UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

Big Data Management

Implementation of a Big Data Management Backbone - P2

Hadiqa Alamdar Bukhari

Marwah Sulaiman

Sara Saad

June 2025

Contents

1 Project Idea and P1 recap

ScholAmigo: a Scholarship Finder that helps students discover scholarships in the EU tailored to their profiles and aspirations, and it guides them through the application process with valuable support tools.

The platform offers the following key features:

- **Advanced Scholarship Search:** Students can filter opportunities based on various criteria such as country, funding amount, nationality, field of study, application deadline, and more.
- **Comprehensive Scholarship Information:** Each listing includes clear, structured, and up-to-date details to help users fully understand the opportunity.
- **Community Connections:** Users can connect with current applicants and alumni of the target programs to gain insights and mentorship.
- **Resume Evaluation and Matching:** The platform analyzes users' resumes, matches them with relevant scholarships, and suggests actionable improvements.
- **Alumni Career Insights:** By leveraging alumni data, ScholAmigo highlights trends, potential career paths, and job titles commonly pursued after graduation.

In the scope of Big Data Management, our project involves the collection and integration of data from various sources, broadly categorized into four areas:

- **Scholarship data**
- **The data of current students and alumni of the programs**
- **Resumes**
- **User profiles** (to simulate registered users on the platform)

The following sections provide detailed information on the refined data pipeline, and the Trusted and Exploitation Zones built on top of our landing zone.

2 P2 Architecture overview

The ScholAmigo platform is built on a modular, multi-zone architecture that processes heterogeneous educational data into clean, structured, and intelligent outputs. The system follows a progressive data refinement approach, transitioning raw data through well-defined zones—Landing, Trusted, and Exploitation—before exposing it to user-facing analytics and applications. The BDM pipeline architecture follows the principles outlined in the Data Management Backbone model, which ensures scalability, modularity, and semantic consistency across the data lifecycle. The pipeline is orchestrated using Apache Airflow, and data is stored in Amazon S3.

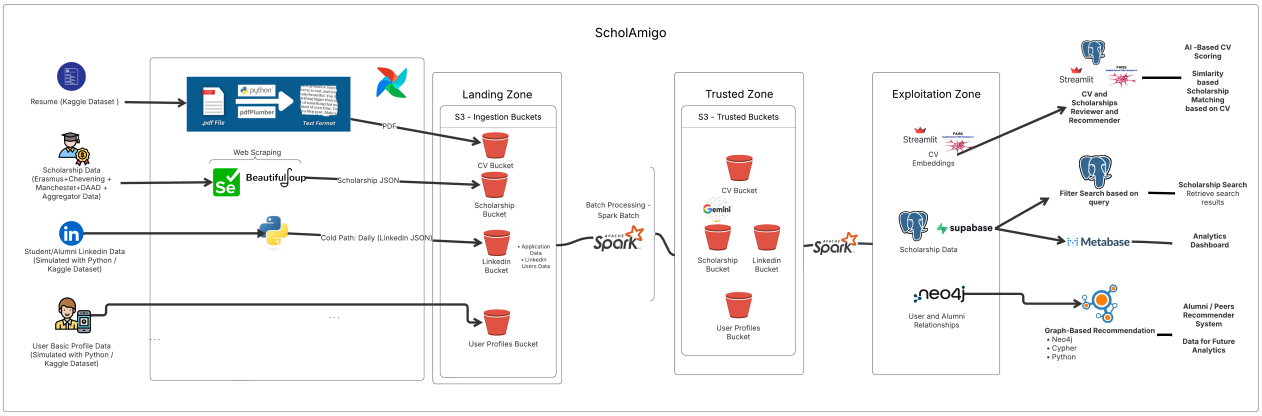

Figure 2.1: Architecture Diagram

2.1 Data Sources and formats

- **Scholarship Data**: Scraped from Erasmus Mundus catalog, Chevening, University of Manchester, and internationalscholarships.com → JSON.
- **Student and Alumni Profiles**: Scraped from LinkedIn using a custom Python crawler → JSON.
- **Resumes**: Collected from a Kaggle dataset → PDF.

- **User Profiles**: Simulated profiles created for testing the platform downloaded from Kaggle → JSON.

2.2 Landing Zone

Each extraction script is scheduled and run using Airflow DAGs, which ensures regular and fault-tolerant data updates. All raw data collected from external sources is first ingested into the Landing Zone in the form of JSON and PDF files are stored with metadata such as source name and file type etc for traceability and version control. Each extraction script is scheduled and run using Airflow DAGs, which ensures regular and fault-tolerant data updates.

2.3 Trusted Zone

The Trusted Zone is the system's data refinement core, responsible for cleaning, standardizing, and structuring raw inputs from the Landing Zone. Spark-based pipelines are employed for scalable processing, including schema validation, entity normalization, and transformation into consistent formats.

