Enterprise Financial Analyzer
Technical Documentation & Solution Architecture

Date: September 27, 2025
Author: Sanjay Anbazhagan

Table of Contents

-----1. Introduction1.1. Project Mission

The mission was to address the "Advanced Debug & Architecture Challenge" by taking an intentionally flawed, bug-riddled prototype and re-architecting it into a secure, scalable, and fully functional enterprise-grade financial analysis system. This required thinking not just as a developer, but as a solution architect to solve deep-rooted issues across the full stack.1.2. Executive Summary

The Enterprise Financial Analyzer is a full-stack web application that transforms unstructured financial documents (PDFs) into structured, actionable insights. It leverages a sophisticated multi-agent AI system, built with CrewAI and powered by Google Gemini, to perform a multi-faceted analysis covering financial

metrics, market research, investment potential, and risk assessment. The system is built on a modern, decoupled architecture with a React frontend and a high-performance FastAPI backend, ensuring both a seamless user experience and robust, scalable performance.1.3. Core System Features

- **Secure User Authentication:** Enterprise-grade JWT authentication with modern argon2 password hashing.
- **Full-Stack Architecture:** A clean, decoupled system with a React & Tailwind CSS frontend and a high-performance FastAPI backend.
- **PDF Document Processing:** Seamlessly upload and manage financial reports through a sleek, responsive user interface.
- **Sophisticated AI Crew:** A multi-agent system powered by Google Gemini & CrewAI provides deep analysis.
- **Asynchronous & Real-Time:** AI analysis runs as a background task, with the frontend polling for live status updates.
- **Persistent History:** All analysis requests and results are stored securely in a MongoDB database.

-----2. Solution Architecture2.1. Architectural Paradigm: Three-Tier System

The application is built on a modern **three-tier architecture**, ensuring a clear separation of concerns between the user interface (presentation), application logic, and data storage. This paradigm is the cornerstone of scalable and maintainable enterprise systems.

- **Presentation Tier:** A React-based Single-Page Application (SPA).
- **Logic Tier:** A high-performance, asynchronous API built with FastAPI.
- **Data Tier:** A MongoDB database for flexible and scalable data storage.

2.2. High-Level System Diagram

[Insert High-Level System Diagram Here]2.3. Architectural Decisions & Rationale

- **Decoupled Frontend/Backend:** This was the most critical architectural decision. It allows the frontend and backend to be developed, scaled, and deployed independently. It also enables the future development of alternative clients (e.g., a mobile app) that can consume the same backend API.
- **Asynchronous Backend (FastAPI):** Financial analysis is a long-running task. An asynchronous framework like FastAPI is essential to handle these tasks in the background without blocking the server or timing out user requests, ensuring the application remains responsive.
- **NoSQL Database (MongoDB):** The output of an AI analysis is semi-structured and can vary in length and complexity. A NoSQL database like MongoDB is perfectly suited to store this flexible, document-based data, whereas a traditional SQL database would require a rigid and complex schema.
- **Multi-Agent AI System (CrewAI):** Instead of using a single, monolithic AI prompt, a multi-agent approach was chosen to break down the complex task of financial analysis into specialized roles.

This mimics a real-world financial team and produces a more comprehensive and higher-quality output by assigning specific goals to each agent (e.g., one for data extraction, one for market research).

-----3. System Components Deep Dive3.1. Frontend Tier (React SPA)

- **Framework:** React for its component-based UI development.
- **Styling:** Tailwind CSS for a modern, utility-first design system.
- **API Communication:** axios is configured with interceptors to automatically manage JWT tokens for secure API calls and to globally handle session expirations (401 Unauthorized errors), providing a robust user experience.

3.2. Logic Tier (FastAPI Backend)

- **Framework:** FastAPI for its high-speed, asynchronous capabilities and automatic API documentation.
- **Security:** JWT tokens are used to protect endpoints. Passwords are never stored directly; they are hashed with argon2 via passlib.
- **AI Engine:** CrewAI orchestrates the workflow of multiple AI agents powered by Google's Gemini models.
- **Asynchronous Tasks:** File analysis is executed as a background task, allowing the API to immediately respond to the user and prevent timeouts.

3.3. Data Tier (MongoDB)

- **Driver:** The motor library provides the essential asynchronous interface to MongoDB, integrating seamlessly with FastAPI.
- **Collections:**
    - users: Stores user credentials and profile information.
    - analysis_requests: Tracks every analysis task, its status, and the final report.

-----4. Request Lifecycle: A User Story4.1. User Authentication Flow

1. A user registers on the React frontend.
2. The request hits the FastAPI /register endpoint.
3. The password is hashed, and the user is saved to MongoDB.
4. The frontend then automatically calls the /token endpoint.
5. FastAPI verifies the credentials and returns a JWT access token.
6. The token is stored in the browser's localStorage and is attached to all future requests.

4.2. Document Analysis Flow

1. An authenticated user uploads a PDF.
2. The React frontend sends the file to the protected /analysis/upload endpoint.

3. FastAPI validates the JWT, saves the file, creates a "pending" record in the analysis_requests collection in MongoDB, and immediately returns a request ID.
4. Simultaneously, FastAPI triggers the CrewAI analysis as a background task.
5. The React frontend begins polling the /analysis/status/{request_id} endpoint every few seconds.
6. In the background, the AI crew analyzes the document, making calls to the Google Gemini and Serper APIs.
7. Once complete, the AI crew's final report is saved to the corresponding record in MongoDB, and the status is updated to "completed".
8. The next time the frontend polls, it receives the "completed" status and the full report, which is then rendered to the user.

-----5. The Debugging & Refactoring Journey5.1. Initial State of the Prototype

The initial codebase was a non-functional prototype intentionally riddled with a wide spectrum of bugs, including:

- **Deterministic Bugs:** Incorrect imports, syntax errors, and typos.
- **Logical Flaws:** The AI prompts were nonsensical and the user authentication flow was incomplete.
- **Architectural Problems:** A monolithic structure with no separation of concerns.
- **Production Gaps:** No error handling, no environment configuration, and critical security vulnerabilities (e.g., no password hashing).

5.2. Key Challenges & Resolutions

- **Dependency Hell:** The most significant challenge was resolving a cascade of pip dependency conflicts between crewai, langchain, and pydantic. This was solved by methodically creating a clean virtual environment and pinning each package to a specific, compatible version.
- **API Authentication & Configuration:** We fixed numerous issues related to the Google Gemini API, including incorrect model names, invalid API keys, and—most critically—the need to enable the Vertex AI API in the Google Cloud project.
- **Frontend/Backend Communication:** We resolved CORS (Cross-Origin Resource Sharing) issues and fixed bugs in the frontend's API calls, including implementing a global error handler to manage expired sessions gracefully.
- **AI Logic:** The original, ineffective AI prompts were completely replaced with a professional, multi-agent crew designed for high-quality financial analysis.

-----6. Setup & Deployment Guide6.1. Prerequisites

- Python 3.11.x
- Node.js v18.x or later
- MongoDB

6.2. Environment Configuration

1. Navigate to the backend directory.
2. Rename .env.example to .env.
3. Fill in the values for GEMINI_API_KEY, SERPER_API_KEY, MONGO_URI, and SECRET_KEY.

6.3. Running the Application Locally

(Requires two separate terminals)

Terminal 1: Start the Backend

```
cd backend
python3 -m venv venv
source venv/bin/activate
pip install -r requirements.txt
python -m uvicorn main:app --reload
```

Terminal 2: Start the Frontend

```
cd frontend
npm install
npm start
```

The application will be available at http://localhost:3000.