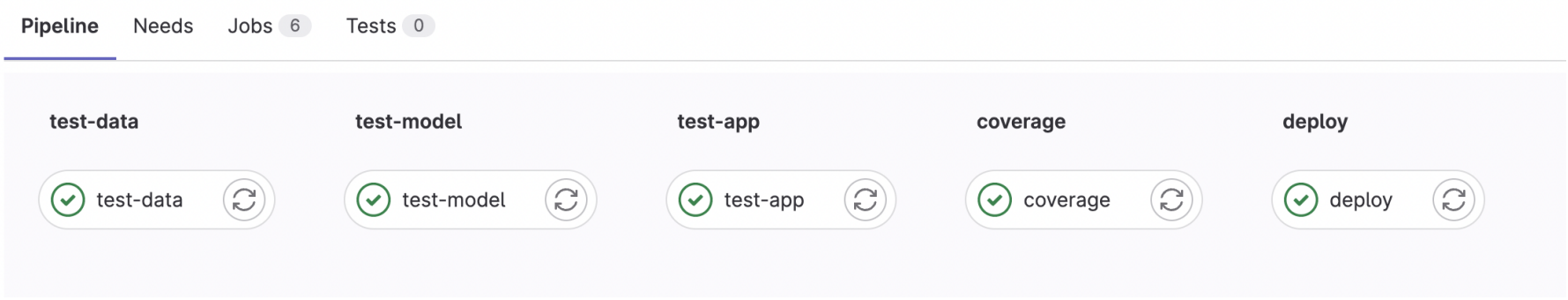
data processing scripts

`end point data processing`

Pipeline:

We implemented an MLOps to get constant feedback during the development process and make sure that our code works seamlessly across different modules (data, ML model, app, etc.).

Pipeline structure:



We configured our pipeline to sequentially run the following different jobs/stages:

- test-data
- test-model
- test-app
- coverage
- deploy

Description of each stage:

- **test-data:** This pipeline runs tests on the data quality and data processing ensuring that the data which is being fed into the model is of good quality.
  - It runs tests like checking if the movies are getting filtered, curl requests are working etc.
  - Works on the following branches: main, development
- **test-model:** This pipeline runs tests on the model and ensures that functionalities like printing recommendations, test-train split, global dataset values etc. are working fine. Based on these tests we get a good benchmark of our model performance.
  - Works on the following branches: main, development.
- **test-app:** This pipeline runs tests on the flask app and ensures that functionalities like testing the endpoint of the flask app and other functions of the API are working fine.
  - Works on the following branches: main, development
- **coverage:** We implemented a pipeline for the code coverage which tells us how much code of the app has been tested by the tests that we developed.
  - This pipeline combines the results of all our tests described previously (data, model, app) and returns the final code coverage report. Currently our test coverage is at 91%.
  - Works on the following branches: main, development
  - Link to our coverage report: https://gitlab.cs.mcgill.ca/comp585_2023f/team-4/-/jobs/3699

Screenshot of our coverage report:

```
$ python3 -m coverage report
Name                                                    Stmts   Miss  Cover
---------------------------------------------------------------------------
__init__.py                                                 0      0   100%
data_processing_scripts/__init__.py                         0      0   100%
data_processing_scripts/fetch_movie_details.py             13      0   100%
data_processing_scripts/filter_movie_details.py            33      5    85%
data_processing_scripts/process_history.py                 21      0   100%
data_processing_scripts/process_rating.py                  27      6    78%
model/__init__.py                                           0      0   100%
model/movie_rec.py                                        110     18    84%
tests/__init__.py                                           0      0   100%
tests/data_processing/__init__.py                           0      0   100%
tests/data_processing/test_data_extraction.py               7      0   100%
tests/data_processing/test_data_preprocessing.py           28      0   100%
tests/model/test_model.py                                  74      0   100%
---------------------------------------------------------------------------
TOTAL                                                     313     29    91%
```

- **deploy:** After all tests have passed, this final pipeline deploys our model and the app to our team server hosted on GitLab. Works on the following branches: main
  - We set this job to work on only the `main` branch and not `development` as we don't want to deploy a new model each time a new commit is pushed to development which is our final testing branch. Only when we are confident of our code we push to the `main` branch which deploys the model.

Reason why our testing is adequate:
As described above, we have tested the most crucial aspects of a machine learning model pipeline i.e. the data quality and the model performance. Additionally we have also tested the app interaction which makes our end-to-end test suite robust.

Links to our test suite and other test files:

- https://gitlab.cs.mcgill.ca/comp585_2023f/team-4/-/tree/development/app/tests
- https://gitlab.cs.mcgill.ca/comp585_2023f/team-4/-/blob/development/app/test_app.py

Reason for making the jobs sequential: We wanted to make sure that in case any of the jobs like `test-dat`, `test-model` and `test-app` fails, we don't proceed forward unless we have fixed the issue concerned with the failing pipeline.