- **Scholarships** are cleaned and standardized using PySpark, LLM-assisted extraction (Gemini), and custom classification logic. The data is output in Parquet files and stored in an S3 bucket.
- **User profiles and Linkedin Profiles** are normalized and validated for downstream compatibility. The output files are in the JSON/JSONL format to support high-throughput access and long-term durability in Amazon S3 buckets.
- **Resume** pdf files are processed and the text extracted is normalised and stored in the JSON format in an S3 bucket.

2.4 Exploitation Zone

The Exploitation Zone hosts specialized databases that enable advanced analytics, intelligent recommendations, and rich data interactions. Data from the Trusted Zone is transformed into interconnected entities and relationships using Apache Spark and Python threading for scalable graph construction. This zone provides the intelligence backbone of the platform. It includes:

- PostgreSQL relational database for structured scholarship data
- FAISS-powered semantic vector database for resume to scholarship similarity
- Neo4j graph database for user-alumni networking.

2.5 Consumption Tasks

The final layer of the system delivers refined data to end-user applications, visualizations, and analytics tools. It supports dynamic querying of both relational and graph databases to uncover career trajectories, scholarship opportunities, and meaningful peer or alumni connections. Advanced statistical and relational analyses enable the exploration of geographic and demographic trends, as well as mapping in-demand skills across regions and populations. Additionally, the platform allows users to upload their own CVs to receive personalized scholarship recommendations and targeted feedback on how to improve their profiles for better eligibility. This layer is powered by semantically rich, rigorously validated datasets, ensuring accurate insights and a seamless, data-driven user experience.

3 Trusted Zone Data Processing

3.1 Scholarship Aggregator Data

Transformation Pipeline

The pipeline consists of four main phases:

1. Data Ingestion:

Spark is used to read NDJSON files. Logged metadata includes source origin and processing timestamps. Schema compliance is checked to filter out corrupted records.

2. Standardization:

DD/MM/YYYY is the common format for all date formats. When feasible, currency amounts are standardized to EUR while remaining in their original form. ISO 3166 is used to standardize country names. A common taxonomy is used to map educational levels: bachelor's, master's, and doctoral. Fields of study are standardized to consistent categories.

3. Field Mapping: The ScholAmigo schema is used to align fields from raw sources. Based on scholarship titles, duplicate records are eliminated, types (string, list, and struct) are validated, and defaults are applied for missing values.

4. Data Cleaning: Special characters are eliminated, whitespace is normalized, and entries that are duplicated or corrupted are eliminated. The final product complies with semantic clarity and structural requirements.

Unified Scholarship Schema

Each scholarship is standardized into the following structure by the final schema:

Listing 3.1: Standardized Scholarship JSON Schema

```
{
  "name": "Scholarship Title",
  "description": "Detailed scholarship description",
  "level": "Bachelor | Master | PhD",
  "fields_of_study": ["Engineering", "Data Science"],
  "countries": ["Germany", "Spain"],
  "requirements": "Eligibility conditions",
  "required_documents": "Application requirements",
  "deadline": "15/10/2025",
  "universities": ["TU Delft", "KU Leuven"],
  "funding": {
    "monthly_allowance": "1000 EUR",
    "tuition_coverage": true,
    "installation_costs": "1000 EUR",
    "details": "Fully funded"
  },
  "url": "https://example.org",
  "status": "active"
}
```

Error Handling

Errors in transformation, invalid date formats, and corrupted JSON entries are handled gracefully. Unresolved issues are noted and removed from the final output.

Output Processing

The cleaned data is saved as line-delimited JSON files. Metadata such as source and processing time is appended. Final files are uploaded to Amazon S3 for durability.

3.2 Erasmus and UK Scholarship Data

This part developed a comprehensive data processing pipeline for Erasmus Mundus scholarship data and UK scholarships data since they had similar landing zone format. The technology stack of AWS S3, PySpark, and Google's Gemini AI model were used for information extraction, transformation and storage.

1. ***Data Ingestion:*** The pipeline retrieves raw JSON scholarship data from S3 landing zone bucket and loads data into PySpark DataFrame with optimized memory configurations to handle large datasets

efficiently.

