Universidad Autónoma de Madrid

Escuela Politécnica Superior

25|26

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

excelencia Campus Internacional 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

agosto 2025

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 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 engi-

neering 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 crucial 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.

This review is divided into four main areas. First, a high-level overview of LLMs, the core technology enabling the conversational capabilities of the platform, is provided. Second, the field of Recommender Systems is briefly discussed and examined from an engineering and integration perspective. Third, essential topics in data management and scalability are explored, which are critical for building a robust and efficient system. Finally, a review of the modern web application frameworks utilized to build the user-facing components of the platform is presented.

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 fundamental 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 knowledge 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. Graphs created using FalkorDB may be managed through the FalkorDB Browser application, as illustrated in Figure 2.1.

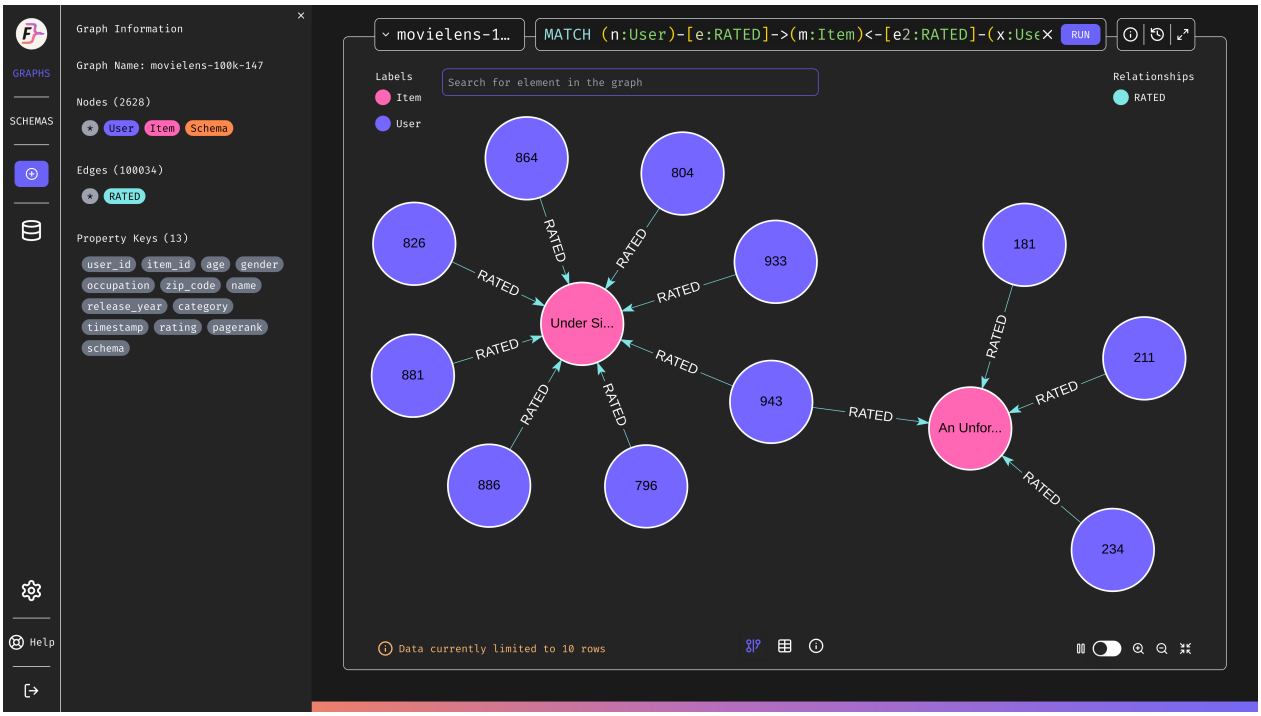


Figure 2.1: FalkorDB Browser Interface.

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 headless (decoupled) architecture, with a distinct frontend application responsible for rendering the User Interface, and a separate backend API with independent libraries that handle domain logic, data processing, and communication with other services. This design separates presentation concerns from server-side logic, offering greater flexibility and scalability compared to traditional, monolithic (tightly-coupled) architectures where the server is also responsible for delivering the User Interface (as illustrated in Figure 2.2).

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 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.

Principal 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.

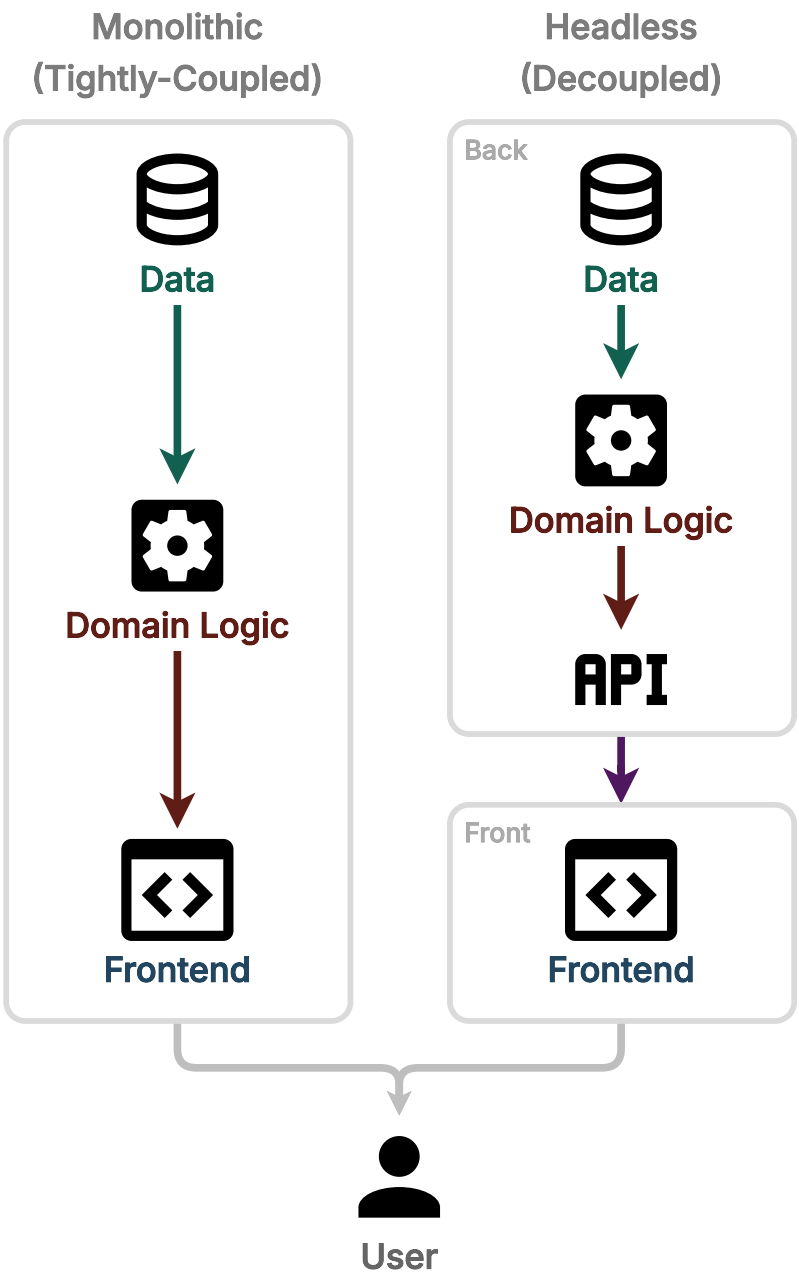


Figure 2.2: Comparison between Monolithic and Headless web architectures.

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.

3.2.1. System Components

The primary components of the system 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:** A central, high-performance API developed in FastAPI. This component acts as an orchestrator, handling all domain 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 hybrid solution with two data storage services, consisting of a Supabase [28] PostgreSQL instance for structured metadata, and a FalkorDB graph database (NoSQL) for storing and querying both user-item interaction data and chat history records.

Each of these components can be containerized using Docker, ensuring environmental consistency and portability across different development stages. In production, though, the Frontend and the Supabase services are deployed in cloud environments, while the rest is deployed on-premises and accessi-

ble through the Backend API. This modular design, illustrated in Figure 3.1, is fundamental to achieving the system's non-functional requirements of scalability and maintainability.