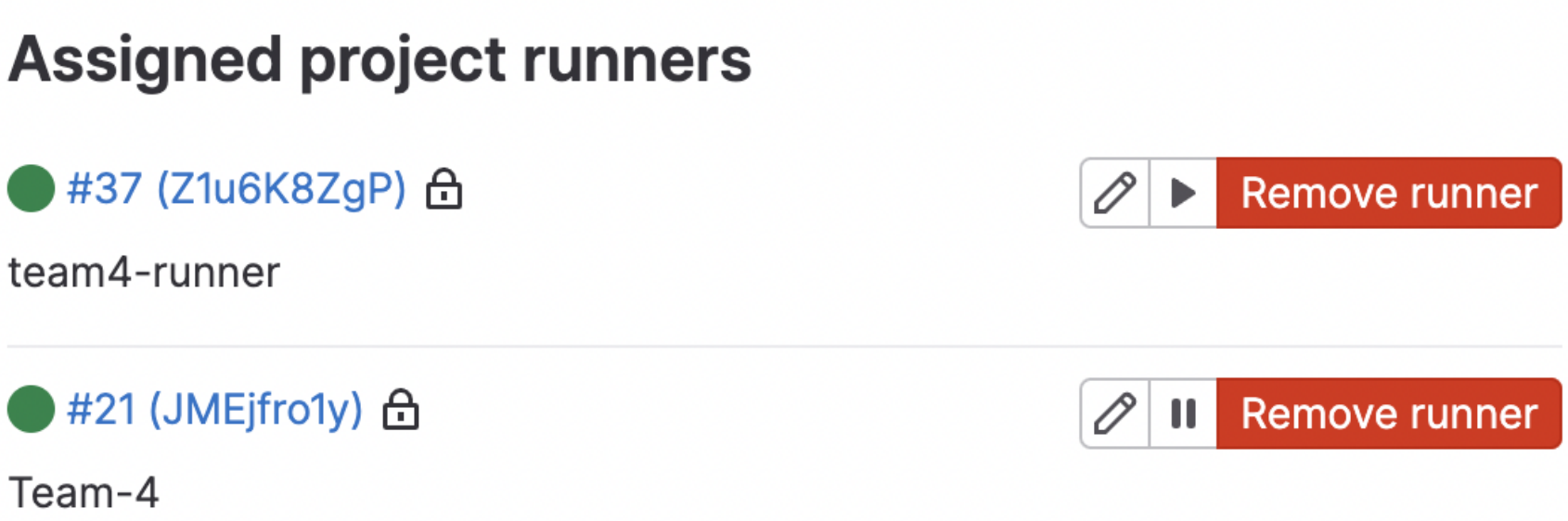
Continuous integration

In addition to the various stages of the pipeline as described above, we also focussed on the following:

1. Infrastructure:

We have configured our systems to act as GitLab runners. This allows for efficient and rapid execution of CI/CD jobs in a Doker environment, ensuring that our code integrations are validated in real-time. By hosting our runners, we could manage the CI process to match our project's requirements.

Following is a screenshot of our runners:



2. Automated Model Testing:

The model testing phase not only evaluates the model's correctness but also benchmarks its performance. This ensures that the model not only produces the right results but also operates within the expected time and resource constraints.

Service: The CI process is integrated within our GitLab repository.
To access the platform and monitor the CI jobs, please refer to our GitLab repository URL:
https://gitlab.cs.mcgill.ca/comp585_2023f/team-4/-/settings/ci_cd

Unit tests for data quality

We have done some unit test to be sure that the data is consistent. In these unit tests, we test:

- If all the user_ids in the ratings dataset are existing user_ids in the users dataset
- If all the movie_ids in the ratings dataset are existing movie_ids in the users dataset Link to these unit tests: https://gitlab.cs.mcgill.ca/comp585_2023f/team-4/-/blob/development/app/tests/model/test_model.py

