Team 5 Milestone 2 Report

Team Members: Benjamin Lo, Fabrizzio Molfino, Gaurav Iyer, Keyu Yao, Yann Bonzom

Offline Evaluation

In an offline setting, we test our recommender system on a snapshot of our entire database, and not on a separate validation/test set. This is because our recommender does not "learn" from the provided data – recommendations are derived by querying a vector database of movie embeddings generated from a pre-trained sentence transformer called all-MiniLM-L6-v2 based on a user's movie history, ratings, and the popularity of recommendation candidates. Instead, we explore two metrics for our recommender system that deal with different aspects of the recommender system:

1. Hit Rate: The hit rate is the fraction of users for which the correct answer is included in the recommendation list of length L (= 20). A random movie from a user's watch history is removed and recommendations are computed on this modified history. If the removed movie is in the recommendations, then we count that as a "hit".
    a. Pros: Allows us to evaluate a system that doesn't rely on explicit training and can't use traditional ML metrics (like ours), while still giving us an idea of its performance. Easy to understand relative to notions of accuracy for a recommendation system. For example, it is hard to define the accuracy of a system that recommends 20 movies.
    b. Cons: Does not take into account the recommendations order. Our implementation is stochastic in nature. It does not make sense for users with less than 2 movies watched.

    Results: Our recommender achieves a hit rate of only 1.8179%. On a user base of ~130k, only 29594 users had more than 1 movie in their history, on which we achieved 538 hits. We believe that this is not necessarily due to our recommendations being bad, with other reasons including: (1) The amount of data per user is extremely small, resulting in a low hit rate; (2) the previously deployed model recommended the same set of movies due to memory and data constraints in M1. As a result, our collected data is very skewed; (3) evaluation metrics rely on users having predictable preferences. Due to our skewed data, and in general, simulated users of the Kafka stream do not reflect this.
2. Coverage: By coverage, we mean the percentage of movies that can get recommended to users through recommendations. For reasons stated earlier, we compute this on our entire user base. This metric gives us a sense of "diversity" of recommendations.
    a. Local Coverage: We have 23909 movies recorded in our database, and our recommendations come from this set of movies. We call the fraction of this set of movies that can be recommended the "local coverage" = 22082/23909 = 92.3585%.
    b. Global Coverage: This is computed on the complete set of movies, which we know is ~27k. Therefore, the global coverage = 22082/27000 = 81.7852%.

    We find that our model can recommend a majority of the movies given the different movie histories and ratings in the database. We are curious to see how this metric changes when user histories are more detailed and diverse. We used a Kaggle movie dataset and queried the catalog API for each movie to create a larger set of movies.

Some of the evaluation pitfalls we discussed in class include unrepresentative test data, data leaking, and overfitting on test data. We note that we avoid these pitfalls due to not explicitly splitting our data into train and test sets. However, it is likely that our evaluation is imperfect because our inference system has significantly changed and our previously deployed model has had a significant effect on the simulated users' behavior and therefore the data used in the evaluation. Link to the code implementation.

Online Evaluation

To evaluate our recommender with respect to real users, we compare their movie histories and ratings at two timestamps to check whether they used our recommendations. While this is not strictly “online” evaluation, this works for our system because our recommendations only depend on a user’s movie history and their ratings and the set of movies we can recommend is relatively static as it gradually saturates at the total catalog size.. It is also important to note that internally, if a user does not rate a movie, we assign it a rating of 3 as an approximated average rating. The rationale is that the user did not like or dislike a movie enough to care to rate it.

Our methodology was as follows: We randomly sampled 50k users from our database, and checked their movie and ratings history on the evening of 27th October and morning of 28th October. We find that 7913 users out of the 50k sample have had a change in their histories between these two timestamps. We use the movie histories and ratings at the first timestamp to generate recommendations, and check (a) if any movies from our recommendations pop up at the later timestamp and (b) if so, how the movie was rated. Out of these 7913 users, we find that 5433 users had one movie from the generated recommendations in their watch list, while the rest had 0. All 5433 users rated these movies with a 3 (it is more likely that they did not rate them at all). Thus, 68.6591% users reacted to our recommendations, but likely left felt average about the recommendations.