2. ***HTML Content Processing:*** The system extracts and structures HTML content using BeautifulSoup, removes non-content elements (nav, footer, scripts), and organizes content into logical sections with headings to prepare it for LLM processing.
3. ***AI-Powered Information Extraction:*** The pipeline uses Google's Gemini-1.5-flash model to process each scholarship through detailed prompts, extracting 30+ fields including scholarship details, financial information, requirements, dates and deadlines, and participating institutions.
4. ***Data cleaning and Standardization using PySpark:***
    - **Normalizes text fields and dates**: Converts scholarship names to title case while removing irregular spacing and non-standard characters. Standardizes all dates to DD/MM/YYYY format from various input formats (YYYY-MM-DD, DD-MM-YYYY, MM/DD/YYYY), maintaining associated descriptions and handling parsing errors gracefully.
    - **Standardizes country names and durations**: Uses country_converter library to map various country name formats to consistent short names, with special handling for edge cases. Converts program durations to standardized years with half-year precision, recognizing units like semesters, months, weeks, and days.
    - **Classifies admission seasons**: Maps various expressions for admission periods to standardized "fall" and "spring" categories, recognizing terms like "autumn" while preserving non-standard periods in original form.
    - **Computes application status**: Implements intelligent status classification by analyzing deadline-related keywords in date descriptions, comparing against current date to automatically classify scholarships as "open", "closed", or "unknown".
5. ***Financial Data Processing:***
    - **Special handling of Erasmus+ scholarship amounts**: Processes nested financial structures with inconsistent field naming, converting text-based availability (for current intake) to boolean values while handling edge cases and formatting inconsistencies.
    - **Distinguishes program/partner country benefits**: Separates financial benefits for programme countries vs partner countries, each with different benefit structures including tuition coverage, monthly allowances, travel allowances, and installation costs.
    - **Standardizes monetary values**: Extracts numeric values from text containing currency symbols and formatting, handles conditional travel allowances based on distance thresholds, and distinguishes between full/partial/no tuition coverage scenarios while preserving detailed coverage information.

Output

The pipeline saves processed data to S3 in Parquet format (`trusted_zone_data/`), preserving schema structure, all needed data, and generates clean datasets ready for organization in the next zone. Parquet was chosen due to its efficient columnar storage, which aligns with our need for columnar access during the later stage. Additionally, our tests showed that Parquet files were processed faster than JSON.

3.3 CV Data Processing

CVs are processed using a multi-step transformation pipeline that includes cleaning, section segmentation, normalization, and PDF text extraction.

PDF Extraction. Text is extracted using the `pdfplumber` library:

Listing 3.2: Extracting Text from PDF

```
def extract_text(pdf_path):
    with pdfplumber.open(pdf_path) as pdf:
        return "\n".join(page.extract_text() or "" for page in pdf.pages)
```

Section Identification. Typical CV sections include summary, education, experience, skills, projects, certifications, languages, and publications.

Text Normalization. Text is cleaned through lowercasing and whitespace compression:

Listing 3.3: Text Normalization Function

```
def normalize_text(text):
    text = text.lower()
    text = re.sub(r"\s+", "␣", text)
    return text.strip()
```

Final CV Format. Transformed CVs are stored using the following format:

Listing 3.4: Cleaned Resume Format

```
{
  "file_name": "resume_sara.pdf",
  "sections": {
    "education": "Extracted education text",
    "experience": "Extracted experience text",
```

```
    "skills": "Extracted skills",
    "other": "Miscellaneous sections"
  }
}
```

Cleaning Routine. Each record is verified and normalized. Empty or malformed sections are discarded.

Listing 3.5: Cleaning Resume Sections

```
def clean_resume_sections(data):
    if not isinstance(data, dict) or "file_name" not in data:
        return None

    cleaned = {
        "file_name": data["file_name"],
        "sections": {
            k: normalize_text(v)
            for k, v in data["sections"].items() if isinstance(v, str)
        }
    }
    return cleaned
```

Storage and Output. Each cleaned CV is saved in JSON format to `trusted_zone/data/resume_data/cleaned`. The filename corresponds to the original PDF with a `.json` extension.

3.4 LinkedIn Alumni and User Profiles Data

The system architecture follows a logical data flow that begins in the landing zone, where raw JSON and JSONL files are stored. These files are processed and cleaned before being moved to the trusted zone, where they exist in structured, standardized formats.

3.4.1 Data Ingestion

The pipeline processes two main sources of data.

1. The student profile dataset stored in JSONL format in an S3 bucket:
   `s3://scholamigo/landing_zone_data/users_data/`
2. The Erasmus alumni LinkedIn dataset, stored in JSON format at:

`s3://scholamigo/landing_zone_data/erasmus_linkedin_profiles/`

3.4.2 Data Processing Methodology

Data Quality Framework

To ensure high data quality, several preprocessing steps are implemented. Text fields are normalized by removing excessive whitespace, eliminating non-standard characters, and converting all text to lowercase to maintain consistency. Platform-specific artifacts such as truncated descriptions are removed. Numeric fields, such as age and GPA, are validated to ensure they contain only non-negative values. Furthermore, country names are standardized using a predefined mapping dictionary, and all fields are normalized to maintain uniformity across datasets.