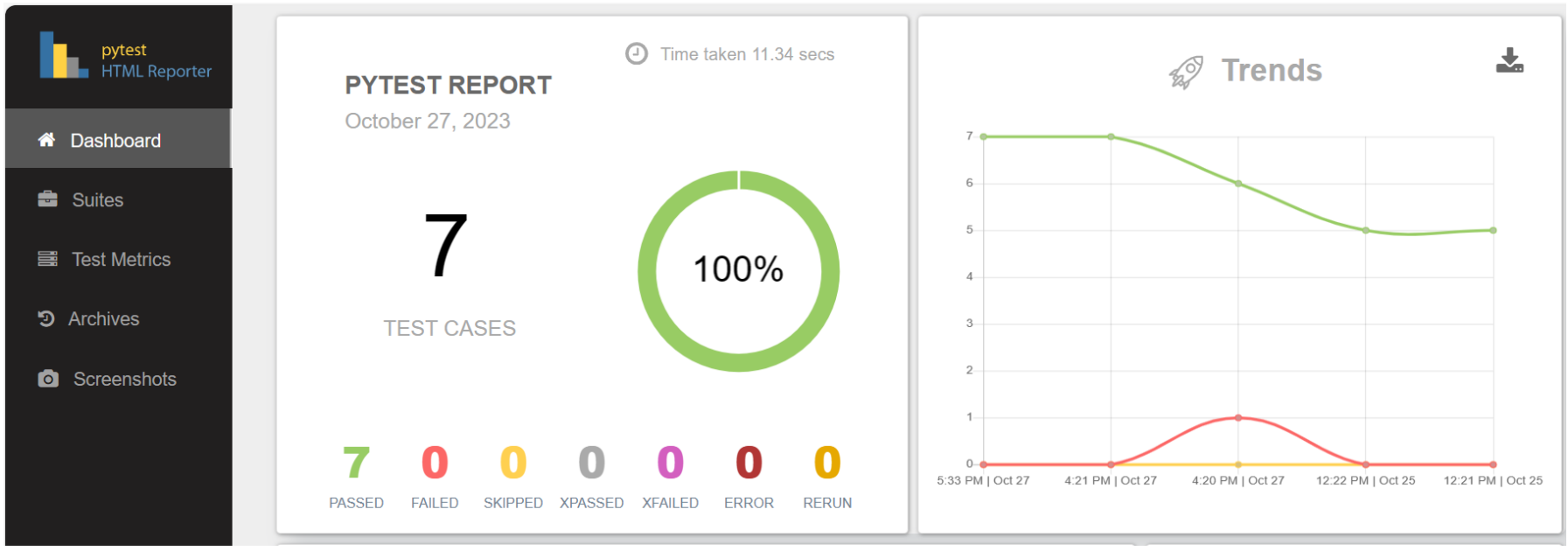
Data processing tests

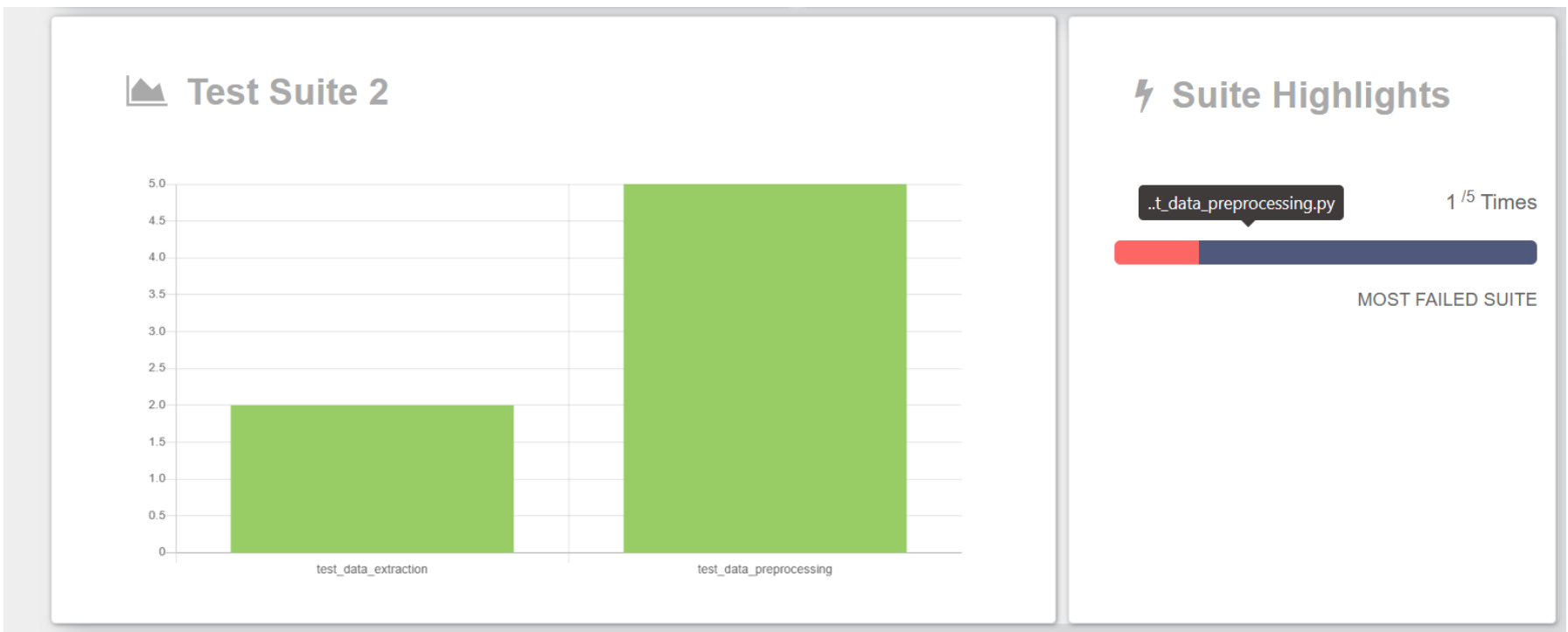
We added made sure to add tests while considering that we do not introduce test specific flows in the programs. The unit tests were added to run before the deployment pipeline so that regression tests were covered. For data preprocessing scripts, unit tests for fetching movies, filtering movies and sending curl requests to API for creating the dataset were covered. We also included negative cases and tests to unpack the ratings and history data. Dummy dataset files were used so that the unit tests do not have to interface with actual data and make these unit tests pretty quick.