We would like to run this test for timestamps that are farther away in the future to more accurately evaluate user reactions over longer time periods – we could not do so at present because we do not have access to older timestamps and our recommender logic has changed significantly. Given that the model is static, doing this over longer timestamps will give us a better idea of this metric. Doing this evaluation in terms of time spent watching movies we recommend could also be an informative way of assessing recommendation quality.. With respect to other telemetry data, we collect the latest minute of a movie watched by the user, as an estimate of how long they watch a given movie. We do not make use of this for recommendations at present.

In our efforts to optimize the recommender system, we track various metrics, including CPU, memory, request rates per minute, the ratio of successfully processed requests (HTTP status 200), and the cumulative requests categorized by their status codes from the time the container was initiated. While the system consistently receives a steady flow of requests, we've observed some issues related to both CPU and memory utilization over time. With a steady increase of both that scale linearly with the number of requests service hinting at potential inefficiencies such as memory leaks. Further compounding the concern is that we found that the system became unresponsive after handling a range of 23,000 to 27,000 total requests. Addressing these challenges is paramount to ensure the reliability and efficiency of our service for our valued users. We made several changes trying to mitigate this that resulted in longer operating time before getting to the point of unresponsiveness. This is one of our top priorities to solve before our next report, yet we decided to keep this new system given its much improved recommendation quality and ease of scalability.

Figure 1: Increase in CPU usage as more request are serviced

Figure 2: Increase in memory usage as more request are serviced

Figure 3:Total request serviced since start time of the container

Code Implementation

Data Quality

When assessing the quality of our data, we notice that outliers occur mainly as a result of one of two possible errors: (1) Incorrect string parsing, where the data entered is valid but segmented incorrectly, and (2) errors due to corrupted units of data, such as incorrect movie or user ID and typos. To deal with (1), we add simple checks in our line-parsing code to skip any entry which deviates from the normal parsing format (see here). To deal with (2), we check that all parsed data is of the correct type, and that there is no duplicate primary key containing the same data in our database before insertion. Any incorrectly formatted data is replaced with NULL values before insertion.

In Azure SQL, we currently have 4 tables in place to aggregate data. Movie, Users, Watched, and Ratings. Movie and User store movie and user IDss and information, while Watched and Ratings store the minutes watched and movie rating of each movie_id/user_id pair (see db_utils). Because we are using Azure SQL to maintain our database, the remainder of our basic data quality checks are performed automatically. Azure DB ensures that all information stored is of the correct type and that no duplicate data exists. We are able to remove rows of data containing NULL values via simple query (which corresponds to removing incorrect data found during parse-time) and do so prior to pulling data from the database. For non-nullable data, such as movie and user ID's, we reference them as foreign keys across all 4 tables, meaning that any incorrect value in one table will cause the entry to be ignored when querying our table. Our next steps are to create a scheduled pipeline which runs an SQLAlchemy command to regularly delete columns in our database which contain NULL values, as well as further improve our data parsing script's error handling to mitigate issues related to incorrectly formatted requests.

Pipeline Implementation and Testing

Pipeline Implementation

Since M1, our pipeline has changed considerably. We have changed from a pickle file-based database to a full Azure SQL server, and moved to a vector database as the basis for our recommendation engine.

