COMP 585 M3 Report

Containerization

Our inference service requires a total of 6 docker containers in order to run properly. First, we have a container to run our kafka-log parser, which sends new kafka data to be stored in our containerized Postgres database. This container has been adapted as we noticed that we were only processing an average of 8.5 messages per second, which meant that our data was outdated and online evaluation in the canary release would not work. After the change, which involved reducing the amount of data saved, we could process an average of roughly 550 messages per second and keep our database up to date.

For the inference service itself, we have 3 containers running: a load balancer container responsible for sorting and distributing the recommendation requests, as well as a container housing our main inference service and one housing our nightly (canary release) service when it is actively running. The load balancer also stores the request history for data provenance and debugging purposes.

Finally, we have one container housing our GitLab runner, which allows us to trigger pipelines to automatically set up these containers from the GitLab CI/CD interface. A high-level description of our automatic container deployment is as follows: pipelines are triggered either via tags or pipeline variables (see here) depending on the container we wish to run. The runner then executes these pipelines, and deploys containers according to their dockerfiles (see parsing, main-inference, nightly-inference and load-balancer dockerfiles). For the inference service itself, the runner executes a Python script called deploy_inference_container.py which handles the creation of the load balancer and the main and nightly containers.

Automated Model Updates

(0.5 pages max): Briefly describe how you automatically retrain and deploy updated models. Provide a pointer to the relevant implementation (preferably a direct GitLab link).

Due to our recommendation logic being based on a vector database, it does not involve an explicit training step but rather makes use of precalculated movie embeddings as a way of retrieving similar movies. So, the other main aspect of our model's logic is in the subsequent step of sorting these candidates and returning the top 20 picks.

In our initial setup, that sorting is based on a mix of information on movie popularity as well as the ratings the user has assigned to the movies used to retrieve the candidates. Consequently, a "model update", here, constitutes an update to this sorting logic and occurs if the candidate sorting logic is altered in our inference/main.py file.

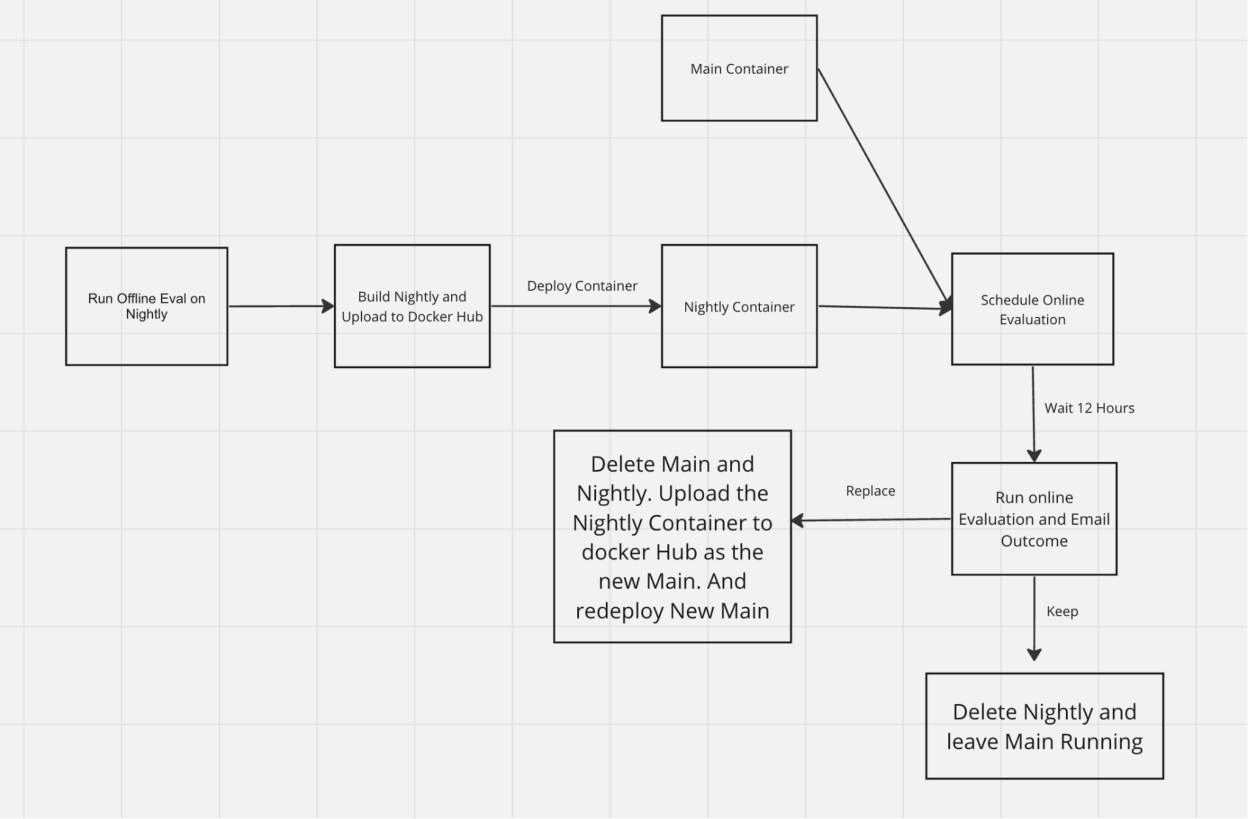
In terms of the vector database, it is essentially static since the movie catalog is limited. As such, a manual ‘filling in’ of the ChromaDB database every now and then permits the use of the most complete set of movies that the movie streaming platform can provide. This was therefore not automated, since in a real system we would have access to the full catalog and would trigger a table insertion whenever a new movie would be added to the catalog.

