Evidi

AI-powered assistant for sourcing, filtering and summarizing job offers.

ROBIN, Héloïse
Data & IA Major
ESILV Paris
Paris, France
heloiserobin@hanyang.ac.kr

KHEYAR¸ Adel
Dept. Computer Science
Hanyang University
Seoul, S. Korea
adelkheyar@hanyang.ac.kr

ARM, Jules
Dept. Computer Science
Hanyang University
Seoul, S. Korea
julesarm@hanyang.ac.kr

DESJONQUERES, Nicolas
Data & IA Major
ESILV Paris
Paris, France
nicolas.desjonqueres@edu.devinci.fr

***Abstract*—The Evidi Job Response Assistant is a webapp that aims to streamline the job application process through automation with artificial intelligence. In the current employment landscape, job seekers face the repetitive task of manually searching, reviewing, and responding to numerous job offers across multiple platforms. This project proposes an automated workflow capable of retrieving job offers from multiple sources, filtering them based on personalized user criteria, giving an AI-generated match score and feedback, summarizing descriptions using language models, and generating a draft for a cover letter.**

The proposed solution's tech stack is as follows: FastAPI serves as the backend API, MongoDB Atlas as the cloud database, n8n for the AI-driven workflows, and React + TypeScript as the frontend UI. In addition, the project leverages Gemini's API for filter extraction, job offer matching, and content creation (AI summaries, cover letters). The app is deployed entirely in the cloud with Vercel for the frontend + backend, and Railway for hosting the n8n workflows.

***Index Terms*—Job search, Artificial Intelligence, Automated workflows, FastAPI, n8n, MongoDB Atlas, React, Vercel, Railway, Natural Language Processing**

I. Role Assignments

TABLE I: Role Assignments

Role	Name	Task Description
User	Jules	Use the app as a real user. Provide feedback on its use of the app. Report bugs, and suggest improvements.
Customer	Adel	Provide requirements and feedback. Validate the functional design and user experience. Evaluate deliverables during milestones (UI mockups, MVP demo). Ensure the project meets academic or business goals.
Frontend developer	Nicolas & Héloïse	Implement frontend in React (UI, API calls). Develop backend API. Design and integrate the database (MongoDB Atlas). Configure n8n workflows for automation and AI processing. Test, debug, and deploy components on cloud platforms.
Development Manager	Héloïse	Define project scope, timeline, and milestones. Assign tasks to team members and manage version control on GitHub. Supervise testing, deployment, and documentation.

II. Introduction

A. Motivation

In today's world, the process of job seeking has evolved into an increasingly complex and repetitive task. As students actively searching for internships and experiencing the process firsthand, we have encountered various forms of mental fatigue and stress stemming from these challenges. Candidates are now required to navigate a growing number of job platforms, manage many applications to secure a single interview opportunity, and adapt to ever-longer and more intricate recruitment processes.

Because of this, job seekers have to spend more and more time reviewing postings, extracting relevant qualifications, and tailoring application materials for each opportunity. Over time, these repetitive actions contribute to demotivation, reduced efficiency, and even missed opportunities. At the same time, organizations receive vast volumes of generic or mismatched applications, revealing a structural imbalance and inefficiency in the modern recruitment system.

With the rapid progress in Artificial Intelligence (AI), particularly in workflow automation, a unique opportunity arises to reimagine this whole process. Automating the collection, analysis, and filtering of job offers can significantly reduce the burden placed on applicants while improving the relevance and quality of submissions. Such an approach not only streamlines the job search but also promotes more equitable and intelligent access to employment information. The motivation behind this work is therefore to develop a transparent, modular, and accessible tool that leverages automation ethically and effectively, supporting individuals throughout their job search journey and alleviating the cognitive load inherent to current recruitment processes.

B. Problem Statement (User Needs)

Despite the apparent convenience of existing online job boards such as **LinkedIn**, **Glassdoor**, or **Indeed**, users continue to face several persistent pain points. These platforms provide keyword filters and automated alerts, yet they remain fundamentally passive, offering limited personalization and no actionable feedback. Users must still sift through each posting, interpret nuanced requirements, and manually prepare application materials.

Automation services like **Zapier** or **Make (Integromat)** enable general workflow integration, but they are not tailored to employment-specific scenarios and require prior technical knowledge to build effective workflows. They lack semantic understanding of job-related data, resulting in rigid automation pipelines. Moreover, while advanced AI systems such as **ChatGPT** can generate natural language content, they still demand extensive prompting and lack integration with dynamic job feeds or structured filtering mechanisms.

From an educational and research standpoint, this absence of a unified, open, and domain-specific framework poses a challenge for practitioners and students seeking to explore AI-driven automation in realistic settings. Therefore, there is a clear need for a customizable, intelligent, and transparent ecosystem that unifies job data retrieval, summarization, and response generation within a single, cloud-ready platform.

C. Existing Solutions

A comparative analysis of current tools reveals several partial solutions, yet none fully address the multidimensional needs of job seekers.

Platforms such as **Simplify** and **LoopCV** attempt to streamline the application process by automating repetitive form-filling or submission tasks. However, their infrastructures are proprietary and non-extensible, limiting opportunities for customization or academic exploration. Similarly, **Huntr** provides efficient tracking for ongoing applications but lacks an AI-driven decision layer capable of analyzing or ranking offers intelligently.

On the other hand, low-code automation platforms such as **Zapier** and **n8n** allow users to design workflows visually, connecting data sources and web APIs. While powerful, they operate as generic middleware and do not incorporate domain-specific heuristics such as keyword extraction, offer classification, or motivation-letter personalization. Finally, AI-driven assistants like **ChatGPT** or **Claude** can generate text upon request but cannot autonomously interact with job data sources or maintain persistent user contexts across sessions.

This review highlights the fragmented nature of current technological ecosystems and underscores the necessity of an integrated, open-source framework where AI models, automation logic, and user interfaces converge seamlessly.

D. Proposed Solution

The **Evidi Job Response Assistant** is designed to address these gaps by combining the flexibility of modern cloud computing with the intelligence of advanced language models. The proposed system features a modular architecture comprising four key layers: **data ingestion**, **AI-driven processing**, **backend management**, and **interactive visualization**.

Through **n8n**, the system automatically retrieves job offers from diverse sources such as RSS feeds, email inboxes, and public APIs. These offers are then filtered through user-defined criteria (including domain, skills, or salary range) and stored in a **MongoDB Atlas** database managed via a **FastAPI** backend. The backend exposes RESTful endpoints that feed into a **React + TypeScript** frontend, where users can visualize offers, summaries, and application drafts.

An **AI layer**, powered by **OpenAI GPT-4**, performs advanced summarization and generates personalized application responses. The combination of these technologies results in a robust workflow that minimizes manual intervention, enhances precision, and supports reproducible research. Beyond its technical contributions, this project aims to demonstrate a scalable model for **AI-assisted decision-making** and to provide an educational reference for integrating automation, data engineering, and language intelligence in real-world employment contexts.

III. Requirements

A. User session management

The system must provide secure and user-friendly mechanisms for registration, authentication, and session management.

- **Registration:** New users must create an account using a valid email address and password. Upon submission, an email verification code is sent automatically. Only verified accounts gain access to the main interface.
- **Password Security:** All user passwords are hashed using a robust cryptographic algorithm (e.g., **SHA-256** or **bcrypt**) prior to database storage. Plaintext passwords are never stored or transmitted.
- **Login:** The login process validates the provided credentials against stored hash values. Upon success, a **JWT (JSON Web Token)** is issued to manage authenticated sessions securely across the frontend and backend.
- **Account Recovery:** In case of forgotten credentials, the system provides a password reset flow via email-based verification. Expired or invalid tokens are automatically rejected to maintain security.