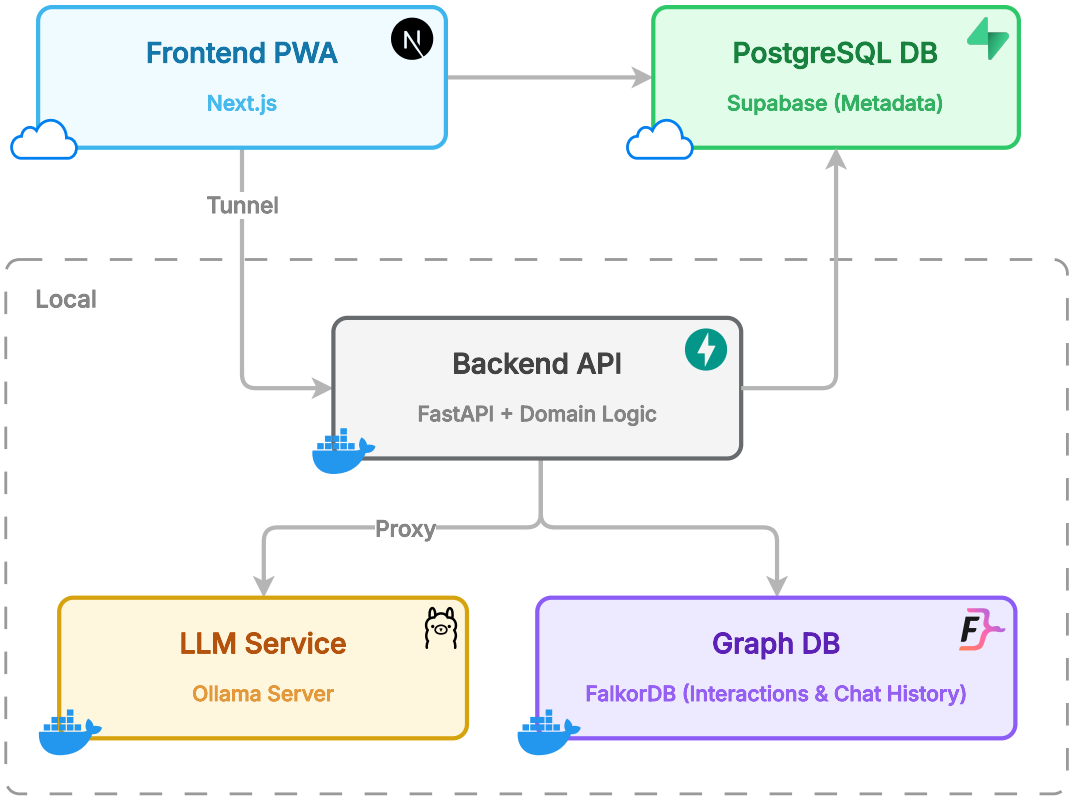


Figure 3.1: High-Level System Architecture Diagram.

3.2.2. Use Case Analysis

To better understand the system's intended functionality from a user's perspective, a use case analysis was performed. To this end, a use case diagram was designed, which is an effective tool for visualizing the interactions between external actors and the system, thereby defining its functional scope.

In this system, the primary actor is the **User**, a general user representing all use cases of the system. Figure 3.2 illustrates the main functionalities available to the actors **User** and **Guest User**, with the latter being, exclusively, a non-authenticated user. These use cases cover the entire spectrum of user interaction with the platform, from initial account management to the creation of and conversation with recommender agents. The principal use cases are:

Manage Account: Encompasses all functionalities related to user account management, including registration, login, password reset, and viewing the personal API access key.

Manage Recommender Agents: Includes the complete lifecycle of an agent: creating a new agent from datasets, browsing the Agent Hub, editing agent metadata, retraining and deleting an agent.

Use Agent Chat: Represents the core conversational feature, where a user engages in a dialogue with a specific, trained recommender agent to receive recommendations.

Use Open Chat: Refers to the general-purpose chat interface where a user can converse freely with an LLM, with support for advanced features like web search and file uploads.

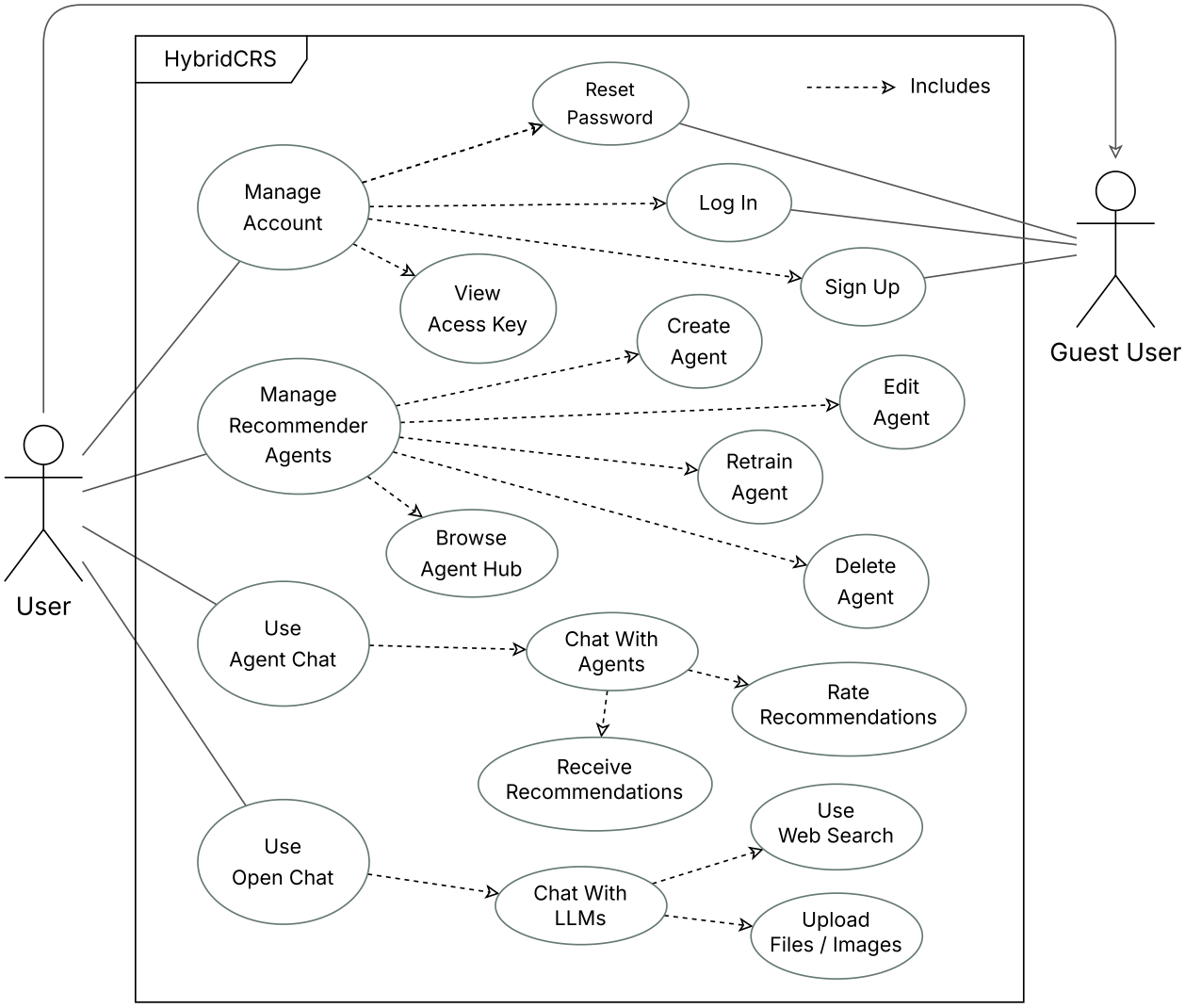


Figure 3.2: System Use Case Diagram.

3.3. Backend Design

The backend serves as the central backbone of the platform. It is designed as a stateless, high-performance API gateway using the FastAPI framework. Its primary role is to process incoming requests from the frontend, orchestrate the necessary domain logic by communicating with various downstream services, and return responses to the client. The design prioritizes a clean separation of concerns, dividing the complex logic into distinct, maintainable modules. This section details this modular struc-

ture, models primary interactions with sequence diagrams, defines the database schema for metadata storage, and illustrates the primary data flows within the system.

3.3.1. Module Definition

The backend's logic is organized into several distinct modules, each responsible for a specific domain of functionality. This modular design enhances maintainability and allows for a clear separation of concerns, as depicted in the module diagram in Figure 3.3.

