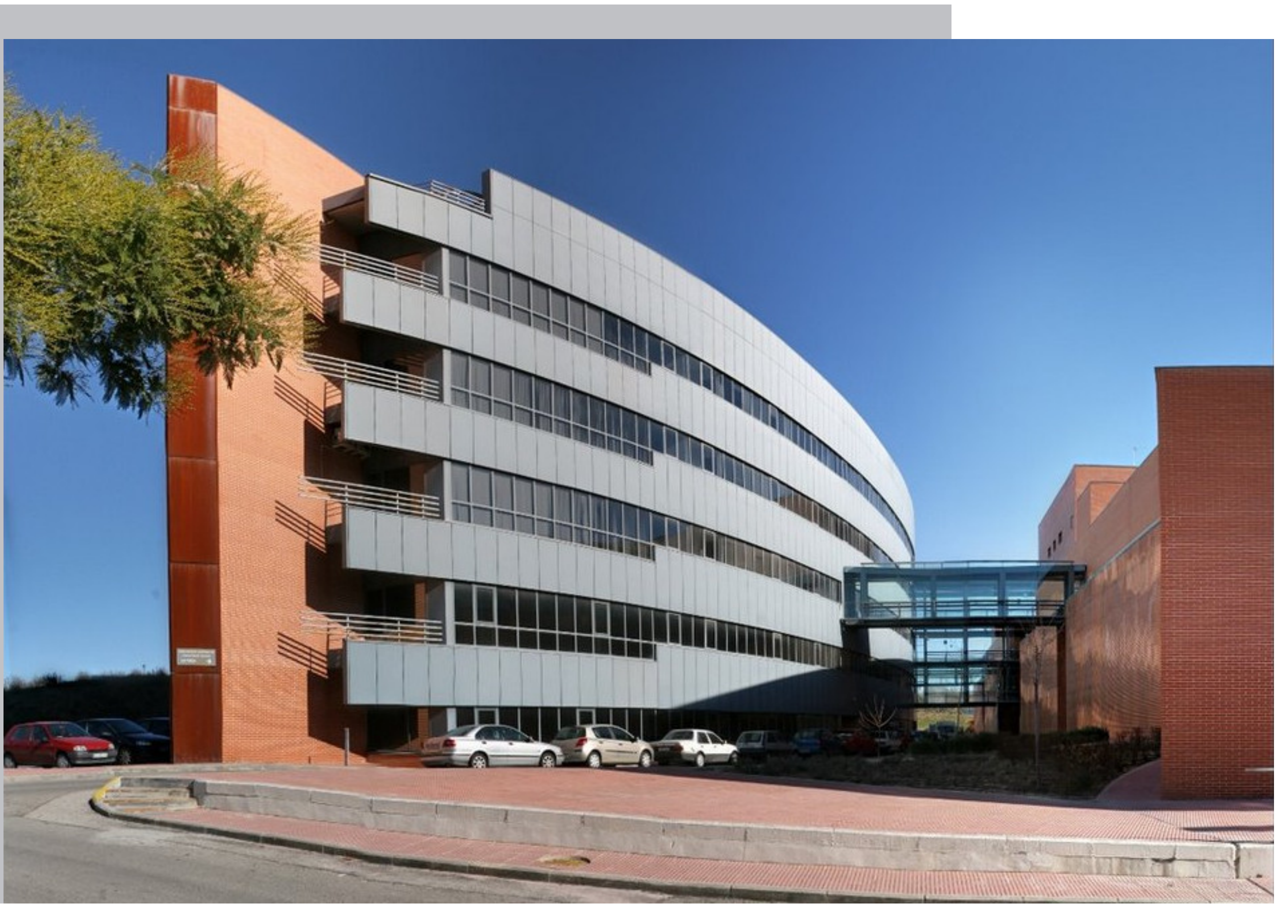
UAM
Universidad Autónoma
de Madrid

Escuela Politécnica Superior

24
25

Master thesis

A Hybrid Conversational Recommender System by integrating LLMs



Javier Wang Zhou

Escuela Politécnica Superior
Universidad Autónoma de Madrid
C\Francisco Tomás y Valiente nº 11

Campus Internacional excelencia UAM CSIC+

www.uam.es

UNIVERSIDAD AUTÓNOMA DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

Master in Computer Science and Engineering

MASTER THESIS

A Hybrid Conversational Recommender System by integrating LLMs

Development of a Scalable Platform for Conversational Recommendations

Author: Javier Wang Zhou
Advisor: Alejandro Bellogín Kouki

julio 2025

© July 13, 2025 by UNIVERSIDAD AUTÓNOMA DE MADRID
Francisco Tomás y Valiente, n^o 1
Madrid, 28049
Spain

Javier Wang Zhou
A Hybrid Conversational Recommender System by integrating LLMs

Javier Wang Zhou
C\ Francisco Tomás y Valiente Nº 11

PRINTED IN SPAIN

To my family & friends

Simplicity is the ultimate sophistication.

Leonardo da Vinci

Agradecimientos

Quisiera agradecer en primer lugar a mi tutor Alejandro Bellogín, por su apoyo y orientación durante este tiempo. Su experiencia y dedicación me han motivado a sacar adelante este trabajo conjunto.

También quiero dar las gracias a todos mis compañeros del Máster, con quienes he compartido estos dos años en los que cada tropiezo me ha permitido aprender y mejorar. Entre cafés en la cafetería de la Escuela y quedadas fuera de ésta, hemos crecido juntos tanto académica como personalmente.

Finalmente, gracias a mi madre por estar siempre ahí, apoyándome en todo momento, y a mi perro Cookie por alegrarme los días que más le he necesitado.

Mi más profundo agradecimiento a todos.

Resumen

La reciente proliferación de los Large Language Models (LLMs) ha impulsado un interés significativo en el desarrollo de sistemas conversacionales avanzados. Sin embargo, la integración de estos modelos en plataformas de Sistemas de Recomendación que sean escalables, robustas y fáciles de usar sigue siendo un desafío complejo de ingeniería. Las soluciones actuales suelen carecer de un marco completo que simplifique el despliegue y asegure la escalabilidad en aplicaciones reales.