B. User Criteria Management

Users can define personalized job search criteria that guide the system's filtering and retrieval mechanisms.

- **Criteria Creation:** Users specify parameters such as target keywords, job titles, industries, required skills, employment type (e.g., remote, full-time, internship), and geographic location.
- **Criteria Persistence:** Each user's preferences are stored in the cloud database under a unique user identifier, allowing the same filters to be reused automatically for subsequent job searches.
- **Dynamic Modification:** The user interface enables modification or deletion of existing criteria. Any change triggers an immediate synchronization across the backend and automation workflows.
- **Validation:** Input validation ensures the accuracy of filter definitions, preventing empty or malformed entries before committing data to storage.

C. Job Offer Ingestion System

The ingestion module is responsible for automatically collecting job offers from multiple external sources in a structured and scalable manner.

- **Sources of Data:** The system supports several channels:
  - RSS feeds from public job boards.
  - REST APIs from professional platforms (e.g., LinkedIn, Indeed, Welcome to the Jungle).
  - Email inbox parsing for newsletters and subscriptions.
- **Automation Pipeline:** Workflows are executed through the **n8n** automation platform. Each workflow consists of nodes for fetching, transforming, and forwarding job data to the backend.
- **Data Standardization:** Collected job data is normalized into a unified JSON structure with standardized fields such as `title`, `company`, `location`, `skills`, `description`, and `source`.
- **Scheduling:** The ingestion process operates at configurable intervals (e.g., every 2 hours) or can be triggered manually by the user through the frontend.

D. Offer Filtering Module

Once data is ingested, the filtering module compares each offer against the user's criteria to identify relevant matches.

- **Matching Algorithm:** A hybrid approach combining keyword search and semantic similarity (e.g., cosine similarity via sentence embeddings) determines the relevance of each offer.
- **Scoring System:** Each offer receives a numerical relevance score between 0 and 1. Only offers above a user-defined threshold (e.g., 0.7) are retained for summarization.
- **Duplicate Detection:** Hash-based identifiers prevent repeated storage of identical job listings from multiple sources.
- **Result Storage:** Filtered offers are saved in the MongoDB database and marked with their corresponding user ID, timestamp, and relevance score.

E. AI Summarization Engine

The summarization engine leverages **OpenAI GPT-4** to generate concise and structured job summaries.

- **Input Data:** The engine receives the normalized job description text and associated metadata from the filtering module.
- **Prompt Template:** A predefined template guides the model to produce summaries containing the following sections: *Position Title, Company Overview, Key Responsibilities, Required Skills, and Application Insights.*
- **Output Format:** Summaries are returned as structured text blocks and stored alongside the original job offers in the database.
- **Quality Control:** The backend verifies that all expected fields are present before saving the result. In case of missing or malformed output, a re-generation is automatically triggered.

F. AI Letter Draft Generation (Optional)

An optional functionality enables users to generate personalized cover letter drafts for selected job offers.

- **Input Context:** The model combines three sources of information: (1) the summarized job offer, (2) the user's stored profile data, (3) previously defined motivation style preferences.
- **Prompt Structure:** The AI is instructed to generate a professional and context-aware letter draft with three sections: introduction, motivation, and conclusion.
- **User Review:** Generated letters are displayed in the frontend editor, allowing the user to modify, approve, or export them in text or PDF format.

G. Data Management Layer

All information produced by the system (job offers, summaries, and user data) is stored and managed within a secure cloud database.

- **Database Engine:** The system uses **MongoDB Atlas**, chosen for its scalability, flexibility, and document-based structure that fits dynamic job data.
- **Data Schema:** Each document includes nested structures for offer metadata, AI-generated summaries, and associated user identifiers.
- **Backup and Retention:** Automatic backup policies ensure data persistence. Obsolete or expired job offers are archived to a secondary collection for future analysis.

H. Notification and Alert System

To enhance user engagement, the system automatically informs users of new or relevant job opportunities.

- **Notification Triggers:** Alerts are generated whenever a newly ingested job offer exceeds the user's relevance threshold.
- **Delivery Channels:** Notifications are dispatched through email or third-party integrations such as Slack or Telegram via **n8n** connectors.
- **Content Format:** Each notification includes the job title, company name, and a short excerpt of the AI summary with a link to view full details on the dashboard.

I. Frontend Visualization Dashboard

The user interface provides access to all system functionalities in an organized and interactive manner.

- **Technology Stack:** Developed using **React** and **TypeScript**, ensuring responsiveness, modularity, and cross-platform accessibility.
- **Views and Components:** The dashboard consists of multiple pages: login, profile settings, job feed, AI summaries, and letter drafts.

- **Interaction Design:** Users can filter, search, and sort offers; review AI-generated content; and trigger workflow actions directly (e.g., "generate letter" or "refresh offers").

J. System Integration Workflow

This component defines the interaction model between all subsystems of the platform, ensuring reliable communication across the automation layer, backend, database, and frontend.

- **Backend API:** Implemented with **FastAPI**, the backend exposes RESTful endpoints for job data retrieval, filtering operations, and AI-driven processing through the Gemini API.
- **Authentication Flow:** Communication between the frontend and backend is secured using JWT-based authentication, ensuring protected access to user-specific functionalities.
- **Automation Orchestration: n8n** workflows, hosted on Railway, periodically fetch and preprocess external job data sources before synchronizing them with the backend database, maintaining up-to-date listings.
- **Deployment Infrastructure:** The frontend and backend are deployed on **Vercel**, offering automated builds and global edge delivery. Automation workflows run on **Railway**, while persistent data storage is managed through **MongoDB Atlas**.

IV. Development Environment

A. Choice of Software Development Platform

The **Evidi Job Response Assistant** is developed as a distributed, cloud-based web application that integrates automation, AI, and workflow orchestration. The project aims to streamline the job search process by automating the retrieval, filtering, and summarization of job offers through AI-driven methods. Given the short project timeline (two months) and the requirement for a scalable, low-maintenance architecture, we selected a modern and cloud-native technology stack.

The backend is implemented in Python using **FastAPI**, chosen for its simplicity, performance, and native support for asynchronous operations. The database layer relies on **MongoDB Atlas**, a cloud-based NoSQL solution well suited for handling unstructured text data such as job descriptions. The automation component is powered by **n8n**, which provides a visual workflow environment to connect APIs and automate data ingestion tasks. The frontend is built with **React** and **TypeScript**, with UI prototyping conducted using **Figma**'s AI-assisted design tools. The system's intelligence layer uses **Gemini**'s API to perform match scoring, job summarization, and draft letter generation.

For deployment, both the backend and frontend are hosted on **Vercel**, selected for its streamlined developer experience and efficient debugging capabilities. The automation workflows are deployed on **Railway**, which also provides the necessary database resources required for operating **n8n**.

This configuration provides a balance between scalability, modularity, and ease of collaboration among our team.