Deduplication Strategy

Student profile deduplication is handled using name-based detection, where duplicate entries are identified and appended with unique suffixes to ensure record uniqueness without data loss. Alumni profiles are processed with a more complex deduplication strategy, detecting duplicates across multiple institutions and resolving them through systematic identifier assignment. This strategy preserves the hierarchical structure of organizational data while ensuring the integrity of individual profiles.

Schema Transformation

For student profiles, raw data fields are transformed into a cleaned schema with normalized field names and validated content. Fields such as name, age, sex, major, GPA, and country are standardized for downstream compatibility. Similarly, the alumni profiles undergo structured transformation, preserving nested arrays of profiles within each institution while cleaning individual fields like name, headline, and about sections. Additional metadata associated with institutions is retained to maintain contextual integrity.

Data Validation Measures

Data validation ensures records meet schema and quality standards through null handling, type checks, range validation, and formatting enforcement before entering the trusted zone.

Original Field	Cleaned Field Description
Name	`name` (lowercase, cleaned)
Age	`age` (non-negative float)
Sex	`sex` (lowercase, standardized)
Major	`major` (lowercase, cleaned)
Year	`year` (lowercase, cleaned)
GPA	`gpa` (non-negative float)
Hobbies	`hobbies` (array of cleaned strings)
Country	`country` (standardized names)
State/Province	`state_province` (lowercase, cleaned)
Unique Quality	`unique_quality` (lowercase, cleaned)
Story	`story` (cleaned text)

Table 3.1: Mapping of Original Fields to Cleaned Student Schema

Error Handling

The system handles record-level failures gracefully, logs errors for transparency, and includes tailored exception handling for Spark, AWS, and JSON, with retry support for transient issues.

Monitoring and Reporting

The pipeline logs record counts, generates sample outputs for verification, and produces status reports on performance and data quality to support monitoring and transparency.

3.4.3 Implementation Details

Apache Spark uses adaptive query execution and automatic partition coalescing for runtime optimization. S3 integration via the S3A filesystem ensures seamless access, with authentication handled through Spark's credential provider chain.

3.4.4 Data Output Specifications

The cleaned student profiles are stored in JSONL format along with alumni profiles in a structured JSON in an s3 bucket in the trusted zone.

4 Exploitation Zone

4.1 Scholarships Relational Database

PostgreSQL was chosen as the main storage system for the scholarships data, which includes Erasmus, UK-based, and aggregator sources. The data shows a clear and consistent structure that can be effectively represented using a relational schema. This made the relational model a suitable choice for our use case. The expected usage of the database involves filtering, searching, and performing analytical queries, tasks for which PostgreSQL is well suited due to its mature query engine and strong indexing capabilities. Additionally, since most queries involve retrieving complete rows rather than individual columns, row-based storage like PostgreSQL is more appropriate than column-oriented systems such as DuckDB.

This is the relational schema we designed and populated using our data 4.1:

Figure 4.1: Scholarships Relational Schema

We coordinated with the other team to define a set of common tables shared across both solutions (highlighted in rose). These tables, which represent the data most frequently used by end-users on the platform, were agreed upon as a foundation for integration.

4.1.1 Schema Overview

- **Core Entity Design** The schema is centered on the table 'scholarships_common', which contains fundamental scholarship attributes shared by all types. It includes standardized fields for funding type, academic level, prerequisite education, intake season, and application status. Each scholarship has a unique sequential key serving as the primary identifier.
- **Specialized Scholarship Tables** Two tables extend the core entity: 'scholarships_erasmus' handles Erasmus Mundus scholarships with detailed financial and program information, while 'scholarships_other' covers other scholarships with a simplified financial structure but consistent core fields.
- **Reference and Lookup Tables** Reference tables ensure consistency: 'countries' uses standardized names; 'universities' links institutions to countries; 'fields_of_study' stores the standard ISCED disciplines.
- **Relationship Management** Junction tables manage many-to-many links. 'scholarships_fields' links scholarships to multiple disciplines. 'origin_countries_sch' and 'program_countries_sch' manage nationality eligibility and study destinations.
- **Temporal Data Structure** Dates are stored in 'important_dates' and 'deadlines' tables with identical structures. Automated classification assigns dates based on keywords, supporting deadline tracking while preserving context via description and comments.
- **Performance Optimization** Indexes on key columns improve query performance. Auto-increment keys support efficient inserts. Foreign key constraints enforce referential integrity maintaining data consistency.

4.1.2 Transformations

In order to populate the discussed schema, We retrieved the data from the trusted zone stored in AWS S3, performed the necessary transformations using PySpark, and populated the PostgreSQL database accordingly.

