Talk2Data - Centralized Ecosystem for College Students

Sanchit Sahay, Neel Gandhi, Kevin Vaishnav, Sitanshu Kushwaha
New York University
{ss19723, njg9191, knv2014, sak9813}@nyu.edu

***Abstract*—We present a portal that is designed to streamline the way students interact with their academic and professional data through the use of Retrieval-Augmented Generation (RAG) and personalized Large Language Models (LLMs). Our application integrates data acquired through separate sources such as Notion and Gmail and stores them in a manner which is easy to access and query. We walk through the various design decisions and performance trade-offs that we made both while designing the application and hosting it on a public cloud.**

***Index Terms*—Retrieval-Augmented Generation, Large Language Models, Data Integration, Academic Management, Automated Workflows**

I. Problem Statement

In today's fast-paced academic and professional environment, students and job seekers often struggle to manage the vast amount of information they encounter daily. Our system addresses this challenge by providing a centralized portal that seamlessly integrates and organizes critical data from various sources. The platform focuses on two key areas: class notes and job applications. For academic content, the system interfaces with Notion to retrieve and store user-provided notes, converting them into embeddings for efficient processing by a Large Language Model. Utilizing Retrieval-Augmented Generation, we ensure that any generated output is strictly based on the user's own notes, thereby minimizing hallucinations and providing personalized, relevant content. On the professional front, the system regularly scans users' email accounts for job-related updates, extracting and structuring vital information such as application deadlines, next steps, and current status into a Notion database. This comprehensive approach enables users to access both their academic materials and job application details in one cohesive interface, facilitating better schedule management and information retrieval. By centralizing these critical aspects of a user's academic and professional life, our system not only streamlines information management but also enhances overall productivity, allowing users to focus on what matters most - their studies and career progression.

II. Motivation

The motivation for developing an ecosystem for students stems from the complex challenges faced in today's rapidly evolving academic and professional landscape. Students are inundated with information from diverse sources, struggling to manage class notes, job applications, and other critical data efficiently. Traditional fragmented systems no longer meet the needs of modern learners, who require seamless integration of their academic and professional lives. There is a growing demand for personalized, adaptive learning experiences that align with individual goals and learning styles. By creating a comprehensive ecosystem, we aim to address these challenges, offering a holistic environment that supports students throughout their educational journey and into their careers. This approach not only streamlines information management and enhances productivity but also fosters a more engaging and effective learning experience. Ultimately, our motivation is to empower students with the tools and resources they need to navigate the complexities of higher education and the job market, preparing them for success in an increasingly competitive and dynamic world.

III. Existing Solutions

A. Job Application Tracking

Job tracking tools like Simplify offer valuable functionality for managing job applications, but they come with certain limitations:

- **Limited tracking scope:** These tools typically only track applications submitted through the browser where the extension is installed. This means applications submitted via mobile devices or other platforms may be missed.
- **Incomplete lifecycle tracking:** Many existing tools struggle to automatically track applications through various lifecycle stages such as applied, online assessment, interview, offer, and rejection. This often requires manual updates from the user, leading to potential inaccuracies and outdated information.

B. Retrieval Augmented Generation for Notes

For Retrieval Augmented Generation (RAG) applications on personal notes, one notable example is:

- **Reor:** An open-source AI note-taking app that implements RAG for personal notes. It offers vector search and RAG-based question-answering capabilities on user notes. However, Reor is still in early stages of development and lacks integration with broader ecosystems like job application trackingreor.

IV. Architecture

The system can be broadly divided into two flows, the *Notes Flow* keeps track of information that is updated by the user, such as class notes. The *Application Tracker* manages a user's inbox to track applications.

Fig 4. delineates the overall architecture diagram for our system, and the following subsections describe the design and trade-offs made to implement these flows in detail.

A. User Credentials & Permissions

As a one-time setup, each user needs to authenticate with our system using a google account on which they want to track applications. Once a user logs in, we store some basic authentication data on a firestore document.

Google Authentication The user would have to log in with their google account, and provide the application with permissions to read their emails. This is required in order for us to track their job applications.

Notion Tokens We need our user to add a new integration on their Notion account, and provide us with their access token. This is the standard practice for Notion integrations. We also need the user to provide us with Page IDs for two notion pages. The first is where the user stores their notes which will be used for our Notes Flow. The second is for storing Job Application updates.

B. Notes Flow

In this section, we discuss the various stages required to retrieve, embed, and serve notes that a user has saved on their Notion pages. At the end of this section, we will discuss how we optimize this flow.

Tracking Changes Once we have the parent page where our users store the data they want us to embed, we leverage Notion APIs to keep a track of new information that the user uploads. Notion provides us with a block level last modified timestamp, we use this to keep a track of which blocks contain information that we previously did not have. We track both changes made manually by the user, and the files that they upload within their notes page.

