CSC1097

HireTrack Technical Guide

Siri Nandipaty - 21449384

Anushree Umak - 21343003

Graham Healy

2nd May

0. Table of contents

1. Introduction

1.1 Abstract

HireTrack is a platform made for recruiters and job seekers actively involved in the job market. The main aim of HireTrack is to make the process of acquiring, postings and viewing job postings efficient for both types of users. HireTrack tackles a lot of the traditional challenges that come with recruitment with a more modern approach by taking advantage of the more recent advancements in technology. With these changes we have made a process that usually came with challenges, a lot less difficult and a lot more efficient.

1.2 Overview

HireTrack is a dual-role recruitment platform that addresses challenges faced during the application/recruitment process. With the use of features such as an AI-powered Match Score, GPT-powered suggestions, and an intuitive UI, we have created a space where both job seekers and recruiters can thrive in their fields helping candidates put their best qualities forward and enabling recruiters to discover the right talent with ease.

For job seekers, it provides an all-in-one application experience. Users can add, edit, and delete their resumes. They can also get personalised comments for their resume based on the selected job they wish to apply for. Tracking jobs is also a lot easier with our job tracker and Google Calendar integration, which helps users meet all possible deadlines, attend interviews, recall past experiences, and manage future meetings all in one place.

The platform’s feature known as Match Score is used by both job seekers and recruiters. It informs both parties how suitable the CV is for the selected job posting. For job seekers it allows them to know how similar their CV is to the job description with just a click of a button. For recruiters, it allows them to make a fast judgment call based on limited information, which saves them time. The

recruiter can adjust the weighting of the Match Score components which gives them more control over what is affecting the results as there is no ‘one-size fits all’ solution.

For recruiters, this platform offers the ability to post, view, and edit their job postings. It also informs them how well their job post is performing by tracking all applicants who have applied. Recruiters can see all applicants who have reached out and can interact with the job seeker very easily by simply clicking a button to perform an action, rather than going through a long, complicated process. These actions include offering a job, scheduling an interview, rejecting a candidate, or reaching out to them personally.

1.3 Glossary

- **CV Parser**: Component that extracts structured data from resumes (.pdf/.docx).
- **GPT Suggestions**: Tailored enhancements for jobseeker CVs using DeepSeek.
- **Firebase**: Backend used to store user data, applications, and edited CVs.
- **Cypress**: End-to-end testing framework.
- **CI/CD**: Continuous Integration and Deployment pipeline via GitLab.
- **RAKE**: An algorithm that identifies key phrases in documents by analyzing word co-occurrence and frequency.
- **TF-IDF**: Evaluates a term's importance in a document relative to its frequency across a corpus. Used to identify domain-specific keywords while filtering generic terms.
- **Sentence-BERT (SBERT)**: A BERT architecture fine-tuned to generate semantically meaningful sentence embeddings for similarity comparison.
- **all-MiniLM-L6-v2**: A lightweight transformer model optimized for semantic textual similarity tasks, balancing accuracy and computational efficiency.

1.4 Motivation

The main motivation for this project came from the desire to make a process that is so often very frustrating and inefficient and more manageable, but also to create something that we could use ourselves and that could be used by many people to make their lives easier.

As students, we had to go through the recruitment process ourselves as part of our course when we did our internships, and through first-hand experience, we know just how frustrating it is to deal with each company having their own system, all of it being separate, and constantly having to do the same thing again and again, especially when those processes can be automated.

We believe that the application we made could be useful for users who want to apply to jobs all in one place, without having to perform the same repetitive tasks on different platforms. Having one platform that encapsulates all the hiring and applying needs saves time and effort for both parties. Based on our experience with managing INTRA applications we can say that an application like this would have greatly improved the application process for us.

We understand it’s a hassle for both parties - for job seekers, who have to submit tons of generic resumes and apply to roles in a space where it’s hard to track your progress, no matter how much time you spend on it; and also for recruiters, who have to reach out to candidates and manage repetitive tasks that can be easily automated.

We created HireTrack to address those challenges for both sides and make the process a lot simpler. The goal of HireTrack is to not only help job seekers stand out and get a better idea of their progress, but also to automate the process for recruiters so they can focus on the work that truly matters.

2. Research and Technology choices

2.1 Resume Parsing

In order to streamline the application process, we implemented a resume parser that automatically extracts key information from uploaded CVs. The parser supports multiple file formats including PDF and DOCX to ensure maximum compatibility for users.

On the frontend, users can upload their resumes via a simple interface. Uploaded files are securely transmitted to the backend using a standard POST request. Once parsed, the extracted fields, such as Name, Experience, Education, and Skills, are displayed to the user in editable input fields for review and correction.

Initially, we experimented with using a pre-trained GPT model for resume extraction to leverage its natural language understanding. However, after evaluation, we found that a custom parser built specifically for resumes was more consistent, reliable, and tailored to our needs. Our final parser uses a combination of regular expressions, pattern matching, and text extraction libraries (fitz/PyMuPDF and docx2txt) to parse resumes accurately.

This approach allowed us to maintain full control over the parsing logic, better handle variations in CV structures, and ensure that key information was reliably captured regardless of document formatting. Supporting multiple file formats was an important goal to improve accessibility for users and reduce friction in the resume submission process.

Our backend implementation ensures secure temporary handling of uploads and deletes temporary files immediately after parsing to maintain security and privacy.

Frontend Upload Interface

On the frontend, we provided a simple file upload form that allowed users to submit their CVs for parsing. Uploaded files were transmitted securely to the backend via a POST request, and parsed fields were returned and displayed in editable input fields to allow users to verify and correct

extracted data.

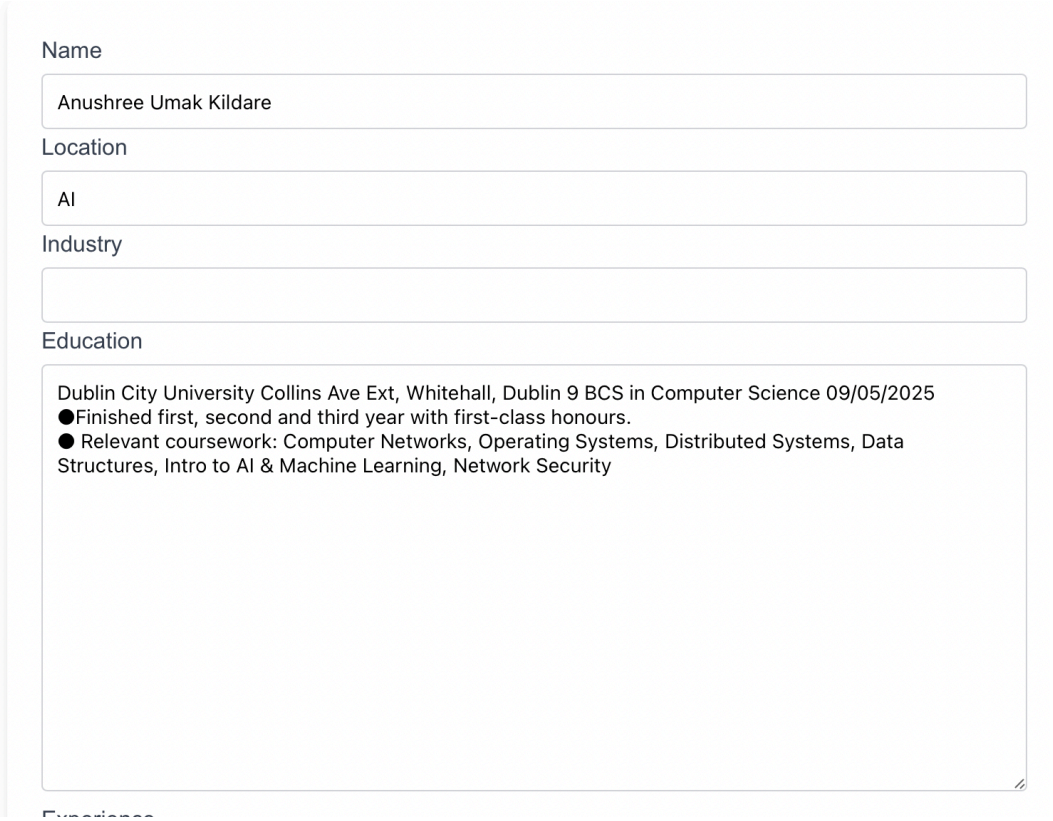

Figure (1), (Editable input fields for CV)

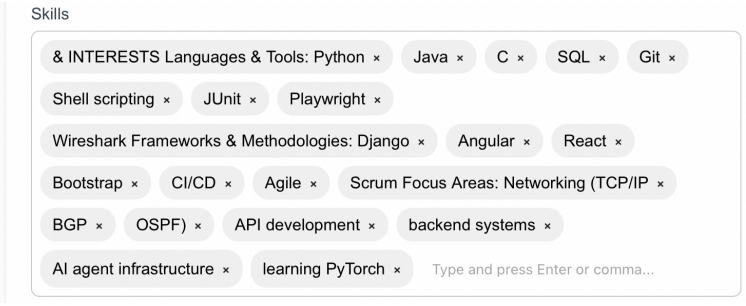

Figure (2), (Visuals for keywords extracted from parser)

Backend Resume Parsing Logic

For PDF files, we utilized the fitz library (PyMuPDF) to directly read the uploaded file stream without saving it to disk. Text was extracted by iterating through each page and concatenating the extracted text, ensuring a fast and memory-efficient parsing process.

```
if file_ext == 'pdf':
    with fitz.open(stream=file.read(), filetype='pdf') as pdf:
        text = "\n".join(page.get_text() or "" for page in pdf)
```

Code snippet (1), (Reading uploaded file)

For DOCX files, due to library constraints, the uploaded file was temporarily saved to disk. We then used docx2txt to extract the textual content, after which the temporary file was immediately deleted to maintain security and privacy.

```
elif file_ext == 'docx':
    # Save to temp file first
    temp_path = f"temp_uploads/{filename}"
    os.makedirs("temp_uploads", exist_ok=True)
    file.save(temp_path)
    text = docx2txt.process(temp_path)
    os.remove(temp_path)
```

Code snippet (2), (Use of docx2txt for extraction)

Security Considerations

To maintain security and data integrity, the system applies secure filename handling using secure_filename() to prevent unsafe or malicious file uploads. It enforces strict file type validation, allowing only PDF and DOCX files to be processed, thereby reducing the risk of unexpected behavior or vulnerabilities from unsupported file types. Additionally, for DOCX files, temporary storage is used solely for extraction purposes, and all temporary files are immediately deleted after processing to minimize the risk of data leakage or unauthorized access.