1. ***Data Extraction and Validation:*** The system retrieves processed files from S3 trusted zone for all scholarship types, validates scholarship availability for current intake using boolean flags, and filters out invalid records with insufficient scholarship info or unavailable status.
2. ***Scholarship Data Normalization:***

- **Core scholarship standardization**: Maps funding types to standardized categories (Full/Partial/Unknown) based on extracted scholarship type information, normalizes intake periods to Fall/Spring format, and standardizes application status to Open/Closed classifications.
- **Duration processing**: Converts various duration formats to standardized float values representing years, with robust parsing that handles decimal formats and invalid entries gracefully.

3. ***Relational Data Transformation:***
   - **Geographic data normalization**: Uses country_converter library to map scholarship program countries and nationality eligibility to standardized country keys, handles special cases like "All" nationality requirements by linking to all available countries.
   - **University relationship management**: Processes university data with automatic deduplication based on name-country combinations, implements safe insertion with conflict resolution to prevent primary key violations, and establishes many-to-many relationships between scholarships and participating institutions (bridge table).
   - **Academic field categorization**: Maps extracted fields of study to predefined standard ISCED categories (0-10), ensuring consistent academic classification across all scholarship records.
4. ***Temporal Data Processing:***
   - **Date parsing and classification**: Converts various date formats (DD/MM/YYYY) to standardized database date objects, and categorizes dates as either deadlines or important dates based on description keywords, and preserves associated comments and contextual information.
5. ***Financial Data Structuring:***
   - **Erasmus-specific financial processing**: Handles complex nested financial structures differentiating between programme country and partner country benefits, preserves detailed tuition coverage information, monthly allowance data,installation cost, and additional benefit details.

4.1.3 Hosting on Cloud

We supported hosting our PostgreSQL database on Supabase, a cloud platform. We created a project on Supabase and migrated the local database using pg_dump. Supabase provides connection pooling and scaling options, which help manage multiple connections and maintain stable performance for data access.

4.2 CV Vector Database

We created a semantic vector database as part of the exploitation zone to index and store CV data. The `SentenceTransformer` model `all-MiniLM-L6-v2` is used to process and convert cleaned and standardized CVs from the trusted zone into dense embeddings. The FAISS similarity search engine is then used to

index these embeddings, making it possible to efficiently retrieve similar CVs for use in subsequent applications like similarity-based search or personalized recommendations.

The loading, embedding, and storing of CVs into the vector database is demonstrated by the script below:

Listing 4.1: Loading CVs from Trusted Zone

```
trusted_cv_path = "trusted_zone/data/resume_data/cleaned"
cvs = []

for filename in os.listdir(trusted_cv_path):
    if filename.endswith('.json'):
        with open(os.path.join(trusted_cv_path, filename), 'r') as f:
            cv_data = json.load(f)
            cvs.append({
                "id": cv_data.get('id', filename.replace('.json', '')),
                "title": cv_data.get('title', ''),
                "content": cv_data.get('content', '')
            })
```

After initializing the model, each CV's vector embeddings are produced:

Listing 4.2: Embedding CVs with SentenceTransformer

```
model = SentenceTransformer('all-MiniLM-L6-v2')
cv_texts = [cv['content'] for cv in cvs]
embeddings = model.encode(cv_texts)
```

FAISS, a quick and memory-efficient similarity search library, is used to index these embeddings:

Listing 4.3: Building and Saving FAISS Index

```
dimension = embeddings.shape[1]
index = faiss.IndexFlatL2(dimension)
index.add(embeddings.astype('float32'))

# Save index and metadata
faiss.write_index(index, 'data/cv_index.faiss')
with open('data/cv_metadata.json', 'w') as f:
    json.dump([{"id": cv["id"], "title": cv["title"], "content": cv["content"]} for cv in cvs]
```

4.3 User-Alumni Network Graph Database

4.3.1 Architecture and Design

The system uses Neo4j for graph storage, Apache Spark for distributed data processing, AWS S3 for scalable storage, and Python threading for concurrent batch operations, ensuring efficient and scalable data handling.

Key Components

The system architecture is composed of several key components that work together to transform raw data into a richly connected graph database.

Data Integration Layer Data is ingested from AWS S3 in JSON and JSONL formats, processed using Spark's distributed computing capabilities to efficiently handle large datasets.

Entity Extraction Engine Multiple entity types are identified and extracted, including personal profiles, educational institutions, professional skills, geographic regions, and achievements.

Graph Construction Framework Nodes are created in batches with uniqueness constraints and parallel processing to optimize performance. Various educational, professional, geographic, and similarity relationships are modeled between entities.

Advanced Similarity Engine A multi-dimensional algorithm scores similarity between students and between students and alumni based on skills, education, institution affiliation, and geographic proximity, supported by indexing, candidate pruning, parallel processing, and sampling for large datasets.