1. *Data Storage*
    a. The Azure SQL database is used to store all the data we continuously aggregate from the Kafka stream through the data parsing container. This allows us to scale to much more data without running into storage risks on the McGill server, and allows us to make highly optimized queries to the database, in particular to retrieve user watch histories.
    b. The ChromaDB open-source vector database stores exclusively the movie embeddings along with their IDs. This is stored on the McGill server as it is sufficiently small (with our current dataset of 23,909 movies, its size is only 93.1MB) and will not scale past the 27K movies in the catalog. Vector databases are a new type of database that are optimized for organizing unstructured data and enable highly efficient retrieval of similar items using their vector embeddings by use of an approximate nearest neighbor (ANN) search.
2. *Data Aggregation*: A data aggregation container using the Python-Kafka client connects to the Kafka stream and continuously parses each line, storing it to the Azure database. Data quality issues are handled in the parser itself – noisy logs are simply skipped over and never make it to the database. Currently the vector database is manually updated with aggregated data from the Azure SQL database. A next step is to automatically register new movies and users in the vector database when database objects are created.
3. *Inference*: the inference container functions as our API endpoint and handles the recommendation logic. This is a straightforward process: upon getting a request with a user ID, we retrieve the user's watch and rating history from the Azure database; next, we loop over each movie and get the top 20 most similar candidates in the ChromaDB database using ANN search; finally, we rank these candidates based on their similarities to seen movies, the movies' popularity scores, and the way the user has rated similar movies. This setup is much more effective than our M1 setup where all the vector embeddings were stored in memory, as it allows us to scale to any number of movies. Additionally, thanks to the use of ANN search, the recommendations are almost always generated in around 100ms, ensuring user needs are met quickly.

Testing the Pipeline

Unit tests were written for both the data aggregation and inference portions of our pipeline.

1. *Inference unit tests:* the API endpoint unit test serves as a simple assessment of whether the Flask API endpoint functions properly. The use of mocks to avoid running more complex underlying logic of the recommendation functions allowed us to verify proper functioning of the /recommend/<user_id> endpoint. Additionally, there are also extensive tests on the recommendation engine's logic. The test of the get_movie_history_and_ratings(user_id) function verifies that, in the case where a movie is not assigned a rating, our default rating of 3 is assigned. The remaining tests verify proper behavior of our recommendation algorithm including returning our default recommendation, working properly with a movie history of more than one movie, and excluding recommendations for movies the user has watched but has rated poorly.
2. *Data aggregation tests*: these unit tests encompassed both the Azure database interactions through testing of the User and Movie classes, as well as testing the data parsing functions to process the Kafka stream. The Azure database tests involve inserting and deleting a custom test user, and verifying that the functions we use work properly. To properly test the data parsing code, it has been refactored to improve modularity and enable a granular assessment of error handling (for instance, in the case that a Kafka log is improperly formatted).

As seen in our coverage report below, the coverage could be higher still. However, we felt it was sufficient as we focused on testing the core functionality of our pipeline. To further improve this, additional modularization of our codebase may be beneficial to facilitate the testing of more granular functionalities.

Name	Stmts	Miss	Cover
db_utils/db_utils/__init__.py	4	0	100%
db_utils/db_utils/azure_db.py	9	0	100%
db_utils/db_utils/movie.py	93	20	78%
db_utils/db_utils/user.py	53	6	89%
inference/main.py	76	21	72%
tests/test_data_parsing.py	0	0	100%
tests/test_db.py	57	1	98%
tests/test_inference.py	32	1	97%
TOTAL	324	49	85%

Continuous Integration

The CI/CD pipeline for our project is a composition of two separate pipelines – one for our inference system, and one for our data aggregation system. Each pipeline has 3 stages: a test-running stage, a build stage, and a deploy stage. The test-running stage is common to both pipelines, and is also executed whenever code is pushed to our main branch. This stage executes all the python unittest test files found in team-5/tests and generates a report of the results in the GitLab pipeline log.

Next, each pipeline has a build stage, which is triggered when we tag our repository with a /release-recommender-*/ or /release-parser-*/ tag, respectively. The build stage builds a Docker image based on the dockerfiles found in /inference and /data_parsing and installs the necessary package requirements as well as our own custom db_utils Python package, which houses all of the code used to interact with our Azure SQL database. The stage then pushes the latest image to Dockerhub. Finally, the deploy stage pulls the latest image from Dockerhub and deploys it on our McGill server, essentially allowing us to re-deploy both containers using only tags. While seemingly simple on the surface, this pipeline took a significantly longer amount of time to create than planned. This was mainly due to Azure SQL dependencies and necessary drivers, as well as the complex environment required to run our containers.