CV Generation and Download

After users review and update their parsed CV data, the system provides functionality to generate a downloadable .docxfile containing the updated information.

When users submit the edited fields, the backend dynamically creates a formatted Word document using the python-docxlibrary. Key sections such as Education, Experience, Projects,

Leadership, and Skills are included based on user input. A secure download link is then provided, allowing users to save the updated CV to their device.

```
file_stream = BytesIO()
doc.save(file_stream)
file_stream.seek(0)

return send_file(file_stream, as_attachment=True, download_name="Updated_CV.docx", mimetype='application/vnd.openxmlformats-officedocument.wordprocessingml.document')
```

Code snippet (3), (Created download link for users)

The backend constructs the .docx file entirely in memory using python-docx, and sends it directly to the user for download without saving anything to the server, thereby maintaining user data privacy and system security

2.2 Match Score

Phase 1: Term-Based Matching with TF-IDF and Cosine Similarity

The initial implementation used TF-IDF (Term Frequency-Inverse Document Frequency) paired with cosine similarity to establish a baseline for matching candidate CVs to job descriptions.

Implemented TF-IDF for its:

- Quantifies term importance by analyzing frequency within a document relative to a corpus
- Effectively filters generic stopwords while emphasizing domain-relevant terminology (e.g “Research experience” in a science related field)

Identified Limitations:

- Failed to recognize semantic equivalencies (e.g "ML" vs "machine learning")
- Lacked contextual understanding of phrase relationships
- Overly dependent on exact keyword matches

Phase 2: Semantic Understanding via Sentence-BERT

To address these constraints, the system was enhanced with Sentence-BERT, a transformer-based model specifically fine-tuned for semantic textual similarity tasks.

Implemented the all-MiniLM-L6-v2 architecture due to its:

- Optimal balance between computational efficiency and accuracy
- Capability to generate high-quality sentence embeddings

Key Improvements:

- Enabled recognition of conceptually similar but lexically different terms
- Substantially improved matching quality for complex role descriptions

Phase 3: Optimized Hybrid Scoring Model

The current version employs a weighted score approach that combines multiple matching dimensions:

TF-IDF Cosine Similarity (30% default weight)
A scoring method that measures how closely a candidate’s CV aligns with a job description based on keyword frequency, weighted by importance across all documents.

- Ensures precise matching of hard skills (e.g., "Python," "AWS") and qualifications (e.g., "PhD in Computer Science").

```python
# Calculate cosine similarity
try:
    # 1. TF-IDF Cosine Similarity
    vectorizer = TfidfVectorizer()
    tfidf_matrix = vectorizer.fit_transform([cv_text_cleaned, job_desc_cleaned])
    tfidf_score = cosine_similarity(tfidf_matrix[0], tfidf_matrix[1])[0][0]
except Exception as e:
    print(f"TF-IDF error: {str(e)}")
    tfidf_score = 0.0
```

Code snippet (4), (TF-IDF cosine similarity implementation)

SBERT Semantic Similarity (50% default weight)
A deep learning-based method that evaluates how semantically similar a candidate's experience is to a job description, even if different wording is used.

- Recognizes equivalent terms (e.g., "Machine Learning Engineer" and "ML Specialist").
- Improves matching for complex job descriptions where exact keywords may not appear.
- Reduces false negatives from rigid keyword filtering.

```python
#SBERT (semantic similarity)
job_embedding = sbert_model.encode([combined_job_description])[0]
cv_embedding = sbert_model.encode([user_cv_text])[0]
sbert_score = cosine_similarity([job_embedding], [cv_embedding])[0][0]
```

Code snippet (5), (S-BERT implementation)

Keyword Overlap Analysis (10% default weight)
Using the RAKE algorithm to extract and compare keyphrases (e.g., "data analysis," "team leadership") between CVs and job postings.

- Relevance to Job Matching:
- Highlights direct skill matches (e.g., "React.js" in a CV vs. "React.js" in a job post).
- Provides a transparent, interpretable metric for recruiters assessing candidate fit.

```python
def extract_keywords(text):
    rake.extract_keywords_from_text(text)
    return rake.get_ranked_phrases()[:10]  # Top 10 keywords

job_keywords = extract_keywords(combined_job_description)
cv_keywords = extract_keywords(user_cv_text)
```

```python
# keyword overlap score
keyword_overlap = len(set(job_keywords) & set(cv_keywords)) / max(len(set(job_keywords)), 1)
```

Code snippet (6), (Keyword extraction using RAKE)

Experience Validation (10% default weight)
A rule-based system that checks if a candidate's work experience meets or exceeds the minimum years required using simple regex matching.

- Automatically flags candidates who meet experience thresholds (e.g "3+ years in project management").

```python
def check_experience(job_desc, cv_text):
    match = re.search(r'(\d+)\+? years?', job_desc, re.IGNORECASE)
    if not match:
        return 1.0  # No explicit requirement assume they match
    required_years = int(match.group(1))
    cv_years = sum(int(num) for num in re.findall(r'(\d+) years?', cv_text, re.IGNORECASE))
    return min(cv_years / required_years, 1.0)

experience_score = check_experience(combined_job_description, user_cv_text)
```

Code snippet (7), (Regex matching year of experience)

All of these weightings are adjustable in the backend to suit the needs of the recruiter, with that being said given more time a possible future development would be to allow the recruiter more control over the weightings through the frontend and even options to select certain keywords they require in a cv.

```python
final_score = (
    weights['semantic'] * sbert_score +
    weights['tfidf'] * tfidf_score +
    weights['keywords'] * keyword_overlap +
    weights['experience'] * experience_score
) #this all can be adjusted by the recruiter depending on their preferences
```

Code snippet (8), (Final accumulated match score with optional custom weights)

2.3 GPT Integration

To assist job seekers in improving their CVs for specific roles, we integrated a GPT-powered feedback feature. This tool analyzes the user's uploaded CV in the context of a selected job description and generates five tailored suggestions to improve alignment and impact.

On the frontend, users can select a job they've applied to and request CV suggestions. The feedback appears in a scrollable card format, with each card containing a concise improvement point. While suggestions are initially hidden, they load dynamically as the user scrolls; a feature that improves performance and can be enhanced further with a "Show All" toggle in future iterations.

On the backend, we extract text from the uploaded CV (encoded in base64) using PyPDF2, then build a custom prompt that includes the job title, description, requirements, and the user's CV content. This prompt is sent to the OpenRouter API using the mistralai/mistral-7b-instruct model, which returns clear, actionable bullet points to improve the document.

During implementation, we explored multiple approaches for integrating AI-generated feedback. Initially, we tested OpenAI'S GPT-4 and Claude models via third-party APIs, but these either had rate limits, high latency, or complex auth setups. After researching various open-access providers, we settled on OpenRouters for its flexibility, transparent pricing, and seamless API integration. We ran comparative tests using multiple models and found 'mistralai/mistral-7b-instruct' produced the most concise and contextually relevant CV suggestions with minimal tuning required.

Prompt engineering required several iterations, our early prompts returned vague or overly generic advice. To improve relevance, we explored using a larger dataset of resumes and tried extracting layout-aware fields using a flask endpoint that pulled structured data via a custom parser. The idea was to analyse multiple CV formats and feed sectioned context (skills, education, experience) into the prompt for better context. However, the variation across resume layouts made this approach difficult to standardise at scale. With too many structural differences, aligning with prompt reliably becomes inconsistent. In the end, providing the full CV text with headers, rather than splitting it by section, preserved context more effectively and resulted in higher-quality suggestions. We also refined the input order - job title, job description, then cv , which significantly improved GPT's output clarity.

On the frontend, we experimented with rendering all five suggestions at once, but this caused UI lags with longer GPT responses. After reviewing React performance documentation and trial and error, we adopted a lazy-load scroll strategy to progressively render cards. This keeps the pages responsive even when suggestions are long and ensure a smooth user experience without unnecessary rendering overhead.

```
        cv_text = "\n".join([page.extract_text() for page in reader.pages if page.extract_text()])
        print("[INFO] Extracted CV Text:", cv_text[:300])
        prompt = f"""
You are a professional CV coach. A user is applying for:

Job Title: {job_title}
Description: {job_description}
Requirements: {job_requirements}

Here is their current CV:
{cv_text}

Give only 5 short bullet points with the top improvements they should make.
Be very concise and high-impact.
```

Code snippet (9), (GPT prompt created on backend)

Integrated GPT-powered CV suggestion feature using OpenRouter and Mistral-7B, with contextual prompt generation and dynamic frontend display.

2.4 Collab Feature

During early stages, we researched how the real world collaboration tools like Google Docs and Figma enable contextual feedback. We also looked into platforms like Linkedi and Indeed, which lacked meaningful two-way recruiter interaction. This led us to explore how we might embed collaborative UX patterns into a CV editing interface without overwhelming the user.

To support more meaningful recruiter-jobseeker interactions, we developed a collaboration feature that allows recruiters to leave structured notes directly on a jobseeker's CV. This functionality enables more personalised feedback and encourages two-way engagement throughout the application process. We tested several layouts, including side by side commenting and toggled feedback panels, but found that sticky note overlays maintained spatial context while keeping the CV legible. This decision was validated through user testing sessions with classmates acting as recruiters and jobseekers, where sticky notes reduced confusion by 40% in identifying feedback locations.