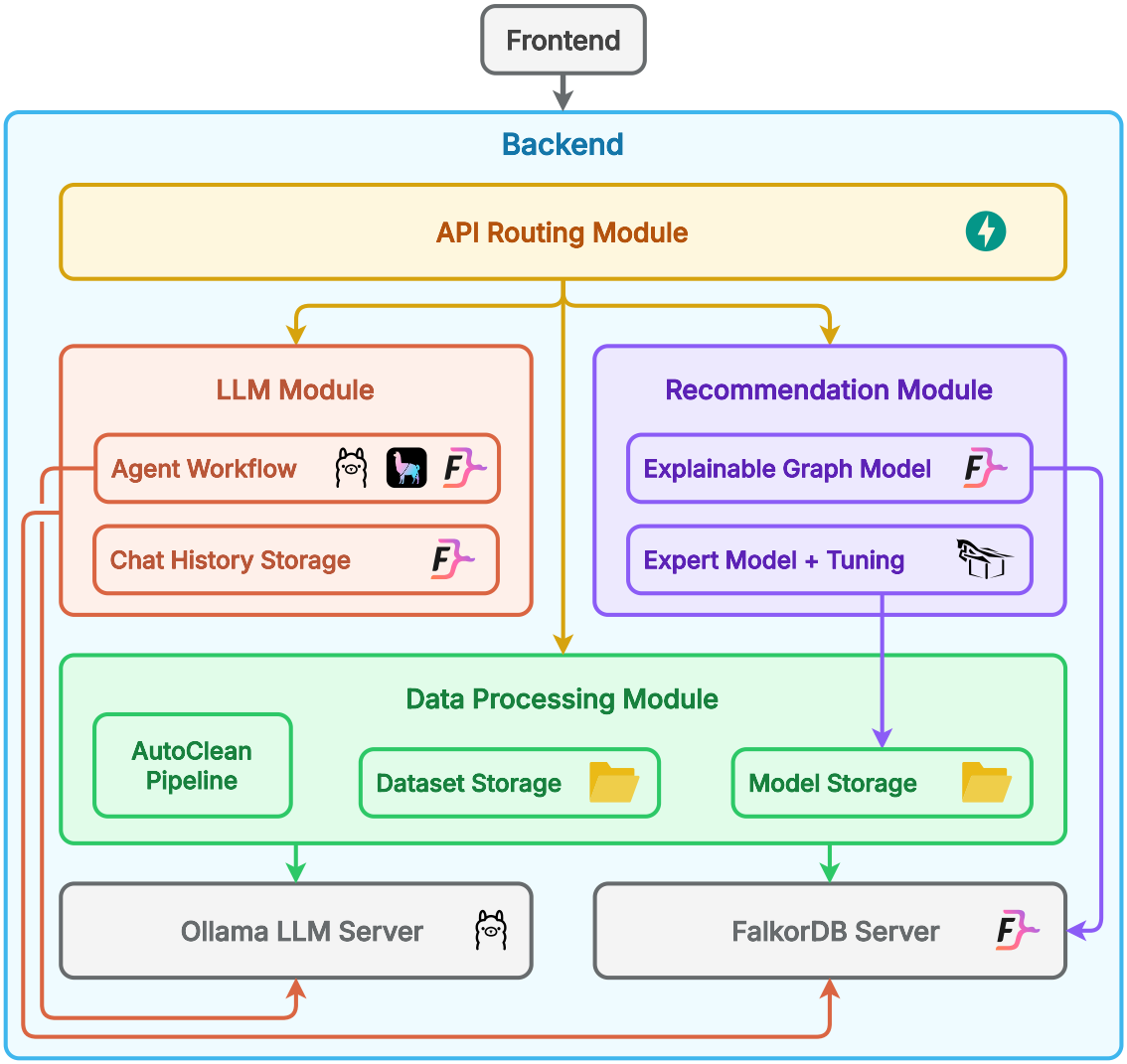


Figure 3.3: Backend Module Diagram.

The core modules are defined as follows:

API Routing Module This is the main entry point of the backend, built using FastAPI. It is responsible for defining all HTTP endpoints, handling request validation through Pydantic schemas, managing user authentication through a middleware, and delegating tasks to the

appropriate domain logic modules.

LLM Module This module encapsulates all interactions with the language models and the graph database. It utilizes the LlamaIndex framework to manage the conversational agent's logic, including streaming events and Function Calling. It is also responsible for managing user profiles during a conversation and persisting chat histories to the graph database as nodes.

Recommendation Module This module contains all logic related to generating recommendations. It is further subdivided into two major components: a graph-based recommender using FalkorDB for real-time, explainable recommendations, and an expert model from RecBole to train on the datasets—including new session interactions—and serve predictions.

Data Processing Module This module handles the asynchronous, time-consuming tasks related to agent creation. When a user uploads new datasets, this module executes a pipeline that cleans the data, prepares it for the different recommender components, and triggers the model training and data ingestion processes.

3.3.2. Sequence Diagrams

To illustrate the dynamic behavior of the system, sequence diagrams may be used to model the flow of interactions between different components, in order to visualize the processes involved. The following diagrams detail three of the most critical use cases in the platform.

Agent Creation Sequence

The creation of a new recommender agent is a complex, asynchronous process that engages all system components. As shown in Figure 3.4, it begins when the user submits dataset files and configuration through the frontend. The backend API validates the request and initiates a data processing pipeline that cleans invalid values, fills missing entries, removes duplicates, and normalizes ratings.

After preprocessing, the system constructs a graph representation, ingesting users and items as nodes and interactions as edges into a FalkorDB graph, creating a reference structure for downstream tasks. With the graph in place, the backend begins training and tuning of an expert model using the processed dataset. Afterwards, the resulting model and dataset artifacts are stored, and the agent's metadata is updated to mark the process as complete. In the case of exceptions—such as failures in preprocessing, ingestion, or training—the system rolls back by deleting temporary files, model and dataset artifacts, graph data, and agent metadata to ensure consistency. Once successful, the recommender agent becomes accessible through the API, ready for conversational recommendations and management tasks.

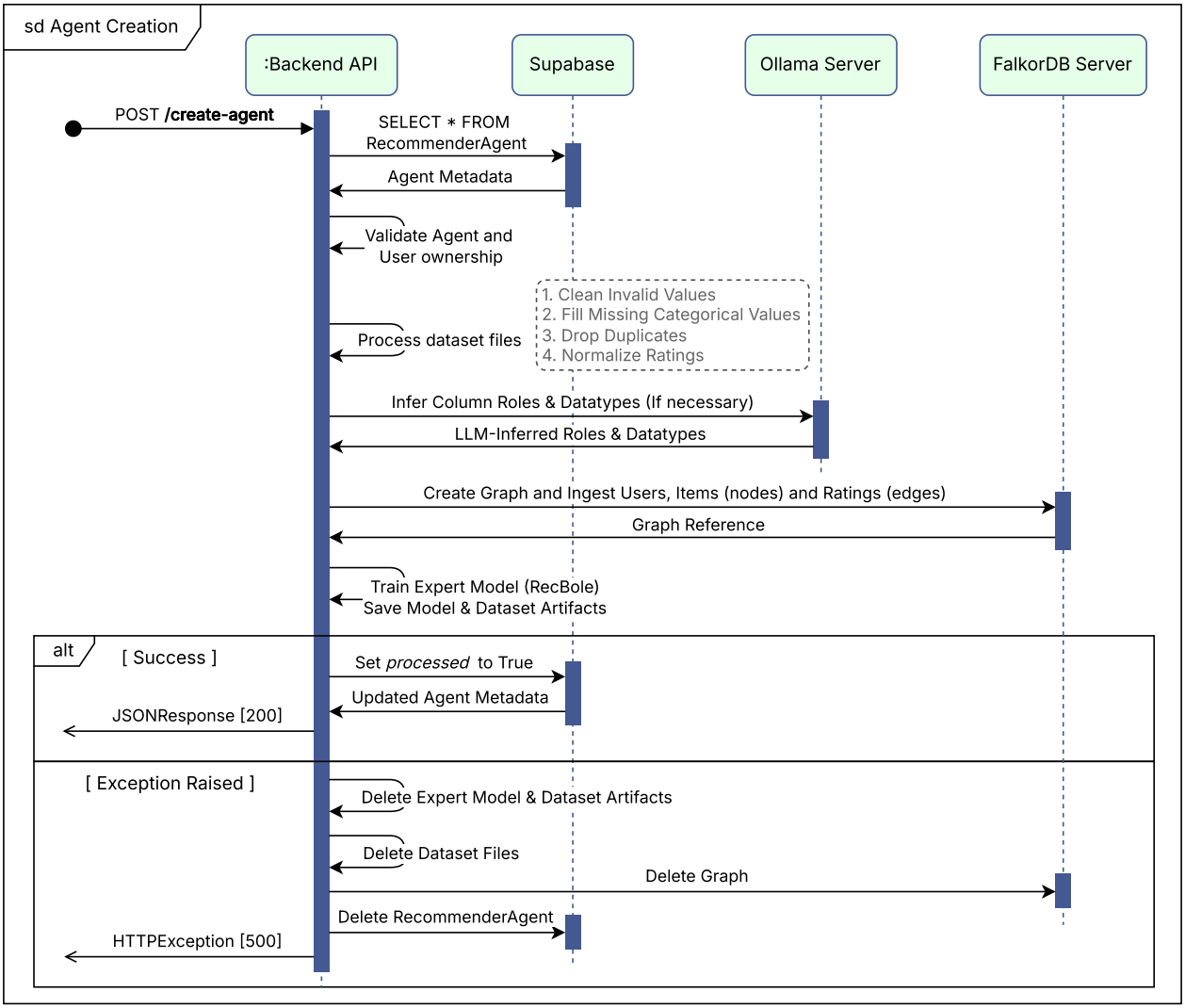


Figure 3.4: Sequence Diagram for Agent Creation.

Agent Chat Interaction Sequence

A typical agent chat interaction involves a conversation between the user, the LLM, and the underlying recommendation engines. As shown in Figure 3.5, a user's message is sent to the backend, which forwards it to the LlamaIndex workflow. The workflow may use Function Calling to invoke the recommendation module (either the graph-based or expert model) to retrieve a list of recommendations and explanations, formatted into a natural language response by the LLM and streamed back to the user. Finally, each message by the user and the agent is saved in the conversation history, just as depicted in the next diagram (Figure 3.6). The internal design and implementation details of the conversational agents are discussed in the complementary research thesis [7].