Releases

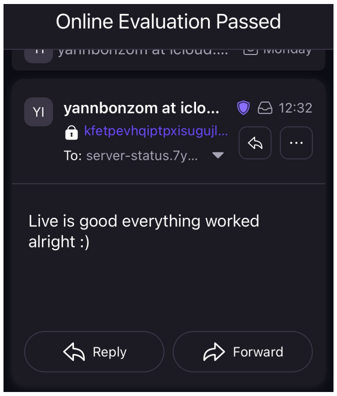
At a high level, the canary release logic is as follows:

1. When a branch with an updated model logic is pushed to the GitLab repository and tagged with *release-recommender-nightly-** (where * is a placeholder for the model version), this triggers the CI/CD pipeline.
    a. First, this runs the offline evaluation to assess whether the new model would perform at a level equivalent to the current main inference system based on historical data.
    b. Second, if the offline evaluation passes, the nightly container is built and pushed to Dockerhub. It is tagged with the version number stored in the * of the tag.
    c. Next, this triggers the deploy_inference_container.py script on the McGill server.
2. On the server, this deploy_inference_container.py handles all load balancer- and inference container-related logic. This essentially involves building the load balancer container if there is none, pulling and starting the main inference container if it is not running, and pulling and running the nightly container if there is already a main container running. In that final case of deploying a nightly, it also schedules a cronjob to run the replace_main_for_nightly.py script in 12 hours.
3. Thanks to the load balancer, the server now splits all traffic between the main version receiving 80% of the requests, and the nightly version receiving the rest. It ensures that request histories and recommendation responses are being saved to the RecommendationsHistory table for data provenance, and also uses the UserModelMap table to auto-assign users to the main and nightly versions and ensure they are always directed to the right version.
4. Finally, once 12 hours pass, the replace_main_for_nightly.py script triggers the online evaluation to compare the new model against the main version based on the data collected. It then, based on whether the new version outperforms the old version, performs the replacement with minimal downtime as it simply takes down both containers, renames the nightly container as the main container, sets it to run on the main port 3001, and pushes the new version as the latest main version to Dockerhub.

With this setup, all nightly versions are properly stored on Dockerhub based on the image tags and the model version (i.e., specific inference logic used) can easily be verified by checking the tagged commit or branch. Storing all these versions on Docker Hub also facilitates easy reversion to earlier versions if we ever run into issues with the new main inference container.


Main Container
Run Offline Eval on Nightly
Build Nightly and Upload to Docker Hub
Deploy Container
Nightly Container
Schedule Online Evaluation
Wait 12 Hours
Run online Evaluation and Email Outcome
Replace
Delete Main and Nightly. Upload the Nightly Container to docker Hub as the new Main. And redeploy New Main
Keep
Delete Nightly and leave Main Running

Due to our server not running due to technical difficulties on the final day of the deadline, we are not able to show evidence of an aborted release. However, we can show evidence of an email notification for the successful release of our nightly container after a short duration of online evaluation – the same message can be seen scripted in the replace_main_for_nightly.py file


Online Evaluation Passed

yannbonzom at icloud.... Monday

yannbonzom at iclo... 12:32
kfetpevhqiptpxisugujl...
To: server-status.7y...