TABLE II: Tools and Language Choice

Tools and Language	**Reason**
FastAPI (Python)	FastAPI is a modern, high-performance Python framework optimized for building APIs. Its ASGI-based asynchronous support enables efficient handling of concurrent requests, which is essential for AI summarization calls and webhook-based integrations. Built-in OpenAPI generation, Pydantic validation, and type safety contribute to a robust and maintainable backend.
MongoDB Atlas	MongoDB Atlas provides a fully managed, cloud-hosted NoSQL database ideal for dynamic and semi-structured data such as job listings. Its flexible document model allows variable fields without strict schemas. The service offers high availability, effortless scalability, and seamless integration with Python, reducing infrastructure maintenance overhead.
n8n	n8n offers a visual, low-code automation platform capable of orchestrating workflows across APIs, databases, and AI services. It automates RSS ingestion, filtering, and forwarding logic to the backend, significantly reducing custom scripting. Its cloud-friendly deployment and transparency make it accessible to both technical and non-technical contributors.
React + TypeScript	React provides a modular, component-based architecture for building responsive and dynamic user interfaces. TypeScript adds static type checking, improving reliability and reducing runtime errors. This combination accelerates frontend development while ensuring maintainability and smooth integration with backend services.
Gemini API	The Gemini API delivers advanced natural language processing capabilities for filter extraction, summarization, match scoring, and cover letter draft generation. It enables efficient transformation of long job descriptions into concise outputs and supports scalable experimentation through prompt engineering in an API-first architecture.
Vercel + Railway	Vercel offers seamless deployment and continuous integration for both the backend and frontend, providing a unified developer experience and efficient debugging environment. Railway hosts the automation workflows and provides the database resources required for operating **n8n**, ensuring reliable background job execution.

B. Cost Estimation

Ensuring the reliability and scalability of the **Evidi Job Response Assistant** requires cloud services that minimize operational overhead while providing sufficient performance. We selected free-tier and low-cost cloud options to maintain budget efficiency during the prototype phase while retaining professional-grade capabilities.

Vercel provides free-tier hosting for both the FastAPI backend and the React frontend, supporting continuous deployment, SSL-secured endpoints, and seamless integration with GitHub. Workflow automation and background tasks are deployed on **Railway**, whose cheapest plan (approximately 0.40 USD per day) offers persistent execution and the infrastructure required for running **n8n**. For data storage, **MongoDB Atlas**'s M0 cluster tier supports several thousand job entries at no cost, offering built-in backups and high availability.

API-driven intelligence is powered by the **Gemini** free plan, which provides sufficient quota for summarization, match scoring, and draft generation during development without incurring usage fees.

Overall operational costs remain minimal. The only recurring expense is the **Railway** plan, amounting to roughly 12 USD per month, keeping the total cost low while ensuring reliable cloud-based operation of the system.

TABLE III: Hosting and AI Tools

Tools and Services	Reason
Vercel (Backend & Frontend Hosting)	Vercel provides a unified platform for hosting both the FastAPI backend and the React frontend. Its automated CI/CD pipelines, global edge network, and built-in HTTPS support enable seamless deployment and rapid iteration without server management.
Railway (Automation Hosting)	Railway hosts the **n8n** workflows used for automation and data ingestion. Its low-cost plan (0.40 USD/day) offers persistent execution, easy scaling, and integrated resource provisioning, making it suitable for handling recurring background tasks.
MongoDB Atlas (Database Hosting)	MongoDB Atlas's M0 free-tier cluster provides a fully managed NoSQL database with automated backups, monitoring, and high availability. It is well suited for storing unstructured job listings and requires no local infrastructure.
Gemini API (AI Processing)	The Gemini API delivers advanced natural language processing capabilities for summarization, match scoring, and draft letter generation. The free plan offers sufficient quota for prototype-level workloads without introducing additional operational costs.

C. Software in Use

1) Existing Systems / Tools:

Simplify (Simplify Copilot) is a browser-based tool that automates repetitive job application tasks. It provides features such as auto-filling application fields, tracking applications, tailoring resumes, and identifying missing keywords in one's CV. While Simplify excels at streamlining the manual data entry portion of applications, it does not (publicly) provide a unified backend, cross-source ingestion pipeline, or AI summarization embedded in a web app.

LoopCV is another job search automation platform that matches job seekers with listings and automates part of the application process. It uses AI to optimize CVs, track applications, and even directly apply on behalf of the user. LoopCV's strength lies in its end-to-end workflow (search $\rightarrow$ apply $\rightarrow$ track), but it is a closed, commercial product with limited transparency into its internal pipelines.

Huntr offers features for job application tracking, AI-assisted resume and cover letter generation, and auto-filling application forms. However, Huntr is primarily a productivity / tracking tool; it does not appear to automate ingestion from RSS or provide full workflow orchestration with external services like n8n.

LazyApply automates job applications across job platforms via AI, handling form filling and submission. Its value is in applying at scale, but challenges include handling custom fields, CAPTCHA, and job boards with complex forms.

From the research perspective, **ResumeFlow: An LLM-facilitated Pipeline for Personalized Resume Generation and Refinement** introduces a pipeline that takes job descriptions and resumes and produces tailored, optimized CVs using LLMs. This aligns with our AI-driven summary / draft generation module. It demonstrates the viability of leveraging LLMs for alignment between job descriptions and user profiles.

2) Comparison & Gap Analysis:

- **Scope of ingestion:** Existing tools like Simplify or LazyApply typically rely on browser extension or manual input, whereas our project plans to support ingestion via RSS feeds, APIs, and emails through automation workflows.
- **AI summarization / drafting:** While tools offer resume optimization or cover letter suggestions, few expose summarization of full job descriptions or draft generation from user profile + job content. Our approach explicitly integrates that.
- **Transparency and extensibility:** Commercial tools are black-box; you cannot inspect or customize their pipelines. We provide modular architecture (n8n + API backend) that is open, testable, and extensible.
- **Integration & orchestration:** Our system orchestrates ingestion, filtering, AI processing, and persistence together. Tools like Simplify partly automate, but lack seamless end-to-end pipeline control integrated with a web app interface.
- **Flexibility & deployment:** Because ours relies on open stack (FastAPI, MongoDB, n8n, React), we can adapt features, scale, modify workflows, or replace components, which is typically impossible with off-the-shelf tools.

V. Specifications

A. Requirement 1 – User Management

Goal: Allow users to register, authenticate, and manage their profile and preferences.

Implementation:

- **Frontend:** Login and Register pages with full name, email, password, and confirmation fields. Settings page for profile update, password change, and notification preferences. Axios handles communication with backend.
- **Backend:** FastAPI endpoints `/register`, `/login`, `/user/preferences`, `/user/profile`, and `/user/password`. JWT-based authentication and bcrypt password hashing.
- **Database:** MongoDB collection `users` storing { `_id, full_name, email, password_hash, preferences, settings` }.

- **Security:** JWT tokens for authentication, password validation, and session protection.

Pseudocode:

```
POST /register:
  receive {full_name, email, password}
  hash = bcrypt.hash(password)
  insert into db.users({full_name, email, hash})
  return success

POST /login:
  check email exists
  verify bcrypt(password, hash)
  return JWT_token
```

B. Requirement 2 – Criteria & Filter Management

Goal: Allow users to define, update, and store job search preferences.

Implementation:

- **Frontend:** Filters page with tech stack tags, experience level, include/exclude keywords, location preferences, and job type options.
- **Backend:** FastAPI endpoints `/criteria/update` and `/criteria/get`.
- **Database:** MongoDB collection `criteria` linked to user ID.
- **Integration:** n8n workflows retrieve criteria for dynamic filtering.

Pseudocode:

```
POST /criteria/update:
  user_id = JWT_token.user
  update db.criteria where user_id
  return success
```

C. Requirement 3 – Job Offer Ingestion

Goal: Retrieve and store job offers automatically from public and integrated sources.

Implementation:

- **Automation:** n8n workflows scheduled every 3 hours to fetch from RSS feeds, APIs, or email triggers.
- **Backend:** FastAPI webhook `/webhook/jobs` receives job payloads and stores them in MongoDB.
- **Frontend:** Sources page to add, edit, or remove sources with manual sync and status indicators.

