Implementing Advanced Design Patterns

Application or use cases of various design patterns

Assignment #3

Professor:

Maxim Glaida

Students:

Lucía Digón (se24m504@technikum-wien.at)
Saúl Ariel Castañeda (se24m027@technikum-wien.at)
Axel Preiti Tasat (se24m503@technikum-wien.at)

Due date:

Sunday 20th April 2025

Link:

https://github.com/saulex16/restaurant-management

Application idea

The application idea of this assignment consists of an expansion of the previous one (assignment #1). The idea consists of a restaurant management API that helps order handling, inventory tracking and kitchen operations. The system allows waiters to see the menu, check dish availability and place orders for different tables. The kitchen staff can monitor incoming orders, to help a more efficient preparation. Additionally, the API includes a warehouse management component that tracks ingredients and sends notifications when stock is added or runs out, making it easier to manage dish availability.

The API is designed to be adaptable, as restaurants can integrate it with a UI to have an efficient digital ordering system but it can also be used as the backend for a restaurant simulation video game, for example.

In this assignment, the idea is expanded to include an AI-powered service, including RAGs and LLMs, that allows you to upload recipes from a pdf file, and then have the possibility to ask, as a restaurant manager, which new recipes you could add to the menu taking into account the ingredients you currently have in stock. The idea would be to have a chat that allows you to ask about the files uploaded, such as the one we can see in Figure 1.

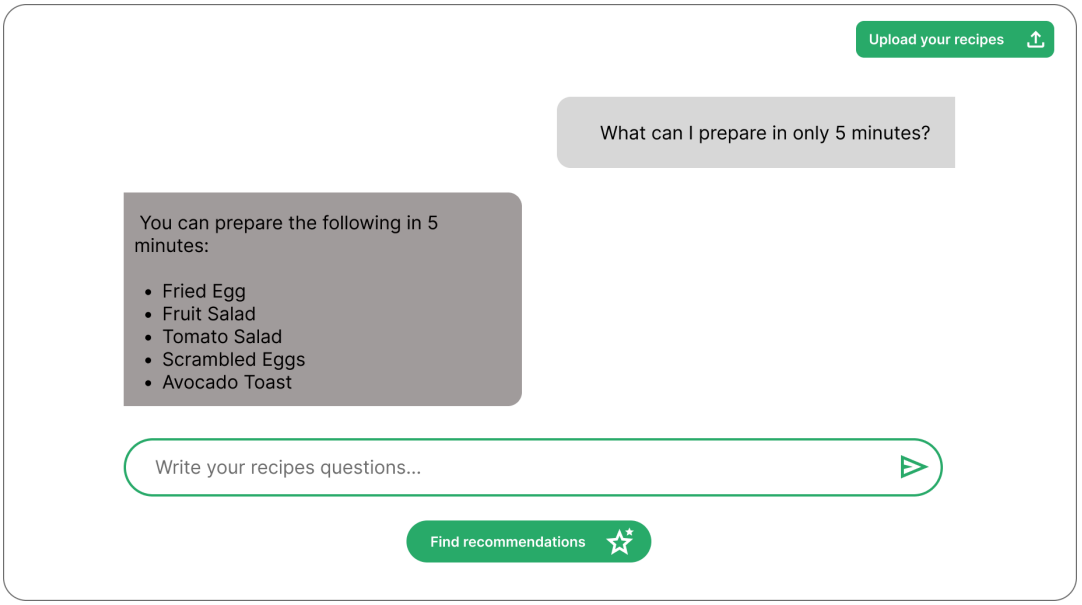

Fig 1. Visualization of the AI expansion

Limitations and possible improvements

The system has some limitations and opportunities for improvement. Firstly, the prompt can be refined, giving more detailed and clear requirements. In other words, a thorough process of prompt engineering should be carried out. Additionally, the tokenization of the chunks in the vectorial DB could be enhanced to achieve more accurate responses.

Secondly, the applied resilience design patterns can be sophisticated in order to consider more complex cases in the communication between both APIs. Furthermore, to reach a higher software quality, it would be recommended to incorporate more patterns to both services.