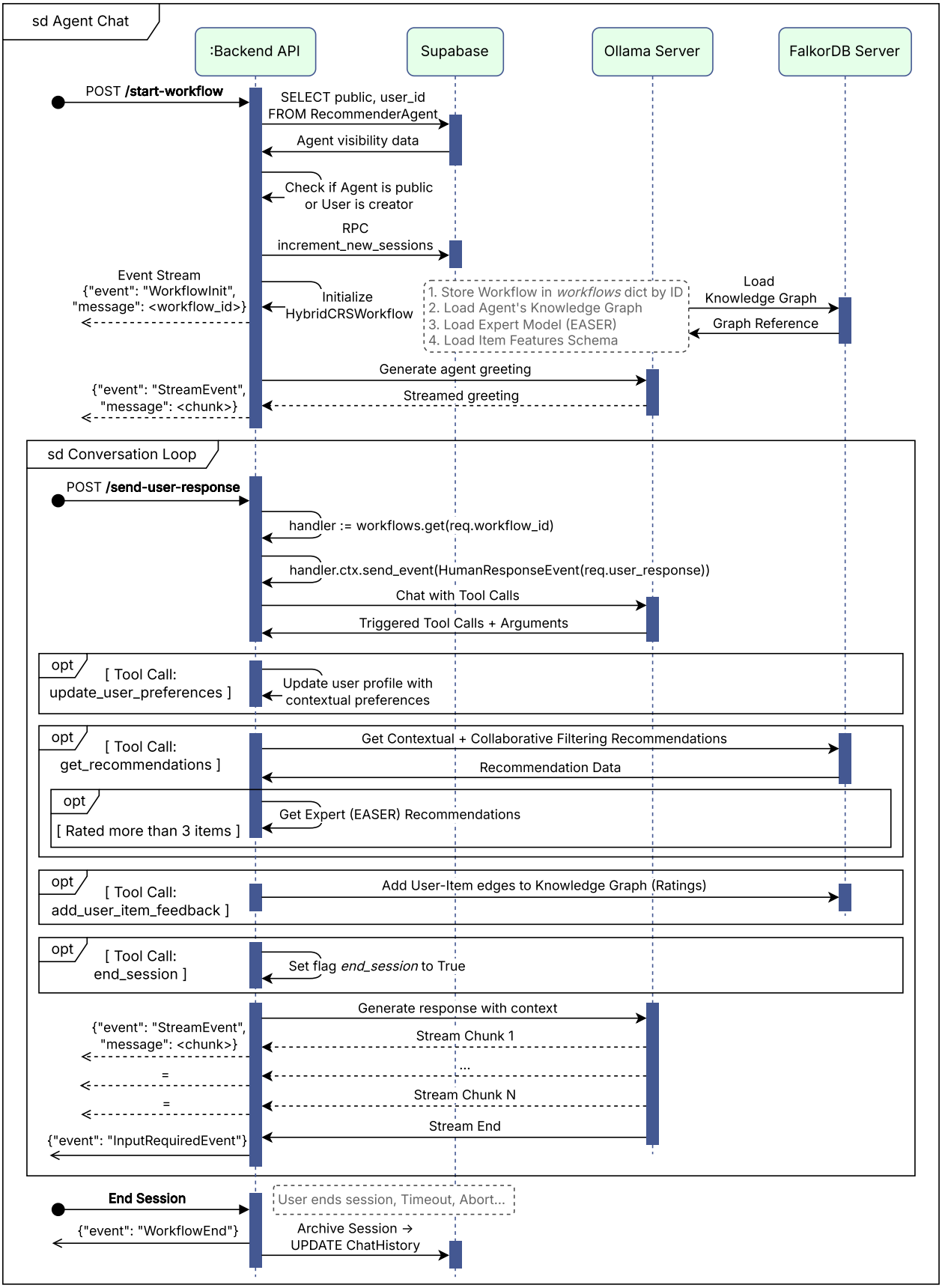


Figure 3.5: Sequence Diagram for an Agent Chat interaction.

Open Chat Interaction Sequence

This interaction describes the open chat functionality, where the user interacts freely with any LLM available in the platform. The sequence diagram in Figure 3.6 illustrates how the user sends a chat request to the backend, which then proxies it to the Ollama service. Any PDF attachments are processed beforehand through the API by extracting all the text and including it in the user message. The model generates a response and streams the reply back to the user through the backend server. Finally, each message by the user and the model is saved in the conversation history, as a node in a FalkorDB graph.

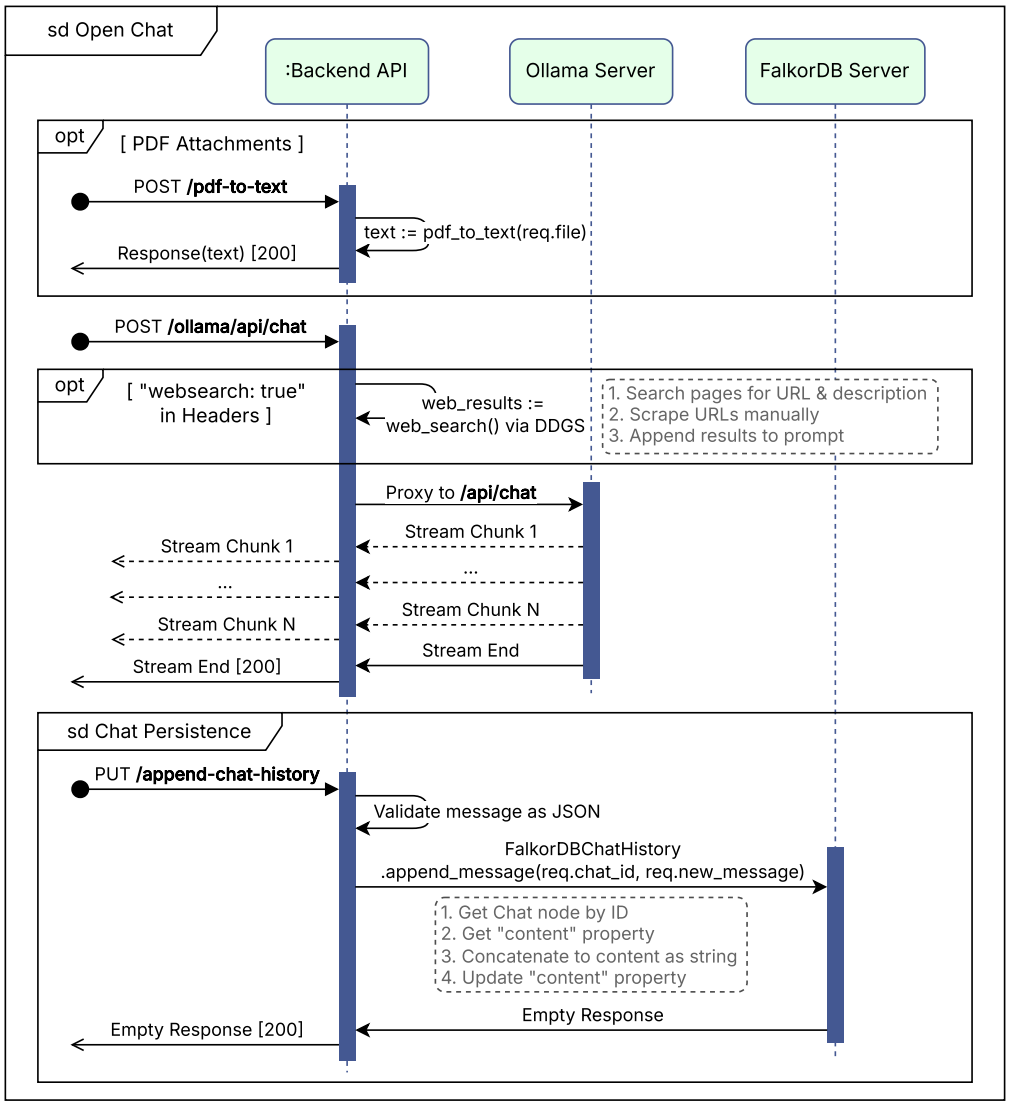


Figure 3.6: Sequence Diagram for an Open Chat interaction.

3.3.3. Entity-Relationship Diagram

The platform's primary metadata, user information, and agent configurations are stored in a relational database managed by Supabase, which runs on PostgreSQL. The logical structure of this database is represented by the physical Entity-Relationship (ER) diagram shown in Figure 3.7. This schema is designed to efficiently store and retrieve information about users, the agents they create, and their conversation histories, while also implementing specific workarounds for the platform's real-time features.