Pseudocode:

```
RSS Trigger -> Filter (new posts only)
-> HTTP POST to FastAPI /webhook/jobs
-> Insert into db.jobs
```

D. Requirement 4 – Offer Filtering

Goal: Match job offers against user-defined criteria.

Implementation:

- **n8n:** "IF" nodes filter job payloads based on keywords and location.
- **Backend:** Python regex fallback for keyword matching.
- **Frontend:** Jobs page filter dropdown (All, Matched, Applied, Rejected) with match badges.

Pseudocode:

```
for job in new_jobs:
  if any(keyword in job.description for keyword in
  user.criteria):
    insert into db.jobs_filtered
```

E. Requirement 5 – AI Summarization

Goal: Generate concise AI summaries for each job description.

Implementation:

- **n8n:** HTTP request node calls OpenAI API (GPT-4/4o-mini).
- **Backend:** FastAPI endpoint `/ai/summarize` for manual trigger.
- **Frontend:** Job Detail Modal's "AI Summary" tab shows results with regeneration option.
- **Database:** MongoDB collection `summaries`.

Pseudocode:

```
POST /ai/summarize:
  input = job.description
  prompt = "Summarize in 5 bullet points"
  response = openai.ChatCompletion(prompt)
  db.summaries.insert({job_id, summary: response})
```

F. Requirement 6 – AI Letter Draft Generation

Goal: Automatically generate a motivation letter based on user's CV and job description.

Implementation:

- **n8n:** Uses OpenAI API with user profile and job data.
- **Backend:** Endpoint `/ai/draft` for manual regeneration.
- **Frontend:** "Cover Letter" tab in Job Detail Modal with edit, regenerate, copy, and download options.

Prompt Example:

```
"Write a short motivation paragraph for
this position based on user's experience
and the job description."
```

G. Requirement 7 – CV Upload & Analysis

Goal: Analyze uploaded CVs to extract skills and job preferences.

Implementation:

- **Frontend:** CV Analysis page with drag-and-drop upload, file card, and AI analysis results.
- **Backend:** Endpoint `/cv/analyze` using OpenAI model for extraction.
- **Database:** Stores extracted skills, experience level, and preferences in `cv_analysis` collection.

Workflow:

```
Upload CV -> Analyze with AI -> Extract
Skills and Preferences -> Store in db.cv_analysis
-> Apply to Filters
```

H. Requirement 8 – Notification System

Goal: Notify users of new job matches via multiple channels.

Implementation:

- **n8n:** Slack, Email, or Push notification nodes.
- **Backend:** Endpoint `/api/notify` for message dispatching.
- **Frontend:** Settings page toggles for email, push, and weekly digest.

Message Example:

```
"New job matching your skills: Data Engineer at
XCorp."
```

I. Requirement 9 – Dashboard & Analytics

Goal: Display user's job search metrics and activity overview.

Implementation:

- **Frontend:** Dashboard with stats cards (Total Jobs, Matched, Applied, Response Rate), recent activity feed, and quick actions.
- **Backend:** Endpoint `/api/dashboard` aggregates metrics.
- **Database:** Activity logs stored for history display.

J. Requirement 10 – Frontend Dashboard and Navigation

Goal: Provide a modern, responsive interface for all application modules.

Implementation:

- **Framework:** React + TypeScript + Tailwind CSS.
- **Global UI:** Header with logo, theme switcher, settings, logout.
- **Navigation:** Tabs for Dashboard, Jobs, Sources, Filters, CV Analysis, and Settings.
- **Theme Support:** Default, Dark, Deep Blue, and Green themes.

K. Requirement 11 – Data Storage

Goal: Securely persist all user and job-related data.

Implementation:

- **Database:** MongoDB collections for users, criteria, jobs, summaries, letters, CV analyses, and notifications.
- **Access Control:** JWT authentication required for all write operations.

Schema:

```
users: { _id, full_name, email,
password_hash, preferences, settings }
criteria: { user_id, tech_stack[],
keywords_include[], keywords_exclude[],
location[], job_type[], experience_level[] }
jobs: { job_id, title, company, description,
tags[], location, source, matched }
summaries: { job_id, summary, ai_model,
updated_at }
letters: { job_id, user_id, content, updated_at }
cv_analysis: { user_id, extracted_skills[],
experience_level, preferences }
notifications: { user_id, message, type,
timestamp, read }
```

L. Requirement 12 – Logging, Monitoring & Error Handling

Goal: Ensure observability, traceability, and efficient debugging across all components of the system.

Implementation:

- **FastAPI:** Request/response logging handled through middleware, including timestamps and status codes. Exceptions are captured via FastAPI's global error handler for structured output.
- **n8n (Railway):** Built-in workflow execution logs, node-level error traces, and retry policies for failed tasks.
- **Cloud Hosting (Vercel & Railway):** Vercel provides real-time logs for backend and frontend deployments, including build diagnostics and runtime errors. Railway logs workflow execution events, webhook calls, and system-level failures.
- **Frontend:** Browser-side error boundaries, toast notifications, and retry mechanisms handle user-facing failures and network instability.

M. Requirement 13 – Testing and Validation

Goal: Ensure application stability through automated testing.

Implementation:

- **Backend:** Pytest for unit testing and Postman for integration tests.
- **Frontend:** React Testing Library for component testing.
- **Automation:** GitHub Actions CI pipeline for continuous testing.

N. Requirement 14 – Integration Workflow

Goal: Maintain full interoperability among system components.

Architecture Overview:

- **n8n:** Manages ingestion, filtering, and AI summarization.
- **FastAPI:** Handles authentication, validation, and persistence.
- **MongoDB:** Centralized data storage.
- **OpenAI API:** Provides summarization and letter generation.
- **React Frontend:** Displays user-facing data and controls.

Data Flow:

```
[RSS / API / Email Source]
        ↓
    [n8n Workflow]
        ↓
 [AI Summarizer Node / Letter Draft Node]
```

```
        ↓
[FastAPI Webhooks (/webhook/jobs, /ai/...)]
        ↓
       [MongoDB]
        ↓
[React Frontend Dashboard & Pages]
```

O. Requirement 15 – UX Enhancements (Cross-Page Features)

Goal: Enhance user experience through modern interaction patterns.

Implementation:

- Toast notifications for success, error, and info messages.
- Loading indicators, skeleton loaders, and progress feedback.
- Responsive mobile design with touch-friendly inputs.
- Keyboard accessibility for modals and navigation.
- Smooth theme transitions and persistent preferences.

VI. Architecture Design & Implementation

This section presents the architectural structure of the Evidi system, with a focus on its modular decomposition and implementation strategy. Each module is described in detail, including its purpose, responsibilities, internal components, and reasons for selection. Visual representations are omitted for simplicity but can be added as figures if needed.