Este Trabajo de Fin de Máster trata de abordar este problema, detallando el diseño y la implementación de una plataforma modular y escalable para la creación de agentes conversacionales de recomendación híbrida. El objetivo principal es desarrollar una solución completa *full-stack* que automatice todo el ciclo de vida de un agente conversacional, desde su creación y el procesamiento de datos hasta su despliegue e interacción con el usuario.

La plataforma está diseñada sobre una arquitectura contenerizada con Docker para garantizar modularidad, portabilidad y escalabilidad. El *backend* se construye sobre FastAPI, un *framework* de alto rendimiento que expone una Application Programming Interface (API) RESTful para gestionar agentes, procesar conjuntos de datos y manejar la lógica conversacional. Para el *frontend*, se desarrolló una Progressive Web Application (PWA) *adaptativa* e instalable usando Next.js, que proporciona una interfaz intuitiva para que los usuarios administren y conversen con sus agentes de recomendación. La capa de datos del sistema utiliza Supabase, una base de datos relacional en la nube para almacenar metadatos de los agentes y las conversaciones; y FalkorDB como base de datos de grafos de alto rendimiento para modelar las relaciones entre usuarios e ítems, y para almacenar historiales de chat. Una característica destacable es el *pipeline* de procesamiento de datos automatizado que se encarga de la ingesta, limpieza y estructuración de los conjuntos de datos subidos por el usuario.

El resultado final de este proyecto es una plataforma *end-to-end* completamente funcional que permite a los usuarios crear, gestionar e interactuar de manera fluida con agentes de recomendación conversacionales personalizados. La robusta arquitectura prioriza la eficiencia, la escalabilidad y la reproducibilidad, proporcionando una base sólida para futuros desarrollos e investigaciones en la intersección de la IA conversacional y los sistemas de recomendación.

Palabras clave

Sistema de recomendación conversacional, modelos extensos de lenguaje, desarrollo full-stack, arquitectura escalable, diseño de sistemas, bases de datos de grafos, FastAPI, Next.js, Docker

Abstract

The recent proliferation of Large Language Models (LLMs) has catalyzed significant interest in the development of advanced conversational systems. However, integrating these models into scalable, robust, and user-friendly Recommender System platforms remains a complex engineering challenge. Current solutions often lack an end-to-end framework that simplifies deployment and supports real-world scalability.

This Master's Thesis addresses this problem by detailing the design and implementation of a scalable and modular platform for the creation of hybrid conversational recommender agents. The primary objective is to develop a comprehensive, full-stack solution that automates the entire lifecycle of a conversational agent, from creation and data processing to deployment and user interaction.

The platform is designed using a containerized approach with Docker to ensure modularity, portability, and scalability. The backend is built upon the high-performance FastAPI framework, which exposes a RESTful Application Programming Interface (API) to manage agents, process datasets, and handle conversational logic. For the frontend, a responsive and installable Progressive Web Application (PWA) was developed using Next.js, providing an intuitive interface for users to manage their recommender agents and engage in conversations. The system's data layer leverages Supabase, a cloud-based relational database for storing agent and conversation metadata; and FalkorDB as a high-performance graph database to model the relationships between users and items, and to store chat histories. A key feature is the automated data processing pipeline that handles the ingestion, cleansing, and structuring of user-uploaded datasets.

The final result of this project is a fully functional, end-to-end platform that empowers users to seamlessly create, manage, and interact with personalized conversational recommender agents. The robust architecture prioritizes efficiency, scalability, and reproducibility, providing a solid foundation for future development and research at the intersection of conversational AI and Recommender Systems.

Keywords

Conversational recommender system, large language models, full-stack development, scalable architecture, system design, graph databases, FastAPI, Next.js, Docker

Table of Contents

Lists

List of codes

List of equations

List of figures

List of tables

1

Introduction

The paradigm of human-computer interaction is undergoing a significant transformation, largely driven by the advent of powerful LLMs. These models have unlocked new possibilities for creating dynamic and intuitive conversational experiences. This chapter introduces a project situated at the confluence of this technological wave and the established field of Recommender Systems [1]. It begins by exploring the motivation behind developing a novel, scalable platform for conversational recommendations, born from the need to bridge the gap between the potential of LLMs and the practical engineering challenges of their implementation. Subsequently, the specific objectives guiding this engineering effort are outlined, followed by a description of the structure of this document.

1.1. Motivation

The recent and rapid proliferation of LLMs has sparked a transformative wave across numerous technological domains. These models, exemplified by systems like OpenAI's Generative Pre-trained Transformer (GPT) [2] and Google's Gemini [3], have demonstrated remarkable capabilities in understanding and generating human-like text, leading to a surge in the research and development of conversational systems for a wide array of applications. While rule-based and pattern-matching conversational recommenders have existed for some time [4], the integration of the contextual understanding of LLMs into the specialized field of Recommender Systems is still a relatively unexplored topic [5].

This integration, however, is not a trivial task. It presents significant engineering challenges related to system architecture, data management, scalability, and the seamless fusion of conversational interfaces with complex recommendation algorithms. The challenge lies not only in leveraging the conversational power of LLMs but in building a robust, end-to-end platform that can be easily configured, extended, and deployed.

This project is motivated by the opportunity to address these challenges with a practical approach. The main goal is to harness the potential of LLMs to design and implement a versatile platform for creating Conversational Recommender Systems (CRSs). This platform will be engineered to be domain-agnostic and automatically extensible with external datasets. By employing modern software engineering practices and technologies, this work aims to create a system that is not only powerful in its functionality—offering natural conversations to elicit user preferences [6, Conversational Preference Elicitation] and generating explained recommendations—but also scalable, reproducible, and maintainable. The emphasis is on the practical engineering aspects required to build a sophisticated, full-stack application that translates the theoretical potential of LLMs into a practical solution for Recommender Systems.