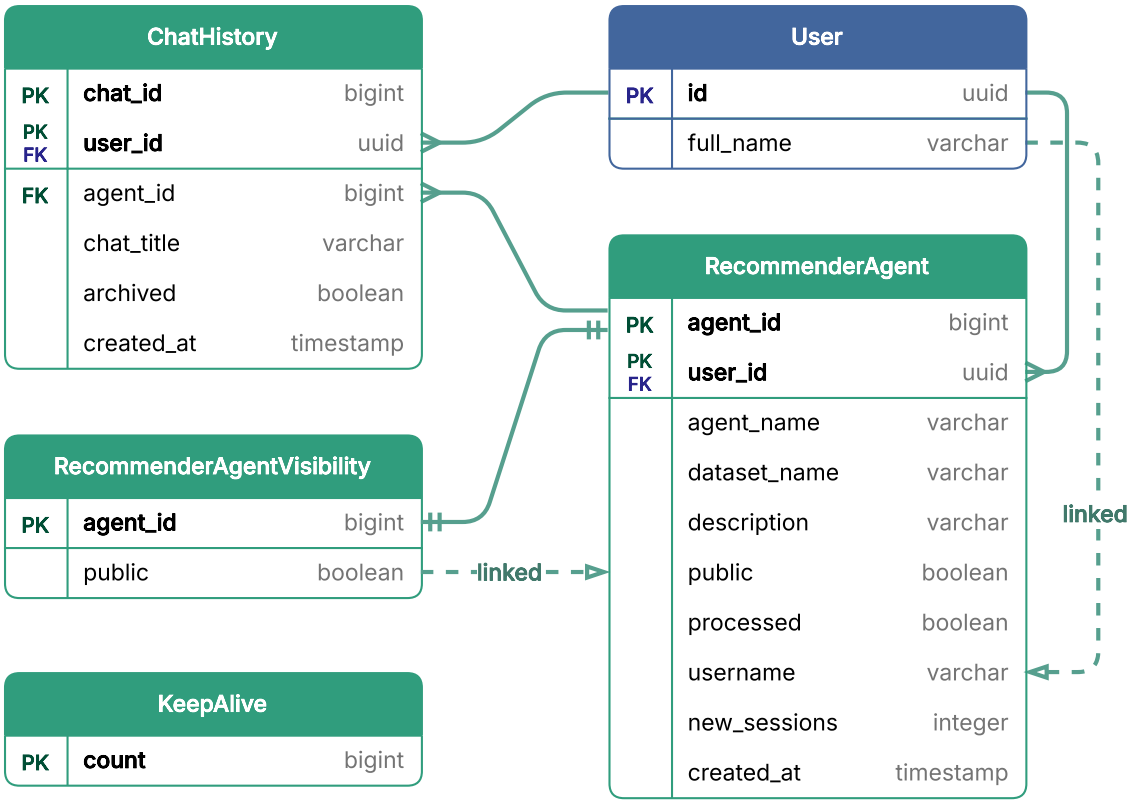


Figure 3.7: ER Diagram for the Metadata Database.

The core entities are defined as follows:

- **User:** This entity is implicitly managed by Supabase's built-in `auth.users` table. It is not defined directly in the public schema but is referenced via a `user_id` foreign key in other tables, linking data to a specific authenticated user.
- **RecommenderAgent:** This is the central table for storing all metadata associated with a created agent. It includes attributes such as the agent's name, the dataset it was trained on, its description, its processing status, and its visibility (public or private). Additionally, `new_sessions` tracks the number of new conversations users had with an agent since its last retraining, as an indicator of the agent's engagement for the creator. Each agent is directly linked to a creator via the `user_id`, and the user's username is also duplicated for public visibility.
- **RecommenderAgentVisibility:** This auxiliary table was created to address a specific challenge with

Supabase's real-time capabilities. When Row-Level Security is active, the frontend cannot listen for real-time changes on rows that a user does not have access to (e.g., when an agent becomes private). This table, which is fully public, mirrors the `agent_id` with a foreign key and the `public` status with an automatic trigger, allowing the frontend to subscribe to its changes and correctly update the UI when an agent's visibility changes.

- **ChatHistory:** This table archives the records of conversations. Each entry is linked to a `user_id`, and optionally to an `agent_id` (which is `null` for Open Chat histories), storing the conversation title and an `archived` flag to distinguish active from read-only sessions.
- **KeepAlive:** A utility table with a single auto-incrementing counter. Its sole purpose is to be updated by a scheduled daily job to prevent the database from being paused by Supabase's inactivity policy.

3.3.4. Data Flowchart

The flow of data through the system is most critical during the creation of a new recommender agent. This process transforms raw user-uploaded files into a structured, multi-faceted data asset that powers the agent's conversational and recommendation capabilities. The end-to-end data flow for this process is illustrated in the flowchart in Figure 3.8.

The process can be summarized in the following steps:

1. An entry is created in the `RecommenderAgent` table in the PostgreSQL database, and a corresponding row is inserted in the `RecommenderAgentVisibility` table through a trigger, to manage its real-time visibility on the frontend.
2. The user uploads raw CSV files (interactions, items, users) and column configuration via the frontend interface.
3. The backend API receives the files and checks the existence of the agent metadata. If the entry exists, it stores the files temporarily.
4. The Data Processing Module initiates its pipeline, performing data cleansing and normalization. Afterwards, each file is persisted in a local, unique directory.
5. The cleaned data is then used in two procedures:
   a) The FalkorDB recommender component ingests the user-item interaction data, building a property graph and computing graph metrics like PageRank.
   b) An expert model [29] is trained through RecBole on the cleaned data, and the resulting model and dataset artifacts are saved.
6. Upon successful completion of both procedures, the agent's `processed` flag is switched to `True` in the PostgreSQL database, making the agent available on the platform.

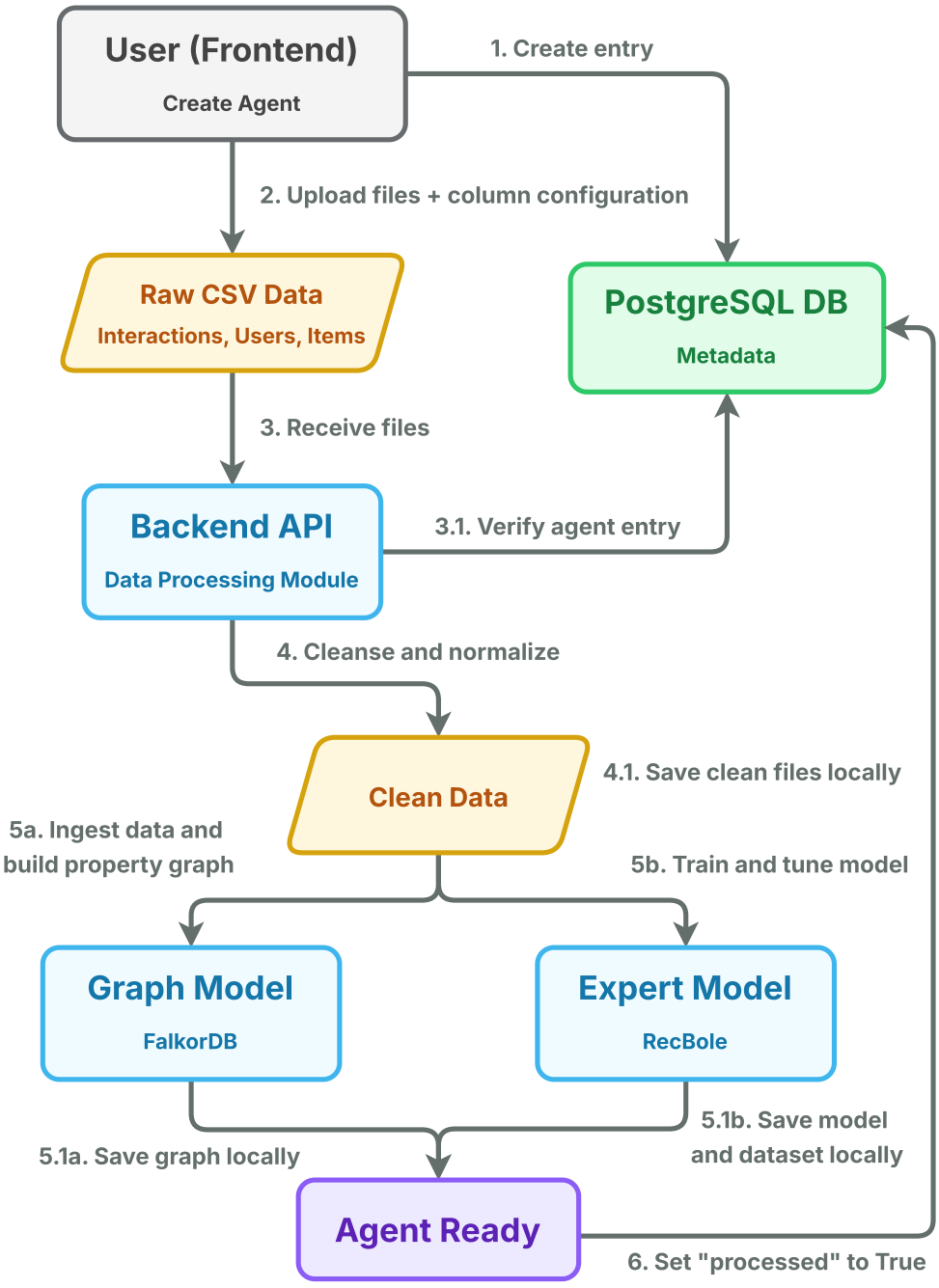


Figure 3.8: Data Flowchart for the Agent Creation process.