A. Global Technical Architecture

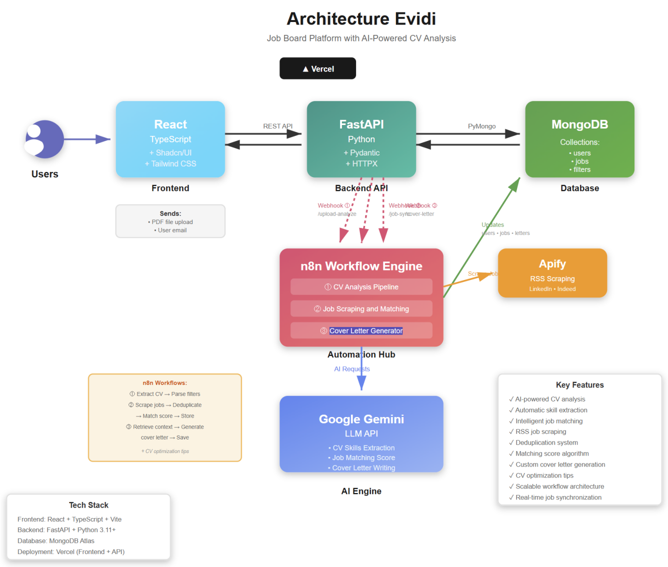

Fig. 1: Global architecture

B. Module 1 — Backend API (FastAPI)

Purpose: The Backend API is responsible for managing all server-side logic, data persistence, user authentication, and communication between the frontend, the database, and the AI workflow engine (n8n). It acts as the central orchestrator of the Evidi architecture.

Functionality:

- Provides RESTful endpoints for job offers, user criteria, summaries, and authentication.
- Validates and processes all data received from the frontend.
- Interacts with MongoDB Atlas to store and retrieve structured job and user data.
- Receives data from n8n workflows (job ingestion, summarization, AI matching).
- Ensures secure JWT-based authentication to safeguard user sessions.

Location of Source Code:

- Stored in the `/backend/` directory of the project repository.
- Deployed automatically through Vercel's serverless backend environment.

Class and Component Structure:

- **main.py** — Initializes the FastAPI application and registers all routers.
- **routers/auth.py** — Handles login, registration, and password recovery.
- **routers/jobs.py** — Exposes CRUD operations for job offers.
- **routers/criteria.py** — Manages user-defined filtering rules.
- **database/mongo.py** — Defines the MongoDB connection client.
- **models/.py** — Defines Pydantic models for validation and consistency.

Origin and Rationale: FastAPI was chosen because:

- It offers high performance through asynchronous execution.
- It integrates seamlessly with Pydantic for type-safe data models.
- Its auto-generated Swagger/OpenAPI documentation accelerates development.
- It fits well with serverless deployment on Vercel.

Graphical Representation (Description):

- The module sits at the center of the system architecture.
- Communicates upstream with the frontend, downstream with MongoDB, and laterally with n8n.

C. Module 2 — Automation Workflows (n8n)

Purpose: This module automates the ingestion, transformation, filtering, and AI-based enrichment of job offers. It serves as the intelligence pipeline that feeds the backend with structured and enriched job information.

Functionality:

- Fetches job offers from external sources (RSS, APIs, email inbox).
- Normalizes raw job data into a common JSON structure.
- Extracts job requirements and skills using AI prompts.
- Sends summarized job descriptions and match scores to the backend API.
- Schedules repeated workflows (e.g., every 2 hours).

Location of Workflow Files:

- Stored and executed directly within the n8n cloud workspace.
- Exportable as JSON files to the project repository under `/workflows/`.

Component Breakdown:

- **HTTP Request Nodes** — Retrieve job postings from external APIs.
- **RSS Nodes** — Subscribe to job boards and periodic updates.
- **Email Parsing Nodes (IMAP)** — Extract content from newsletters.
- **Transform Nodes** — Map and clean scraped fields.
- **OpenAI/Gemini Nodes** — Perform match scoring, text summarization, and cover letter generation.
- **Webhook / API Nodes** — Push structured data into the FastAPI backend.

Origin and Rationale: n8n was chosen because:

- It provides a no-code/low-code environment suited for rapid workflow construction.
- It supports integration with virtually any API via native nodes.
- It drastically simplifies automation processes that would otherwise require extensive backend coding.
- It can run fully in the cloud, enabling fast deployment on Railway.

Graphical Representation (Description):

- The workflows form a directed graph: *Data Source → Transformation → AI Processing → Backend API.*
- The system repeats this flow periodically to keep job data up to date.

D. React Frontend

Purpose: The React Frontend provides the user interface for interacting with the Evidi platform. Its purpose is to deliver a fast, responsive, and intuitive user experience while communicating with the backend API and presenting AI-processed job data clearly and efficiently.

Functionality:

- Displays job offers, summaries, match scores, and user-filtered results.
- Provides forms for user authentication, criteria creation, and job exploration.
- Sends requests to the FastAPI backend and renders returned data dynamically.
- Manages user session state through JWT tokens stored securely.
- Handles loading states, error feedback, pagination, and UI transitions.

Location of Source Code:

- Entirely stored in the `/frontend/` directory.
- Deployed on Vercel for seamless CI/CD and automatic static optimization.

Component and Class Structure:

- **App.tsx** — Root component, router setup, global layout.
- **pages/.tsx** — Page-level components (Dashboard, Login, Job Details).
- **components/.tsx** — Reusable UI elements such as cards, modals, lists, and filters.
- **services/api.ts** — API wrapper for communicating with backend endpoints.
- **context/AuthContext.tsx** — Handles authentication state and token persistence.
- **utils/.ts** — Helper methods for formatting, parsing, and error handling.

Origin and Rationale: React with TypeScript was selected because:

- It provides a component-based architecture ideal for scalable UI development.
- TypeScript ensures type safety, reducing runtime errors.
- Strong ecosystem support (hooks, libraries, Next.js-style patterns).
- Perfect compatibility with Vercel hosting and CI/CD automation.

Graphical Representation (Description):

- The frontend communicates exclusively with the FastAPI backend via HTTPS.
- Data flows from the backend to the UI components, which update reactively.
- Authentication context wraps the app and controls route access.

E. Module 4 — Database Layer (MongoDB Atlas)

Purpose: The Database Layer is responsible for persistent storage of all application data including user accounts, job offers, filtering criteria, AI-generated summaries, cover letters, and system logs. MongoDB Atlas provides a scalable, cloud-hosted NoSQL solution.

Functionality:

- Stores user authentication data with hashed passwords.
- Maintains collections for jobs, criteria, summaries, drafts, and CV analyses.
- Provides indexing for fast query performance on frequently accessed fields.
- Enables flexible schema evolution as new requirements emerge.
- Offers automated backups and disaster recovery capabilities.

Location of Configuration:

- Database connection strings stored in `/backend/.env` file.
- Schema definitions and models in `/backend/models/`.
- MongoDB Atlas dashboard for cluster management and monitoring.

Collection Structure:

- **users** — User authentication and profile information.
- **criteria** — User-defined job search filters and preferences.
- **jobs** — Raw and processed job offer data from various sources.
- **summaries** — AI-generated job summaries linked to job IDs.
- **letters** — Draft cover letters generated for specific applications.
- **cv_analysis** — Extracted skills and preferences from uploaded CVs.
- **notifications** — System notifications and alerts for users.

Origin and Rationale: MongoDB Atlas was selected because:

- It provides a flexible document-based schema suitable for varying job data structures.
- Cloud hosting eliminates infrastructure management overhead.
- Native JSON support aligns perfectly with REST API data exchange.
- Free tier (M0) provides sufficient capacity for prototype development.
- Built-in security features including encryption at rest and in transit.