Despite the existence of conversational libraries that have become quite common nowadays, this project was built as a standalone system to avoid relying on tools that evolve quickly and may become obsolete. It also provided an opportunity to understand how such conversational agents work by implementing the entire pipeline independently, allowing full control over its architecture and behavior.

1.2. Objectives

The primary objective of this Master's Thesis is the design and implementation of a scalable, modular, and reproducible platform (Conversational Recommender System) for building and deploying hybrid conversational recommender agents. The focus is centered on the engineering and architectural challenges inherent in creating a robust, full-stack application– whereas the complementary research thesis [7] is devoted to the research component and evaluation of the conversational and recommendation components. The specific objectives are detailed as follows.

O-1.– Detailed System Design: To define and document a resilient and scalable architecture that integrates multiple components, including a frontend user interface, a backend API with all the application logic, a LLM service, and data storage solutions. This architecture will be designed for modularity to facilitate independent development, testing, and deployment of each component.

O-2.– High-Performing Backend Service: To develop a backend using the FastAPI framework that serves as the backbone of the platform. This includes creating API endpoints for managing the lifecycle of recommender agents, handling user interactions, processing data, starting and interacting with conversational workflows and proxying requests to the LLM.

O-3.– Automated Data Processing Pipeline: To build a system capable of automatically ingesting, cleaning, and structuring user-provided datasets. This pipeline will standardize heterogeneous data files into a format suitable for both the graph-based recommender and the expert RS model, ensuring data integrity and consistency.

O-4.– Responsive Frontend Application: To build a modern PWA using the Next.js framework. This interface will provide users with a comprehensive dashboard to create and manage their recommender agents, as well as an intuitive chat interface to interact with them. Emphasis will be placed on usability, accessibility, and providing a seamless user experience.

O-5.– Scalability and Reproducibility with Containerization: To utilize Docker to containerize each component of the system (backend, database, LLM service). This approach will ensure that the entire platform is portable, easy to deploy, and horizontally scalable, adhering to modern DevOps practices.

O-6.– System Testing and Evaluation: To perform comprehensive testing, including unit and integration tests for backend logic, API load testing to measure performance under stress, usability testing with real users, and a comparative analysis of the platform's contributions against established benchmarks or similar systems where applicable.

1.3. Work Structure

This document is structured into six chapters, designed to provide a comprehensive overview of the project, from its design to its implementation and evaluation.

Chapter 1 — Introduction This initial chapter provides the motivation for the project, outlining the current landscape of LLMs and CRSs and identifying the engineering challenges this thesis aims to address. It also defines the main objectives of the work and presents this overview of the document's structure.

Chapter 2 — State of the Art This chapter reviews the key technologies and existing research that form the foundation of this project. It provides an overview of LLMs and their associated frameworks, discusses their application in Recommender Systems, and examines technologies required for data management and scalability, such as vector and graph databases. Finally, it reviews the web application frameworks used in the implementation.

Chapter 3 — System Design This chapter presents a detailed blueprint of the platform's architecture. It begins with an analysis of the system's functional and non-functional requirements. It then describes the high-level architecture, including use case diagrams, and delves into the specific design of the backend and frontend components, detailing module definitions, sequence diagrams, database schemas, and data flowcharts.

Chapter 4 — Implementation This chapter details the technical execution of the design presented in the previous chapter. It covers the development of the backend services, including the integration of the LLM and the RS models, and the creation of the frontend PWA. It also describes the deployment strategy, encompassing the frontend deployment, the use of Docker for containerization, the Continuous Integration and Continuous Deployment (CI/CD) pipeline, and the tunneling solution for exposing the backend service.

Chapter 5 — Testing and Evaluation This chapter focuses on the validation and assessment of the implemented platform. It describes the testing environment and methodologies, covering unit and integration testing for all major components, performance and load testing of the API, and a high-level summary of the usability testing results. A comparative analysis of the platform's contributions is also discussed.

Chapter 6 — Conclusions and Future Work The final chapter summarizes the key achievements and contributions of this thesis, reflecting on how the initial objectives were met. It also discusses the limitations of the current work and proposes potential avenues for future research and development, suggesting ways to extend and improve the platform.

2

State of the Art

This chapter provides a review of the key concepts, technologies, and existing research that form the foundation of this project. The aim is to establish the context in which this work is situated and to justify the technological choices made during the design and implementation phases.

The review is divided into four main areas. First, we provide a high-level overview of LLMs, the core technology enabling the conversational capabilities of the platform. Second, we briefly discuss the field of Recommender Systems, which will be examined from an engineering and integration perspective. Third, we explore essential topics in data management and scalability, which are critical for building a robust and efficient system. Finally, we review the modern web application frameworks that were utilized to construct the user-facing components of the platform.

2.1. Large Language Models

The main component that enables the advanced conversational capabilities of the developed platform is the Large Language Model. These models, built upon the transformative attention mechanism introduced by the Transformer architecture [8], have demonstrated an unparalleled ability to comprehend, reason, and generate human-like text. This section provides a high-level overview of LLMs and the key techniques used to augment their capabilities within our system.

To overcome the inherent limitation of their static training data, modern LLM applications employ advanced techniques to connect models with external, real-time information and tools. Two of the most prominent techniques are Retrieval-Augmented Generation (RAG) and Function Calling. RAG allows an LLM to retrieve relevant information from an external knowledge base to ground its responses in factual, specific data [9]. Function Calling, on the other hand, enables the model to interact with external software and APIs, allowing it to perform actions beyond simple text generation. A detailed exploration of these methods is conducted in the complementary thesis [7].

The field is populated by both proprietary models, such as those from OpenAI [2], Google [3], and Anthropic [10], and a rapidly growing ecosystem of open-source alternatives like the Qwen series of models [11]. This project leverages the flexibility of open-source models, which can be hosted locally for greater control and privacy.

2.1.1. LLM Frameworks

Developing applications that effectively harness the power of LLMs requires more than just access to a model. LLM frameworks have emerged as essential tools that abstract away the complexity of building, training, and managing interactions with these models. These frameworks provide a structured approach to prompt engineering, state management, and the integration of various components like data sources and external APIs.