Because our inference model is a lazy-learner, no pipeline to train our model was implemented. To run this pipeline, a dedicated GitLab runner was deployed on the McGill VM with the Docker executor. The .yml file for the pipeline can be found here, and the Wiki entry about our pipeline is here. Next steps involve figuring out a way of doing this without Dockerhub, and fixing our requirements.txt files as they can be further optimized to reduce image sizes and thus improve our setup.

Monitoring

Setup

We employ Prometheus, in conjunction with several other accessory containers that add metrics to prometheus like cAdvisor, in order to obtain container-specific metrics and utilize the Grafana container for visual analytics. Prometheus facilitates the extraction of essential information from various containers, notably the availability, CPU, and memory usage for both our Kafka parser and recommender system containers. It further provides detailed performance metrics for the recommender system, such as the total count of HTTP request return codes, and the proportion of 200 return codes relative to other status codes.

Alerting mechanisms have been established based on the availability of containers and the proportion of 200 return codes, to promptly notify us of any container downtimes or if the recommender system encounters issues

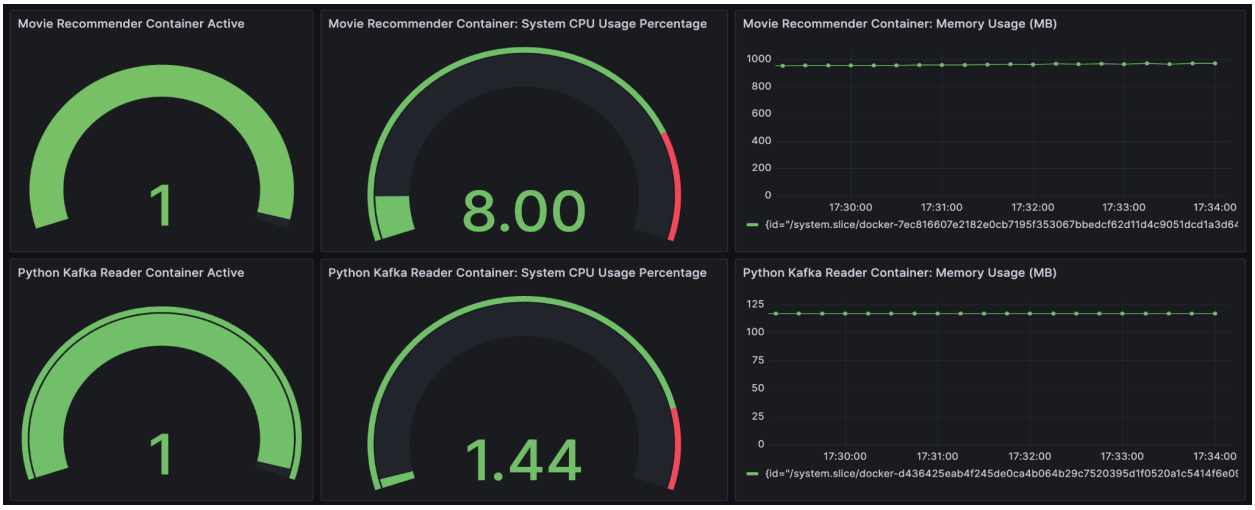
in processing requests. We use a Docker compose and a Prometheus configuration file to deploy all the containers with the right runtime configurations and configure the Prometheus data sources. These scripts can be found in the /monitoring directory.