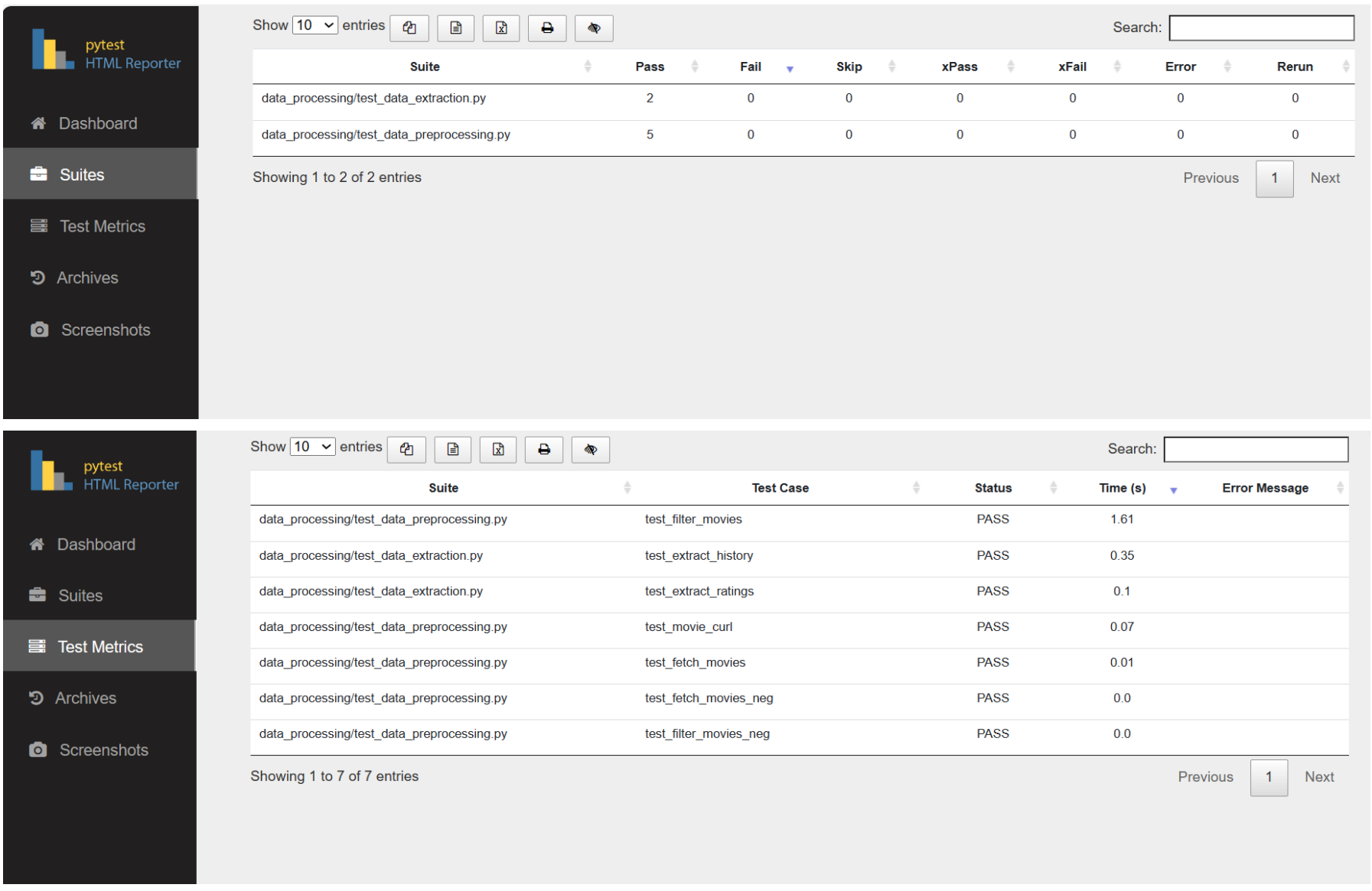
Test reports and runners

Most of the unit tests were written in pytests. They were kept under tests directory on the app folder. Some tests had to be kept outside because of relative module import issues. We used html-report package added as a plugin to the pytest runner so that we can generate reports for our test suites. The report runner keeps track of the history of the past runs. It saved in the archival files which are read everytime report is generated.