3.4. Frontend Design

The frontend is the primary interface between the user and the platform's complex backend functionalities. It is designed as a modern and responsive Progressive Web Application, with the core responsibility of providing a robust, intuitive, and responsive user experience, while efficiently managing client-side state and communicating with the backend API. This section outlines the design patterns and principles that guided its development, as well as the specific considerations made for accessibility and internationalization.

3.4.1. Design Patterns and Principles

The frontend architecture is built upon several key design patterns and principles to ensure it is maintainable, scalable, and provides a high-quality user experience.

Component-Based Architecture Following the core philosophy of the React library [30], the entire UI is constructed from a series of reusable, self-contained components. This promotes modularity and code reuse. The project leverages a combination of a general component library, `shadcn/ui`, for common elements like buttons and forms, and a specialized library, `assistant-ui`, for the complex conversational chat interface.

Centralized State Management To manage application-wide state, such as user session data, the currently selected LLM, or conversational context, the application employs React's Context API. A hierarchy of context providers is established at the root of the application, allowing any component to access or modify shared state without the need for "prop drilling" (passing properties down through multiple layers of components).

Responsive Design A responsive, mobile-first design principle was adopted to ensure a seamless and consistent experience across all device types, from large desktop monitors to small mobile screens. This is achieved through the use of Tailwind CSS [31], a utility-first Cascading Style Sheets (CSS) framework that facilitates the creation of adaptive layouts.

3.4.2. Accessibility

Accessibility was taken into consideration during the design of the frontend to maximize the platform's usability for the broadest possible range of users. The main accessibility enhancements are focused on the chat interface, which is the primary mode of interaction. The platform integrates the Web Speech API to provide:

- **Speech-to-Text:** This allows users to dictate their messages using their microphone, offering an alternative to keyboard input.

- **Text-to-Speech:** This enables the system to read LLM responses aloud, which is particularly beneficial for users with visual impairments.

It is noted that due to browser-specific implementations of the Web Speech API, these features offer the best performance in Google Chrome at the time of writing.

In addition to these explicit features, best practices for web accessibility, such as using semantic HTML and ensuring keyboard navigability, are largely upheld through the use of the compliant component library, `shadcn/ui`.

3.4.3. Localization and Internationalization

To cater to a global user base, the platform was designed and built with internationalization (i18n) as a core principle. The goal is to provide a broadly localized experience where users can interact with the interface in their preferred language.

This is achieved using the `next-intl` library [32], a dedicated solution for adding i18n to Next.js applications. All user-facing strings within the UI are externalized from the components and stored in separate JSON files, one for each supported language. As of the completion of this project, the platform is fully localized for seven languages: English, Spanish, German, French, Italian, Portuguese, and Chinese.

A language selector component is integrated into the application's main header, allowing users to dynamically switch their desired language. The `next-intl` library handles the loading of the appropriate message file and re-rendering the UI with the localized strings, offering a user-friendly translation experience.

4

Implementation

This chapter transitions from the architectural design presented previously to the concrete implementation of the platform. It details the translation of the system's blueprints into functional code, highlighting the specific libraries, frameworks, and patterns used to build each component.

A core principle of the implementation is **modularization through containerization**. The entire platform is designed to run within Docker containers, which encapsulates each primary service—the FastAPI backend, the Next.js frontend, the FalkorDB graph database, and the Ollama LLM server. This approach, orchestrated via Docker Compose, ensures an efficient development workflow, simplifies deployment, and guarantees portability and scalability. By isolating dependencies and standardizing the runtime environment, containerization provides a reproducible foundation for the entire system.

4.1. Backend Development

The backend was implemented as a Python application using the FastAPI web framework. It serves as the central API gateway, responsible for handling all client requests, managing the various backend services, and enforcing domain logic and authentication. The implementation closely follows the modular design outlined in the previous chapter, with a clear separation of concerns between API routing, data processing, LLM interaction, and recommendation logic.

4.1.1. LLM Integration

The conversational capabilities of the platform are powered by the integration with open-source LLMs served via Ollama. The core of this integration is the `HybridCRSWorkflow` class, which is built using the LlamaIndex framework [13]. This class defines a stateful, event-driven workflow for managing a conversation between a user and a recommender agent.

At a high level, the workflow is responsible for maintaining the conversational context, parsing user inputs, and coordinating the use of tools—specifically, the recommender functions. When a recommendation is required, the workflow triggers a function call to the appropriate method in the recommendation

module. The results are then fed back into the LLM to be synthesized into a natural language response, which is streamed back to the user. This high-level oversight abstracts the complexity of the LLM interaction. The specific design of the conversational flow, prompt engineering, and user-preference elicitation strategies are research topics explored in greater detail in the complementary thesis [7].

4.1.2. Recommender System Integration

The platform integrates a dual-component recommendation engine to provide both real-time, explainable recommendations and highly-accurate, pre-trained expert recommendations.

The graph-based component is implemented in the `FalkorDBRecommender` class. This class interfaces directly with the FalkorDB graph and provides methods for contextual, collaborative filtering, and hybrid recommendations. Importantly, it also implements a method to generate explanations for any given recommendation, via Cypher queries that obtain liked items with similar features, neighbors for collaborative evidence, and general popularity.

The expert model component is managed by a set of utility functions, which use the RecBole [19] library—a comprehensive framework for building and evaluating Recommender Systems built on the PyTorch [33] machine learning library—to handle the lifecycle of the model. Specifically, the model of choice is the Embarrassingly Shallow Autoencoder for Sparse Data (`EASER`) [29], a surprisingly simple RS model which excels at making recommendations by finding hidden patterns in sparse user data. For instance, the `hyperparam_grid_search` function automates the tuning process with exhaustive search, and the `retrain_on_dataset` function is used to train the final model on the full dataset. These models are then loaded for prediction when required by the conversational agent. A detailed comparative analysis of these models is presented in the complementary thesis [7].

4.1.3. Automated Data Preprocessing

The automated data preprocessing pipeline is a critical component that enables the platform to ingest heterogeneous datasets and transform them into a standardized format. The core logic was designed as part of the Data Processing Module, is present in the majority of the data flow depicted in Figure 3.8, and is one of the fundamental components in the creation of recommender agents. Therefore, the pipeline is designed to be robust and flexible, suitable for a wide variety of datasets while minimizing the need for manual intervention. This ensures that users can easily prepare their data for training recommender agents, regardless of its initial structure or quality.

The pipeline executes a series of steps upon receiving new dataset files from the user, starting from the frontend interface and ending with the processed data being saved to the backend:

1. **Column Role Inference:** For user convenience, the system first attempts to infer the semantic

roles of columns (e.g., user ID, item ID, rating). This is achieved through a specific API endpoint using helper functions which send the column names to an LLM and ask it to return the roles in a structured JSON format.

2. **Data Type Inference:** Similarly, a utility function infers the data type (e.g., `token`, `token_seq`, `float` & `float_seq`) for each column by sending a sample of its values to the LLM. After this step, the user may modify column roles and types through the frontend interface, if necessary, before sending the dataset files and column configuration to the backend.
3. **Normalization:** For the `ratings` column, a normalization function scales all values to a consistent range of [0, 5]. Furthermore, list-like columns are standardized into a space-delimited format (e.g., ["item1", "item2", "item3"] is converted to "item1 item2 item3").
4. **Data Cleansing:** Data cleansing is performed by making use of the *AutoClean* library [20] to handle common data quality issues like duplicates and missing values.
5. **Standardized Output:** Finally, the processed, cleaned, and standardized `DataFrames` are saved to a specific directory with a unique filename based on the dataset name and agent identifier, ready to be ingested by the recommendation and graph database modules.

It is worth mentioning that, for the task of DataFrame processing (reading, filtering, manipulating...), `FireDucks` [34]—a library for efficient DataFrame operations by leveraging multithreading and compiler optimization—was used, in conjunction with `Pandas` [35] for compatibility. `NumPy` [36] was also utilized for optimized numerical operations and array manipulations.

4.1.4. API Endpoints

The backend exposes a comprehensive RESTful API using FastAPI. Request and response bodies are strictly typed and validated using Pydantic models defined in `schemas.py`, with model classes detailed in Appendix A. This ensures data integrity and consistency between the frontend and backend. The main endpoints are grouped by functionality:

Agent Management: The `/create-agent` endpoint handles the complex, multi-part form data containing the agent configuration and uploaded dataset files. It triggers the asynchronous data processing and model training pipeline. The `/delete-agent` and `/retrain-agent` endpoints manage the lifecycle of an existing agent.

Conversational Workflow: The `/start-workflow` endpoint initiates a new conversational session with an agent. It returns a `StreamingResponse` that allows the server to push events (such as LLM tokens or state changes) to the client in real-time. The `/send-user-response` endpoint allows the client to inject the user's reply back into the running workflow.

Chat History Management: To manage chat histories, a set of endpoints were implemented.

The `/create-chat-history`, `/get-chat-history`, `/append-chat-history`, and `/delete-chat-history` endpoints handle the creation, retrieval, updating, and deletion of conversation logs, respectively.

LLM Proxy: A general-purpose proxy endpoint, `/ollama/api/{endpoint}`, securely forwards requests to the Ollama server. This allows the frontend to interact with the LLM (e.g., to list available models or pull new ones) without exposing the Ollama service directly. It also intercepts chat requests to inject context from web search, by using the DDGS metasearch library [37].