Several frameworks have gained prominence in this area. LangChain [12] offers a general-purpose and comprehensive library for creating elaborate LLM-powered agents. LlamaIndex [13], conversely, adopts a more data-centric approach, providing robust tools specifically for connecting LLMs to custom data sources, making it particularly well-suited for RAG pipelines and workflows. Other frameworks like Haystack [14] also offer end-to-end solutions for building applications with LLMs.

For the practical deployment and serving of local models, several tools are available. Platforms like Ollama [15], based on llama.cpp [16] simplify the process of running and managing open-source LLMs on personal hardware. For performance-critical applications, inference engines such as vLLM [17] offer optimized memory management and throughput for serving LLM models efficiently.

In this project, LlamaIndex was chosen to manage the conversational workflow and data integration, for its lean but efficient feature set and overall modular pipeline. In addition, Ollama was used due to its simplified deployment of local models.

2.1.2. LLMs in Recommendation

The application of LLMs to the domain of Recommender Systems is a growing field of research that promises to redefine how users interact with recommendation services [5]. Traditionally, users interact with Recommender Systems through rigid interfaces. By integrating LLMs, it is possible to create dynamic, conversational experiences where users can express their preferences in natural language, ask for clarifications, and receive recommendations that are not only relevant but also contextually explainable by the language model.

The primary role of the LLM in this context is to act as a natural language interface between the user and the underlying recommendation engine. It can parse user queries, maintain conversational context, and format the output from the recommender into a coherent and helpful response. This thesis focuses on the engineering aspects of building a platform that facilitates this integration, creating a scalable system where conversational agents can be powered by expert recommender models. The specific research questions regarding the effectiveness of different conversational strategies and their impact on user satisfaction, recommendation quality and transparency are explored in greater detail in the complementary research-focused thesis [7].

2.2. Recommender Systems

Recommender Systems are a cornerstone of modern web platforms, designed to help users navigate vast catalogs of items by predicting their preferences and suggesting relevant content. These systems are critical for personalization and user engagement in domains ranging from e-commerce to media streaming. The underlying algorithms can be broadly categorized into several families.

The most common approaches are Content-Based Filtering, which recommends items based on their intrinsic properties, and Collaborative Filtering, which depends on the behavior of a community of users to find items a target user might like, such as ItemKNN [18]. Many state-of-the-art systems employ hybrid methods, which combine these and other strategies to mitigate their individual weaknesses and improve recommendation quality.

This thesis does not aim to propose novel recommendation algorithms. Instead, the focus is on the engineering challenge of integrating existing, pre-trained recommender models as “expert” components within a larger, scalable conversational platform. To facilitate this, the project utilizes RecBole [19], a established recommendation library which provides a unified and efficient framework for working with a wide range of recommendation algorithms. A detailed investigation of library comparison, model classification, evaluation metrics, and the specific model choices for this project are covered in the complementary research thesis [7].

2.3. Data Management & Scalability

Building robust and effective applications powered by LLMs requires a strong foundation in data management and system scalability. The performance of techniques like RAG and the overall reliability of the platform are highly dependent on how data is processed, stored, and retrieved. This section reviews the essential data management practices and technologies applied in this project, covering the entire data lifecycle from initial cleaning to efficient storage in specialized databases tailored for Artificial Intelligence (AI) workloads.

2.3.1. Data Preprocessing

Raw data sourced from the real world is inherently noisy, inconsistent, and often incomplete. Before this data can be effectively used to train a recommender model or populate a knowledge base for an LLM, it must undergo a thorough preprocessing stage. This critical step, often referred to as data cleansing, involves a series of transformations to improve data quality and ensure its usefulness for downstream tasks.

Common preprocessing operations include handling missing or null values, identifying and removing duplicate entries, correcting structural errors, and standardizing data formats. For numerical data, normalization and outlier detection are often necessary to prevent skewed model behavior. In the context of this project, an automated pipeline employing libraries like AutoClean [20] was developed to perform these tasks systematically, ensuring that the datasets used by the recommender agents are clean, consistent, and reliable.

2.3.2. Document Chunking

One of the primary limitations of LLMs is their finite context window, which restricts the amount of information that can be processed in a single query. To apply these models to large documents or extensive knowledge bases, a technique known as chunking is employed, particularly within RAG frameworks [9].

Chunking is the process of breaking down large texts into smaller, semantically meaningful segments. The goal is to create chunks that are small enough to fit within the model's context window, yet large enough to retain their original meaning and relevance. The strategy used for chunking—whether it involves fixed-size splits, sentence-based division, or more advanced content-aware methods—has a direct and significant impact on the quality of the retrieval process. Effective chunking ensures that the information retrieved and presented to the LLM is coherent and relevant, which is fundamental to generating accurate and contextually-aware responses [21]. Although this is a primary technique for LLM contextualization, it was not employed in this project due to the tabular nature of the datasets.

2.3.3. Vector Databases

Vector databases have become a foundational technology for modern AI applications, especially those utilizing embeddings to represent data like text or images. These databases are specialized systems designed to store and query high-dimensional vector representations of data efficiently.

In a typical RAG pipeline, after documents are divided into chunks, each chunk is passed through an embedding model to convert it into a numerical vector. These vectors are then stored in a vector database, which uses specialized indexing algorithms (e.g., Hierarchical Navigable Small World - HNSW) to enable fast and scalable similarity searches [9]. When a user submits a query, it is also converted into a vector, and the database is queried to find the stored vectors that are "closest" in the embedding space. This rapid retrieval of the most relevant document chunks is essential for providing the LLM with the right context to answer the query. While this project's architecture supports vector stores, its primary data engine for recommendations is a graph database.

2.3.4. Graph Databases

While vector databases excel at semantic similarity search, graph databases are optimized for managing and querying data with intricate and meaningful relationships. In a graph database, data entities are represented as nodes (or vertices) and the relationships between them as edges. This model is exceptionally well-suited for the domain of Recommender Systems, where the ecosystem of users, items, and interactions can be naturally represented as a graph.