Retrieving Information Retrieval is done in a scheduled manner, we run a serverless function (SFN-1) every 5 minutes. This function talks to firestore to retrieve the list of all users from our database, it then runs one instance of SFN-2 and passes them 10 users each. SFN-2 now uses Notion APIs to check which of these users have fresh changes on their notes.

Embedding Once we have a list of user and blocks changed pairing, SFN-2 captures both the raw texts that were stored in the block, and any new files in the page. We extract the text content from any file that we receive, and then concatenate it to the raw texts.

Once we have each users' changes in a string format, we divide this data into chunks of 1,000 characters. The Chunk IDs are generated using the user's email, the page ID, and an extra numerical suffix for each of the chunks. This serves two purposes. One, it lets us update our embeddings in place, since every location changed in the user's Notion will generate the same Chunk ID. Two, it lets us use a user's email as a primary key to ensure that we only search chunks associated with this user.

Finally, we use *VertexAI* to convert these chunks into embeddings and store these. Along with these embeddings we also store Chunk IDs as a reference to enable us to properly implement the updating and querying of these embeddings.

Querying Once a user's query has been passed to your backend, we send it to a serverless function that processes these queries (SFN-3). SFN-3 first embeds the user's query so that it can be matched against our Vector DB. Using *VertexAI*, and the user's email as a primary key, we match the embedded query against all the Chunks who's IDs start with the user email. Once we find the query's nearest neighbors, SFN-3 has access to both the query and the context within which this query was made.

As a final step before returning the output to the frontend, SFN-3 sends the query and the context to the *text-embedding-005* LLM model along with a prompt which asks it to properly format the context as required by the query.

Scaling & Tradeoffs

Unlike the application tracker that we will discuss later, the notes flow is less sensitive to latencies. We are therefore free to pick a schedule for our initial cron job in a more relaxed manner.

The main design decisions we contended with are the process of information retrieval from Notion. Instead of employing a web-hook that will send us a push notification every time a user's page changes, we decided to periodically poll Notion for this information. This lets us batch process multiple user changes in a single invocation of our SFN-2 function, which is the most computationally expensive function in this flow. To scale out the embedding process, we wanted to leverage multiple concurrent nodes. However, without a load balancer and a scheduler, it would be hard to eliminate two concurrent nodes processing the same user's query. This is why we decided to pass over these responsibilities to SFN-1.

As mentioned earlier, SFN-1 runs every 5 minutes, and only invokes 1 instance of SFN-2 for every 10 users. This parameter can be easily tuned in case we deal with differing traffic patterns. All these jobs runs in parallel. If one of these invocations crash before processing the embeddings, SFN-1 can simply re-invoke SFN-2 for the batch of users that failed. In our estimations, SFN-2 takes much less than 5 minutes to process 10 users.

Finally, our choice of vector search engines and LLM models were limited by cloud costs. We were only able to leverage a very basic LLM model (text-embedding-005) on Vertex AI. This model produces 768 dimension embeddings, however, in an ideal scenario, we would use a high-end embedding model to generate better embeddings in higher dimensions.

C. Application Tracker

In this section, we discuss the various stages that our application tracker goes through, and finally we talk about how these stages are connected and how well they can scale.

Retrieval Using Google Cloud Platform's GMail API, we can set up a simple serverless function (SF-1) which is triggered every time a user in our database receives a new email.

This function extracts the content of the email and pushes them to a queue (Queue-1) for further processing. Each message contains the receiver's email address and the content of the email.

Classification The key intuition behind this is that we need a cheap way to detect whether the email in question is related to a job application. This helps to avoid unnecessary computation in the form of processing every single email a user receives.

It is trivial to train a simple TF-IDF model that can detect whether an email is likely to be relevant based on the frequency of certain words and phrases. As we shall see in the next step, it is acceptable to have false positives at this stage. For our project, we trained one such model and stored it on a bucket on GCP's Cloud Storage.

Next, we created a serverless function (SF-2) that when triggered reads messages from Queue-1, loads the NLU model from Cloud Storage, and passes the emails through our model. The model outputs whether an email is relevant or not. All relevant emails are passed to another queue (Queue-2). Irrelevant messages are dropped.

Extraction It's important that we extract information that all relevant emails are likely to have. This will help us store this information in a format that is easy to read for the user. From all relevant emails, we extract the company applied to, the position applied for, status of the application, any upcoming deadlines, and a basic note summarizing the contents of the email.

To perform this, we run a serverless function (SF-3) that calls GCP's *gemini-flash-1.5* LLM model that can be accessed through the *VertexAI* service endpoints. We send this LLM a list of email bodies, and ask it to return us the information in a JSON format. Any data not present in the email is later substituted with a default value.