F. Directory Organization

TABLE IV: Project Directory Structure

Directory	File Names	Module

`/frontend/src/`	`App.tsx`
	`index.tsx`
	`App.css`
		React Frontend (Module 3)

`/frontend/src/pages/`	`Login.tsx`
	`Register.tsx`
	`Dashboard.tsx`
	`Jobs.tsx`
	`Filters.tsx`
	`Settings.tsx`
	`CVAnalysis.tsx`
		Frontend Pages

`/frontend/src/components/`	`JobCard.tsx`
	`JobModal.tsx`
	`FilterForm.tsx`
	`Header.tsx`
	`Sidebar.tsx`
		Reusable UI Components

`/frontend/src/services/`	`api.ts` `auth.ts`	API Client & Auth Service
`/frontend/src/context/`	`AuthContext.tsx` `ThemeContext.tsx`	React Context Providers

`/backend/`	`main.py`
	`requirements.txt`
	`.env`
		Backend API (Module 1)

`/backend/routers/`	`auth.py`
	`jobs.py`
	`criteria.py`
	`ai.py`
	`cv.py`
		FastAPI Route Handlers

`/backend/models/`	`user.py`
	`job.py`
	`criteria.py`
	`summary.py`
		Pydantic Data Models

`/backend/database/`	`mongo.py` `schemas.py`	Database Connection & Schemas

`/backend/utils/`	`auth.py`
	`validation.py`
	`email.py`
		Utility Functions

`/workflows/`	`job_ingestion.json`
	`ai_processing.json`
	`notifications.json`
		n8n Workflow Exports (Module 2)

`/.github/workflows/`	`deploy.yml` `test.yml`	CI/CD Configuration

`/docs/`	`API.md`
	`INSTALL.md`
	`USER_GUIDE.md`
		Documentation

G. Module 5 — Authentication & Security Layer

Purpose: This module ensures secure user authentication, authorization, and data protection across the entire system. It implements industry-standard security practices to safeguard user credentials and sensitive information.

Functionality:

- Handles user registration with email verification.
- Implements JWT-based stateless authentication.
- Hashes passwords using bcrypt before storage.
- Provides password reset functionality via email.
- Enforces role-based access control (RBAC) for future extensions.
- Validates all incoming requests for proper authentication tokens.

Location of Source Code:

- Authentication logic: `/backend/routers/auth.py`
- Token utilities: `/backend/utils/auth.py`
- User models: `/backend/models/user.py`
- Frontend auth context: `/frontend/src/context/AuthContext.tsx`

Key Components:

- **JWT Token Generation** — Creates signed tokens containing user ID and expiration.
- **Password Hashing** — Uses bcrypt with salt for one-way encryption.
- **Token Validation Middleware** — Verifies tokens on protected routes.
- **Email Verification Service** — Sends verification codes for account activation.
- **Session Management** — Stores tokens in secure HTTP-only cookies or local storage.

Origin and Rationale: JWT (JSON Web Tokens) and bcrypt were chosen because:

- JWT enables stateless authentication, reducing server-side session storage.
- bcrypt provides robust password hashing resistant to brute-force attacks.
- Industry-standard implementations ensure security best practices.
- Easy integration with FastAPI and React ecosystems.

H. Module 6 — AI Processing Service

Purpose: This module interfaces with OpenAI's GPT models (specifically GPT-4) to provide intelligent text processing capabilities including job summarization, match scoring, and cover letter generation.

Functionality:

- Generates concise summaries of job descriptions highlighting key information.
- Scores job offers against user criteria to determine relevance.
- Creates personalized cover letter drafts based on job requirements and user profile.
- Extracts structured data from unstructured job postings (skills, requirements, etc.).
- Provides feedback and suggestions for improving application materials.

Location of Source Code:

- AI endpoints: `/backend/routers/ai.py`
- Prompt templates: `/backend/utils/prompts.py`
- Integration called from n8n workflows: `/workflows/ai_processing.json`

Component Breakdown:

- **SummarizationService** — Processes job descriptions into structured summaries.
- **MatchingService** — Compares job requirements with user criteria.
- **DraftService** — Generates personalized application materials.
- **ExtractionService** — Identifies key entities (skills, locations, salary ranges).
- **PromptManager** — Manages and versions AI prompt templates.

Origin and Rationale: OpenAI GPT-4 was selected because:

- State-of-the-art natural language understanding and generation capabilities.
- Reliable API with extensive documentation and community support.
- Excellent performance on complex text analysis tasks.
- Flexible prompt engineering allows customization for specific use cases.
- Cost-effective for prototype-scale usage with reasonable token limits.

VII. Use Cases

This section demonstrates the practical functionality of the Evidi system through concrete use cases. Each use case addresses specific requirements outlined in Section II and shows the complete user workflow.

A. Use Case 1 — User Registration and Profile Setup

Requirement Addressed: Requirement 1 (User Management) and Requirement 2 (Criteria Management)

Objective: A new user registers an account, verifies their email, and sets up their job search preferences.

Step-by-Step Description:

1) **Navigate to Registration Page**
   - User opens the Evidi web application at `https://evidi.vercel.app`
   - Clicks on "Sign Up" button in the navigation bar
2) **Complete Registration Form**
   - User enters full name: "Jean Dupont"
   - User enters email: "jean.dupont@example.com"
   - User creates password meeting security requirements (min. 8 characters)
   - User confirms password in second field
   - User clicks "Create Account" button
3) **Email Verification**
   - System displays message: "Verification email sent to jean.dupont@example.com"
   - User opens email inbox and finds verification message
   - User clicks verification link or enters 6-digit code
   - System confirms: "Email verified successfully"
   - User is redirected to login page
4) **First Login**
   - User enters email and password
   - System generates JWT token and redirects to dashboard
   - Welcome message displays: "Welcome to Evidi, Jean!"
5) **Configure Job Search Criteria**

- User clicks ”Filters” tab in navigation
- User selects tech stack tags: ”Python”, ”React”, ”FastAPI”
- User sets experience level: ”Junior (0-2 years)”
- User adds include keywords: ”internship, stage, apprentissage”
- User adds exclude keywords: ”senior, manager”
- User selects locations: ”Paris”, ”Remote”
- User chooses job types: ”Internship”, ”Full-time”
- User clicks ”Save Preferences”
- System confirms: ”Filters saved successfully”

Expected Results:

- User account created in MongoDB `users` collection
- Password stored as bcrypt hash
- JWT token generated for authenticated session
- User preferences stored in `criteria` collection
- User can now receive personalized job recommendations

Screenshot Description:

- Figure 1: Registration form with all fields filled
- Figure 2: Email verification confirmation screen
- Figure 3: Filters page showing configured preferences with tags and options

B. Use Case 2 — Automated Job Ingestion and AI Summarization

Requirement Addressed: Requirement 3 (Job Offer Ingestion), Requirement 4 (Offer Filtering), and Requirement 5 (AI Summarization)

Objective: The system automatically retrieves job offers from configured sources, filters them based on user criteria, and generates AI summaries.

Step-by-Step Description:

1) **Configure Job Sources**
   - User navigates to ”Sources” page
   - User clicks ”Add Source” button
   - User selects source type: ”RSS Feed”
   - User enters RSS URL: `https://www.welcometothejungle.com/fr/jobs.rss`
   - User names the source: ”Welcome to the Jungle - Data Jobs”
   - User enables the source and sets sync frequency: ”Every 3 hours”
   - User clicks ”Save Source”
2) **Trigger Manual Sync (Optional)**
   - User clicks ”Sync Now” button next to the source
   - System displays loading indicator: ”Fetching jobs...”
   - n8n workflow activates and begins RSS feed retrieval
3) **Automated Workflow Execution (Backend)**
   - n8n RSS trigger node fetches new job postings from feed
   - Transform node normalizes data into standard JSON format
   - Filter node compares job data against user criteria stored in MongoDB
   - Jobs matching criteria are marked with `matched: true`
   - HTTP Request node sends job data to FastAPI webhook: `POST /webhook/jobs`
4) **AI Summarization Process**
   - For each matched job, n8n triggers OpenAI API node
   - System sends prompt: ”Summarize this job description in 5 concise bullet points highlighting: position title, key responsibilities, required skills, company overview, and application details.”
   - GPT-4 generates structured summary
   - Summary is stored in MongoDB `summaries` collection linked to job ID
5) **View Results in Dashboard**
   - User navigates to ”Jobs” page
   - Filter dropdown set to ”Matched Jobs”
   - Job cards display with green ”Match” badge showing score (e.g., 85%)
   - User clicks on a job card titled: ”Junior Data Engineer - Paris”
   - Job Detail Modal opens with three tabs: Overview, AI Summary, Cover Letter
6) **Review AI Summary**
   - User selects ”AI Summary” tab
   - Summary displays in bullet point format:
     - Position: Junior Data Engineer focused on ETL pipeline development
     - Responsibilities: Design and maintain data pipelines using Python and SQL
     - Required Skills: Python, SQL, FastAPI, Docker, basic cloud experience
     - Company: Tech startup specializing in data analytics for retail
     - Application: Send CV and motivation letter to careers@company.com by Dec 15
   - User can click ”Regenerate Summary” for alternative version

