Executive Summary

This project is a web application designed to assist with anomaly detection in financial data. Users can select the type of invoice and date range they want to analyze, and the app will automatically identify and display anomalies. Each anomaly can be explored in detail, providing insights that support faster and more accurate decision-making. The tool is built to streamline manual review processes and enable data-driven anomaly investigation.

Idea

The problem addressed in this project is the detection and analysis of anomalies in financial invoices. Specifically, we focus on identifying unusual patterns in the total invoice amounts aggregated by day invoice type.

The first step is to detect days that present anomalous behavior in terms of their total invoice amounts. Once these days are identified, the second step is to understand the *reason* behind each anomaly. To do this, the system leverages a **financial analysis agent** that examines all invoices from the anomalous day and generates a detailed report explaining the irregularities.

By the end of this process, users have a clear view of which days are anomalous and the likely causes—empowering them to take informed action based on the insights provided.

Project Log

1. **Exploring Belvo's Documentation**
   I started by thoroughly reading Belvo's documentation to understand the available products and how to extract data. This step took approximately 30% of the total project time due to several issues encountered with the test data. Only one test user from Brazil could be successfully authenticated. Additionally, understanding how to integrate the widget and implement the webhook-based approach required a significant time investment.

2. **Exploratory Data Analysis (EDA)**
   I evaluated three Belvo products: *Banking Brazil (OFDA)*, *Fiscal Mexico*, and *Employments Mexico*. The analysis for each dataset can be found in the notebooks within the repository.
   After comparing data quality, I decided to work with **Fiscal Mexico**, as it offered the most usable data:

   - The Brazilian banking user had over 1,200 transactions, but all shared the same amount and distribution.

   - The employment data from Mexico included only one user with a few records.

    - Fiscal Mexico, on the other hand, included multiple users, and I found one with over **160 invoices** of varying types and amounts.

3. **Solution Design Thinking**
   At this point, I began defining what kind of feature or solution to build. One option was to create a traditional **RAG (Retrieval-Augmented Generation)** interface where users could query their data conversationally. However, I opted instead to focus on **anomaly detection**, as it offered higher practical value for financial analysis use cases.

4. **Development Phase**
   I envisioned a simple web app where users could explore their data, see detected anomalies, and inspect each one in detail. However, by this stage, I had already used most of the challenge's proposed time limit (8–10 hours).
   To deliver a useful and coherent demo, I focused on building a core flow that demonstrated the product's value, leaving additional improvements for future work.
   I used **Windsurf**, an AI-powered IDE, as the main development tool. Approximately 70% of the code in the repository was AI-assisted (generated with Windsurf), which helped accelerate development. I reviewed and tested all generated code to ensure reliability.

5. **Productization and Documentation**
   The final step involved setting up productization features such as CI/CD pipelines, testing, linters, and deployment workflows. I also created a comprehensive `README.md` to document the repository and this technical report.
   I continued to use Windsurf to assist in creating tests, CI/CD configuration, and code quality tools. I chose **Heroku** as the deployment platform due to its simplicity and low cost.

Technical Decisions

Language and Frameworks

The project was developed in **Python**, primarily due to its rich ecosystem for data and AI development. For the AI components, I used **LangChain** and **LangGraph**—two powerful frameworks that significantly simplify the development of LLM-powered solutions.

For the frontend, I used **Streamlit**, a Python-based framework designed specifically for data-centric applications. It allows for the rapid creation of intuitive and interactive user interfaces with minimal code. In my opinion, Streamlit is ideal for demo apps or MVPs where demonstrating value is more important than building a fully custom UI.

Cloud Provider

I deployed the application on **Heroku**, a platform I've used in the past. For small-scale projects, Heroku offers a great balance of simplicity and functionality. Unlike more complex providers like AWS or GCP, Heroku does not require extensive configuration (e.g., VPCs, IAM roles), making it a fast and cost-effective choice for prototypes or side projects.

LLM Selection

Two different language models were used for distinct parts of the application:

- For the **Anomaly Detection** module, I used **OpenAI GPT-4.1-nano**. This model offers a solid trade-off between performance, speed, and cost, making it ideal for real-time detection in user-facing workflows. Its lightweight nature ensures fast response times without sacrificing too much accuracy.

- For the **Anomaly Analysis Agent**, I used **GPT-4o-mini**. This model was selected for its **greater reasoning and analysis capabilities**, which are essential when interpreting financial data and generating meaningful explanations for anomalies. Although slightly heavier, it provides significantly better depth in natural language reasoning—making it more suitable for detailed analysis tasks.