Once we have the outputs from the LLM, we match these with the user emails and send them to a final queue (Queue-3) from where we can push it asynchronously to each individual user's NotionDB.

The decision to use a pre-trained LLM model for this task was taken after comparing the costs and performances that self-hosted models could offer. It was pertinent that the model in the extraction stage have high performance, otherwise it would be impossible to store any meaningful information on Notion. In our experience, the costs associated with hosting even small models such as LLaMa-2 would add significant costs to our system while still sacrificing on the outputs.

Storing Finally, once we have all the relevant application data for each user, we need to push this to a NotionDB. For this, we run SF-4 which reads from Queue-3, and then makes API calls to Notion to store each application update.

One important point to note is that Notion provides us with granularity for API keys on a user level. This implies that we can run parallel calls to write the final output to Notion. Since each API key comes with its own rate limits, we do not need to throttle these calls. Each API key would be writing a limited number of entries to their respective NotionDBs, and a single user is unlikely to have 700 application updates in a second.

Pipeline & Scaling As is obvious, we connect various stages of our pipeline through the use of queues - on GCP, the implementation for message queues is *Pub/Sub*. The use of message passing instead of API calls during this flow helps us decouple these functions from one another. In the case of a failure at any stage, we can fix the crashing services without losing any data. Moreover, each message is only acknowledged if it has been properly processed and pushed to either the next queue or to the final NotionDB.

An important concern for us was to take into account how well this pipeline would scale computationally for multiple users. Apart from this, we also had to take into account that the application updates are often time-sensitive and need as low of a latency as possible. Traditionally message passing systems either have functions triggered after a certain number of messages have reached a queue (batch processing), or to run functions on a schedule. In our case, it made more sense to use a different technique for different components. We will now walk through these decisions.

SF-1 receives a push-notification as soon as a new email arrives, which it then places into Queue-1. This follows the standard practices prescribed by Google. Since all this function does is store these emails in a queue, it is not computationally expensive to run. On average, this function runs for X seconds at every invocation.

SF-2 and SF-3 are more computationally expensive to run. The former needs to load a model from Cloud Storage and the latter invokes an LLM. It is obvious that we need to perform batch processing for these functions however this causes our latencies to take a hit. Suppose we ask SF-2 to be triggered at every 100 messages, this might lead to cases where 99 messages are left unprocessed for a really long time and have to wait for one of our users to receive a new update.

The solution we came up with is to run SF-2 every 2 minutes and process all emails that are received in the interim in one go. Suppose across all our users combined we receive 1000 emails every minute, our SF-2 will need to load the classification model only once to process 2000 emails. We can easily tweak the invocation schedule to deal with higher traffic. We can even come up with a staggered schedule to invoke multiple instance of SF-2 for this.

SF-2, once it has classified these 2000 emails, will batch all the relevant messages into one single message and forward it to SF-3. This allows SF-3 to be triggered for every new message, avoiding the reliance on a cron job. In this scenario, SF-3 gets triggered every 2 minutes as well in case we have a steady flow of relevant emails. If we consider that 10% of all user emails are related to job applications, we need to invoke the LLM only once for roughly 200 emails. Another important aspect of not triggering SF-3 on a schedule is that for the period of time where there are no relevant emails, we do not need to invoke SF-3, further reducing costs. This approach helps to couple these two related functions, while still providing granularity in how we update or fix bugs within these components.

Finally, we can trigger SF-4 either periodically or on trigger. This part of the pipeline requires the least amount of scaling optimizations, given that our calls end up going to different endpoints with different rate limits as discussed earlier.

D. Frontend Application

For our User Interface, we run a simple Flask application that hosts our website. This Flask application is deployed on GCP's App Engine. This application is scaled using Load Balancers to properly distribute user traffic.

The application has two pages -

- **Settings Page** which handles the login flow, and allows a user to enter their Notion token, and Page IDs.
- **Query Page** which has a simple text box for a user to search their notes. This calls SFN-3 function and passes it the query and the logged in user's email.