Modeling data in this way allows for powerful and intuitive querying of complex relationships. For example, collaborative filtering can be implemented by traversing the graph to find users who have rated the same items, and explanations for recommendations can be generated by identifying the paths that connect a user to a suggested item.

This project relies on FalkorDB [22], a high-performance graph database built on Redis, which uses sparse matrices and linear algebra to accelerate graph operations. It serves as the primary data store for user-item interactions, enabling efficient graph-based recommendation and explanation generation, which are central to the system's functionality.

2.4. Web Application Frameworks

The development of a modern, full-stack platform requires the careful selection of web application frameworks for both the client-facing frontend and the server-side backend. These frameworks provide the foundational structure upon which the application is built, significantly influencing development velocity, performance, maintainability, and the end-user experience.

This project employs a decoupled architecture, with a distinct frontend application responsible for rendering the user interface and a separate backend API that handles business logic, data processing, and communication with other services. This approach, illustrated in contrast to a traditional monolithic architecture in Figure 2.1, allows for greater flexibility and scalability. This section provides an overview of the primary frameworks chosen for this architecture: Next.js for the frontend and FastAPI for the backend. It also discusses general design principles for user interfaces in the context of LLM-powered applications.

2.4.1. Next.js

For the frontend component of the platform, Next.js was selected as the development framework [23]. Next.js is a production-grade React framework that provides a rich set of features designed to

Figure 2.1: Comparison of Monolithic and Decoupled Web Architectures.

build fast, scalable, and user-friendly web applications. Its architecture is particularly well-suited for creating intricate, data-driven interfaces like the one required for this project.

Key features of Next.js that motivated its selection include its flexible rendering strategies, such as Server-Side Rendering (SSR) and Static Site Generation (SSG), which enhance performance and Search Engine Optimization (SEO). The framework's file-based routing system simplifies the organization of pages and components, leading to a more intuitive development workflow. Furthermore, Next.js has first-class support for TypeScript, which enables the development of type-safe, robust, and more maintainable code. While Next.js also supports the creation of backend API routes, this project utilizes a dedicated Python backend to better separate concerns and leverage Python's extensive data science ecosystem.

2.4.2. FastAPI

FastAPI is a modern, high-performance web framework for building APIs with Python [24]. Its design philosophy prioritizes development speed and runtime performance, making it an excellent choice for services that must handle potentially high loads and complex logic.

One of the highlights of FastAPI is its automatic generation of interactive API documentation. It depends on Python type hints and the Pydantic library to automatically validate, serialize, and deserialize data, while also creating a detailed OpenAPI (formerly Swagger) schema for the documentation. This feature proves pragmatic for development, testing, and potential integration with other services.

Furthermore, FastAPI's performance is among the best in the Python ecosystem, as it is built atop the Starlette ASGI framework [25]. This ensures that the API can handle concurrent requests efficiently, essential for a responsive user experience, especially when mediating long-running tasks involving LLM inference or data processing. These features collectively make FastAPI an ideal choice for the robust and scalable backend required by this project.

2.4.3. UI/UX in LLM Applications

Designing an effective User Interface (UI) and User Experience (UX) for LLM-powered applications presents unique challenges that differ from traditional web applications. The interaction model shifts from predictable, form-based inputs to fluid, open-ended conversations, which requires a more dynamic and responsive interface.

A key principle in conversational UI design is managing user expectations and providing continuous feedback. Due to the potential latency of LLM responses, it is favorable to stream tokens to the user as they are generated. This creates an immediate sense of activity and reduces perceived wait times. Additionally, the UI should clearly indicate the system's current state, for example, by showing when it is processing a request, retrieving information, or using an external tool. Beyond the conversational aspects, adhering to fundamental principles of modern web design is also important. This includes implementing a fully responsive design to ensure the platform is accessible and usable across a wide range of devices, from large desktop monitors to mobile phones. A responsive layout is critical for a PWA, as it guarantees a consistent and high-quality experience regardless of the device from which the user accesses the application.

To achieve a polished and intuitive user experience, this project utilizes modern UI component libraries. Specifically, it leverages `shadcn/ui` [26], a collection of beautifully designed components, and `assistant-ui` [27], a React library specifically tailored for building AI chat interfaces. These tools provide the necessary building blocks to implement features like streaming responses, displaying conversation history, and handling complex user interactions, ensuring the final application is both functional and visually appealing.

3
System Design

This chapter transitions from foundational concepts to the specific design of the proposed platform. It serves as the architectural blueprint for the project, detailing the methodologies and patterns used to construct a scalable and maintainable system. First, it begins with a thorough analysis of the system's requirements, which are categorized into functional and non-functional specifications. This analysis forms the basis for all subsequent design decisions. Following the requirements, the chapter presents a comprehensive overview of the system's architecture, including its modular design and the interactions between its various components.

3.1. Requirements Analysis

One of the first and most important steps in any engineering project is the elicitation and definition of system requirements. This process establishes a clear set of goals and constraints that guide the architectural design and implementation. The requirements have been divided into functional specifications, which describe what the system must do, and non-functional specifications, which define the system's quality attributes.

3.1.1. Functional Requirements

FR-AUTH-1.– User Authentication and Management: The system must provide secure mechanisms for user authentication and account management.

- **FR–1.1.–** Users must be able to register using an email and password combination or via a Google OAuth provider.
- **FR–1.2.–** Registrations via email must trigger a confirmation email to verify the user's address.
- **FR–1.3.–** Registered users must be able to log in to access the platform.
- **FR–1.4.–** A password reset mechanism must be available for users who have forgotten their password.
- **FR–1.5.–** The system must provide a secure session key for authenticated users to interact with the API.

FR-AGENT-1.– Recommender Agent Lifecycle Management: The platform must provide comprehensive functionalities for users to create, manage, and interact with conversational recommender agents.

FR–1.1.– Creation: Users must be able to create a new agent by uploading datasets for interactions, item features, and user features. The system must support asynchronous processing of these files, including data cleansing, ingestion into the graph database, and the training of an expert recommender model.