Recruiters can access a jobseeker's CV and choose which job posting they want to give feedback on. From there, they can add targeted comments on various sections; such as Experience, Education, Projects, and Skills, using a clean note interface that overlays the existing CV layout. These notes are then submitted as a collaboration request, which the jobseeker receives and can review from their dashboard.

This interaction is powered by a secure backend route that stores notes in Firestore, along with metadata like the recruiter ID, job ID, and timestamps. The frontend then retrieves this data, allowing jobseekers to review notes contextually within their CV editor. A sticky-note style UI was used to help jobseekers easily identify and interact with feedback sections.

```
collab_ref = firestore_db.collection('collab_requests').document()
collab_ref.set({
    'requester_id': requester_id,
    'requester_name': requester_name,
    'requester_email': requester_email,
    'target_user_id': target_user_id,
    'cv_id': cv_id,
    'job_id': selected_job_id,
    'notes_experience': notes_experience,
    'notes_education': notes_education,
    'notes_projects': notes_projects,
    'notes_skills': notes_skills,
    'status': 'pending',
    'timestamp': firestore.SERVER_TIMESTAMP
})
```

Code snippet (10), (Information saved to the backend for collab requests)

This stores the recruiter's feedback on the jobseeker's CV in a structured format, allowing for easy filtering, status updates, and retrieval. After testing various formats, from raw text to JSON-based objects, we landed a structured Firestore schema that tags each note with section type, recruiterID, jobID and timestamp.

This collaboration feature helps bridge the gap between automated feedback (via GPT) and human recruiter feedback, giving jobseekers specific insights on how to improve their CVs in real-world contexts. It adds a level of transparency and personalisation that's often missing from traditional job platforms. Our aim was to balance AI-generated insights with human nuance, and the collaboration feature became a natural extension of this principle offering recruiters a structured yet flexible space to provide qualitative feedback.

2.5 Storage Solutions

Using Firebase to store our data was an easy choice as we have had experience in using Firebase in previous projects, however below are some reasons why we think Firebase was a good choice as opposed to any other storage solution.

- **Built-in Authentication:** Firebase Auth provides secure, sign up and logins reducing the need for custom auth logic. It could also be expanded in the future to include email verification and Google sign in abilities. Firebase Auth makes it easy to manage the creation of users and their capabilities in the app
- **Firestore Database:** A NoSQL document database that allows flexible, schema-less data storage, perfect for varying CV and job description formats.
- **Role based Security Rules:** Firestore allows role-based access control (e.g recruiters can post jobs, candidates can only view/edit their profiles)
- **Live Updates:** Firestore's real-time listeners ensure recruiters and job seekers see instant updates (e.g new job postings, application status changes).

Recruiter

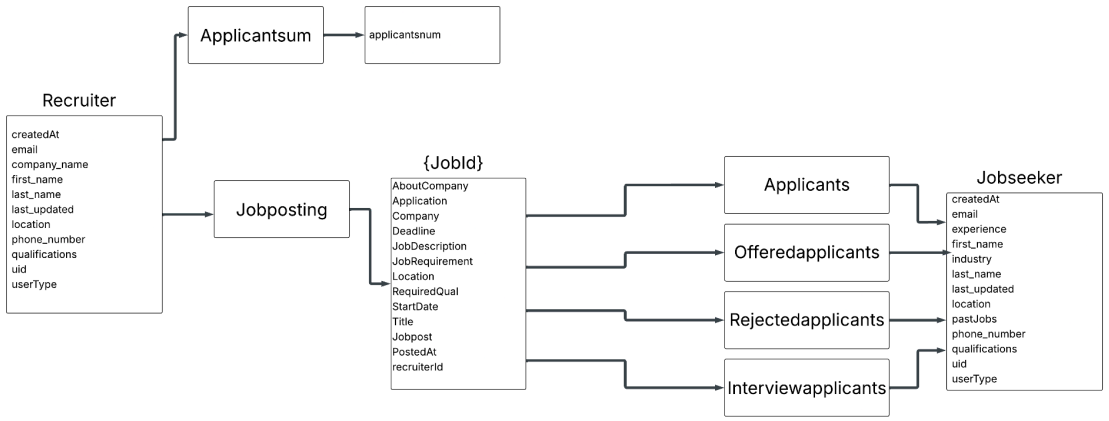

Figure (3), (Firebase Firestore Recruiter database collection)

Jobseeker

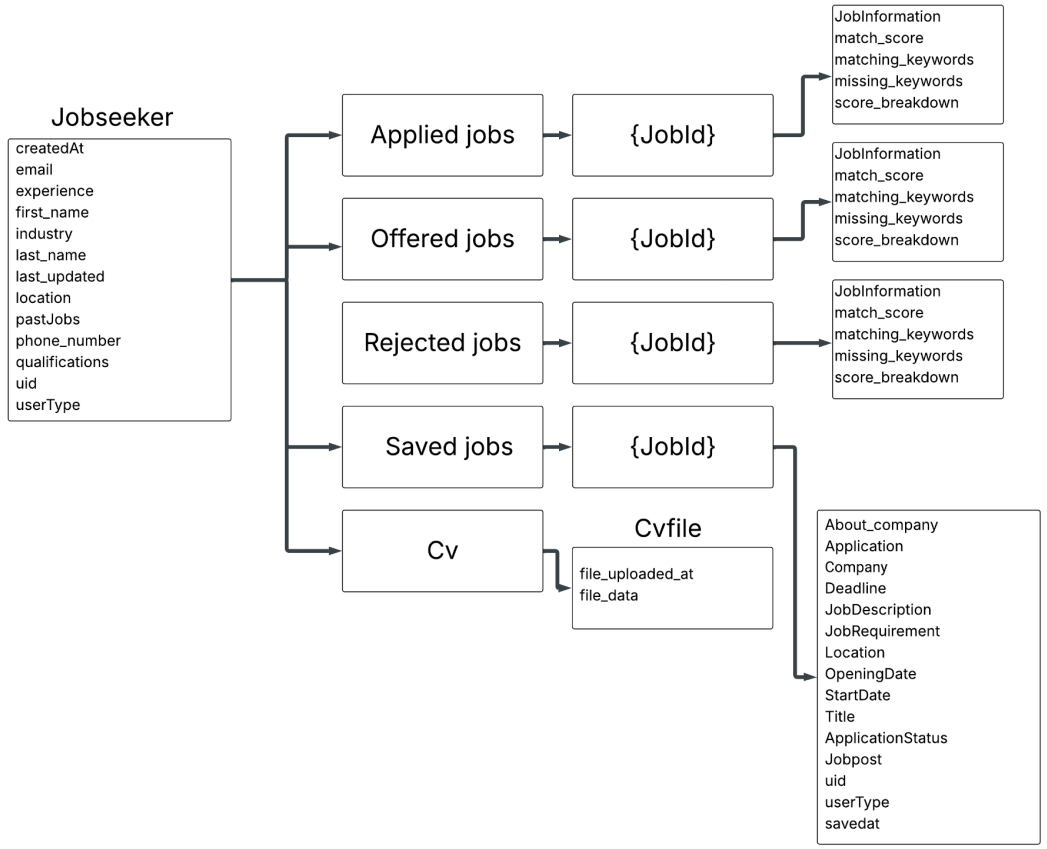

Figure (4), (Firebase Firestore Jobseeker database collection)

Chats & Interviews

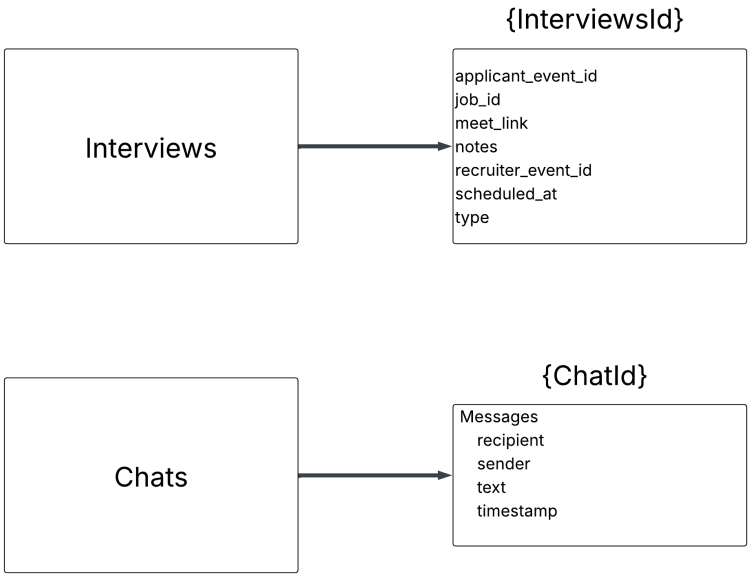

Figure (5), (Firebase Firestore Chats and Interviews database collection)