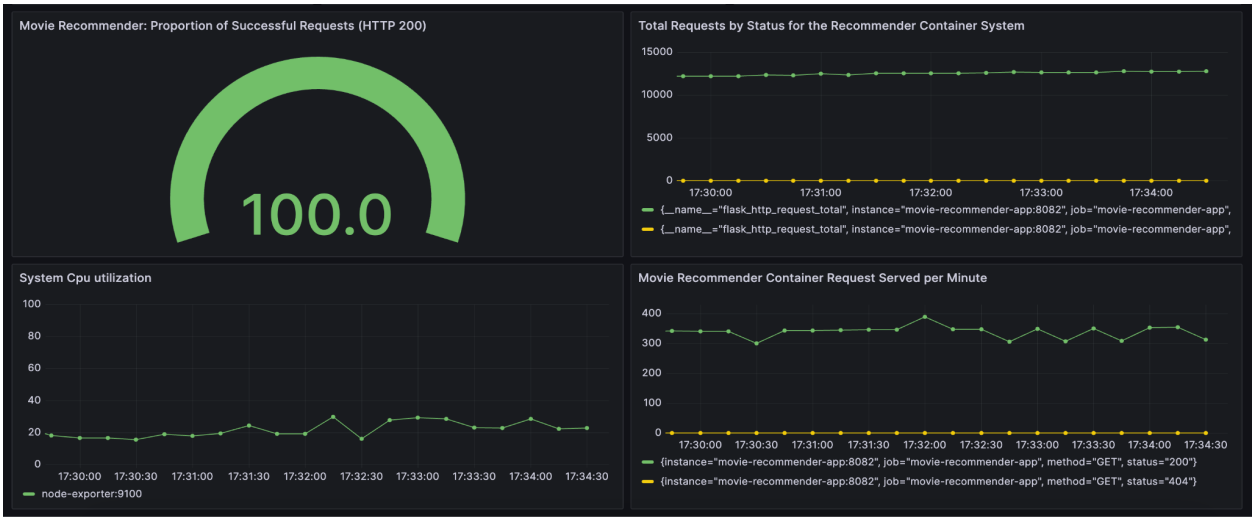
Metrics Monitored

For both the python_kafka_reader parser and the movie-recommender-app inference server: status, system CPU usage as a percentage, and memory usage. For requests: request served per minute to track the rate of requests, the proportion of successful requests (HTTP 200) to measure service quality, and the total number of requests by status. For the system, we measure total CPU utilization of our entire setup.

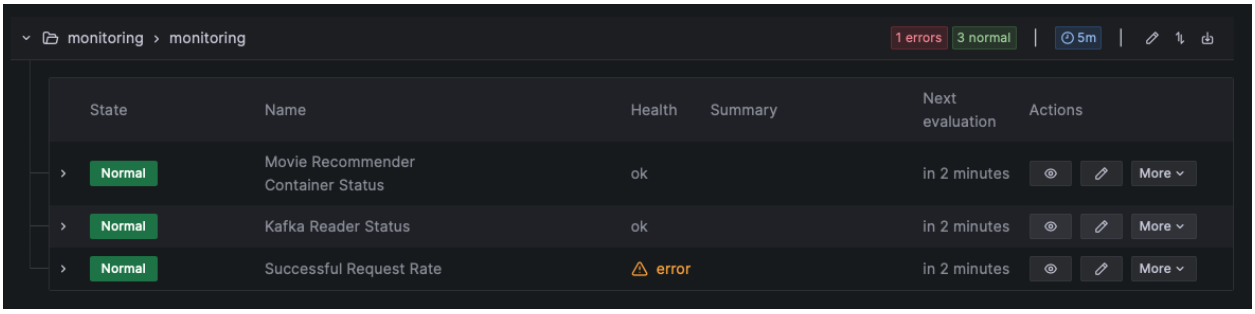
Monitoring Service Details

The monitoring service via Grafana is active on port 3000 on the team server. It can be accessed locally at localhost:3000 on the server or by forwarding the server's port 3000 to your machine using tools like Visual Studio Code. The dashboard is titled "Recommender System Monitoring Dashboard". The credentials are username “admin” and password “team5”.





Alerts



We set up alerts that monitor the active status of the recommender and Kafka Reader containers, and the proportion of successful requests. Any alerts triggered from these conditions are sent to our "grafana-alert" Slack channel. This ensures that the team is immediately informed about potential issues, allowing for swift responses.

Individual Contributions and Meeting Notes

Link to meeting notes

Benjamin Lo

- Designed and implemented the Azure SQL database, including writing SQL table-generating code and setting up table entries and relationships.
- Created a custom private Python package, db_utils, (wiki here) which uses sqlalchemy to map Python objects to SQL table row-entries, (movie.py, user.py) and manages ODBC connections to our database via Session objects (azure_db.py).
- Created the CI/CD pipeline for the project, including runner setup, pipeline script, etc.
- Created unit tests for database insertions (see here)
- Was the point of contact for database-related work, working closely with Yann and Gaurav throughout the development of the inference container and offline evaluation creation.
- Wrote the data quality and CI/CD sections of the report. Helped write the pipeline implementation portion with Yann.