Following are the artifacts from our test reports:







Monitoring

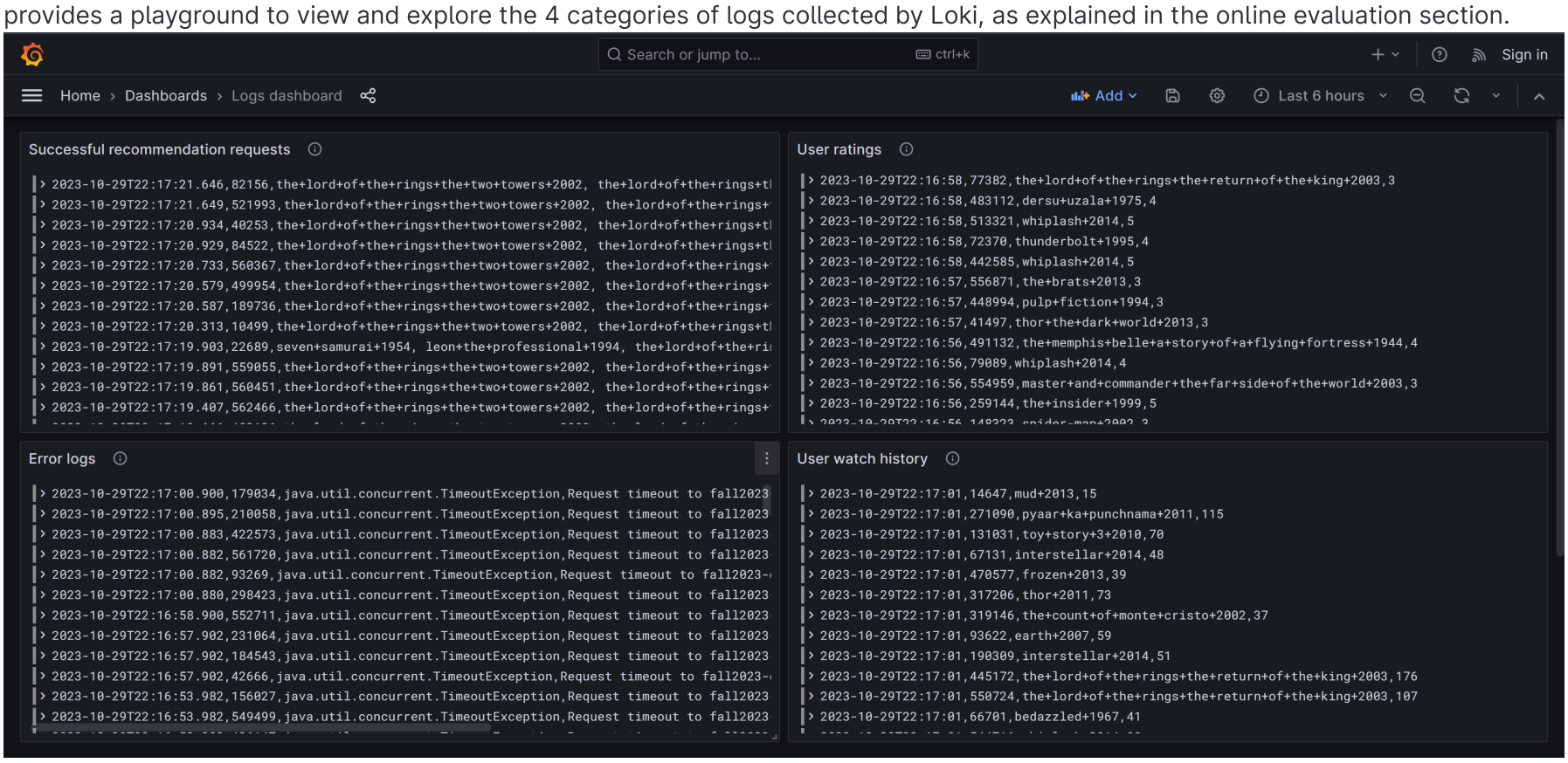
Our monitoring system operates on four main components: the telemetry collection system (described in the online evaluation section), the system metrics collection stack involving Prometheus, cAdvisor and node-exporter, the Grafana dashboard and the alert manager (Docker configuration)

Prometheus

Prometheus was set up along with cAdvisor and node-exporter to scrape metrics from our running containers, such as resource usage, filesystem usage etc. In addition, Prometheus was configured to scrape metrics from Loki and Promtail as well. Custom metrics were defined in Promtail to collect the number of successful and failed recommendation requests, and a histogram of response time.