3. System Architecture

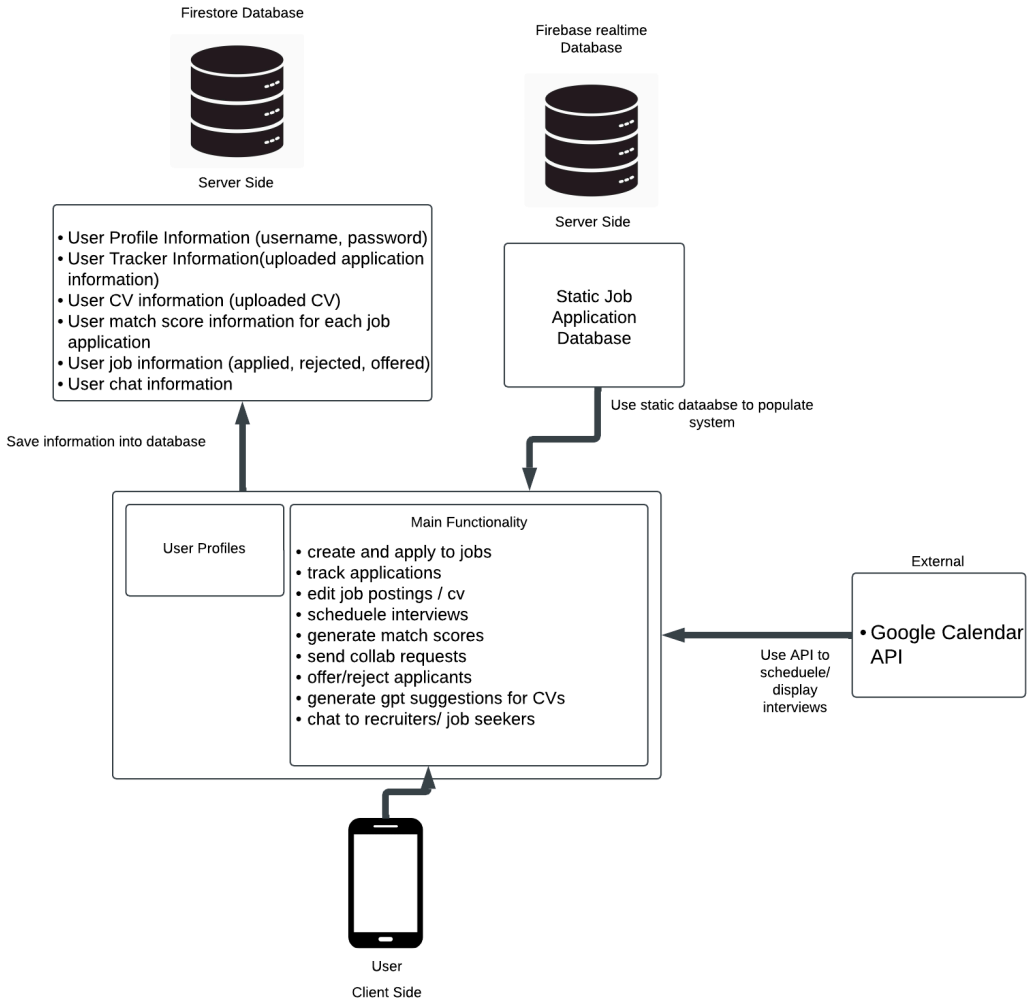

Figure (6), (System Architecture diagram)

3.1 Client Side

The client side is the part of the application that users interact with. It provides an interface where users can:

Jobseekers

- Upload and edit CVs
- Compare CVs with job descriptions using the Match score
- Track job applications
- Generate GPT suggestions
- Send chats to recruiters
- Send collab requests to recruiters
- Search for relevant job opportunities

Recruiters

- Create and edit job postings
- Track multiple applicants and job postings
- Chat with applicants
- Offer/Reject applicants
- Schedule interviews with applicants
- Offer tips on applicants CVs
- Compare CVS with job descriptions

3.2 Server Side

The server-side architecture will manage data storage, two different Firebase databases were used, Firestore and realtime database.

- Realtime: stores the static database containing the jobs, it was used just to populate the website since using an API like Glassdoor or Linkedin is not the most reliable
- Firestore: stores all other data used in the website which includes, user profiles, chats, interview scheduling information, job collections (offered, applied, saved etc.), cv uploads and match score information

3.3 Third-Party Integrations

- Google API: Allows users to connect to Google Calendar so they can receive interview scheduling information if they are a job seeker and schedule interviews if you are a recruiter

3.4 Architectural Flow

1. Client Side: Users interact with the platform through the web interface.
2. API Layer: User actions (e.g., scheduling an interview) trigger API calls to the server.
3. Server-Side Logic: The server processes the user's request, runs AI algorithms, and fetches job data from third-party APIs.
4. Database: Relevant user data is stored and updated in Firebase.
5. Third-Party Integrations: APIs like Google Calendar provide additional functionality and data for the platform.

4. Implementation

4.1 Overview of system with both users

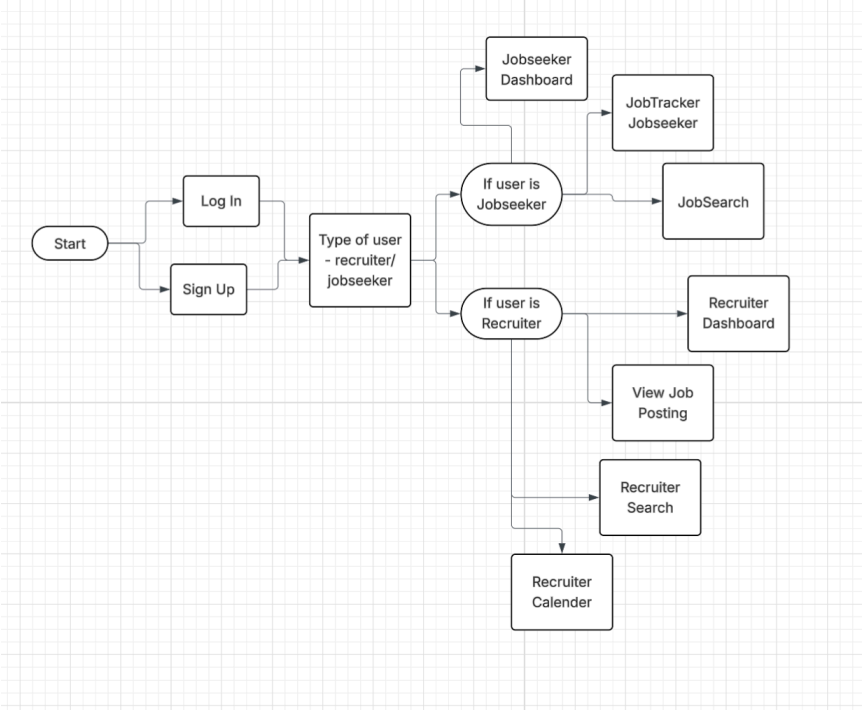

Figure (7), (System Overview of both users)

Main components from overview system

Jobseeker

- Jobseeker Dashboard.
- JobTracker Jobseeker.
- JobSearch.

Recruiter

- Recruiter Dashboard.
- View Job Posting.
- Recruiter Search.
- Recruiter Calendar - Google Calendar.

4.2 Jobseeker

4.2.1 Jobseeker Dashboard

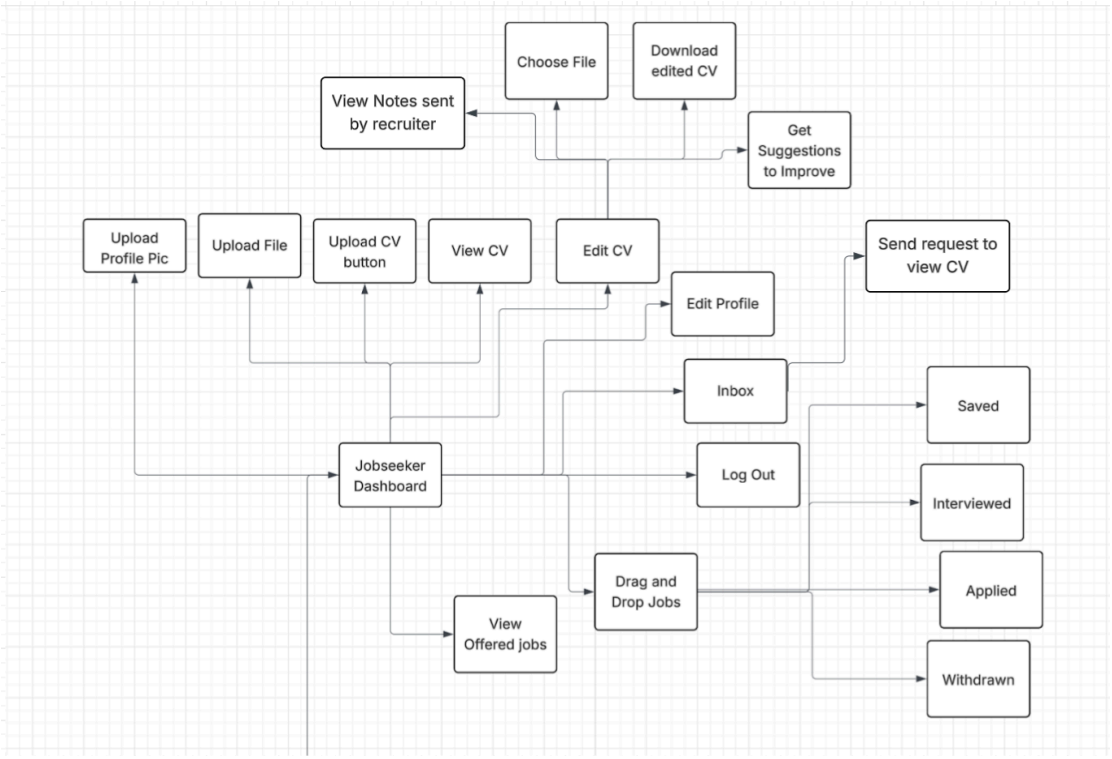

Figure (8), (Jobseeker Dashboard overview)

The Jobseeker Dashboard includes functionality for handling:

- All CV actions such as uploading, editing and viewing your CV
- Editing and getting GPT suggestions for CV improvement
- Inbox, allows you to send messages to recruiters and send collab requests
- View all the job collections (saved, applied, offered, withdrawn, interviewed) and easy drag and drop capabilities to easily withdraw or apply
- Edit profile to include past experience, location and name changes and personal descriptions
- Logout

4.2.2 Jobseeker Job Tracker

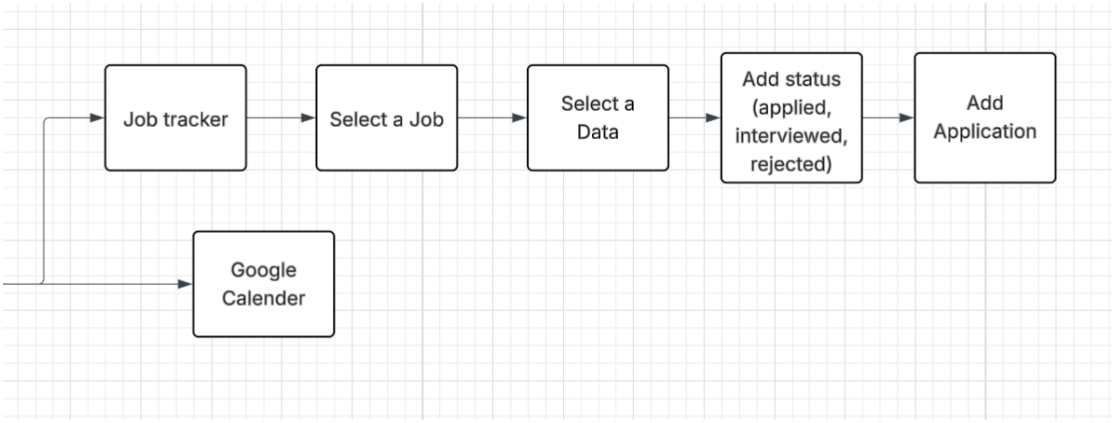

Figure (9), (Jobseeker Tracker overview)

The Job Tracker includes functionalities to add items into a chart to visualise how many applications you have sent out, rejected from and have interviewed for. This page also has google calendar integration which will allow you to view any interviews you have been scheduled for by a recruiter.