4.3.2 Performance Optimizations

The system uses configurable batch sizes, connection pooling, memory optimizations, multi-threading, and asynchronous operations to enhance throughput and resource management.

Batch Processing Architecture: The system employs configurable batch sizes to optimize database write operations, balancing throughput and resource usage. Connection pooling is used to efficiently manage Neo4j database connections, reducing overhead. Memory management leverages Spark's adaptive query execution and optimization features to handle large datasets while minimizing memory consumption.

Concurrent Processing: Parallelism is achieved through multi-threading, allowing batch operations to

execute simultaneously and speed up processing. Asynchronous operations ensure non-blocking database writes, improving responsiveness. Additionally, resource management routines automatically clean up connections and other resources, maintaining system stability and preventing leaks.

5 Consumption Tasks

5.1 Scholarship CV Matcher System

Our web-based intelligent assistant called the *Scholarship CV Matcher* uses AI-driven semantic similarity to match applicants' resumes with appropriate scholarships. It assesses and suggests scholarship opportunities based on the user's academic and professional profile by combining cutting-edge natural language processing, embedding models, and cloud-based generative AI.

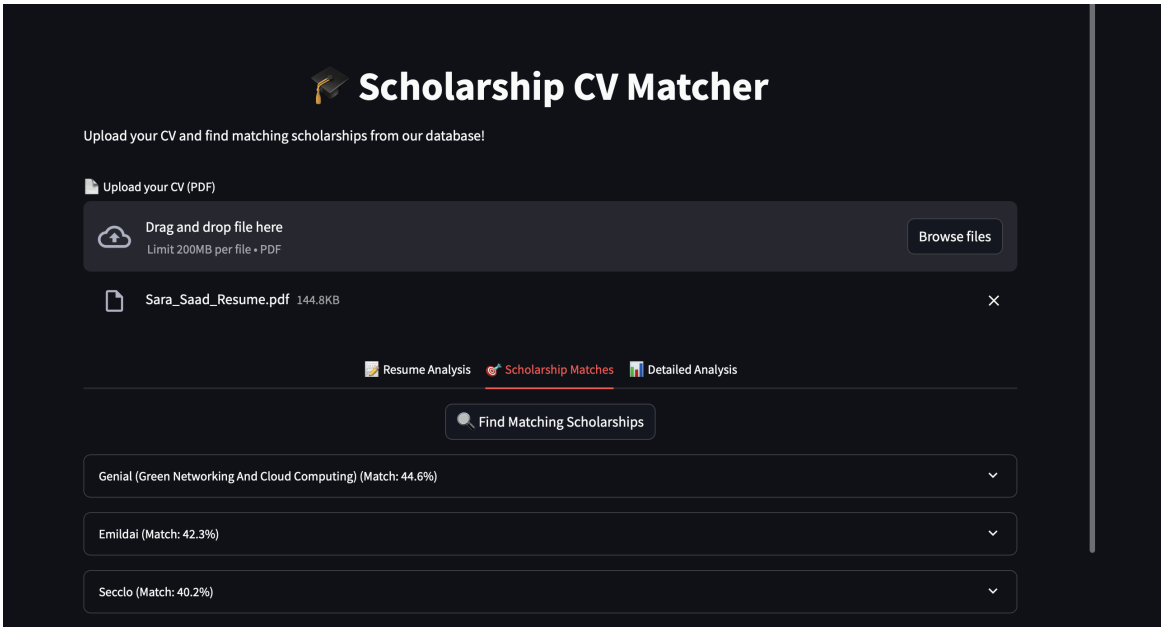

Figure 5.1: Scholarship Matching App

5.1.1 System Architecture

The architecture can be logically divided into the following layers:

1. **Ingestion Layer:** Handles PDF uploads and scholarship data loading.
2. **Processing Layer:** Performs text extraction, embedding creation, and similarity matching.
3. **AI Analysis Layer:** Uses Gemini AI for content generation and explanation.
4. **Interface Layer:** Built with Streamlit for user interaction.

5.1.2 CV Upload and Text Extraction

Listing 5.1: Text Extraction from PDF

```
def extract_text_from_pdf(pdf_path):
    text = ""
    try:
        with pdfplumber.open(pdf_path) as pdf:
            for page in pdf.pages:
                text += page.extract_text() or ""
    except:
        pass
    if len(text.strip()) < 100:
        images = convert_from_path(pdf_path)
        text += "\n".join(pytesseract.image_to_string(img) for img in images)
    return text
```

5.1.3 CV Semantic Embedding

The extracted CV text is segmented into four weighted sections: `education`, `skills`, `experience`, and `other`. Each section is encoded into vector embeddings using a pre-trained `SentenceTransformer` model, and then combined using a weighted average.