FR–1.2.– Discovery: The platform must feature an "Agent Hub" where users can browse, search, filter, and sort all agents they have access to.

FR–1.3.– Modification: Agent creators must be able to edit their agent's metadata (e.g., name, description, visibility) and retrain the underlying model with updated data.

FR–1.4.– Deletion: Agent creators must be able to permanently delete their agents, which must trigger the removal of all associated artifacts, including dataset files, database entries, chat histories and trained models.

FR-CHAT-1.– Conversational Interaction: The system must offer a robust and feature-rich conversational interface.

FR–1.1.– A dedicated chat interface must be available for interacting with each specific recommender agent.

FR–1.2.– Conversation histories with agents must be archived and available for users to review in a read-only format.

FR–1.3.– A general-purpose "Open Chat" must be provided for direct and unrestricted conversation with a selected LLM. This will be used to evaluate LLM model capabilities and for general-purpose queries and structured prediction.

FR–1.4.– The "Open Chat" interface should support advanced features present in modern conversational interfaces, such as web search, the ability to upload files for context, and message editing.

FR–1.5.– The conversations in the "Open Chat" interface must be persisted and may be resumed or deleted by the user.

3.1.2. Non-Functional Requirements

NFR-PERF-1.– Performance: The system must be highly responsive and efficient.

NFR–1.1.– The backend API must maintain low latency, even under concurrent loads.

NFR–1.2.– LLM responses in the chat interface must be streamed token-by-token to minimize perceived latency.

NFR–1.3.– Database queries for filtering and searching in the Agent Hub must be optimized for speed.

NFR-USABIL-1.– Usability and Accessibility: The platform must provide a high-quality, intuitive, and accessible user experience.

NFR–1.1.– The user interface must be fully responsive, ensuring a seamless experience on both desktop and mobile devices.

NFR–1.2.– The platform must support internationalization (*i18n*) with translations for multiple languages.

NFR–1.3.– Accessibility must be enhanced through features such as speech-to-text and text-to-speech in the chat interface.

NFR-SEC-1.– Security: The system must ensure the confidentiality and integrity of user data.

NFR–1.1.– All API endpoints must be protected against unauthorized access.

- **NFR–1.2.–** Row-Level Security (RLS) policies must be enforced in the database to ensure users can only access their own private conversations and agents.

NFR-MAINT-1.– Maintainability and Scalability: The system must be designed for long-term maintenance, portability, and scalability.

- **NFR–1.1.–** The architecture must be modular to allow for independent development and deployment of its components.
- **NFR–1.2.–** The entire application stack must be containerized using Docker to ensure portability and ease of deployment.
- **NFR–1.3.–** The architecture must support horizontal scaling to accommodate a growing user base.

3.2. Architecture Overview

The platform is founded on a decoupled architecture designed to promote modularity, scalability, and maintainability. This architectural style separates the system into several distinct, independently deployable components, each with a specific responsibility. This approach prevents the tight coupling of a monolithic system, allowing for greater flexibility in development, technology choice, and scaling strategies. Communication between these modules is handled through a well-defined API, which serves as the contract between the frontend client and the backend services.

The primary components of the architecture are:

- **Frontend Application:** A client-side PWA built with Next.js, which serves as the sole point of interaction for the end-user. It is responsible for rendering the entire user interface and managing client-side state.
- **Backend API Gateway:** A central, high-performance API developed in FastAPI. This component acts as an orchestrator, handling all business logic, user authentication, and routing requests to the appropriate downstream services.
- **LLM Service:** A dedicated container running Ollama, responsible for hosting, managing, and serving inferences from the open-source LLMs.
- **Data Services:** A collection of data storage solutions, including a Supabase PostgreSQL instance for structured metadata, and a FalkorDB graph database for storing and querying both user-item interaction data and chat history records.

Each of these components is containerized using Docker, ensuring environmental consistency and portability across different deployment stages. This modular design, illustrated in Figure 3.1, is fundamental to achieving the system's non-functional requirements of scalability and maintainability.

3.2.1. Use Case Diagram

Figure 3.1: High-Level System Architecture Diagram.

3.3. Backend Design

3.3.1. Module Definition

3.3.2. Sequence Diagrams

3.3.3. Entity-Relationship Diagram

3.3.4. Data Flowchart

3.4. Frontend Design

3.4.1. Design Patterns and Principles

3.4.2. Accessibility

3.4.3. Localization and Internationalization

4

Implementation

4.1. Backend Development

4.1.1. LLM Integration

4.1.2. Recommender System Integration

4.1.3. Automated Data Preprocessing

4.1.4. API Endpoints

4.1.5. API Documentation

4.2. Frontend Development

4.2.1. UI Components

4.2.2. State Management

4.2.3. API Integration

4.2.4. Authentication and Authorization

4.3. Deployment

4.3.1. Docker Containerization

4.3.2. Frontend Deployment

4.3.3. Backend Tunneling

4.3.4. CI/CD Pipeline

5 Testing and Evaluation

After the implementation of the platform, a comprehensive testing and evaluation phase was conducted to validate its functionality, performance, and overall quality. This chapter details the methodologies and results of this evaluation. The objective is to provide empirical evidence that the system meets its technical requirements and performs robustly under expected usage patterns.

5.1. Testing Environment

All performance, unit and integration tests were conducted on a single machine with the following hardware specifications, representing a typical modern developer-grade laptop:

- **CPU:** Intel® Core™ i7-11800H @ 2.30 GHz (16 Cores)
- **RAM:** 16 GB DDR4 @ 3200MHz
- **GPU:** NVIDIA® GeForce® RTX 3050 Laptop GPU with 4GB VRAM

The software environment was managed entirely through Docker Compose. For the duration of the performance tests, the FastAPI backend was configured to run with 4 Uvicorn workers to make better use of the available CPU cores. It is important to note that for production use, the backend is limited to a single worker due to the in-memory dictionary used to manage active conversational workflows. The testing environment also included the containerized Ollama and FalkorDB services.