4.2.3 Jobseeker Job Search

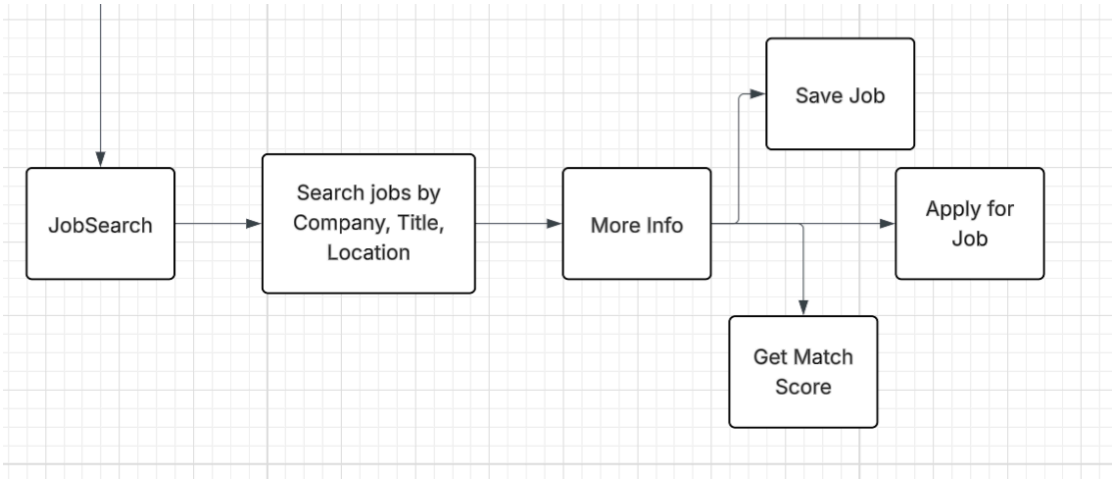

Figure (10), (Jobseeker Job Search overview)

The Job search page loads in a static database hosted on Firebase realtime database which the user can search through using Fuse.js the fuzzy search library which deals with matching jobs by company, location and title. Users can view the details of a job taking them to a page where they can save and apply for the job. They can also see how compatible their CV is with the job posting by using the Match score button.

4.3 Recruiter users

4.3.1 Recruiter Dashboard

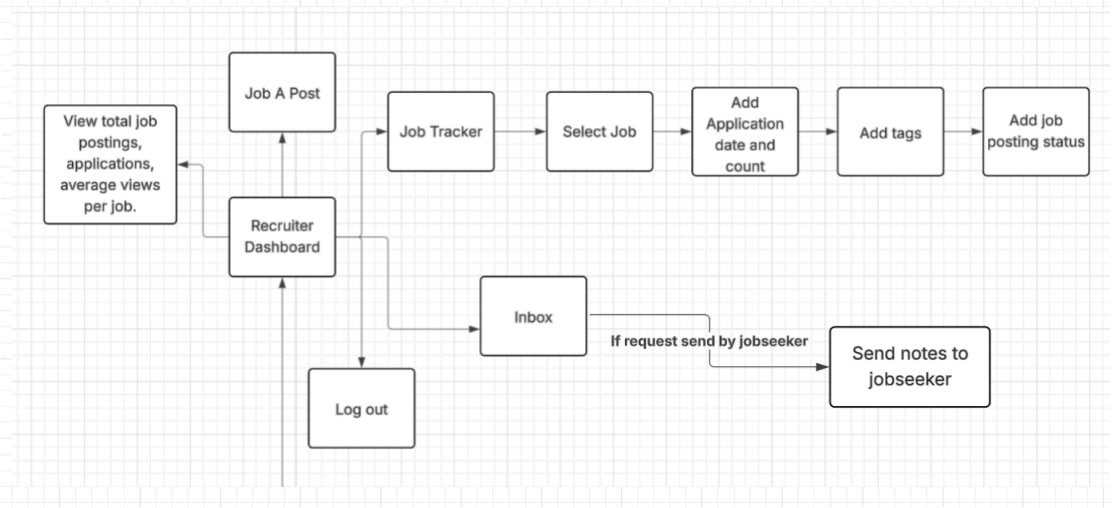

Figure (11), (Recruiter Dashboard overview)

The Recruiter dashboard includes:

- A page to create a new job posting and setting the match score parameters.
- Visuals on how many job postings each job has and how many applicants there are for each.
- The job tracker which allows recruiters to visualise the statistics of their job postings.
- Inbox which allows the recruiter to send messages to job seekers and also is used in the collab feature to connect with the job seeker and add suggestions to their cv.
- Logout.

4.3.2 View Job postings

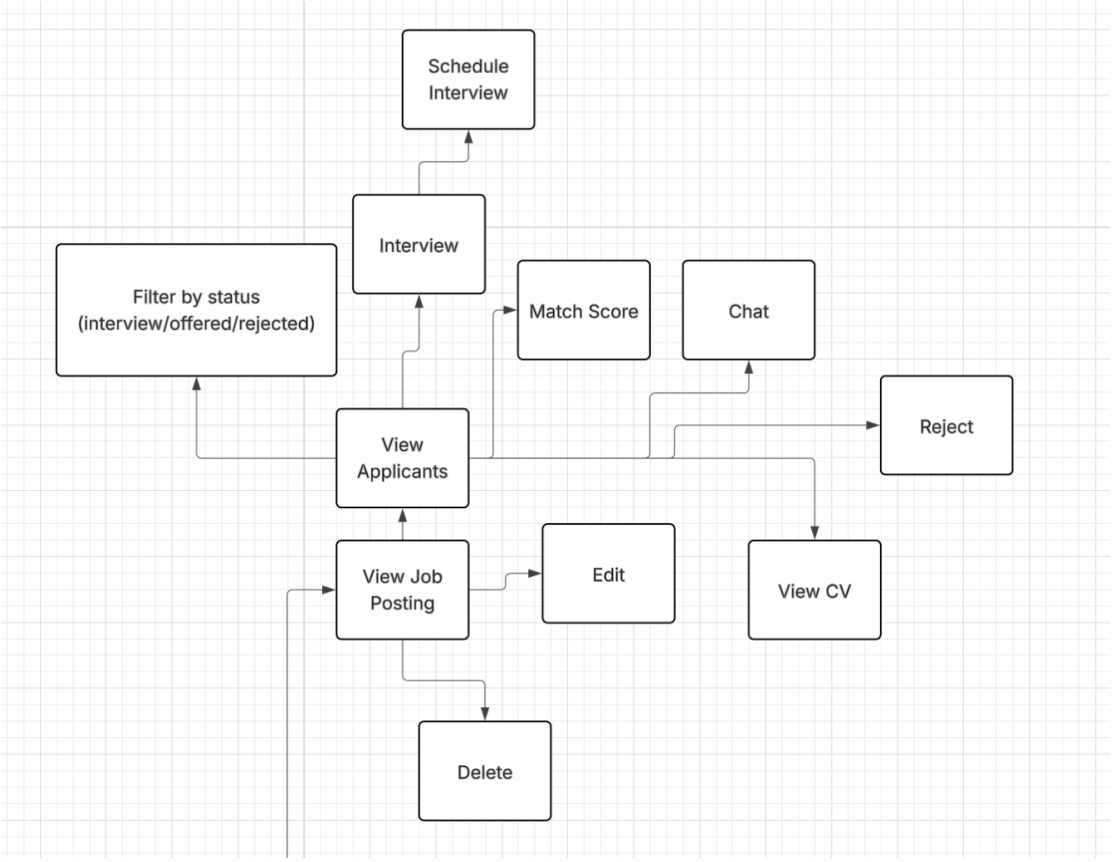

Figure (12), (Recruiter view job postings page)

The View job postings page allows the recruiter to see all their created job postings and allows them to:

- Edit the job posting and the match score parameters and save the new version.
- Delete the job posting.
- View all applicants for each job.

  For each applicant the recruiter can:

  - Schedule an interview with a particular applicant by creating a google calendar event and an email reminder.
  - Reject the applicant.
  - Offer the applicant the job which sends an automatic message to the applicant's inbox.
  - Chat button, which opens a new chat with the applicant.
  - Match Score button, which shows the recruiter the match score and also a breakdown of how the match score was generated (tfidf, sbert, matching and missing keywords).