Expected Results:

- RSS feed successfully configured in n8n workflow
- New job postings automatically retrieved every 3 hours
- Jobs filtered based on user’s criteria (Python, React, Junior level)
- Matched jobs stored in MongoDB with relevance scores
- AI summaries generated and linked to job records
- User receives concise, actionable information without reading full descriptions

Screenshot Description:

- Figure 4: Sources page showing configured RSS feeds with status indicators
- Figure 5: Jobs page displaying matched job cards with relevance badges
- Figure 6: Job Detail Modal with AI Summary tab showing bullet points

- Figure 7: n8n workflow diagram showing RSS → Filter → AI → Webhook flow

C. Use Case 3 — CV Upload and Analysis

Requirement Addressed: Requirement 7 (CV Upload & Analysis)

Objective: User uploads their CV for AI-powered analysis to automatically extract skills and preferences.

Step-by-Step Description:

1) **Navigate to CV Analysis Page**
    - User clicks "CV Analysis" tab in navigation
    - Page displays drag-and-drop upload area
2) **Upload CV File**
    - User drags CV file (PDF format): "Jean_Dupont_CV.pdf"
    - Alternatively, user clicks "Browse" and selects file
    - System validates file type and size (max 5MB)
    - File card appears showing filename and size
3) **Trigger AI Analysis**
    - User clicks "Analyze CV" button
    - System displays progress indicator: "Analyzing your CV with AI..."
    - Backend sends file to OpenAI API with analysis prompt
4) **View Analysis Results**
    - Analysis completes in 5-10 seconds
    - Results display in structured cards:
        - **Extracted Skills:** Python (Advanced), JavaScript (Intermediate), SQL (Advanced), Docker (Beginner), Git (Intermediate)
        - **Experience Level:** Junior (1 year professional experience + internships)
        - **Preferred Roles:** Data Engineer, Backend Developer, Software Engineer
        - **Education:** Master's in Data Science - ESILV Paris
        - **Languages:** French (Native), English (Fluent), Korean (Basic)
5) **Apply to Filters**
    - User clicks "Apply to My Filters" button
    - System automatically updates user criteria with extracted information
    - Confirmation message: "Your job search filters have been updated based on your CV"
    - User navigates to Filters page to review and adjust auto-populated fields

Expected Results:

- CV file uploaded to temporary storage
- OpenAI extracts structured information from unstructured document
- Skills, experience level, and preferences stored in `cv_analysis` collection
- User criteria automatically updated with relevant information
- User saves time by avoiding manual filter configuration

Screenshot Description:

- Figure 8: CV Analysis page with drag-and-drop upload interface
- Figure 9: Analysis results showing extracted skills and preferences in card format

D. Use Case 4 — AI-Generated Cover Letter

Requirement Addressed: Requirement 6 (AI Letter Draft Generation)

Objective: User generates a personalized cover letter draft for a specific job application.

Step-by-Step Description:

1) **Select Job Offer**
    - User browses matched jobs on Jobs page
    - User clicks on job: "Junior Data Engineer - TechCorp Paris"
    - Job Detail Modal opens
2) **Navigate to Cover Letter Tab**
    - User clicks "Cover Letter" tab
    - If no draft exists, system shows: "No cover letter generated yet"
    - User clicks "Generate Cover Letter" button
3) **AI Generation Process**
    - System displays loading animation: "Crafting your personalized cover letter..."
    - n8n workflow combines:
        - Job description and requirements
        - User's CV analysis data
        - User's profile information
    - OpenAI API generates draft with structured sections:
        - Introduction paragraph
        - Motivation and skills alignment
        - Conclusion with call to action
4) **Review and Edit Draft**
    - Generated letter appears in editable text area
    - User reviews content for accuracy and tone
    - User makes minor edits to personalize further
    - Example generated content:

        "Dear Hiring Manager,
        I am writing to express my strong interest in the Junior Data Engineer position at TechCorp. As a recent graduate with a Master's in Data Science from ESILV Paris and hands-on experience in Python and ETL pipeline development during my internship at DataCorp, I am excited about the opportunity to contribute to your data infrastructure team.
        My technical background aligns well with your requirements, particularly my proficiency in Python, SQL, and FastAPI, which

I used to build automated data processing pipelines that reduced processing time by 40%. Additionally, my experience with Docker containerization and cloud deployment on AWS would enable me to contribute effectively to your microservices architecture..."

5) **Save and Export**
   - User clicks "Save Draft" to store in database
   - User has options to:
     - "Copy to Clipboard" for pasting into application form
     - "Download as PDF" for formal submission
     - "Regenerate" for alternative version
   - System confirms: "Cover letter saved successfully"

Expected Results:

- Personalized cover letter generated in under 15 seconds
- Content aligns user's skills with job requirements
- Professional tone and structure maintained
- Draft stored in MongoDB `letters` collection
- User can iterate on multiple versions
- Significant time saved compared to manual writing

Screenshot Description:

- Figure 10: Cover Letter tab showing generated draft in editable text area
- Figure 11: Action buttons (Save, Copy, Download, Regenerate) below the draft

E. Use Case 5 — Notification System for New Matches

Requirement Addressed: Requirement 8 (Notification System)

Objective: User receives real-time notifications when new jobs matching their criteria are found.

Step-by-Step Description:

1) **Configure Notification Preferences**
   - User navigates to Settings page
   - User toggles ON "Email Notifications"
   - User toggles ON "Browser Push Notifications"
   - User selects notification frequency: "Immediately for new matches"
   - User enables "Weekly Digest" for summary of all activity
   - User clicks "Save Settings"
2) **Automated Job Discovery**
   - n8n workflow runs scheduled ingestion (every 3 hours)
   - New job posting found: "Python Developer Intern - Remote"
   - Filtering logic determines match score: 92%
   - Job exceeds user's relevance threshold (70%)
3) **Notification Trigger**
   - n8n notification node activates
   - System composes notification message
   - Notification dispatched through multiple channels:
     - Email sent to jean.dupont@example.com
     - Browser push notification if user granted permissions
     - In-app notification badge on dashboard
4) **Receive Email Notification**
   - User receives email with subject: "New Job Match: Python Developer Intern"
   - Email contains:
     - Job title and company
     - Match score and badge
     - Brief excerpt of AI summary
     - "View Full Details" button linking to Evidi dashboard
5) **Review in Dashboard**
   - User clicks email link or opens Evidi app directly
   - Notification bell icon shows badge: "1 new"
   - User clicks bell to open notifications panel
   - Panel displays recent notification with timestamp
   - User clicks notification to open Job Detail Modal
   - User reviews job and decides to apply

Expected Results:

- User notified within minutes of new job discovery
- Multi-channel delivery ensures message is received
- Concise notification format enables quick decision-making
- Notification logged in MongoDB `notifications` collection
- User engagement increased through timely alerts

Screenshot Description:

- Figure 12: Settings page showing notification toggle options
- Figure 13: Email notification with job match information
- Figure 14: Dashboard notification bell with badge and dropdown panel

VIII. Discussion

A. Project Challenges and Learning Experiences

Throughout the development of the Evidi Job Response Assistant, our team encountered various technical and non-technical challenges that significantly shaped the final product and our understanding of modern software engineering practices.

