**Recap Aggregator**

CSCA 5028 Final Project – **Ryan Matzke**

**Overview**

For my final project, I created a recap aggregator application. This application gathers video metadata from a variety of movie recap channels on YouTube, analyzes the video metadata to determine the genre of the movie being recapped, and presents the analyzed data to a web application where users can filter by genre.

The reason I chose to build a recap aggregator is because I enjoy watching movie recaps, but there are a lot of recap channels and they tend to go on streaks where they only upload videos of a certain genre for a significant stretch of time. I tend to only be interested in recaps from a few genres, so I thought it would be nice to have a place where I can go to see only the recaps for my preferred genres.

**Source Code**

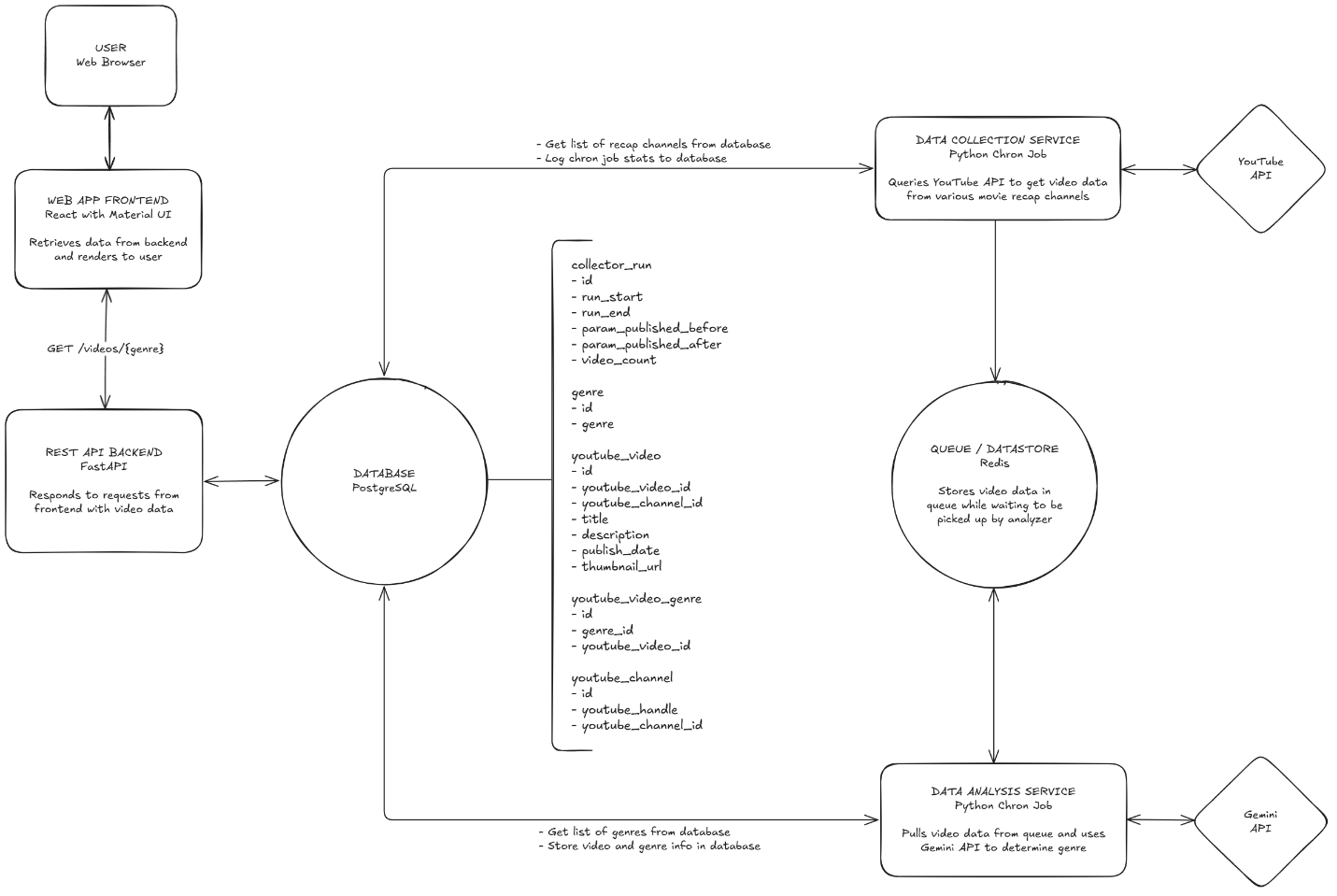
You can view the source code repository here: [rmatzke1/csca-5028-final-project](https://github.com/rmatzke1/csca-5028-final-project)