4.3.3 Recruiter Search

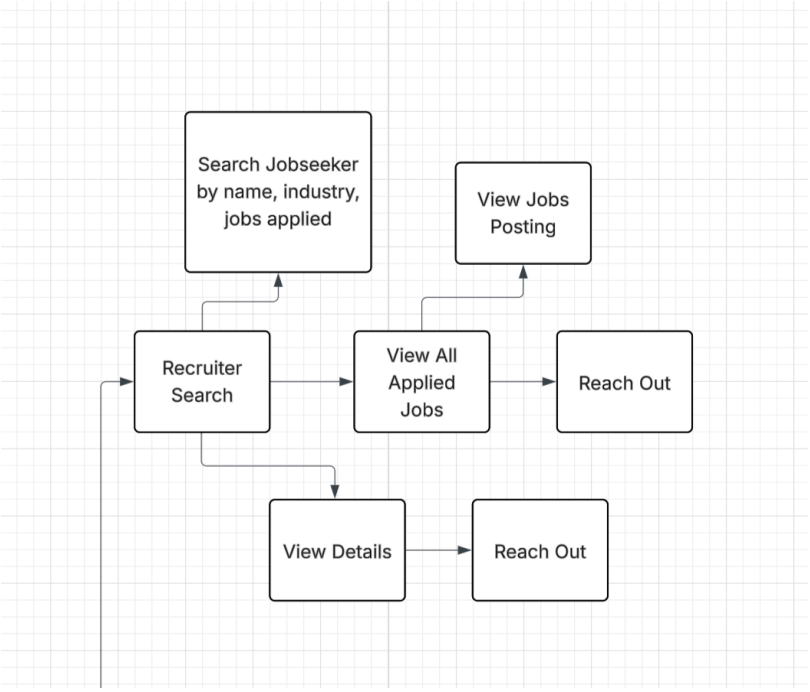

Figure (13), (Recruiter Search page)

The Recruiter Search allows the recruiter to:

- Search from a database of all jobseekers by name and industry.
- The recruiter can view all the jobs that a particular applicant has applied to within their company.
- The recruiter can ‘reach out’ which starts a new chat between the recruiter and the applicant.

4.4 General Implementation Information

4.4.1 File Structure

We organised the file system in a way that prioritised organisation and ease of use. Due to the size of the project, organisation was a key factor as we needed a lot of different components to interact with each other. Having two separate dashboards made the project quite large but it also allowed for an easy division in the frontend. All the frontend for the job seeker dashboard was kept separate from the recruiter and any frontend components that could be reused were created outside of those two folders with ‘chat’ and ‘auth’ folders that were reused in both dashboards as seen in Figure(14).

```
backend
  routes
  tests
components
pages
  >admin
  >auth
  >chat
  >jobseeker
  >recruiters
  >signup
```

Figure (14), (File system overview)

4.4.2 Backend use of Blueprints

We implemented the backend using Flask which is lightweight and straight forward, only needing to create API routes. We created blueprints for each key piece of functionality as seen in Code snippet(11).

```
create_job_bp = Blueprint('create_job', __name__)
```

Code snippet (11), (Create_job_bp blueprint)

This allowed us to easily import all the functions in App.py and to easily organise all similar functions into the same page which relied on each other or were used in the same location in the application. This made it easier to understand how code connected together and was much better than having many different individual functions to keep track of. For example all the functions for google calendar integration (storing tokens, creating an event, fetching events) in the backend are all under the @google_cal_bp.route blueprint.

4.4.3 Backend to Frontend integration

The React frontend interacts with the Flask backend primarily through HTTP requests (GET, POST, PUT, DELETE) using fetch(). Each function is only specified to perform certain methods.

```
@edit_job_bp.route('/fetch_job/<user_id>/<job_id>', methods=['GET', 'OPTIONS'])
@cross_origin()
def fetch_job(user_id, job_id):
```

Code snippet (12), (Edit_job_bp to show example of backend HTTP methods)

An example of how we have designed the routes in the backend, specifying the parameters the routes take (in this case <user_id> and <job_id>) and the methods that are allowed with this route shown in Code snippet(13).

```
try {
  const user = getAuth().currentUser;
  if (!user) throw new Error('User not authenticated');

  const idToken = await user.getIdToken();
  const user_id = user.uid;
  const response = await fetch(`http://localhost:5000/update_job/${user_id}/${id}`, {
    method: 'PUT',
    headers: {
      'Content-Type': 'application/json',
      'Authorization': idToken,
    },
    body: JSON.stringify(jobData),
  });
```

Code snippet (13), (Update_job to show an example of Frontend integration of backend)

This is an example of the integration of the backend in the frontend. The route is being called from the frontend using the same parameters and we use that JSON response to inflict change in the application (in this case editing a job posting).

Iterative API testing was implemented during the entire development of the backend mainly in the use of Postman which is a platform for API testing, Postman helped us understand the errors we were encountering better and allowed for us to create functions that handled errors well.We mainly used Postman to:

1. **Verify endpoint functionality** across all HTTP methods
2. **Validate parameter handling** for various input scenarios
3. **Stress-test error conditions** including:
    - Invalid IDs
    - Malformed JSON
    - Unauthorized access attempts
    - Missing required fields

4.4.4 Reusable Code

Certain pages like the jobseeker dashboard and the view all applicants pages had functions which did very similar things. On the view applicants page many different firebase collections need to be accessed to retrieve different types of applicants for a job posting (all applicants, rejected, offered, interviewed). Initially there were multiple functions which just retrieved from different collections which cluttered the page and increased the lines of code in a way that did not add value.

With refactoring we were able to cut down significantly on unnecessary components and create reusable code which allowed for more general use across different features. Code snippet (14) is a display of the example I described, as you can see we are using different parameters that can be modified on the frontend to allow for more flexibility.

```
@viewapplicants_bp.route('/fetch-collection-applicants/<recruiter_id>/<jobposting_id>/<collection>', methods=['GET'])
def fetch_collection_applicants(recruiter_id, jobposting_id, collection):
```

Code snippet (14), (Fetch-collection-applicants backend function to show use of reusable code)

Code snippet (15) is a display of fetching lists of jobs for a specific job seeker to display on the dashboard which also allows for accessing different kinds of lists when needed.

```
@seeker_dashboard_bp.route('/fetch-jobseeker-jobs/<job_list>', methods=['GET'])
@cross_origin()
```

Code snippet (15), (Fetch-jobseeker-jobs backend to show use of reusable code)

5. High-Level Design

5.2 Data Flow Diagram

5.2.1 Jobseeker

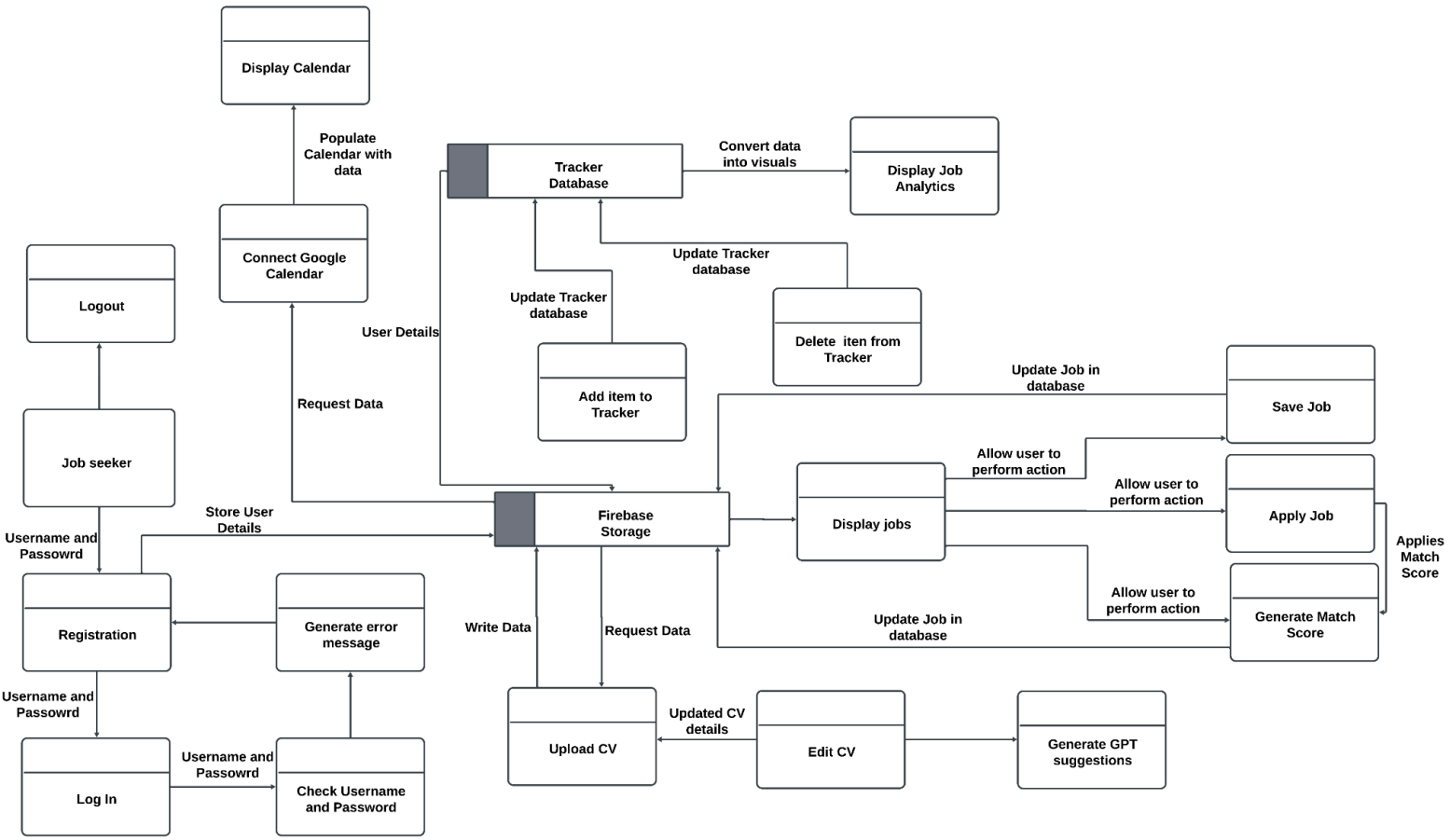

Figure (15), (Jobseeker Data Flow Diagram)

- Login/Registration: Job seekers can register or log in using a username and password. Credentials are validated through a check, and if correct, access is granted; otherwise, an error message is generated.

- Logout: Allows the job seeker to exit the system.