Data Handling Strategy

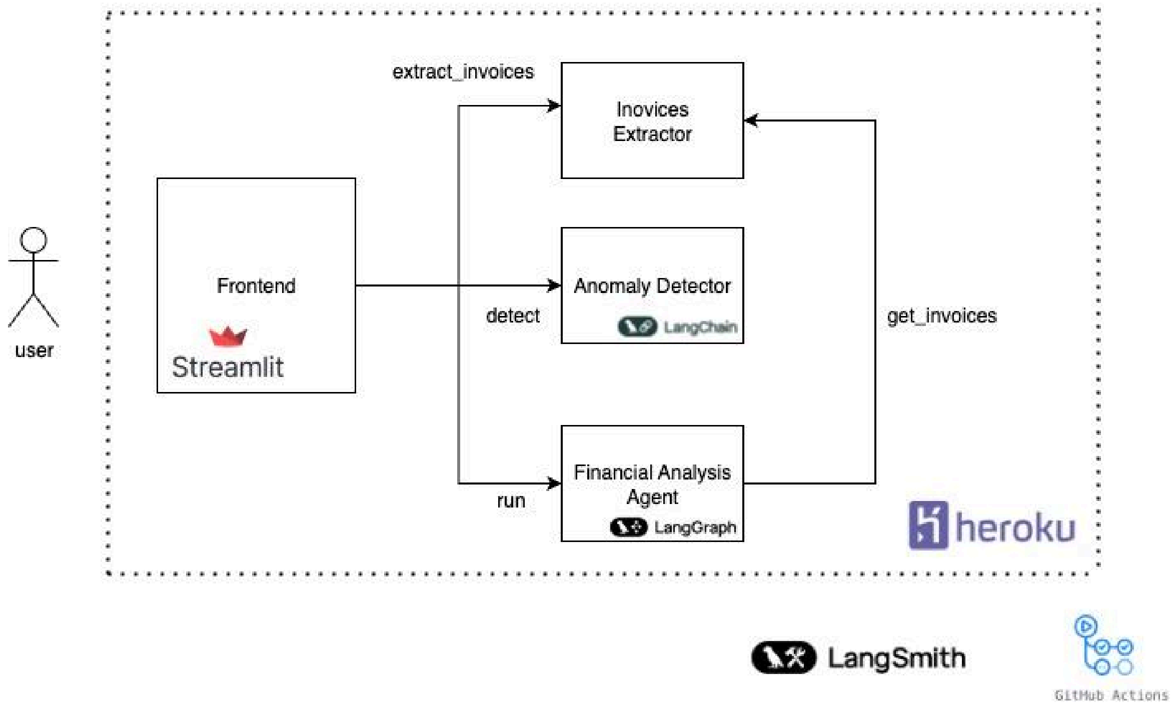
For this demo, the invoice data is stored in a local **CSV file**. Given the scope of the challenge and the limited volume of data (~160 records), setting up and configuring a full database felt unnecessary. Making API calls to retrieve the same small dataset repeatedly would have also introduced unnecessary complexity and latency.

The focus was on **demonstrating the solution's value**, not optimizing for production-scale data architecture. In a real-world scenario, the appropriate storage strategy would depend on the expected data volume. Options could include a traditional SQL database or more scalable solutions like **Snowflake** or **BigQuery**, depending on performance and integration needs.

Testing Strategy

Unit tests were implemented for the **Invoices Extractor** component to ensure that data grouping and retrieval logic works as expected. However, no tests were written for the **LLM-based components** (e.g., the Anomaly Detector and the Financial Analysis Agent). Since these modules operate through prompt-based input/output interactions with language models, traditional unit testing was less applicable. The behavior of LLMs is non-deterministic and context-dependent, so validating outputs would require predefined expectations for natural language responses—something that's better handled through **manual evaluation** or **prompt tracing tools** like LangSmith.

System Architecture



The system is composed of four main components, as illustrated in the diagram:

Frontend (Streamlit)

The user interface is built using **Streamlit**. This single-page app provides controls for selecting invoice types and date ranges, and for viewing anomaly reports. The Streamlit app also encapsulates the logic for interacting with the other backend components.

Invoices Extractor

This component acts as a repository and access layer for the invoice data. It provides two key methods:

- `extract_invoices`: Returns invoices grouped by date and invoice type. This grouped data is used by the anomaly detector to identify irregularities.
- `get_invoices`: Returns the raw invoices for a specific date and invoice type. This method is used by the Financial Analysis Agent when further investigation is needed.

Anomaly Detector (LangChain)

This component is implemented using **LangChain** and is responsible for detecting anomalies in the invoice data. It processes data **grouped by day and invoice category**, a deliberate design choice to reduce the overall input size. Grouping at the day level allows for more efficient scaling, as analyzing each invoice individually would significantly increase the number of tokens sent to the language model—impacting both performance and cost. This approach balances granularity and efficiency, enabling effective anomaly detection without compromising scalability.