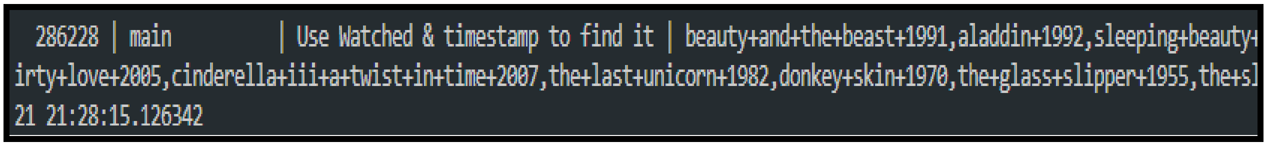
Live is good everything worked alright :)

Provenance

To understand how provenance works in our system, we first need to explain how our model deployment process operates. Since the inference model in our recommendation system is a lazy learner, there is no need to periodically retrain the model on newer data – our model automatically utilizes the most recent data at inference time. This means that the model release pipeline is manually triggered only when significant changes to the model logic are made. When this happens, our pipeline releases the new version of the model (known as the nightly model) in parallel with our currently operating model (known as main), redirecting 20% of the total recommendation requests to this nightly model. As such, one of the necessary requirements for proper provenance tracking is to keep track of whether a recommendation originated from the main or nightly model version of the pipeline. Since recommendations for a user depend entirely on the state of the database at recommendation time, knowing the state of the database at inference time as well as the model version is enough for full-scale model analysis and debugging purposes.

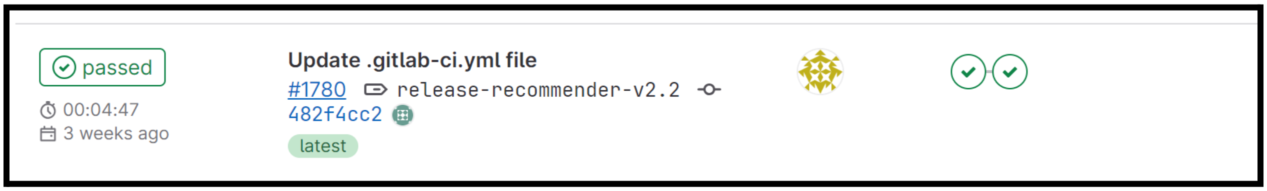
The general process to track the provenance of a given recommendation is as follows: we first locate the recommendation entry in our RecommendationsHistory table found in our Postgres database. This entry contains the user_id, the recommendation, and which model it originated from during the pipeline (either main or nightly). Then, given the timestamp of the entry, we locate the pipeline it originated from in our GitLab CI/CD interface. This provides us with the model's tag, indicating which model version the recommendation originated from, and also which instance of the model (main or nightly) corresponds to that recommendation. At the same time, the timestamp allows us to find the database backup made closest to the time of recommendation, which we can then use to restore the database to its previous version for data analysis.

Below is a specific example using a past recommendation in the RecommendationsHistory table. Suppose we want to find the provenance of this recommendation, made by user 286228 on the main container on the 21st of November at roughly 9:30 pm.



```
  286228 | main          | Use Watched & timestamp to find it | beauty+and+the+beast+1991,aladdin+1992,sleeping+beauty+
irty+love+2005,cinderella+iii+a+twist+in+time+2007,the+last+unicorn+1982,donkey+skin+1970,the+glass+slipper+1955,the+sl
21 21:28:15.126342
```

To find the pipeline it originates from, we find the corresponding release at the time of the timestamp in our CI/CD pipeline. As can be seen, the latest stable release originates from version 2.2 of our recommender, released on November 4th.



passed

00:04:47

3 weeks ago

Update .gitlab-ci.yml file

#1780 release-recommender-v2.2

482f4cc2

latest

This release also has an associated main and nightly image on DockerHub, allowing us to easily run that model version on a specific user to replicate the result. In this particular instance, we know the recommendation originated from the main model, so would rebuild the main image corresponding to that timestamp on DockerHub. Lastly, to obtain the data present at inference time, we restore the database version from our most recent backup stored locally in our virtual machine in /db_backups. Unfortunately, at the time of writing, both our virtual machine and database are unreachable due to last-minute technical issues, so no images of our database backups could be presented. To restore the database to this previous version, we simply run our pipeline specifying the backup filename as value to the CI/CD variable $CI_RESTORE_FROM_BACKUP. This pipeline executes a docker command which will restore our database to its previous state (provided there are no active connections to our database). With both the model version image and the database restored, we now have all of the required infrastructure to perform analysis on this specific recommendation.