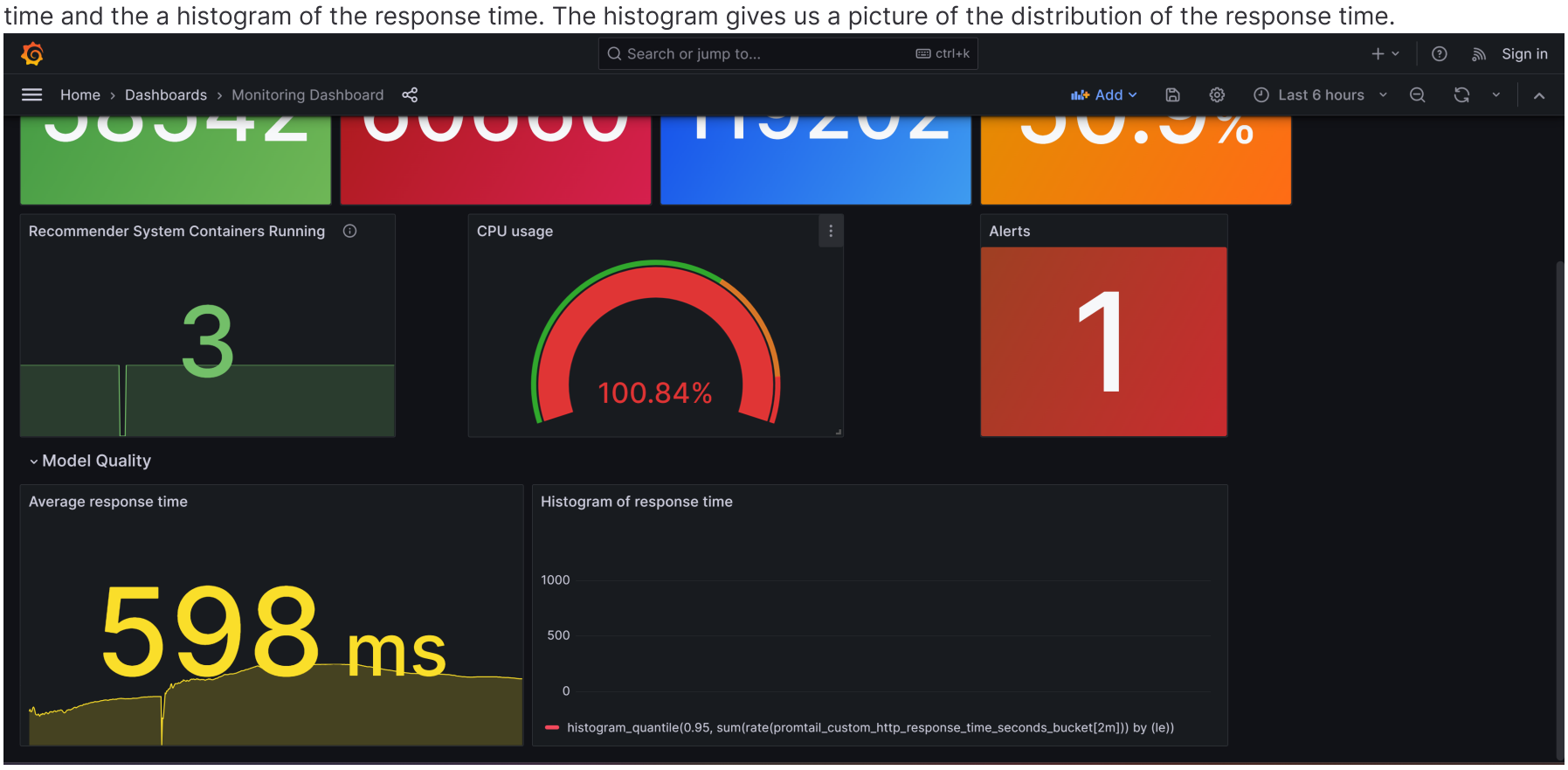
Grafana

Two dashboards were created in Grafana: the Logs dashboard (link) and the Monitoring dashboard (link). The Logs dashboard provides a playground to view and explore the 4 categories of logs collected by Loki, as explained in the online evaluation section.



The Monitoring dashboard is made up of two parts. The first row provides visualizations and statistics to monitor the availability of our recommendation system. In particular, it display the number of successful and timed out requests obtained from the custom Promtail metrics, along with the percentage of failed recommendations. There is a panel to monitor the health of our recommender service infrastructure, that displays the number of associated running containers. Our recommender service has 3 containers, one nginx load balancer and two instances of our Flask service. Thus, this metric should always have a value of 3. We also provide the CPU usage to monitor that our service is not overloaded.