5.2. Unit & Integration Testing

To ensure the correctness and reliability of the platform's codebase, the testing strategy incorporated both **unit tests** and **integration tests**. This approach validates the system at different levels: verifying individual components in isolation and ensuring that these components interact correctly as a cohesive whole. All tests were developed and executed using the `pytest` framework and are organized in a dedicated `/tests` directory for maintainability.

Unit tests were written for discrete functions and classes, often using mocking to isolate the component under test from its external dependencies. Integration tests, on the other hand, were designed to validate the interactions between different modules, such as testing an API endpoint's complete logic flow or the interaction between the recommender class and the graph database.

5.2.1. Backend

The backend's business logic, which encompasses data processing, recommendation generation, and conversational state management, was thoroughly evaluated with unit tests to ensure its reliability. Excerpts of each test are provided in Appendix **??**.

The data processing pipeline, implemented in `data_utils.py`, was validated through `test_data_utils.py`. These tests confirm the correctness of the LLM-based inference for identifying column roles and data types, ensuring the automated data preparation is robust.

The core recommendation logic was tested in `test_falkordb_recommender.py`. Using a `pytest` fixture, a temporary mock dataset is created to test the `FalkorDBRecommender` class. The test suite validates the entire lifecycle, including data ingestion, graph creation, and the output of the different recommendation methods (context-aware, collaborative filtering, and hybrid recommendations), and the generation of explanations.

Finally, the stateful components of the conversational workflow were validated. The `test_user_profile.py` script tests the `UserProfile` class for managing user preferences mid-conversation, while `test_falkordb_chat_history.py` ensures the reliable persistence of conversation logs in the graph database.

5.2.2. API

To validate the API layer, integration tests were written using FastAPI's `TestClient`, which simulates HTTP requests to the application without needing to run a live server. The tests, located in `test_api.py`, make use of the `pytest-mock` library to isolate the API from its downstream dependencies.

This mocking strategy is paramount for focused testing. For example, in `test_create_agent`, all backend logic–including calls to Supabase, FalkorDB, and the RecBole training functions–is mocked. This allows the test to verify that the `/create-agent` endpoint correctly handles multipart form data, parses the request body, and calls the appropriate backend functions, without executing the time-consuming processes of data ingestion and model training. Similarly, the JWT authentication middleware is mocked in all tests to bypass the need for a valid token, allowing the focus to remain on the endpoint's logic.

5.2.3. Frontend

While the primary focus of the testing strategy was on the backend's complex logic, no unit or integration tests were carried out for the frontend. However, the frontend implementation benefits from multiple automated checks that help maintain code quality and catch common issues early in development.

The entire frontend codebase is written in TypeScript, which enables static type checking at compile time. This helps detect many potential runtime errors—such as type mismatches or incorrect property access—before the code is even run, contributing to a more robust and self-documenting codebase.

Additionally, as discussed in Chapter 4, a pre-commit hook managed by Husky automatically runs ESLint on every commit. ESLint provides static code analysis, enforcing a consistent coding style and flagging potential bugs, anti-patterns, and logical errors in the React and TypeScript code. These measures help uphold a high standard of frontend quality despite the absence of runtime testing.

5.3. Performance Testing

Performance testing is crucial for quantifying the system's responsiveness and stability. The evaluation focused on two key performance indicators: the backend API's ability to handle concurrent requests and the local LLM's inference throughput.

5.3.1. API Load Testing

To measure the backend's performance under stress, load tests were conducted using Siege [28], an open-source HTTP/HTTPS benchmarking utility. The tests were configured to simulate 10 concurrent users making continuous requests over a 30-second period. All requests included a valid JWT to ensure the authentication middleware was also evaluated. Three key endpoints were selected to test different aspects of the backend:

- **Root (`/`):** A baseline test to measure the raw request-handling speed of FastAPI.
- **Ollama Proxy (`/ollama/api/version`):** To measure the overhead introduced by proxying requests to the Ollama service.
- **FalkorDB Query (`/get-chat-history`):** To measure the latency of querying the graph database (with indexed identifiers), getting an example history of 6072 bytes.

Endpoints interacting with the Supabase cloud API were not tested due to bandwidth limitations. The results, summarized in Table 5.1, demonstrate that the backend is highly performant, capable of handling thousands of transactions per second with an average response time in the single-digit milliseconds.

Metric	Root (/)	Ollama Proxy	FalkorDB Query
Transaction Rate (trans/sec)	5620.67	2428.50	2964.35
Average Response Time (ms)	1.75	4.08	3.34
Concurrency Rate	9.84	9.91	9.89
Throughput (MB/sec)	0.13	0.04	17.17
Successful Transactions	170,194	74,555	89,879
Failed Transactions	0	0	0

Table 5.1: API Load Test Results (10 concurrent users over 30s).

5.3.2. LLM Inference Testing

The perceived speed of a conversational agent is directly tied to the token generation throughput of its underlying LLM. A test was conducted to measure the raw inference speed of the locally hosted `qwen2.5:3b` model running on the NVIDIA RTX 3050 GPU.

A standard prompt ("Why is the sky blue?") was sent to the model, which generated a 354-token response. The total generation time was 6.74 seconds. This yields a calculated throughput of approximately **52.5 tokens per second**.

This result is highly favorable. It is comparable to the performance of cloud-based flagship models like GPT-4o and significantly exceeds the community-accepted standard of 7-10 tokens per second for a good user experience. Furthermore, given that the average human reading speed is around 4 tokens per second [29], this level of performance ensures that text is generated faster than the user can read it, creating a fluid conversational experience.

5.4. Usability Testing

5.5. Comparative Analysis

6 Conclusions and Future Work

6.1. Conclusions

TODO

The repository containing all the code and data used throughout the project, along with steps to set up the web application, can be found at `https://github.com/Acervans/Hybrid-CRS`.

6.2. Future Work

TODO

Bibliography

[1] F. Ricci, L. Rokach, B. Shapira, and P. B. Kantor, *Recommender Systems Handbook*. Springer, 2022.