Conceptual Analysis of Potential Problems

Our process for detecting feedback loops and fairness issues was done by checking different aspects of the data stored in our PostgreSQL database and by analyzing what our system does and where problems may arise. We make use of SQL queries to do this more efficiently and present our findings and considerations below.

Feedback Loops

One feedback loop in our system is the effect of a set of movies being recommended repeatedly due to downtimes in our data collection container. These downtimes may occur for a variety of reasons, but the consequence is that any user's watch and rating history is not saved for a period of time. This results in recommending the same set of movies for any given user, because our system depends only on the user’s watch history (and associated data, such as ratings) to generate recommendations. Users tend to watch movies we recommend even if they have watched the movie before, causing an inordinate number of watches and ratings for a few movies. While this does not cause issues with what we recommend later (once our data collection container starts working again), it does make the telemetry data we collect inaccurate and difficult to analyze.

Detecting this problem is equivalent to detecting when our data parsing container goes down and reducing the amount of time it stays down. Docker containers go down silently – one solution is to set up alerts to our Slack channel/emails for when our containers go down. In that way, someone can immediately troubleshoot and deploy the data parsing container as soon as possible. Another complementary approach would be to make sure that the container goes down less frequently by making it more robust to noisy logs and ensuring general operation. One consideration is that robustness often comes at the cost of speed – parsing logs too slowly creates a growing lag between real-time data and stored data in our database, making evaluation harder to perform. Our main way of combating this issue was to set up the parsing container so that it always automatically relaunches whenever it crashes. That way, even with potential bugs, we can easily ensure that most of the data is still constantly being collected.

A second potential feedback loop in our system is a bias in which movies are watched and rated due to the usage of “default” movie recommendations for users with no watch histories as a way of handling cold starts. In the case where a user who has watched no movies, we recommend our defaults, and they watch one of the top default movies. The next time this user requests recommendations, they will receive recommendations based on the one movie they have watched. It is reasonable to believe that there are many new users who watch the top default movie and essentially build up the same watch history. Furthermore, very few users actually rate movies according to our data. As a result, we would be left with many users who have the exact same watch histories and would receive the exact same recommendations, further perpetuating the feedback loop. **We have not convincingly detected this feedback loop yet, but we believe it is possible**. One way to reduce this would be to add more data to the decision-making process that differentiates users – such as user demographics, the amount of time they watch a certain movie, etc. The tradeoff that should be kept in mind here, is that fairness issues may arise due to biases such as a user’s gender, geographical location, or economic/occupational status. Another possibility would be to maintain a dynamic list of “default” recommendations instead of a static one, but similar considerations would have to be made when deciding how this dynamic list is to be generated.

We created the inference/feedback_loop.py to identify movies that are recommended with an unusually high frequency. This helps us better detect instances where certain movies may be disproportionately favored in our recommendation system, which may be a phenomenon that could stem from diverse factors such as inherent popularity biases or the influence of default rating mechanisms.

Fairness Issues

One major fairness issue in our recommender system is that of popularity bias. We rank our candidate recommendations by a combination of popularity and user ratings. Since we know that many users don’t rate movies (in which case they default to the average rating), popularity plays a dominant role in these rankings. As a result, popular movies are recommended higher and more often on the list of requested recommendations, skewing watch histories and ratings in their favor, and movies that are less visible (non-English or indie movies, for example) are often effectively ignored. While our approach may be considered fair in the sense of group fairness with respect to genre or language, it is not fair in the sense of individual fairness – two movies may be similar in all aspects other than the budget and general hype/popularity, but do not get treated similarly by our recommendation algorithm.

One way to combat this might be to introduce some stochasticity to our recommendation system, where movies that are not as visible are randomly recommended toward the top of the list to give them more visibility. These could be separately ranked according to some notion of similarity to the user and data specific to them. Another idea is to progressively but carefully give less weight to a movie that is frequently seen, to strike a sort of balance between popular and obscure movies. To us, it does not seem like there is a permanent solution that will work indefinitely, and some changes and tuning to the algorithm would have to be frequently made to maintain this balance. In a real-world context, for instance, with Netflix, it is questionable whether accommodating such unfairness is worthwhile: providing popular movies is an effective heuristic for whether users will watch a movie from a given set of recommendations and spend more time on the platform, which is the goal.