1) Technical Challenges: **Integration Complexity:** One of the most significant technical hurdles was establishing reliable communication between n8n workflows, the FastAPI backend, and the MongoDB database. Initially, we struggled with webhook authentication and data format inconsistencies. Jobs retrieved from different sources (RSS feeds, APIs, emails) had vastly different structures, requiring extensive normalization logic. We solved this by implementing a strict JSON schema validation layer in our backend and creating custom transformation nodes in n8n. This experience taught us the importance of data contracts and API versioning in distributed systems.

AI Prompt Engineering: Achieving consistent, high-quality outputs from the OpenAI API required extensive experimentation with prompt templates. Early iterations produced summaries that were either too verbose or missed critical information like salary ranges or required qualifications. We developed a systematic approach to prompt testing, creating a evaluation dataset of 50 sample job descriptions with manually verified "ideal" summaries. This allowed us to iteratively refine our prompts until achieving a satisfaction rate above 85% based on team review.

Asynchronous Operations: Handling concurrent AI processing requests while maintaining responsive API performance proved challenging. When multiple jobs were ingested simultaneously, OpenAI API rate limits were quickly exceeded, causing workflow failures. We implemented a queuing system with exponential backoff retry logic and added parallel processing limits to n8n workflows. This taught us valuable lessons about designing for scalability and graceful degradation.

2) Deployment and DevOps: The deployment phase revealed gaps in our initial infrastructure planning. Vercel's serverless architecture required adapting our FastAPI application to handle cold starts efficiently. We had to restructure our database connection logic to use connection pooling rather than maintaining persistent connections. Railway's resource constraints for n8n workflows forced us to optimize workflow execution times and reduce unnecessary data transformations.

Environment management across development, staging, and production environments initially caused configuration issues. We resolved this by adopting a strict .env file convention and implementing GitHub Actions for automated testing and deployment. Setting up proper CI/CD pipelines taught us the value of infrastructure as code and automated quality checks.

3) Team Collaboration Challenges: **Communication and Coordination:** Working across different time zones (Paris and Seoul) required establishing clear communication protocols. We adopted daily asynchronous updates via Slack and weekly video synchronization meetings. Initially, task dependencies were not clearly defined, leading to blocked work and duplicated efforts. We resolved this by implementing a Kanban board on GitHub Projects with explicit task dependencies and acceptance criteria.

Code Review Processes: Early in the project, we merged code directly to main without thorough review, resulting in several integration bugs. We established a pull request review policy requiring at least one approval before merging. This slowed initial development but dramatically improved code quality and knowledge sharing across the team.

Role Flexibility: While we assigned specific roles (frontend developer, backend developer, development manager), the reality required greater flexibility. Team members needed to work across the full stack to unblock dependencies. This challenged us to improve our documentation and code clarity so that anyone could contribute to any component. We learned that rigid role separation can be counterproductive in small, agile teams.

4) API Cost Management: Managing OpenAI API costs became a concern as usage scaled during testing. We implemented request caching to avoid re-generating summaries for identical job descriptions and added usage monitoring dashboards. We also discovered that GPT-4 could be replaced with GPT-4o-mini for certain tasks (like keyword extraction) without significant quality loss, reducing costs by approximately 60% for those operations.

5) User Experience Iterations: Initial usability testing with classmates revealed that our interface was too technical and overwhelming. Users were confused by too many configuration options and unclear feedback messages. We conducted three rounds of UX improvements, simplifying the onboarding flow, adding contextual help tooltips, and improving error messages. This taught us that technical correctness must be balanced with user-centered design principles.

6) Security Considerations: Implementing proper authentication and authorization required careful attention to security best practices. We initially stored JWT tokens in localStorage, which exposed them to XSS attacks. After security review, we migrated to HTTP-only cookies with CSRF protection. We also implemented rate limiting on authentication endpoints to prevent brute-force attacks and added input validation to prevent SQL/NoSQL injection attempts.

7) What Worked Well: Despite challenges, several architectural decisions proved highly effective. The choice of MongoDB's flexible schema allowed us to adapt quickly to changing requirements without complex migrations. FastAPI's automatic OpenAPI documentation accelerated frontend development by providing clear API contracts. n8n's visual workflow editor enabled rapid prototyping of automation logic and made the system accessible to non-technical team members for testing.

The modular architecture we adopted from the beginning paid significant dividends. When we needed to replace an RSS feed source with a different API, the changes were isolated to a single n8n workflow without affecting the backend or frontend. This confirmed the value of separation of concerns and loose coupling in system design.

8) Areas for Improvement: If we were to restart this project, we would invest more time in comprehensive testing from the beginning. Our test coverage remained below 60% for most of development, leading to regression bugs that could have been caught earlier. We would also establish clearer API versioning strategies from the start rather than retroactively adding them.

Documentation was consistently deprioritized under time pressure, making onboarding new team members slower than necessary. In future projects, we would treat documentation as a first-class requirement with the same priority as code.

9) Skills Developed: This project significantly expanded our technical capabilities across multiple domains. We gained practical experience with cloud-native architectures, microservices design patterns, and serverless deployment models. The integration of AI services taught us prompt engineering techniques and how to design human-in-the-loop systems that balance automation with user control.

Beyond technical skills, we developed important soft skills in remote collaboration, asynchronous communication, and cross-cultural teamwork. Managing this project taught valuable lessons about realistic scope estimation, risk management, and the importance of iterative development with frequent user feedback.

10) Impact and Future Directions: The Evidi platform successfully demonstrates that AI-powered automation can meaningfully reduce the cognitive burden of job searching while maintaining personalization and user control. Early testing with fellow students showed that users saved approximately 3-4 hours per week on job search activities while actually applying to more relevant positions.

For future development, we envision several enhancements: integration with major job platforms through official APIs, collaborative features allowing users to share and rate job opportunities, analytics dashboards showing application success rates, and machine learning models that learn from user preferences over time to improve matching accuracy.

We also see potential for extending the platform beyond job search to other domains requiring information aggregation and AI-assisted decision-making, such as scholarship applications, graduate school research, or grant opportunities.

11) Conclusion: The Evidi project provided invaluable hands-on experience in building a complete, production-ready web application with modern technologies and AI integration. The challenges we faced—and overcame—taught us far more than theoretical coursework could have. We emerged with not only a functional product but also a deeper appreciation for software engineering principles, the complexities of distributed systems, and the critical importance of user-centered design.

The most significant insight from this experience is that successful software development is not purely technical. It requires balancing technical excellence with user needs, team dynamics, resource constraints, and evolving requirements. The ability to adapt, communicate clearly, and iterate based on feedback proved as important as coding skills.

This project represents a foundation we are proud of and a learning experience that will inform our approach to software engineering throughout our careers. We are grateful for the opportunity to work on a real-world problem that affected us personally as job seekers, and we believe Evidi demonstrates the transformative potential of thoughtfully applied artificial intelligence in everyday life.