[2] OpenAI, “ChatGPT.” `https://chat.openai.com/chat`, 2023.

[3] Google DeepMind, “Gemini,” 2023.

[4] D. Pramod and P. Bafna, “Conversational recommender systems techniques, tools, acceptance, and adoption: A state of the art review,” *Expert Systems with Applications*, vol. 203, p. 117539, 2022.

[5] Z. Zhao, W. Fan, J. Li, Y. Liu, X. Mei, Y. Wang, Z. Wen, F. Wang, X. Zhao, J. Tang, and Q. Li, “Recommender Systems in the Era of Large Language Models (LLMs),” *IEEE Transactions on Knowledge and Data Engineering*, vol. 36, p. 6889–6907, Nov. 2024.

[6] O. Sar Shalom, H. Roitman, and P. Kouki, *Natural Language Processing for Recommender Systems*, pp. 447–471. In [1], 2022.

[7] J. W. Zhou, “A Hybrid Conversational Recommender System by integrating LLMs: Conversation Design & User Profiling for Explainable Recommendations,” 2025. Master’s Thesis, Universidad Autónoma de Madrid.

[8] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, “Attention is All you Need,” in *Advances in Neural Information Processing Systems* (I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, eds.), vol. 30, Curran Associates, Inc., 2017.

[9] Y. Gao, Y. Xiong, X. Gao, K. Jia, J. Pan, Y. Bi, Y. Dai, J. Sun, M. Wang, and H. Wang, “Retrieval-Augmented Generation for Large Language Models: A Survey,” 2024.

[10] Anthropic, “Claude,” 2023.

[11] J. Bai, S. Bai, Y. Chu, Z. Cui, K. Dang, X. Deng, Y. Fan, W. Ge, Y. Han, F. Huang, B. Hui, L. Ji, M. Li, J. Lin, R. Lin, D. Liu, G. Liu, C. Lu, K. Lu, J. Ma, R. Men, X. Ren, X. Ren, C. Tan, S. Tan, J. Tu, P. Wang, S. Wang, W. Wang, S. Wu, B. Xu, J. Xu, A. Yang, H. Yang, J. Yang, S. Yang, Y. Yao, B. Yu, H. Yuan, Z. Yuan, J. Zhang, X. Zhang, Y. Zhang, Z. Zhang, C. Zhou, J. Zhou, X. Zhou, and T. Zhu, “Qwen Technical Report,” *arXiv preprint arXiv:2309.16609*, 2023.

[12] H. Chase, “LangChain.” `https://github.com/langchain-ai/langchain`, October 2022.

[13] J. Liu, “LlamaIndex.” `https://github.com/jerryjliu/llama_index`, 11 2022.

[14] M. Pietsch, T. Möller, B. Kostic, J. Risch, M. Pippi, M. Jobanputra, S. Zanzottera, S. Cerza, V. Blagojevic, T. Stadelmann, T. Soni, and S. Lee, “Haystack: the end-to-end NLP framework for pragmatic builders.” `https://github.com/deepset-ai/haystack`, November 2019.

[15] Ollama developers, “Ollama,” 2025. Open-source platform for running Large Language Models locally.

[16] G. Gerganov and ggml-org community, “llama.cpp: LLM inference in C/C++.” https://github.com/ggml-org/llama.cpp, 2023. Accessed: 2025-07-02.

[17] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica, “Efficient memory management for large language model serving with pagedattention,” in *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

[18] M. Deshpande and G. Karypis, “Item-based top-N recommendation algorithms,” *ACM Trans. Inf. Syst.*, vol. 22, p. 143–177, Jan. 2004.

[19] W. X. Zhao, S. Mu, Y. Hou, Z. Lin, Y. Chen, X. Pan, K. Li, Y. Lu, H. Wang, C. Tian, Y. Min, Z. Feng, X. Fan, X. Chen, P. Wang, W. Ji, Y. Li, X. Wang, and J.-R. Wen, “RecBole: Towards a unified, comprehensive and efficient framework for recommendation algorithms,” 2021.

[20] E. Landman, “AutoClean: Automated data preprocessing & cleaning.” https://github.com/elisemercury/AutoClean, Mar. 2022. Version 1.1.3 (latest release Aug 19, 2022); MIT License.

[21] S. Karzhev, “Advanced RAG Techniques,” 9 2024.

[22] FalkorDB developers, “FalkorDB: A high-performance graph database leveraging sparse matrices and linear algebra.” https://github.com/FalkorDB/FalkorDB, 2025.

[23] V. Inc., “Next.js: The React Framework.” https://github.com/vercel/next.js, 2025.

[24] S. Ramírez, “FastAPI.” https://fastapi.tiangolo.com, 2020.

[25] T. Christie and contributors, “Starlette: The little ASGI library that shines.” https://github.com/encode/starlette, 2018.

[26] shadcn, “shadcn/ui.” https://github.com/shadcn-ui/ui, 2023.

[27] Farshid, Simon and contributors, “assistant-ui: React components for AI chat.” https://www.npmjs.com/package/@assistant-ui/react, 2025.

[28] J. Fulmer, “Siege: HTTP/HTTPS load testing and benchmarking utility.” https://github.com/JoeDog/siege/, 2022. Latest release version 3.0.9; Licensed under GPLv3; website: joedog.org/siege-home.

[29] S. E. Taylor, “Eye Movements in Reading: Facts and Fallacies,” *American Educational Research Journal*, vol. 2, no. 4, pp. 187–202, 1965.

Acronyms

AI Artificial Intelligence.

API Application Programming Interface.

CI/CD Continuous Integration and Continuous Deployment.

CPU Central Processing Unit.

CRS Conversational Recommender System.

GPT Generative Pre-trained Transformer.

GPU Graphics Processing Unit.

LLM Large Language Model.

PWA Progressive Web Application.

RAG Retrieval-Augmented Generation.

RAM Random Access Memory.

RS Recommender System.

UI User Interface.

UX User Experience.

Appendices

UAM

Universidad Autónoma
de Madrid