Another issue is the default rating bias. We give every unrated movie a default average rating of 3 (assuming movies can only be rated from 1-5), which can skew the system’s understanding of these movies' quality and may not accurately reflect user preferences. It may lead to unfairness to those who are less inclined to rate movies or tend to rate only movies they feel strongly about. This approach may not accurately capture the real viewing preferences of such users, leading to a distorted representation of their tastes. Moreover, for users who rate less frequently or not at all, this method might oversimplify their viewing habits, overlooking the subtleties in their movie choices. The consequence is a recommender system that might not effectively cater to the diverse tastes of its user base, especially those who interact differently with the rating feature. One way to combat this is to introduce a more sophisticated approach to handling unrated movies, such as inferring potential ratings through user similarity, movie rating averages from other sources such as IMDB, or developing strategies that accommodate the varying degrees of user engagement with the rating system.

Analysis of Problems in Log Data

The first step we took in analyzing problems in our telemetry data was to check what movies were getting recommended widely. We find that the majority of the movies recommended are from the default list of movies that we recommend to new users. While we do agree with the feedback given that this is a bit of circular analysis and reveals very little information, it does confirm that the majority of the users on our platform are first-time viewers who have no watch history and is an important piece of information with respect to the demographic.

Based on feedback, we try to deepen our analysis by also taking a look at the top 20 movies that are actually watched by users. We find that about half the movies watched are from the default recommendations, we can see that the impact of default recommendations on user behavior within our recommender system is still big. However, the most watched movies are not in the order of the default and are scattered throughout the list. By defaulting to a set of popular movies for new users, we initially thought this approach would provide a safe starting point, exposing users to widely appreciated films. Although our telemetry data indicates that this strategy might be influencing user preferences more significantly than anticipated, it is reasonable for a recommendation system to start off offering some well recognized contents.

This implies that the users have agency to some degree in the movies that they watch and don't only watch movies that are popular i.e. while our default recommendations do significantly impact what users watch, the effect is not dominant to the point that diversity in recommendations is compromised.

We also take a look at the top 20 movies that are recommended when not taking into consideration the default recommendations, and again find that diversity is maintained and that movies are recommended in fairly uniform numbers throughout the list – movie id "five+corners+1987" is at the top of this list and is recommended ~27.7k times, while movie id "baraka+1992" is at the last spot and is recommended ~17.6k times, across a gradient of 20 movies. The movies are also diverse in their genres – quite a few of them are animated, but movies also span action, comedy, documentaries, crime thrillers, and mystery among others.

Overall, our decision to give default recommendations by popularity to new users did cause a few problems in terms of fairness and feedback loops – most popular movies are implicitly biased towards movies with large budgets and studios backing them, and are often in the action/comedy genres. On the other hand, if we look beyond this factor, our system is quite diverse in the recommendations that it provides, which is a silver lining to the fairness aspect of our system.

The recommendation results can be viewed here:

https://docs.google.com/document/d/1x3Sk5ehMEA2TUyRKC4c3PEwrEyNcexkpi0dng2CBiRo/edit?usp=sharing

Script used:
inference/feedback_loop.py
(Since the server was down on the last day of the submission, we couldn't commit the most recent script, but the results were up-to-date.)

Reflection on Recommendation Service

Looking back, the most challenging parts of the project were those found in Milestone 1. Specifically, the creation and maintenance of a database, and the creation of an inference model. These two tasks were challenging for many reasons, foremost among them being that these were the most fundamental components of our entire project, and we had to undertake them at a time where we had the least experience and were the least prepared. This meant that we initially had to sacrifice a lot of time to experiment with implementations that ultimately would not work.

During Milestone 1, we designed and scrapped two database methods, and created two different models, only one of which was expanded upon while the other remained unused. In addition, we changed our evaluation method multiple times due to a misinterpretation of the project instructions. This additional and ultimately unfruitful workload led to a snowball-effect, where many of our downstream tasks were delayed due to frequent changes with our models and/or databases. For example, throughout our project, our database migrated on three occasions: first, we used pickle-files but these were not scalable; second, we moved to an Azure-SQL server, but that proved costly and resulted in high latency; and finally, we moved to a local Postgres server which was consequently free and provided very low latency (on the range of a few milliseconds vs. 100s of milliseconds for Azure). This hindered our ability to gather consistent data, and impacted our ability to perform both offline and online evaluations as we could not generate evaluations without complete user data.