Core Functionalities and Data Flows

1. CV Management
   - Upload CV: Allows users to upload their resume to Firebase Storage.
   - Edit CV: Enables users to make updates to their CVs, with changes written back to Firebase.

- Generate GPT Suggestions: Offers AI-generated suggestions to enhance the content and quality of the CV.

2. Job Interaction
   - Display Jobs: Loads available job listings from Firebase for user viewing.

   - Apply Job: Allows job seekers to submit applications; this action also triggers the Generate Match Score function.
   - Save Job: Enables users to bookmark or store jobs for later action.
   - Generate Match Score: Compares job details with CV information and returns a compatibility score.
   - Update Job in Database: Saves user actions (like saved or applied jobs) back to Firebase.

3. Job Tracking
   - Add Item to Tracker: Adds a job application or saved job to the tracker's database for monitoring progress.
   - Delete Item from Tracker: Removes a tracked job from the user's list.
   - Display Job Analytics: Converts tracker data into visual insights to help users understand their job search trends and activity.

Integration & Calendar Management

1. Google Calendar

   - Connect Google Calendar: Requests access to the user's calendar.
   - Populate and Display Calendar: Adds scheduled job-related events (e.g., interviews) and displays them for the user.

Data Hub: Firebase Storage

Acts as the central repository for:

- User credentials and account details
- CV uploads and updates
- Job listings and user interactions (applied/saved jobs/offered jobs/ interviewed jobs/ withdrawn jobs)
- Calendar events
- Tracker updates

Firebase handles both data storage and retrieval for every core feature in the system.

5.2.1 Recruiter

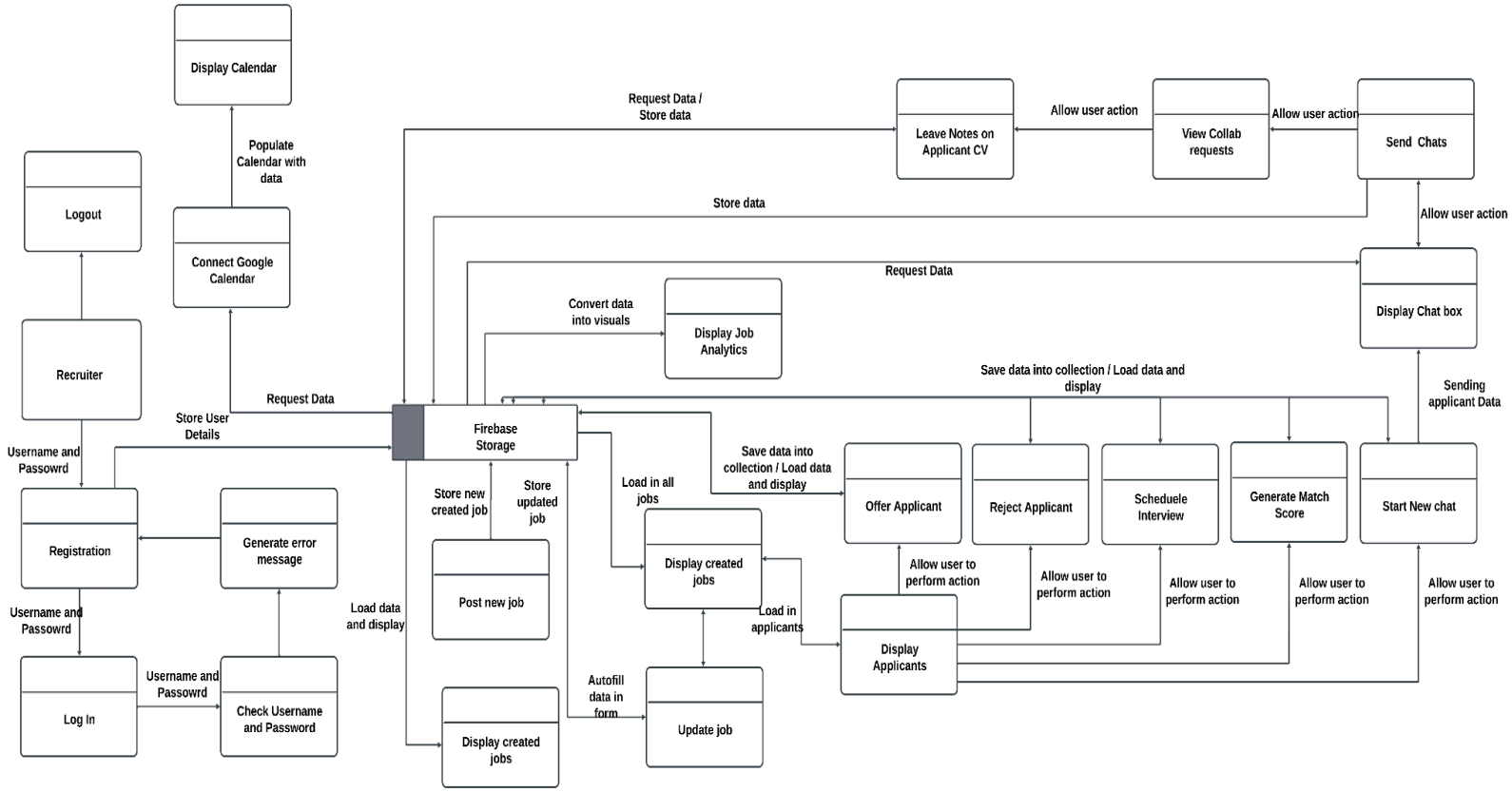

Figure (16), (Recruiter Data Flow Diagram)

- Login/Registration: Recruiters can register or log in using a username and password. Credentials are validated through a check, and if correct, access is granted; otherwise, an error message is generated.
- Logout: Allows the recruiter to exit the system.

Core Functionalities and Data Flows

1. Job Management

- Post New Job: Allows recruiters to create and store new job listings in Firebase.
- Update Job: Enables editing of existing job postings, which are then updated in Firebase.
- Display Created Jobs: Retrieves all jobs from Firebase for viewing and editing.

2. Applicant Management

- Display Applicants: Loads applicants related to a job from Firebase.
- Recruiters can:
    - Offer Applicant
    - Reject Applicant
    - Schedule Interview
    - Generate Match Score

Each of these actions saves data into Firebase and updates the display.

3. Chat and Collaboration

- Start New Chat: Initiates a chat session with an applicant.
- Display Chat Box and Send Chats: Allow conversation handling.
- View Collab Requests and Leave Notes on Applicant CV: Allows recruiter to suggest changes on an applicants CV

Integration & Analytics

1. Google Calendar

- Connect Google Calendar: Requests user data.
- Populate and Display Calendar: Imports interview or task schedules and visualizes them.

2. Analytics

- Display Job Analytics: Visual representations (charts/graphs) of recruitment data, powered by Firebase.

Data Hub: Firebase Storage

Acts as the central repository for:

- User credentials and details
- Job postings
- Applicant data
- Chat logs
- Calendar events
- Analytics-related data

Firebase handles both data storage and retrieval for every key action in the system.

5.3 Sequence Diagram

Jobseeker Data Flow Diagram

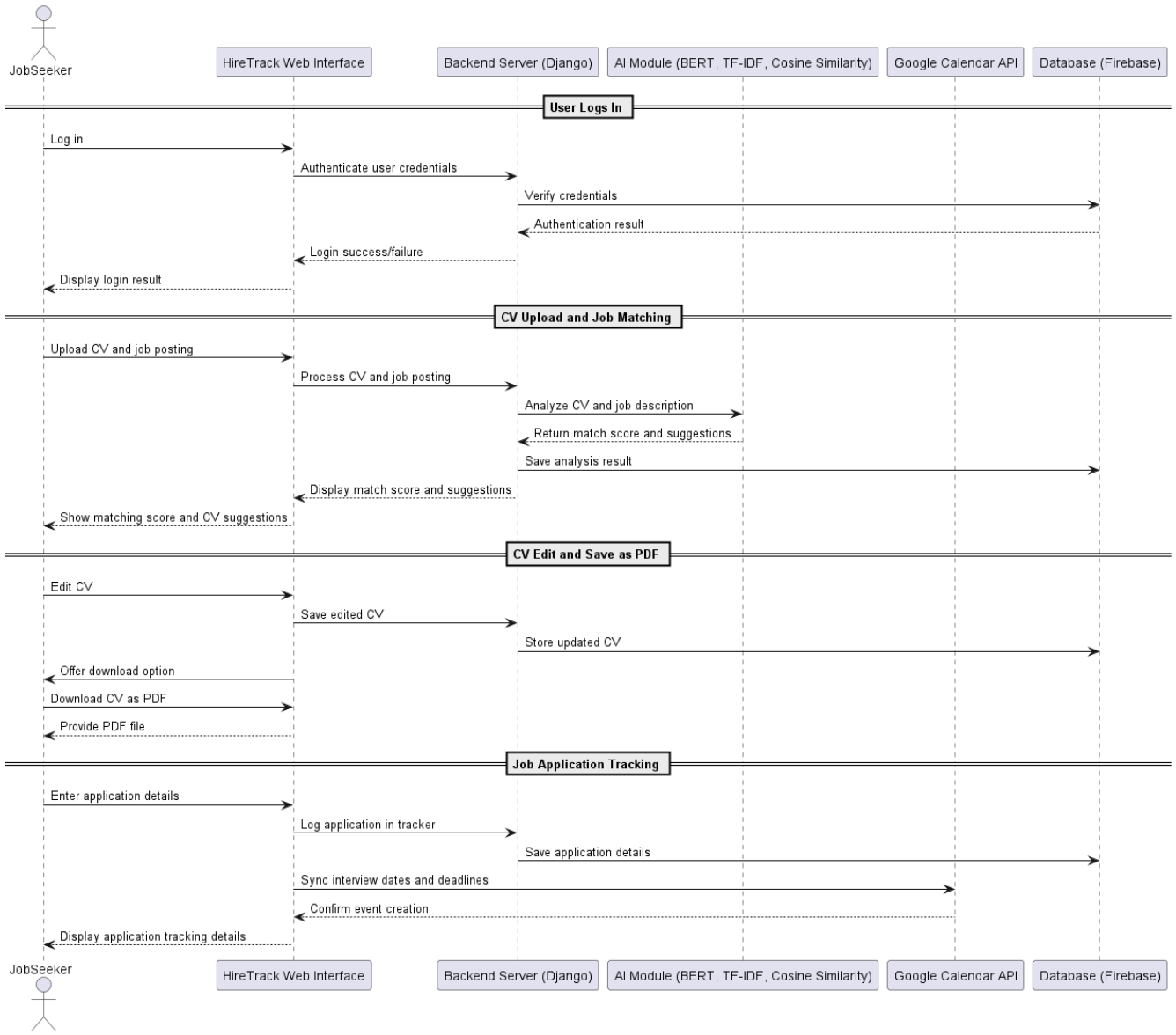

Figure (16), (Jobseeker Sequence diagram for Dashboard)

1. **User Logs In**: The job seeker logs in, and the system verifies their credentials with Firebase.
2. **CV Upload and Job Matching**: The job seeker uploads their CV and a job posting, which is analysed by the Match Score (using BERT, TF-IDF, etc.). The match score and improvement suggestions are saved and displayed to the user.
3. **CV Edit and Save as PDF**: The user can edit the CV, save the changes, and download it as a PDF.
4. **Job Application Tracking**: The job seeker logs an application, which includes syncing dates with Google Calendar for reminders. This data is stored and displayed in their application tracker.