Data Utilities: Helper endpoints like `/infer-column-roles` and `/pdf-to-text` provide utility functions to the frontend, offloading tasks like LLM-based inference or file parsing with PyMuPDF [38] to the backend.

4.1.5. API Documentation

A significant advantage of using FastAPI is its native support for the automatic generation of interactive API documentation. By leveraging Python type hints and the Pydantic models defined for each endpoint, FastAPI automatically creates a detailed OpenAPI specification for the entire backend.

This specification is then used to render an interactive documentation interface (Swagger UI), available at the `/docs` endpoint. To enhance security and usability, a custom function `auth_openapi` was implemented to inject a JWT bearer token authentication scheme directly into the Swagger UI, as shown in Figure 4.1. This allows authenticated users to view their access key on the frontend and use it to test every API endpoint directly from their browser, providing a powerful tool for debugging and development.

4.2. Frontend Development

The frontend of the platform was implemented as a modern, responsive, single-page PWA using the Next.js 15 framework and TypeScript. The primary goal was to create a clean, intuitive, and highly interactive user interface that effectively abstracts the complexity of the underlying backend services. In the main repository, the frontend code is located in the `frontend` directory as a submodule, which allows for independent development and deployment processes.

Development followed best practices for code quality and consistency. The project was configured to use Husky, a tool that manages Git hooks, to run automated scripts before each commit. Specifically, a pre-commit hook was set up to trigger Prettier for code formatting and ESLint for code linting. This automated process ensures that all contributions to the codebase adhere to a consistent style guide and are free from common syntactical errors, significantly improving code readability and maintainability.

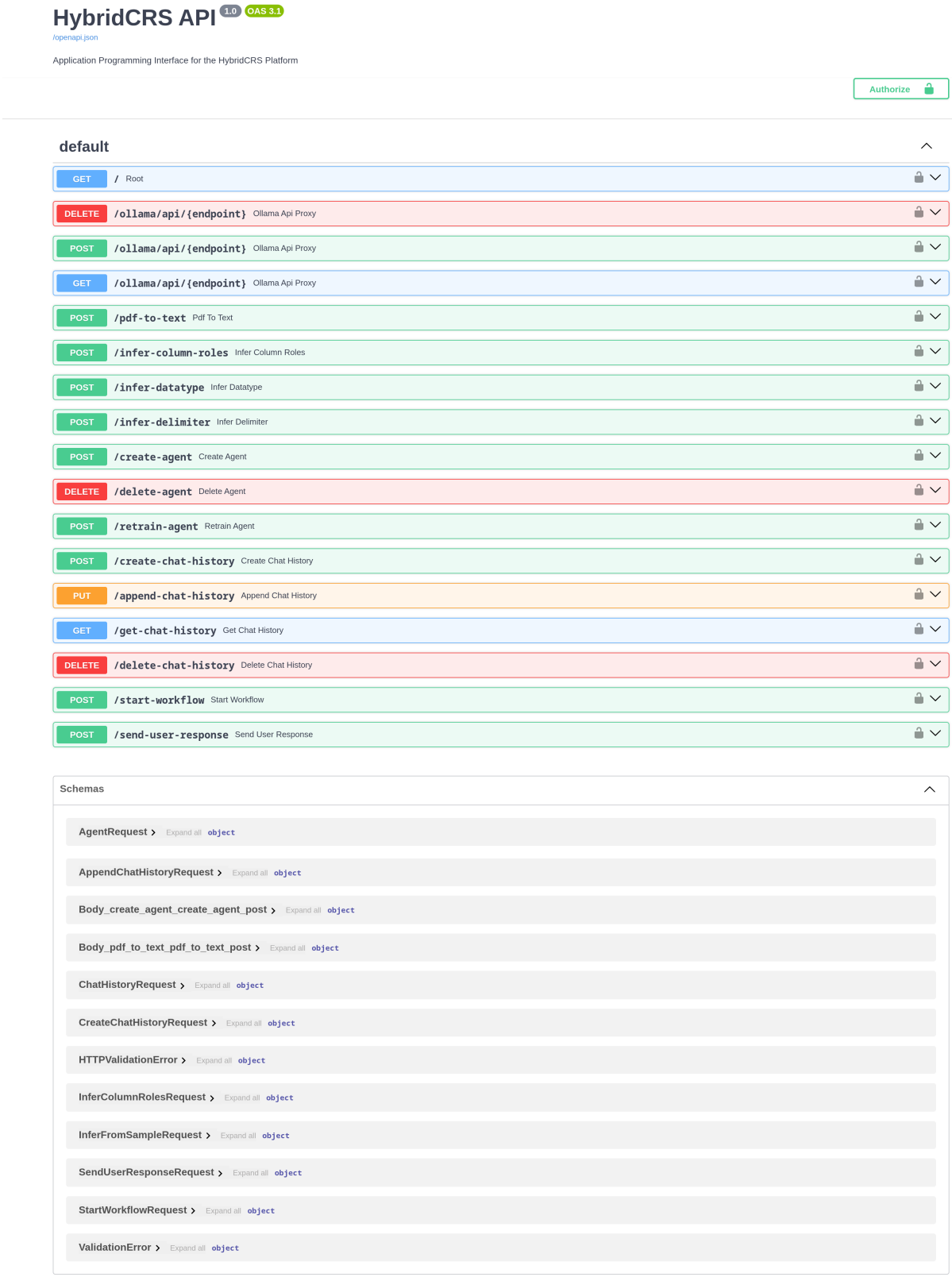


Figure 4.1: Screenshot of the interactive Swagger UI with authorization enabled.

4.2.1. UI Components

The user interface is constructed from a hierarchy of reusable React components, following a component-based architecture. This modular approach was facilitated by two main libraries: `shadcn/ui` and `@assistant-ui/react`.

For the main application shell and dashboard views, `shadcn/ui` was used extensively. This library provides a set of styled, accessible, and composable components that serve as the foundational building blocks. The interface is wrapped by a sidebar with navigation links to the main sections of the platform, such as “Agent Hub”, “Open Chat”, and “Create Agent” (as well as an “API Documentation” external link). The lower section of the sidebar contains a menu for the user profile, to reset their password, see their access key for the documentation page, and log out. The dashboard also includes a header on top of the page, with a button to toggle the sidebar, breadcrumbs to aid in the navigation, and three icon menus on the right side: one for LLM selection, one for language selection, and another for theme selection (dark, light or system mode).

The platform includes several forms for authentication, password reset, and agent creation. These forms utilize `shadcn/ui`’s form components, which are built on top of `react-hook-form` for efficient form state management. Validation was implemented using `Zod` [39], a TypeScript-first schema declaration and validation library, ensuring that all user inputs have correct values before being processed.

The main `AgentHub` view features the `AgentCard` component, which is built by composing `shadcn/ui`’s `Card`, `Button`, and `Tooltip` components to create a custom UI element. These cards are then orchestrated within a grid display of agents in the view, which provides a comprehensive dashboard for browsing, searching, and managing all available recommender agents, as shown in Figure 4.2.

To enhance usability, the state of the Agent Hub’s search, filter and paging controls is persisted in the URL’s query parameters. This implementation allows users to bookmark a specific filtered view or share a direct link to it with others, ensuring a consistent and reproducible experience. Each search query is also debounced to prevent excessive query requests. Additionally, all searching, filtering and paging operations are executed as database queries, ensuring scalable and efficient retrieval.

For the conversational interfaces, the `@assistant-ui/react` library was of significant value. It provides a set of primitives specifically designed for building AI chat applications. The `Thread` component leverages these primitives to create the entire chat experience, including the message list, the composer for user input, and action bars for interacting with messages. This results in a feature-rich and intuitive conversational interface, as illustrated in Figure 4.3 and Figure 4.4. Particularly for the Agent Chat view, a custom component was developed to enhance the user experience for displaying recommendations, allowing the user to rate each recommended item with a rating slider, and sending the feedback to the backend as a JSON string.