Listing 5.2: Embedding Weighted CV Sections

```
weights = {'education': 3.0, 'skills': 2.0, 'experience': 2.0, 'other': 1.0}
section_embeddings = {s: model.encode([cv_sections[s]])[0] for s in cv_sections}
weighted_embedding = sum((weights[s] * section_embeddings[s]
                          for s in section_embeddings)) / sum(weights.values())
```

5.1.4 Scholarship Data Processing

Scholarship descriptions are stored in Postgres Database , in order to be able to deploy the model with Streamlit a python script was generated to convert it into a structured JSON format. Each record includes:

- Name and description
- Required skills and education level
- Eligible countries and institutions
- Funding details and required documents

These are processed and embedded into vectors similarly to the CV.

Listing 5.3: Create Scholarship Embeddings

```
def create_scholarship_embeddings(scholarships, model):
    texts = [f"{s['name']} {s['description']} ... {s['funding']}" for s in scholarships]
    embeddings = model.encode(texts)
    index = faiss.IndexFlatL2(embeddings.shape[1])
    index.add(embeddings.astype('float32'))
    return index, scholarships
```

5.1.5 Semantic Similarity Matching

A high-speed similarity search engine called `FAISS` is used to perform a similarity search after the CV and scholarships have both been represented as vectors. The top k scholarships that best fit the CV are found using the cosine similarity (or its proxy via L2 distance).

Listing 5.4: Top-K Scholarship Matching

```
distances, indices = index.search(weighted_embedding.reshape(1, -1), top_k)
matches = [{'scholarship': scholarships[i], 'score': 1 - d}
           for i, d in zip(indices[0], distances[0]) if i != -1]
```

5.1.6 Generative Analysis with Gemini AI

Google Gemini AI is used for two main things :

1. **CV Analysis:** Evaluates the CV and provides feedback on strengths, experience, and skills and gives a score for the cv.

2. **Scholarship Match Justification:** Generates an explanation for the match between CV and scholarship and gives recommendation on hwo to improve your cv to get accepted into the scholarship.

Listing 5.5: Gemini CV Analysis Prompt

```
prompt = f"""
You are an expert reviewer. Analyze this CV and return:
1. Score
2. Skills
3. Education
4. Improvements
CV:
{resume_text}
"""
response = genai.GenerativeModel("gemini-1.5-flash").generate_content(prompt)
```

5.1.7 Streamlit Frontend Interface

Streamlit is used to deploy the application as an interactive web application. It includes:

- A file uploader for CVs
- Tabs to view Resume Analysis, Matching Scholarships, and Detailed Analysis
- Expandable panels for each scholarship's information and match score

Find Similar CV Function using Vector Database

An uploaded CV is compared to a pre-built vector database of previously processed resumes using the *Find Similar CV* function. The system uses a SentenceTransformer model to create an embedding after extracting the text from a user's newly uploaded resume. The closest matches are then found using vector similarity by querying this embedding against the FAISS-based CV vector database. By computing the cosine similarity between the uploaded CV and the stored vectors, the system finds the top k most similar CVs. This feature can be used to find users with similar profiles, support segmentation and clustering for scholarship recommendations, or compare the uploaded resume to previous applicants to identify areas for improvement or opportunities.

5.1.8 Technologies Stack

Component	Technology
Text Extraction	pdfplumber, pytesseract
Semantic Embeddings	SentenceTransformer
Similarity Search	FAISS
Generative AI	Google Gemini AI
Web Interface	Streamlit
Data Format	JSON

Table 5.1: Technology stack for Scholarship CV Matcher

In order to customize the scholarship discovery process, this system exhibits a hybrid architecture that blends generative AI, semantic search, and structured data processing. When FAISS and Gemini AI are combined, recommendations can be made effectively and intelligibly, giving end users transparency and relevance.