Additionally, the RAG implementation has some limitations that could be improved (see RAG pattern section below). Even after refining the prompt used in the recipe recommendation command, the answers can still be inaccurate. While testing with a well-structured PDF of recipes, we manually reviewed the LLM's recommendations and noticed that, in several cases, it suggested recipes that included some of the available ingredients, but also required others that weren't in stock. This issue appeared more frequently than fully accurate recommendations.

Throughout the development process, we worked with a well-structured PDF. However, less structured documents, especially those missing key fields like titles or ingredient lists, could cause the system to fail. A potential future improvement would be to handle these edge cases by curating the extracted information more carefully to ensure a certain level of quality in the uploaded documents.

Design patterns implemented

In this section, we will explain the design patterns that were implemented for this project, and how they helped to decouple and improve the flexibility of the application business logic.

Retrieval-Augmented Generation (RAG)

The **RAG** pattern was implemented to process uploaded recipes and check if the restaurant has the necessary ingredients to prepare any of them. For this, we decided to use Langchain as the LLM orchestration tool and Gemini as the LLM for both embedding and generative tasks within the RAG pipeline.

As mentioned before, RAG enhances the restaurant management system's ability to handle question-and-answer tasks related to uploaded PDF recipe documents. To support this, the pattern requires a preprocessing step where the PDF is split based on a fixed "chunk_size" with a "chunk_overlap", which helps maintain context across chunks and avoids breaking coherent paragraphs. Initially, we implemented the upload step up to this point, storing chunks with embedded text in a vector store, but without much useful information.

Later on, when developing the recipe recommendation feature, we decided to improve the upload process by including the recipe title and ingredients as metadata. This change added more value to the stored documents. To extract this structured metadata, we added an extra LLM call that prompts the model to return a JSON object with the title, ingredients, and content of each recipe. This approach leads to higher-quality entries than storing plain text alone.

For the retrieval and generation steps, we implemented a question-answering chain that retrieves the top-K most relevant documents based on the user's query. These documents, along with the query, are then passed to the LLM to generate a response. The chain, provided by Langchain, includes its own prompt to ensure the model answers strictly based on the retrieved context. If the context isn't sufficient, the model responds accordingly without relying on external or foundational knowledge.

Additionally, to leverage the LLM's capabilities further, we implemented a command to suggest recipes based on available ingredients (referred to as "Stock" in our domain). This process involves retrieving a list of current ingredients, using that list to filter relevant recipes

from the vectorstore, and then extracting the titles and ingredients from the associated metadata. Finally, an LLM is prompted to determine which of those recipes can be prepared with the given stock.

Health Check

The **Health Check** pattern is used to verify the connection between the Java API and the AI service. We added a new endpoint in the Java application at /ping. When this endpoint is called and responds with "pong", it confirms that the connection is working properly.

Circuit Breaker

The **Circuit Breaker** pattern builds on top of the health check to monitor the connection between the Python and Java services. When the application starts, the system assumes a "semi-open" state, meaning it's not sure yet if Java is reachable. It then performs a health check:

- If the health check succeeds, the circuit is set to closed, meaning everything is working fine and Java is available.
- If it fails, the circuit goes to open, which indicates the Java service is down or unreachable. In this case, further health checks are scheduled periodically (with exponential backoff) to keep checking if Java comes back online.

Whenever a request needs to go to the Java API:

- If the circuit is open, the request is immediately blocked and an error is returned
- If it's semi-open or closed, the request goes through:
    - If the request fails (e.g., times out), the circuit switches back to open, and health checks start again.
    - If the request is successful, the system considers Java to be healthy and keeps the circuit closed.

This pattern helps avoid flooding an already failing service and gives it time to recover, while keeping the rest of the system responsive and stable.

Reflection

In this third assignment, we saw how advances in AI can significantly expand the scope and complexity of a system without necessarily increasing development time. By integrating tools like RAG, we were able to add intelligent features, like recipe suggestions based on stock, in a relatively short time. Compared to our first project, this time many patterns (like Health Check and Circuit Breaker) emerged naturally from the challenges of integrating two different services.

While AI offers clear advantages, saving time and replacing complex logic with simpler, smarter solutions, the responses from language models can be unpredictable. Because of this, it's important to validate and refine the outputs to ensure reliability. More

testing tools are needed to guarantee that the results are of a quality that is acceptable for the application.

Overall, it was interesting to see how design patterns and AI tools can complement each other, making it possible to build more powerful and responsive systems more efficiently.