Financial Analysis Agent (LangGraph)

Built with **LangGraph**, this agent performs in-depth analysis of all invoices from an anomalous day. It uses the `get_invoices` tool from the Invoices Extractor and generates a report describing potential causes of the anomaly. This step adds context and explanation to the raw anomaly detection results.

Deployment and Automation (Heroku + GitHub Actions)

The application is deployed on **Heroku** for simplicity and speed. Additionally, the project uses **GitHub Actions** to automate key parts of the development workflow:

- Run **linters** to ensure code quality
- Execute **unit tests** for stability
- **Deploy automatically** to Heroku upon updates to the main branch

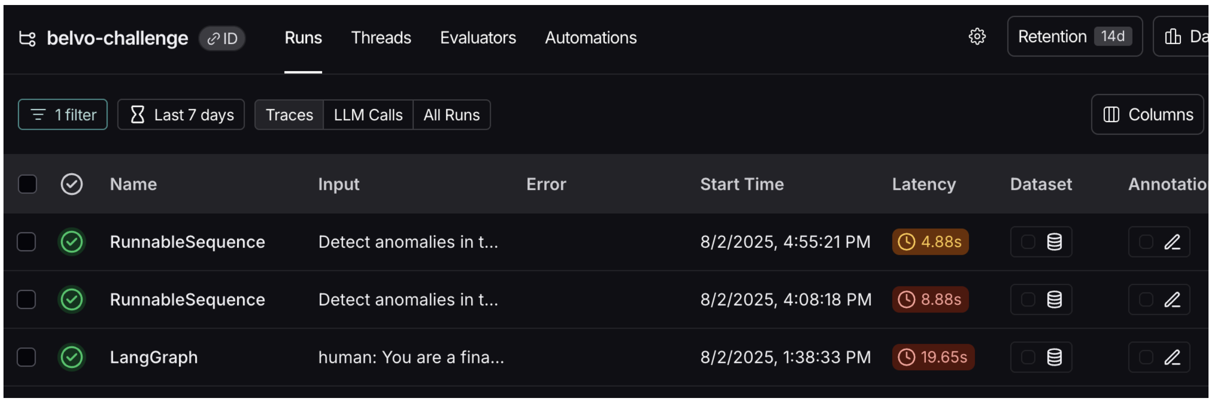
This CI/CD setup ensures that the application remains consistent, tested, and production-ready throughout development.

Monitoring with LangSmith

The app integrates with **LangSmith** to monitor the performance of LLM-based components (LangChain and LangGraph). This enables:

- Real-time tracing of requests and agent behavior
- Visibility into model inputs/outputs and latency
- Debugging and optimization of prompts and tool interactions

Here is an example of the LangSmith dashboard used to trace and evaluate an agent run:



Name	Input	Error	Start Time	Latency	Dataset	Annotatio
RunnableSequence	Detect anomalies in t...		8/2/2025, 4:55:21 PM	4.88s		
RunnableSequence	Detect anomalies in t...		8/2/2025, 4:08:18 PM	8.88s		
LangGraph	human: You are a fina...		8/2/2025, 1:38:33 PM	19.65s		

Future Opportunities

Anomaly Detection Improvements

In a real-world scenario, the anomaly detection logic should be iteratively refined. For this MVP, I defined an anomaly as:

> “A significant increase in the total invoice amount compared to the previous period with available data.”

However, this definition should be validated with both the **customer** and **product teams** to align with business needs. For instance, there’s a difference between detecting anomalies in **volume trends** (e.g., sudden spikes in total amounts) versus detecting **data inconsistencies** (e.g., incorrect names, dates, or fields).

For time-series anomaly detection at scale, specialized models (e.g., statistical models or ML-based forecasters) may offer better results. The current solution is effective for **low to moderate volumes**, but cyclical or seasonal behaviors would benefit from more robust methods.

Data Persistence Strategy

In this prototype, data is stored in a CSV file for simplicity. In a production environment, selecting an appropriate database would depend on data volume and usage patterns. For large-scale use, engines such as **PostgreSQL**, **BigQuery**, or **Snowflake** could provide better performance, scalability, and integration options.

Metrics and KPIs

Defining meaningful metrics is essential to evaluate the product’s value and impact. Suggested metrics include:

- Number of anomalies detected over time
- Total financial value impacted by anomalies

- User engagement with anomaly reports

Automated Execution and Integration

Currently, the solution is designed as a frontend tool where users manually explore data and anomalies. In future iterations, it would make sense to run the detection process **automatically**, for example once a day, and trigger alerts or actions based on the results. This could be valuable for:

- **Analytics teams** that review anomalies regularly
- **Business workflows** that need proactive alerts
- Integrations with messaging or task systems (e.g., Slack, email, or ticketing)

An intelligent agent could also automatically summarize findings and send tailored reports to relevant users—turning this into a fully autonomous monitoring system.