You can view the live web application here: [Recap Aggregator](https://web-ui-dev-255254081323.us-central1.run.app/)

*Disclaimer: The live web application includes video titles and thumbnails pulled directly from YouTube. I apologize in advance if any offensive or inappropriate content is displayed. I have no control over the content that is uploaded to YouTube.*

**System Architecture**

The diagram below provides a high-level overview of the system architecture. More detailed information about each of the components is provided in the next section.



**System Components**

**Web App Frontend**

The web application frontend (web-ui) is where end users will interact with the application. It is a lightweight web page that queries the rest-api application for video data. It was built using Node.js, React, TypeScript, and Material UI. The application is hosted on Google Cloud Platform, as a Cloud Run Service.

**REST API Backend**

The REST API backend (rest-api) serves as a gateway between the frontend and the database. It is hosted on Google Cloud Platform, as a Cloud Run Service. The service is built using FastAPI in Python and contains the following endpoints:

* **GET /api/health** – Used for checking the health of the application
* **GET /api/genres** – Returns a list of the genres defined in the database
* **GET /api/videos** – Returns a list of videos from the database. Supports an optional query parameter **genre\_id**, which filters the list of videos to the specified genre.

**Database**

The primary database for my application is a PostgreSQL database hosted on Google Cloud Platform using Cloud SQL. It contains several tables which are outlined in the system architecture diagram above.

**Queue / Datastore**

Redis is used in this system as both a datastore and a message queue. The data-collector application pushes blob data to the Redis instance, and the data-analyzer application pulls data for analysis. The data is stored using Redis’ SET data structure. This Redis instance is hosted on Google Cloud Platform, using Memorystore.

**Data Collection Service**

The data collection service (data-collector) first queries the primary database for a list of movie recap channels. Next, it queries a YouTube API to retrieve video metadata for each of the movie recap channels, which is then stored in the Redis instance. The YouTube API query is restricted to a specified time window, and each time the job runs the window will be moved forward. Finally, it logs some stats about the run to the primary database. This job is hosted on Google Cloud Platform as a Cloud Run Job. It is scheduled to run every 12 hours. This is to ensure that rate limits for the YouTube API are not exceeded.

**Data Analysis Service**

The data analysis service (data-analyzer) first queries the primary database for a list of allowed genres. Next, it pulls a single video record from the Redis instance and uses the Gemini API to determine what genre of movie is being recapped in the selected video. Finally, it writes to the primary database again to link the selected video to the determined genres. This job is hosted on Google Cloud Platform as a Cloud Run Job. It is configured to run 5 concurrent executions every 6 hours. This is to ensure that the rate limits for the Gemini API are not exceeded.

**Reusable Components**

Although not shown on the system architecture diagram, the source code for this application contains reusable components. This is mainly for interacting with the database, i.e. migration scripts and gateway modules. These components are packaged into Python packages which can be used by the other applications, providing high levels of code reusability.

**GitHub Actions Workflows**

Several GitHub Actions workflows are included in the source code for this application. These workflows handle database migrations, unit and integration tests, and continuous integration and deployment. These are not included in the system architecture diagram, but are instead detailed in the CICD section of this report.