Currently, both our database and our model remain the more unstable sections of our project. This is because the quality of all our other components ultimately depend on the quality of these two services. For our database, for example, we have had to remove certain useful key mappings between tables and data quality checks (i.e. duplicate key checking) in order to speed up the data aggregation pipeline to be able to perform online evaluation on live data. This would need to be fixed before deploying the recommendation service at scale in production.

For our model, while it still remains effective at providing recommendations, opting for a lazy learner (which at the time, was selected partly due to latency constraints) has limited our potential avenues of experimentation and analysis in the long-term. For example, currently we would not be able to experiment with different model parameters, different feature selections or preprocessing decisions, or any other machine learning toolkit available to an eager-learning model. Furthermore, different evaluation metrics, such as precision, recall, or F1 score, also remain unavailable. While not critical, choosing a lazy learner has limited our options in the long-term, and, due to having built much of our architecture around this model, has made it much more difficult to switch models in the future. This would also need to be fixed in the long-term. However, this model setup also comes at the advantage of a greatly simplified setup and very low latency, which proved very helpful.

To fix these issues, we would dedicate a significantly larger portion of time to database engineering and model experimentation. We would run tests using different database and table configurations to see which had the least overhead and were the most efficient data-wise. One way to address this is to have another container running that regularly performs checks on what new movies and users have entered our Watched table, and perform the API calls required to get their details from the course-provided API. Additionally, this service could also automatically insert new movies into the vector database, which is not currently being done yet.

For our models, experimenting with different models on a subset of different datasets now that we have more data would allow us to run inference with more expressive models and evaluation metrics. For the vector database approach in particular, experimenting with various embedding functions such as OpenAI's popular "text-embedding-ada-002" model as well as selecting a different set of features to embed would be interesting to experiment with. In line with these suggestions, refactoring our code to become model and database agnostic would be a priority- this means segregating the logic for our model and database from the rest of our system would allow for easier replacement in the future if need be.

In terms of the overall setup of this entire inference service, one major change we would do in a subsequent iteration would be to use Docker Compose right from the start. We did use it for the monitoring setup with Grafana and Prometheus, and found that it greatly simplified the effort required to orchestrate a large number of containers. Using such a setup for the inference side would have saved us hours on debugging and relaunching broken containers in the right order to ensure that they can all work together properly. This would also tie in nicely with the GitLab CI/CD setup.

Reflection on Teamwork

What Went Well

In our team, we found that our general strategy of task delegation and meeting-scheduling was relatively effective. No interpersonal conflicts occurred, and everyone knew what tasks they had to complete and what their next objective was almost all of the time. The evidence that our initial strategy outlined in Milestone 1 worked was that all deadlines have been met, and nearly all of our given tasks have been accomplished. This is an indication that our teamwork and interpersonal organization work well, albeit not perfectly.

One crucial positive takeaway was the consistency and frequency of communication over Slack. Slack proved to be the perfect balance between formal and informal communication - informal enough that we felt comfortable messaging each other at any hour of the day, without worrying about spelling, grammar, professionalism, etc., but formal enough that it was easy to track threads, requests, deadlines, and follow up with items that needed our review. The constant communication over Slack allowed us to adapt to changes quickly, and make sure everyone remained on the same page related to the project.

Additionally, getting the opportunity to reflect on our teamwork after each milestone enabled us to improve the way we collaborate iteratively. This resulted in slight shifts in our role allocation, where different teammates took on different planning roles to help structure our collaboration more effectively. It also resulted in more structured milestone-planning meetings, where extensive time was spent on planning the individual tasks and making efforts to assign them to all of us to complete by certain dates.

What Didn't Go Well

For a large majority of the time, the team was fragmented due to different time constraints and workload availability. Different members would do coding sprints at different stages, and unpredictable schedules meant that the workload that was originally planned to be done by one person was completed ad-hoc by another, often-times with temporary communication breakdowns in between. These diversions from the initial plan were not necessarily unwarranted, but they did generate some confusion among members and gave off a slight feeling of distrust between members as we couldn't always rely on each other perfectly. Improving such communication, and being comfortable sharing that we can no longer complete what we had agreed to, would be helpful here.

In addition, the independent and asynchronous working-style we developed created large amounts of logistical overhead when trying to merge the work of each individual. In general, big picture ideas were discussed before task assigning, but specific implementation details were overlooked, leading to conflict between stacks. This was, however, in part alleviated by additional planning meetings that went much more into detail to ensure all clearly understood the setup.