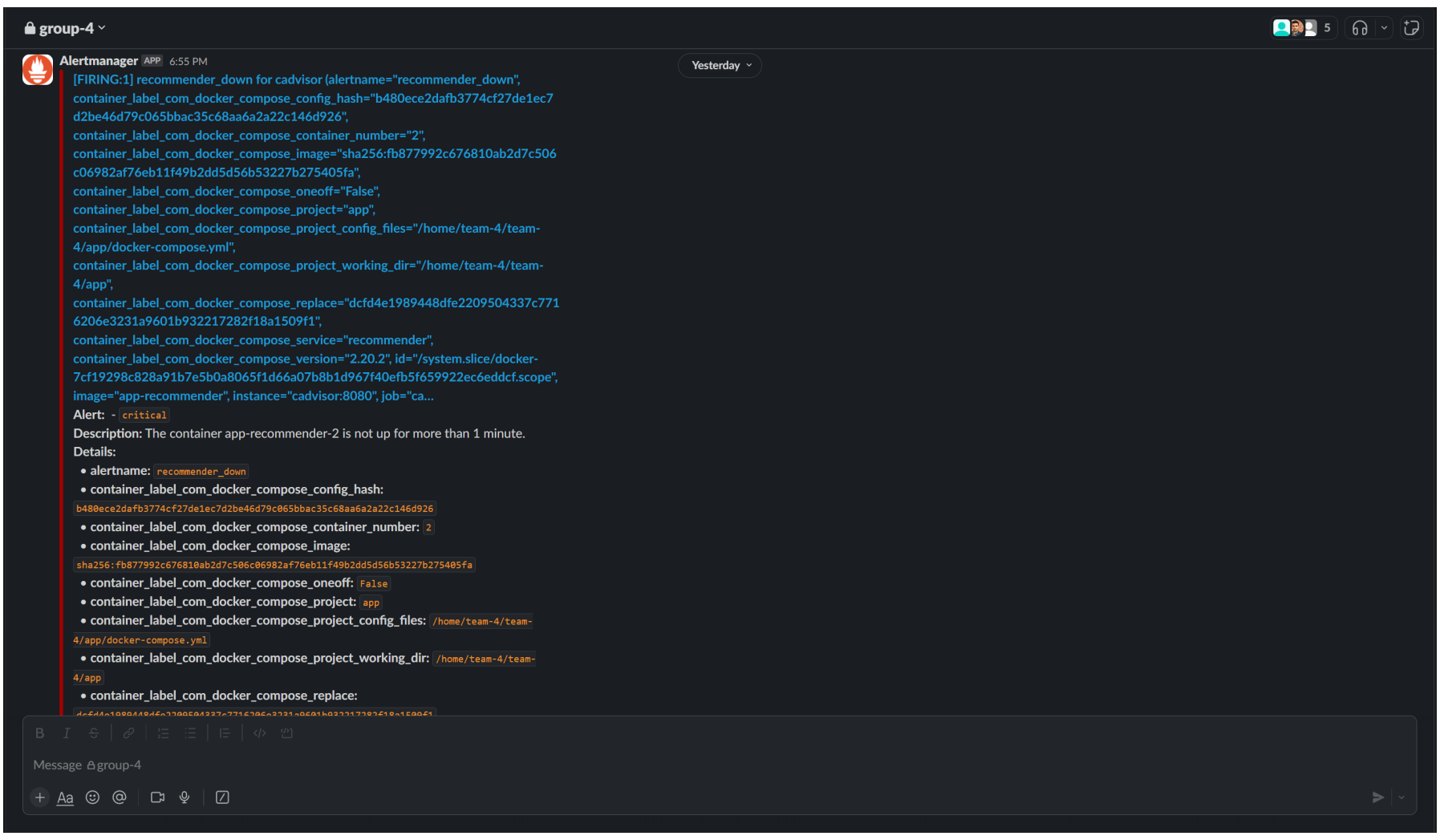
The second row monitors the model quality. Currently, we have two plots derived from the histogram metric: the average response time and the a histogram of the response time. The histogram gives us a picture of the distribution of the response time.



Alert Manager

Three rules were defined in Prometheus to trigger alerts in case of aberrations in the running of the recommendation service. Alert Manager was set up to send the alerts on our private group channel on the COMP585_ISS_A2023 Slack server. (rules.yml)

- Alert when any of the services Prometheus is monitoring is down for more than 2 minutes (cadvisor, node-exporter, promtail, loki). This rule was set up to ensure that the monitoring infrastructure itself is functioning well
- Alert when any one of the three recommendation service containers (1 nginx and 2 flask) goes down. This rule is very important as this alert could imply that there is a disruption in our service or that the server would get overloaded (with just one flask container running)
- Alert when more than 80% of recommendation requests time out. This rule is important as it means that our service is largely unavailable and requires immediate diagnosis.