**Design Considerations**

This section outlines several of the design choices that were made in the development of this application, and provides reasoning and/or justification

**Google Cloud Platform (GCP)**

GCP provides a single location for all components of the application to be hosted. Some of the advantages offered by GCP for this application include:

* Built-in logging for all products
* Health checks for Cloud Run services (rest-api, web-ui)
* Automatic scaling for Cloud Run Services (rest-api, web-ui)
* Scheduling for Cloud Run Jobs (data-collector, data-analyzer)
* Database and datastore scaling and replication (PostgreSQL, Redis)

**Redis**

As mentioned above, Redis is used in this system as both a datastore and message queue. Redis being used as a datastore is a common use case, primarily due to the fact that it stores data in memory, allowing for very fast response times. Redis being used as a message queue is probably a lesser-known use case. The key thing to note here is that Redis is single threaded. This allows the concurrent data-analyzer jobs to pull items from the data set without having to coordinate with each other. Considering the above, and the fact that video data does not need to be processed in any specific order, Redis made an excellent choice for this system.

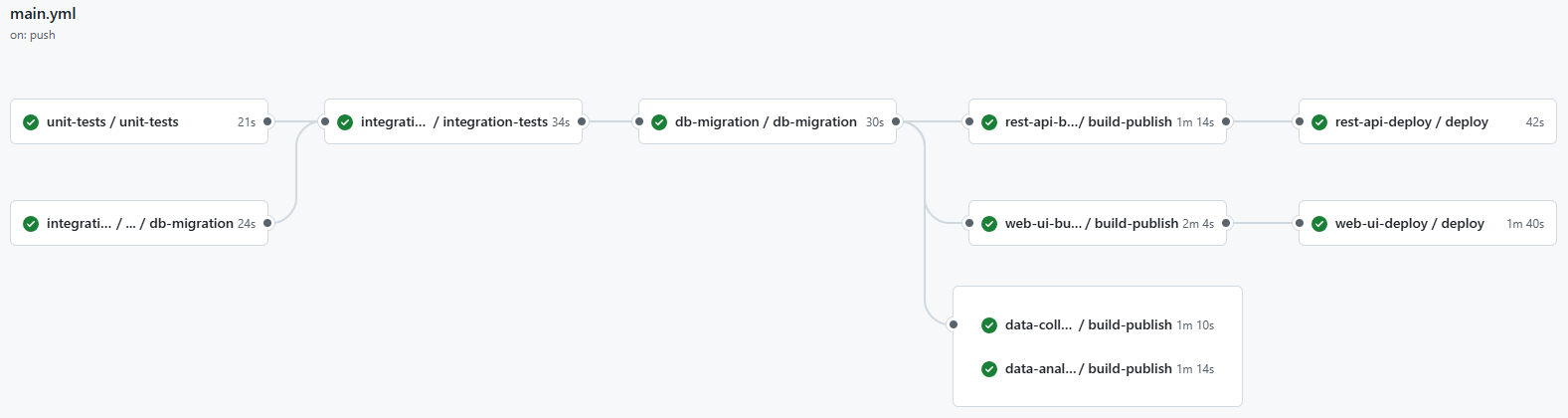
**Web UI and API Decoupling**

Although both the web-ui and rest-api are fairly lightweight, I decided to decouple them so that they can scale independently if needed.

**CI / CD**

Continuous Integration and Continuous Delivery operations for this application are implemented using GitHub actions. The source code repository contains a main workflow, which calls several other workflows. This workflow is triggered whenever there is a push to the main branch.

Here is an overview of the entire pipeline:



**Step 1: Unit Tests and Database Migration**

Unit tests and a database migration is triggered. This database migration is executed against a dedicated test database.

**Step 2: Integration Tests**

Integration tests are kicked if the unit tests and test database migration are successful. These integration tests are also executed against the dedicated test database.

**Step 3: Database Migration**

If the migration and integration tests against the test database were successful, the migration scripts can be executed against the primary (dev) database.

**Step 4: Build and Publish Images**

Images are built for the rest-api, web-ui, data-collector, and data-analyzer applications. These images are then pushed to the GCP Artifact Registry.

The data-collector and data-analyzer jobs are configured to always use the latest image, so that is the final step for these jobs.

**Step 5: Deploy**

This workflow updates the rest-api and web-ui Cloud Run services to point the images built in Step 4. This triggers a new revision, thus deploying the latest code.