V. Architecture Diagrams

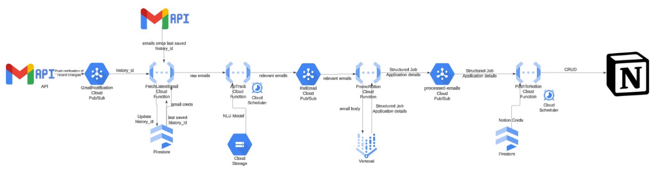
API
emails since last saved history_id
API
Push notification of recent changes
GmailNotification Cloud Pub/Sub
history_id
FetchLatestEmail Cloud Function
raw emails
ApTrack Cloud Function
Cloud Scheduler
relevant emails
RelEmail Cloud Pub/Sub
relevant emails
PremoNotion Cloud Function
Structured Job Application details
processed-emails Cloud Pub/Sub
Structured Job Application details
PushToNotion Cloud Function
Cloud Scheduler
CRUD
gmail creds
Update history_id
last saved history_id
Firestore
NLU Model
Cloud Storage
email body
Structured Job Application details
Vertexai
Notion Creds
Firestore

Fig. 1. Architecture Diagram for the Application Flow

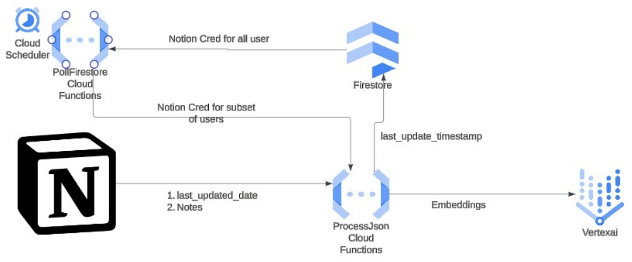

Fig. 2. Architecture Diagram for Retrieval and Embedding of Notes

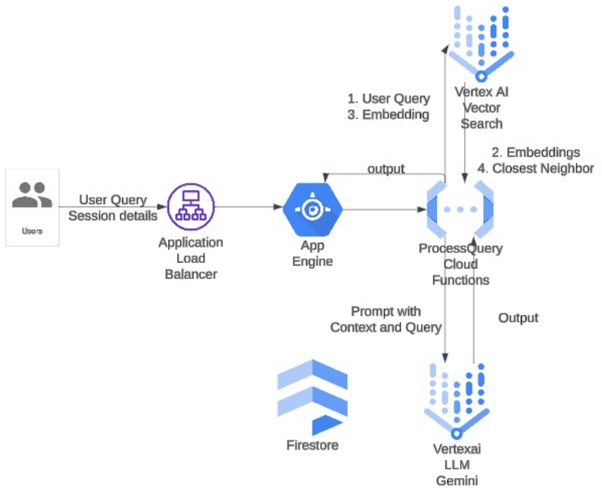

Fig. 3. Architecture Diagram for Querying Notes

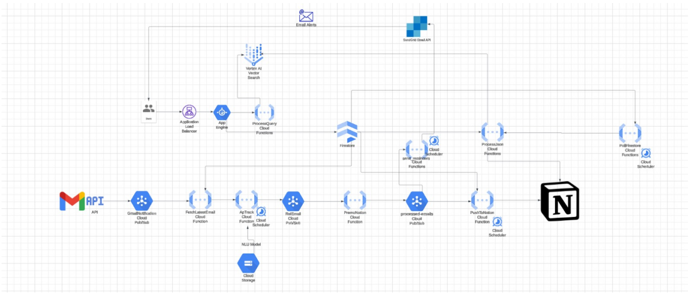
Email Alerts
SendGrid Email API
Vertex AI Vector Search
Application Load Balancer
App Engine
ProcessQuery Cloud Functions
Firestore
send_reminders Cloud Functions
Cloud Scheduler
ProcessJson Cloud Functions
PollFirestore Cloud Functions
Cloud Scheduler
API
GmailNotification Cloud Pub/Sub
FetchLatestEmail Cloud Function
ApTrack Cloud Function
Cloud Scheduler
RelEmail Cloud Pub/Sub
PremoNotion Cloud Function
processed-emails Cloud Pub/Sub
PushToNotion Cloud Function
Cloud Scheduler
NLU Model
Cloud Storage

Fig. 4. Architecture Diagram for the entire application

VI. Future Work

Our ecosystem for students has significant potential for expansion and enhancement. We have identified several key areas for future development:

A. Video Lecture Integration

We plan to integrate our system with video conferencing platforms such as Zoom. This integration will allow students to search for specific information within recorded video lectures This feature will enhance the learning experience by making video content as accessible and searchable as written notes.

B. Advanced Job Application Analytics

While our current system stores job application data on the student's Notion dashboard, we aim to develop more sophisticated analytical capabilities:

- Implement quantitative queries on job application data through our chatbot interface
- Provide statistical insights on application success rates, interview performance, and offer trends
- Develop predictive models to suggest optimal application strategies based on historical data

These enhancements will empower students with data-driven insights to optimize their job search strategies.

C. Expanded RAG Capabilities

We intend to expand our Retrieval Augmented Generation (RAG) capabilities to include:

- Multi-modal RAG incorporating text, images, and video content
- Improved context understanding for more accurate and relevant responses
- Personalized RAG models that adapt to individual student's learning styles and preferences

These advancements will further personalize the learning experience and enhance the utility of our ecosystem for students.