5.2 Scholarship Analytics Dashboard

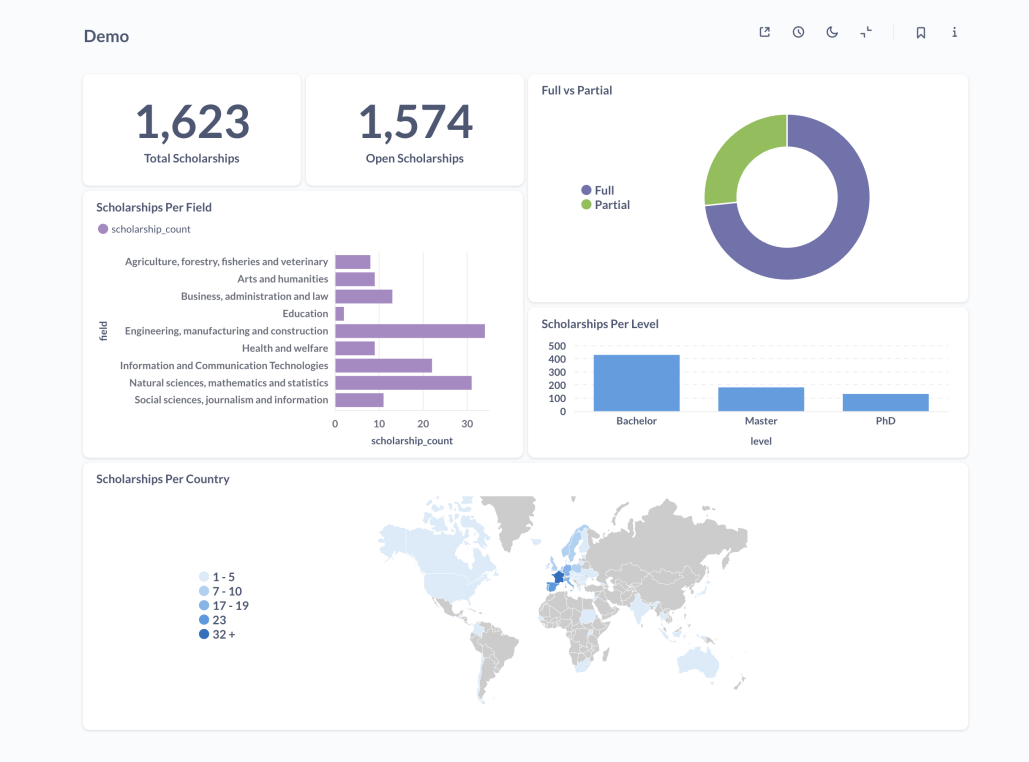

Figure 5.2: Scholarship Analytics Dashboard

Using our PostgreSQL database hosted on Supabase, we utilized Metabase to build an analytics dash-

board. The dashboard was created by running SQL queries that summarize key scholarship data. These queries count total scholarships, filter open scholarships, group scholarships by funding type and academic level, categorize them by fields of study, and aggregate scholarships by program countries. This approach enables clear visualization of scholarship distribution and status.

```
SELECT COUNT(*) AS total_scholarships
FROM scholarships_common;

SELECT COUNT(*) AS open_scholarships
FROM scholarships_common
WHERE status = 'Open';

SELECT funding, COUNT(*) AS count
FROM scholarships_common
WHERE funding IN ('Full', 'Partial')
GROUP BY funding;

SELECT f.field, COUNT(*) AS scholarship_count
FROM scholarships_fields sf
JOIN fields_of_study f ON sf.field_key = f.key
JOIN scholarships_common sc ON sf.sch_key = sc.key
WHERE sc.level <> 'Unknown' and f.field <> 'Generic programmes and qualifications'
GROUP BY f.field, sc.level
ORDER BY f.field, scholarship_count DESC;

SELECT level, COUNT(*) AS count
FROM scholarships_common
WHERE level <> 'Unknown'
GROUP BY level;

SELECT c.name AS country, COUNT(*) AS scholarships_count
FROM program_countries_sch pcs
JOIN countries c ON pcs.country_key = c.key
GROUP BY c.name
ORDER BY scholarships_count DESC;
```

5.3 Student Alumni Graph Database Querying

The consumption of the student alumni network graph involves implementing a comprehensive graph query analyzer using cypher queries that enable efficient data retrieval and analysis from a Neo4j graph database containing student and alumni networking information. This system provides a programmatic interface for extracting meaningful insights from the interconnected network data.

5.3.1 Statistical Analysis Capabilities

The system supports extensive statistical reporting by retrieving metrics such as node and relationship counts, along with distribution patterns across entity types. It performs detailed skill analysis to reveal trends and high-demand competencies among students and alumni. Additionally, it offers insights into educational institutions by summarizing enrollment figures, alumni statistics, and prevalent academic and professional pathways.

5.3.2 Relationship Discovery and Analysis

Advanced relationship discovery mechanisms enable peer similarity matching, helping students find others with shared attributes and interests. The system also supports alumni mentorship recommendations by aligning student profiles with compatible alumni. Networking opportunities are surfaced through analysis of shared skills, institutions, and educational experiences.

5.3.3 Geographic and Demographic Insights:

Geographic analysis tools reveal regional trends in skill development, helping identify in-demand competencies by location. The system also facilitates the formation of study groups by using similarity scores and shared institutional affiliations to connect compatible students for collaboration.

5.3.4 Data Export and Custom Querying:

To support external analysis and flexibility, the system allows users to export selected datasets in CSV format. It includes a custom Cypher query interface for advanced analytical needs, enabling tailored queries on the graph database. Results are formatted consistently to ensure clarity and ease of interpretation.