Fabrizzio Molfino

- Setup the Monitoring service and all of the related containers, configure the Grafana and Prometheus connection(12ec01f3, ac610773, aae32dbf), and developed dashboard metrics and alerts with Keyu's support.
- Assisted on the recommender model refactoring with Gaurav and Keyu by modifying the movie embedding method (523205cd) and changing the selection process of movies (ebc71f83).
- Worked on Monitoring and Online evaluation section of the report

Gaurav Iyer

- Designed and implemented the new recommender system logic, and feature engineered data for embedding generation with Keyu and Fabrizzio (Recommender)
- Researched and implemented offline and online evaluation metrics on the model side (Online, Offline)
- Worked with Ben on integrating the Azure database with the new recommender system (SQL Code Integration)
- Worked with Yann on optimizing and troubleshooting the inference system to make it work with the timeout constraint.
- Made minor contributions to maintaining data quality by filtering noisy Kafka logs (1, 2, 3)
- Wrote the offline evaluation and online evaluation sections of the report for details with respect to the model.

Keyu Yao

- Designed and implemented the new recommender system logic with Gaurav and Fabrizzio.

- Developed the monitoring infrastructure, Grafana dashboard metrics with Fabrizzio.
- Set up the alerts and connect them to our Slack channel.

Yann Bonzom

- Performed research on vector databases and performed the setup for ChromaDB in our inference container. Created notebooks for this and explained the setup to Gaurav, Fabrizzio and Keyu. (notebooks)
- Created a new movie dataset by combining what's both in the Kaggle dataset and on the course movie API, as well as with what's on our Azure database, into one final file. (notebook)
- Adapted the inference logic to use the vector database. Took the new recommendation logic written by Gaurav, Fabrizzio, and Keyu and refactored it to integrate it into our API's main.py file. Restructured code for readability. (main.py)
- Wrote unit tests for the API and performed testing configuration (unit tests)
- Actively kept meeting notes and took a more active role in organizing team efforts by outlining tasks, checking up on progress and the timeline, and coordinating tasks. (meeting notes)
- Wrote the Pipeline Implementation and Testing section as well as portions of other sections. Did a final review of the report prior to submission.

Several members also shared knowledge with the rest of the team through our Gitlab repository. Overall, work was smoother and our progress was more organized in this milestone. We are actively working on improving our communication, and switching up the type of work each member takes on enables an increased understanding of the entire setup as well as improved productivity overall. We aim to further improve these aspects, and create such a timeline earlier on to reduce stress near the deadline.

Pull Requests

Given the higher complexity of the M2 requirements, being more careful in our development practices on GitLab helped improve productivity and ease of collaboration. Though we did not commonly perform code reviews, we did create Merge Requests rather than pushing to Main as we did in our work on M1. Additionally, we would take care to merge to Main only upon communicating with one another and ensuring that the new functionality we have added does indeed work.

An example of a merge request is request !8 by Yann, where the refactoring of the main.py API inference file to incorporate the new recommendation logic has been performed. This involved assigning the review to Ben and Gaurav to ensure that it is in line with their related work, and is good to merge with Main. An example of merge requests from Ben is request !9, which is Ben's largest contribution to the project involving refactoring the code into a package db_utils, and integrating everything into our pipeline. This branch was maintained for several weeks in parallel with Main, and Ben was in close communication with the rest of the team to make sure that his branch was updated with the latest features from Main, and that the CI/CD pipeline would work with whatever future features were planned. Request !1 is another example MR and included the initial migration to the Azure SQL database. Although the merge was temporarily reverted to continue development in a separate branch, this merge communicated the new changes and database pipeline to the rest of the team. Request !3 is an example of a long meeting between Ben and Gaurav, where both members worked together on a call to integrate the new database into our inference system. An example of a small merge request from Gaurav to handle some noisy Kafka logs for the stream parser is request !5.