7. Problems Faced & Solutions

7.1 O-Auth integration with Google Calendar

Initially, my Google Calendar integration had several issues:

1. **Token Expiry Issues**: Users kept getting logged out after 1 hour when access tokens expired and there was no way for them to connect back into google calendar.
2. **Overly Complex UI**: I wrapped everything in an OAuth provider component, making the UI confusing and the implementation much more complicated than it needed to be
3. **Silent Failures**: When tokens expired, the app would fail silently without clear recovery options.

I faced a lot of issues when trying to integrate Google Calendar into the application, mostly because the API was new to me. When testing out the applications API routes, in this case '/store-google-token' and '/get-calendar-events' they would fail on postman because I hadn't implemented refresh token which allow for the user to access google calendar functionalities for an extended period of time without the need to reconnect to their google account. The errors I was receiving on postman were vague and pointed to an issue in the scope rather than the lack of refresh tokens/ incorrectly implemented refresh tokens.

I spent a lot of time trying to wrap the entire app in an O-auth wrapper to try and mitigate this issue but it seemed like I needed to implement a lot of other functions to get it properly working and it was eating away at the time needed for other parts of the project.

While researching a solution I came across this article (see appendices [5]) which explained the need for refresh tokens and the changing of the scope to make it work. I also used better error handling to see where the feature was failing to get a better view of what issues I was encountering.

```
  },
  scope: 'https://www.googleapis.com/auth/calendar.readonly',
  flow: 'auth-code',
  prompt: 'consent',
  access_type: 'offline',
});
```

Code snippet (16), (Shows change of scope to enable refresh token in the backend)

The access type was changed from ‘online’ to ‘offline’ because with an online access type you won’t be able to access refresh tokens.

```
user_ref.update({
    'googleTokens': {
        'accessToken': access_token,
        'expiryDate': time.time() + 10600,  # 1 hr expiration
        'refreshToken': refresh_token,
        'clientId': GOOGLE_CLIENT_ID,
        'clientSecret': GOOGLE_CLIENT_SECRET,
        'tokenUri': 'https://oauth2.googleapis.com/token',
        'scopes': list(granted_scopes),
        'lastRefresh': time.time()
    }
})
```

Code snippet (17), (Section which updates Firebase)

The backend was changed to store refresh tokens in Firebase and fetch them to refresh the connection with Google Calendar.

```
# Refresh token if expired
if creds.expired:
    creds.refresh(Request())
    user_ref.update({
        'googleTokens.accessToken': creds.token,
        'googleTokens.expiryDate': time.time() + 3600
    })
```

Code snippet (18), ()

If the access token did expire I have a function to update them so the Calendar feature doesn’t fail silently.

7.2 Resume Parser & GPT integration

Enhancing the CV editing experience presented multiple challenges. Initially, the resume parser relied solely on regex patterns, which failed to generalize across different CV formats. As a result, field extraction (such as name, experience, and education) was inconsistent. Another issue involved the integration of GPT-based CV suggestions: the raw suggestion text returned by the model was often too long, cluttered the UI, and lacked context. Finally, matching CV improvements to specific job applications was unclear, making the feature less effective for users seeking tailored improvements.

A series of improvements were implemented to overcome these challenges:

- To make the parser more robust, I trained a custom spaCy NLP model using auto-generated .txt and .json datasets derived from existing .docx resumes. This enhanced the parser's ability to generalize across different layouts and improved accuracy in field extraction.
- For the UI display of GPT-powered CV suggestions, I redesigned the interface to use a horizontal scrollable card layout. Each suggestion was displayed on its own card, improving readability and user interaction.
- To ensure suggestions were job-specific, I implemented a dropdown in the CV editor that allowed users to select one of their applied jobs. This context was then used to generate tailored CV improvements relevant to the selected role.

These enhancements made the resume editing and optimization feature significantly more user-friendly, responsive, and intelligent.

7.3 CI/CD

When setting up Continuous Integration and Continuous Deployment (CI/CD) for the project using GitLab pipelines, a number of issues arose. Cypress tests running perfectly in local environments continued failing in the CI pipeline for GitLab. The pipeline was unable to detect the Cypress configuration file (cypress.config.js), and the Flask backend was failing to launch properly during the run of tests. These issues rendered debugging difficult and halted progress on automated test validation.

To address these issues, I used a combination of configuration and debugging solutions:

- I understood that Cypress could not detect the configuration file due to incorrect relative paths and misconfigured working directory. This was resolved by modifying the path to cypress.config.js and modifying the.gitlab-ci.yml file to point to the correct root directory.
- To simulate and test pipeline execution locally, I wrote a custom script called dev-ci.sh that simulated the GitLab CI process, such as launching the backend, frontend, and running Cypress tests.
- In addition, the CI pipeline did not execute the Flask backend initially due to the missing dependencies. To resolve this, I manually placed the pip install flask in the pipeline setup and kept the backend path aligned with the real project folder structure.
- This setup made the pipeline stable and ensured that all end-to-end tests ran smoothly in local and CI environments.

8. Future Work

Customizable Matching

We want to give recruiters more control over how candidates are matched to jobs. Right now, the system automatically calculates match scores, but we know different recruiters might care about different things.

Key Features

Must-Have Keywords

- Recruiters can mark certain skills as required.
- The system will highlight candidates with these key skills.

Flexible Requirements

- Options to be strict or lenient about years of experience.
- Ways to handle equivalent skills (like counting "Python" and "PyTorch" similarly for ML roles).

Simple Controls

- Clean dashboard to adjust settings.
- Clear explanations of what each setting does.

Our current matching system works well, but every hiring manager has their own way of evaluating candidates. These changes will make the tool more flexible by letting people set it up the way it suits them, while still keeping the process fair and consistent.

Notifications

To improve the recruiter and candidate engagement, HireTrack will implement a real-time notification system that keeps users informed about key hiring process updates such as when they have been offered/rejected for a job, scheduled for an interview and when a new job from a company they have previously applied to gets published.

Key Features

Jobseekers:

- Application status updates: offered/rejected/interview.
- Application deadline reminders for saved jobs.
- Interview scheduled for later that day.
- New job recommendations based on previous applications.

Recruiter:

- New high match applicants.

- Jobseeker withdrawing from a job.
- Interview confirmations/rescheduling.

Implementation Approach

These notifications will be implemented and will include notification in the browser but also sent by email with preferences being modified by the user. Firebase Cloud Messaging will be utilised to store notifications in Firestore with metadata on the content, timestamp, isRead and category. These will be triggered based on actions made by the recruiter or applicant (e.g offering a job). The frontend will include a notification dropdown with mark-as-read functionality with visual indicators like badges and toast messages.

Improved User Profiles

Currently job seekers are only able to upload a cv and short descriptions of their work experience and their qualifications. Future developments could include job seekers being able to upload more information to give the recruiter a better view of their capabilities.

Key Features:

- Uploads for references.
- Links to social media like linkedin/portfolio website/code repositories.
- Work samples upload: PDF (research papers, case studies).

Technical Considerations:

Firebase Storage Optimization:

- File size limits (e.g., 10MB for PDFs, 25MB for videos).
- Virus scanning for uploads (via Cloud Functions + Google Cloud Security Scanner).

Performance Impact:

- Lazy-loading for media-heavy profiles.
- Caching frequently accessed data (e.g., LinkedIn previews).

Collab Feature

We aim to enhance the collaboration functionality between recruiters and job seekers, making feedback more organized and actionable while improving communication workflows.

Key Features

- **Rich Feedback Tools**
  - Rich-text formatting for notes (bold, bullets, headings).

    - Visual cues to categorize feedback (e.g., skills vs. experience).
- **Real-Time Notifications**
    - Alerts for new collaboration requests or messages.
    - Firebase Cloud Messaging (FCM) or browser notifications.
- **Extended Communication**
    - Message chains for ongoing conversations (beyond one-time notes).
- **Recruiter-Driven Outreach**
    - Recruiters can proactively contact candidates based on profiles.
- **Dashboard Filtering**
    - Filter collaboration requests by job title, status, or date.

Implementation Approach

- Integrate real-time notifications via FCM with metadata (timestamp, read status).
- Use Firestore to store threaded messages and collaboration.

9. Testing

A separate document can be found titled System Tests which details all of the testing conducted throughout the duration of the project.

10. Appendices

[1] - https://arxiv.org/abs/1810.04805
[2] - https://arxiv.org/abs/2308.04037
[3] - https://www.datastax.com/guides/what-is-cosine-similarity
[4] - https://www.fepbl.com/index.php/csitrj/article/view/859
[5]- https://stackoverflow.com/questions/8942340/get-refresh-token-google-api?rq=3