Pull request reviews

Following procedure was followed by the team members for working on the code and requesting reviews:

Pushing changes:

- Each member has their own branch to work on.
- Once changes are completed in private branch, merge to development.
- Near due date, once **everything** is complete, we will merge it to main branch and tag for submission.

Request reviews:

- Members could directly tag other members in the merge request to the development branch from their own branch. The member who is tagged will get a mail to complete the review. Discussion for each merge request was held through the comment section for merge request on gitlab. Once the reviewer was satisfied, the merge request was accepted.
- The member who gave the reviews would look at the changes included in the merge request and raise issues if need be.

examples:

1. !33 (comment 61433)
2. !38 (comment 61532)
3. !45
4. !46
5. !47
6. !33 (comment 61398)
7. !36

Individual Contributions and Meeting Notes

Meeting notes:

12 Oct: Members [Aayush, Rishabh, Varun, Tamara] - Everyone shared their perspectives. 17 Oct: Members [Aayush, Rishabh, Varun] Updated async: Tamara - Everyone contributed to the meeting. 19 Oct: Members [Aayush, Rishabh, Varun, Luke] Updated async: Tamara - Everyone shared their perspectives. 24 Oct: Members [Aayush, Rishabh, Varun, Tamara, Luke] - Everyone contributed to the meeting. 26 Oct: Members [Aayush, Rishabh, Varun, Tamara] - Everyone shared their perspectives.

Flow of the discussion: We adopted round table based discussion but alot of our ideas were generated over encrypted whatsapp communication.

Role distribution: These were our expectations. You can check the relevant commits and what was actually done. Overall, everyone did their part.

Machine learning specialist: Tamara Software developer: Aayush, Varun, Rishabh, Tamara Project Manager: Aayush DevOps specialist: Rishabh, Varun Data engineer: Varun, Aayush Testing: Rishabh, Luke, Aayush Engineering manager: Varun, Rishabh On-team expert: Tamara Report Writer: Aayush, Varun, Rishabh, Tamara

Contribution by Tamara

Offline evaluation: Tamara

Relevant commit:

- code: !48 (25305ce7), 8ecc2a06, 16e8ef35, 691a68ba, d5be1501
- report: !48 (74475935)

Data quality: Aayush + some unit tests by Tamara

Contribution by Aayush

I worked on the following:

- Writing tests for data processing. Add unit tests for preprocessing (issue #20), Add html report runner (issue #21) commit, Add helpers for data processing commit
- Creating runners for generating test reports through pytest plugins: Commit linked above
- Data quality strategy, code and testing: Create data quality assessment strategy (#25), Add data quality pipeline code (#28), Create tests for it (#32) commit
- Online evaluation strategy and code: Create evaluation strategy (#34), Add scripts for online evaluation (#35), Files for grafana data source (#36) commit
- Report: Add report details: Aayush (#37) commit
- I also took initiative to coordinate among the team members, direct team members (Luke) and worked closely with Varun. team meeting, resolve user directory not found error which was preventing our model from running on team member's machine link
- Merge request reviews: (It is not an exhaustive list.) Asked review from others: merge request 33, Accepted the merge requests: 36

Contributions by Rishabh

Testing: Developed tests for the flask app and API endpoints
Relevant commits:

- fa509da3

Coverage: Developed logic to get the code coverage of all the functions in the system. Relevant commit: !42 (diffs)

Error Handling: Added error handling to most of the code related to the logic of our system. Relevant commit: 9f9e4be4

Pipelines: Worked on setting up the whole end-to-end MLOps pipeline. Relevant commit: 50266386

Continuous integration: Worked on setting up Docker runners for our CI/CD jobss

Pull requests reviewed by Rishabh:

1. !33 (comment 61433)
2. !38 (comment 61532)
3. !45
4. !46
5. !47

- Also brainstormed on the online evaluation model with Aayush and Varun.

Contributions for Varun

- Telemetry collection with Grafana Loki for online evaluation and monitoring (aadc2c40458506528 1e06a770d7b342a234126fb), scripts for scraping(fe5d2c330ccd0eee67ebc83af93bf314ab18f80e)
- Prometheus and metrics collector setup (854e27821bd5acc5dc39c64db312a18d6fda2ed6)
- Grafana dashboard (link, link)
- Prometheus alert manager (854e27821bd5acc5dc39c64db312a18d6fda2ed6, 854e27821bd5acc5dc39c64db312a18d6fda2ed6)
- Report for telemetry collection system and monitoring (252f6f3ab5177a5403bd0bd4d4c81c33369bd5c9)
- Pull requests
  - Requested review: !36
  - Provided review: !41

1. https://grafana.com/oss/loki/ ↩