Better for next time

In meetings, discussing deliverables beforehand – specifically, by agreeing on how each individual's code would be interfaced with (method signature, package specifics, implementation details, etc.), would avoid a lot of the conflicts occurring while merging work later on. In addition, creating a list of guidelines with examples of what is important to communicate and when would help streamline communication and avoid periods of radio silence. For example, if a person makes changes to the stack of another, a guideline would be to send them a detailed message stating what the changes were and why they were essential. Generally, the only reason this was not already done was due to time constraints; however, in hindsight, having such a system of protocols would have paid off later on as project overhead increased.

In addition to this communication aspect, holding more regular mini-meetings just to update each other on our individual work and ensuring that all have the big picture in mind would have been very beneficial. Communication over chat is effective, but it is often easier to grasp the big picture and quickly clarify confusions verbally instead.

Individual Contributions and Meeting Notes

Benjamin Lo

- Implemented the initial CI/CD pipeline logic and infrastructure during Milestone 2 (was done during milestone 2 but were not part of milestone 2 requirements)
- Implemented database backups and restorations, as well as modifications to the CI/CD pipeline to accommodate changes made by Yann and Fabrizzio. (1, 2)
- Worked closely and continuously with Gaurav in testing our online evaluation, and when integrating the various sub-systems (Canary releases, online eval, etc.) into the CI/CD. (1)
- Wrote the Containerization, Provenance, Reflection on Recommendation Service, and Reflection on Teamwork sections.

Fabrizzio Molfino

- Worked with Yann on the Canary Release project, handling key tasks including making the load balancer to evenly distribute traffic across main and nightly containers, and making changes to the pipeline for the canary release process. Also worked on the development of scripts for the scheduling and deployment of containers, and for running online evaluations.(cc9dd37c, ed919292, 1cbb3e66)
- Created the email notifier to notify the outcome of the canary release (a7b8e03c ,384b5b79)
- Adapted monitoring setup to monitor main and nightly container (cc9dd37c)
- Added the Flow diagram in the Release Section

Gaurav Iyer

- Worked on re-implementing online evaluation with feedback from Keyu and Yann to consider how recommendations would be made and how movies would be watched in real-time (1).
- Worked with Ben on integrating evaluation scripts into the CI/CD pipeline along with other components.
- Conceptualized feedback loops and fairness issue possibilities and came up with a few basic ways to implement analyses of these with Keyu.
- Worked on testing the CI-CD pipeline towards the end of the project with Ben to generate successful and aborted releases (of which the former was successfully done) (1)
- Wrote the report sections on Feedback Loops, Fairness Loops, and Analysis of Problems in Log Data with some data provided by Keyu's scripts. Also discussed content and significant points for Reflection on Teamwork with Ben.

Keyu Yao

- Worked with Yann to insert new data into our database.
- Worked on re-implementing online evaluation with Gaurav and Yann.
- Redesigned offline evaluation to use the same metrics as online evaluation with help of Gaurav.
- Worked on the feedback loop with help from Gaurav.
- Wrote the report sections on Feedback and Fairbess, Analysis of Log Data.

Yann Bonzom

- Performed the database migration from Azure to Postgres to reduce cost and latency (Wiki)

- Redesigned the Kafka parser and database structure to increase the rate of messages processed from ~8.5 to ~550 messages per second, allowing us to store live data for online evaluation purposes (new parser logic, updated database utilities)
- Adapted the load balancer written by Fabrizzio to save the histories to RecommendationsHistory table, as well as perform the user-model mapping for the canary release
- Worked extensively with Fabrizzio on the canary release to develop the overall deployment logic, write the .yml CI/CD file, adapt deployment scripts, and perform testing on it all together
- Performed brainstorming with Gaurav on the evaluation script
- Maintained the same team role as in Milestone 2 of leading meetings, taking the time to perform initial planning and brainstorming of tasks, and coordinating task allocations and determining a timeline together with all teammates. In the later stages, this involved in particular the sending of summary messages on Slack and checking in with everyone to help facilitate clarity on what is going on and what needs to be done. Also recorded meeting notes and ideas in the M3 planning document for our reference.
- Wrote Automated Model Updates and most of the Releases sections, as well as performed final editing of the document and adding in additional points to the Reflection on Teamwork section.