For the creation of agents, the `CreateAgent` view is designed to be user-friendly and intuitive, as

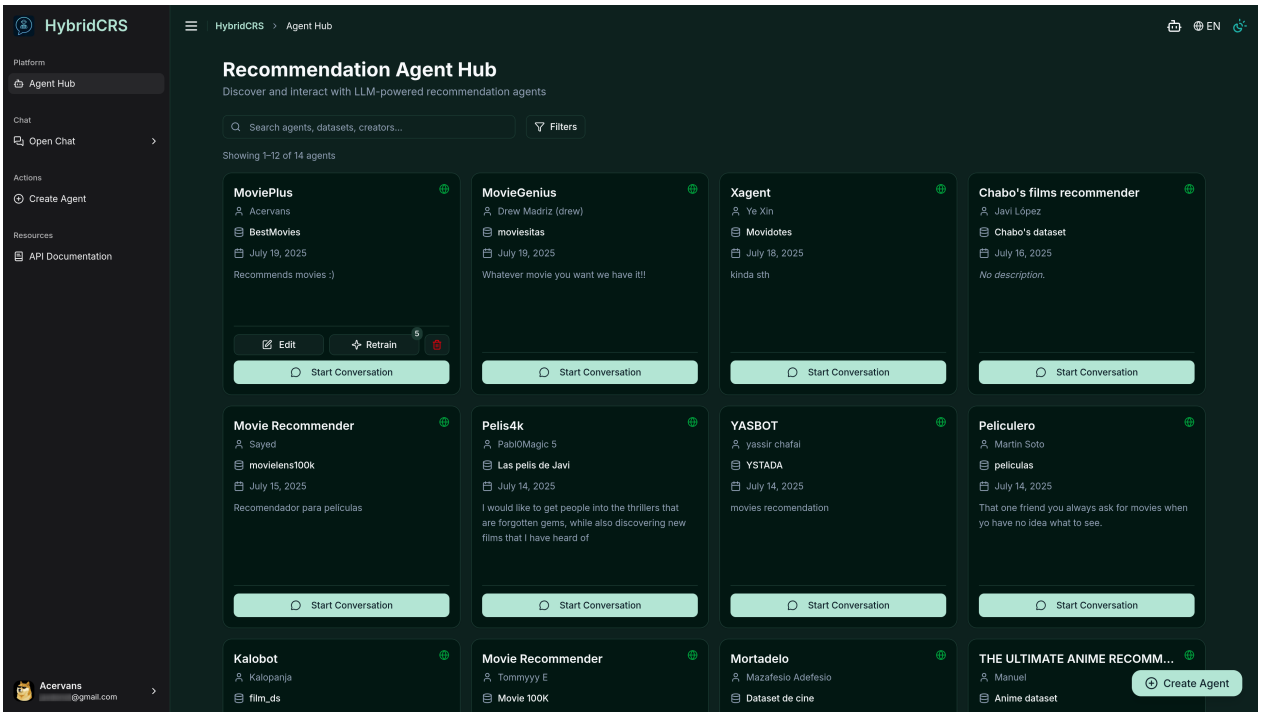


Figure 4.2: The Agent Hub, the platform's main dashboard.

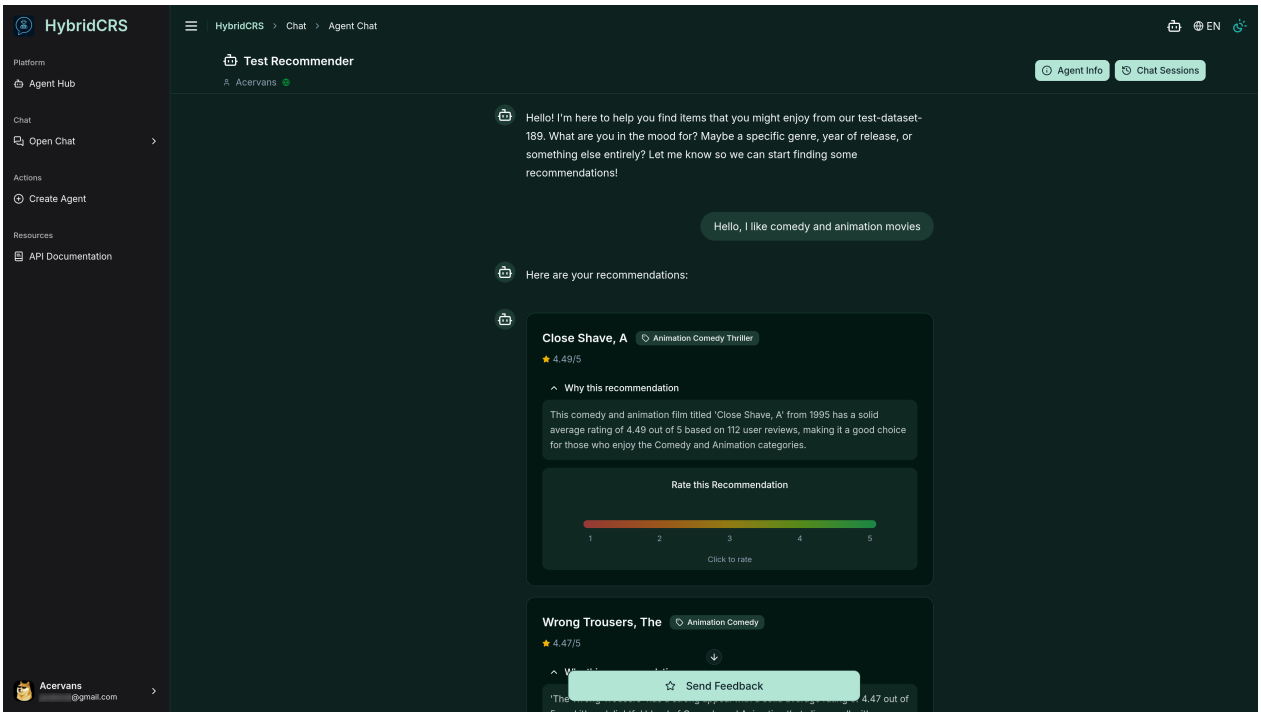


Figure 4.3: The main conversational interface for interacting with an agent.

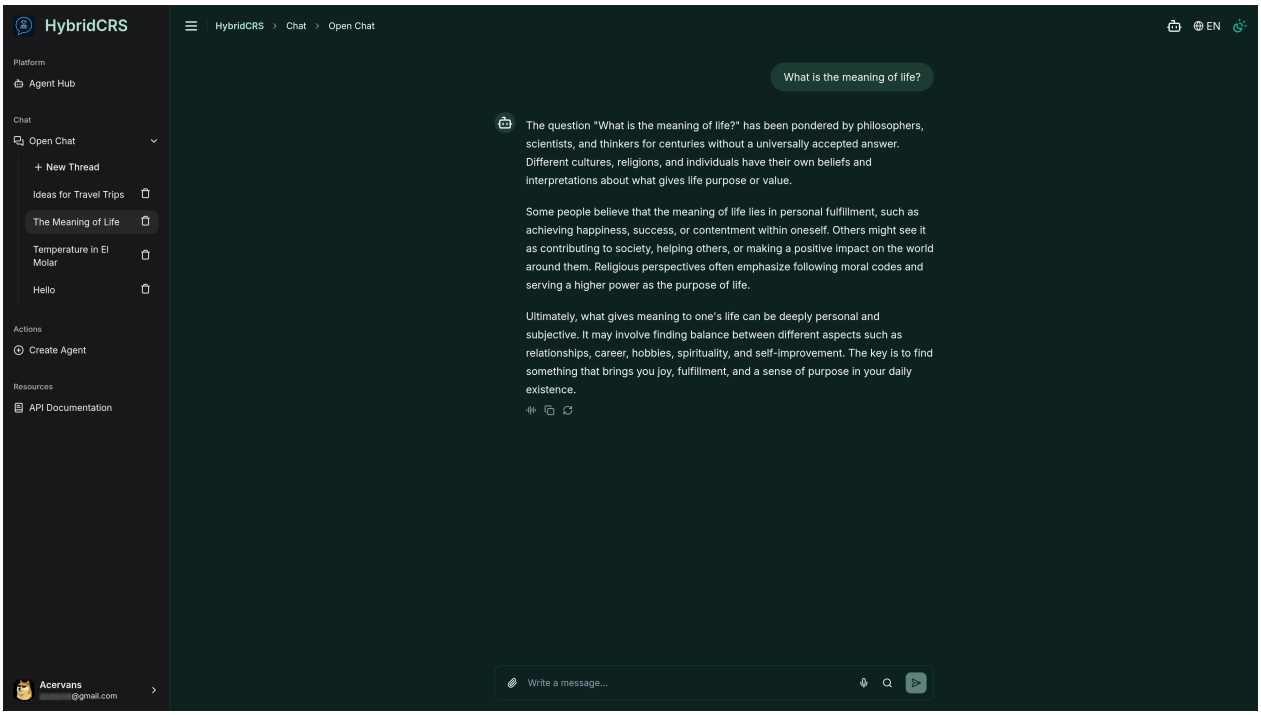


Figure 4.4: The main conversational interface for open conversations with LLMs.

displayed in Figure 4.5. It guides users through the three steps, firstly to define an agent's name, dataset name and description; secondly to upload the datasets to be used for the graph and expert Recommender Systems, with interactive previews of the data to configure the data type and column role of each column; and lastly to review and confirm the agent's configuration before creation.

More views of the platform's user interface can be found in Appendix B, with mobile screenshots of the main views included in Appendix C showcasing the PWA's responsive design.

4.2.2. State Management

To manage the application's global state in a clean and scalable manner, the implementation relies heavily on React's Context API. Instead of a single, monolithic state store, a series of distinct, domain-specific contexts are used to provide shared state to different parts of the component tree. This approach ensures a clear separation of concerns, resulting in a more performant and maintainable state management architecture. Each context is defined in its own module, encapsulating its state, actions, and provider component. The main contexts implemented for the platform include:

- **AssistantProvider:** One of the main providers that wrap the dashboard, providing access to all contexts. It manages the state of the active conversations, threads, and persistence of chat histories.
- **SupabaseContext:** Provides the Supabase client instance and user authentication state throughout the application.

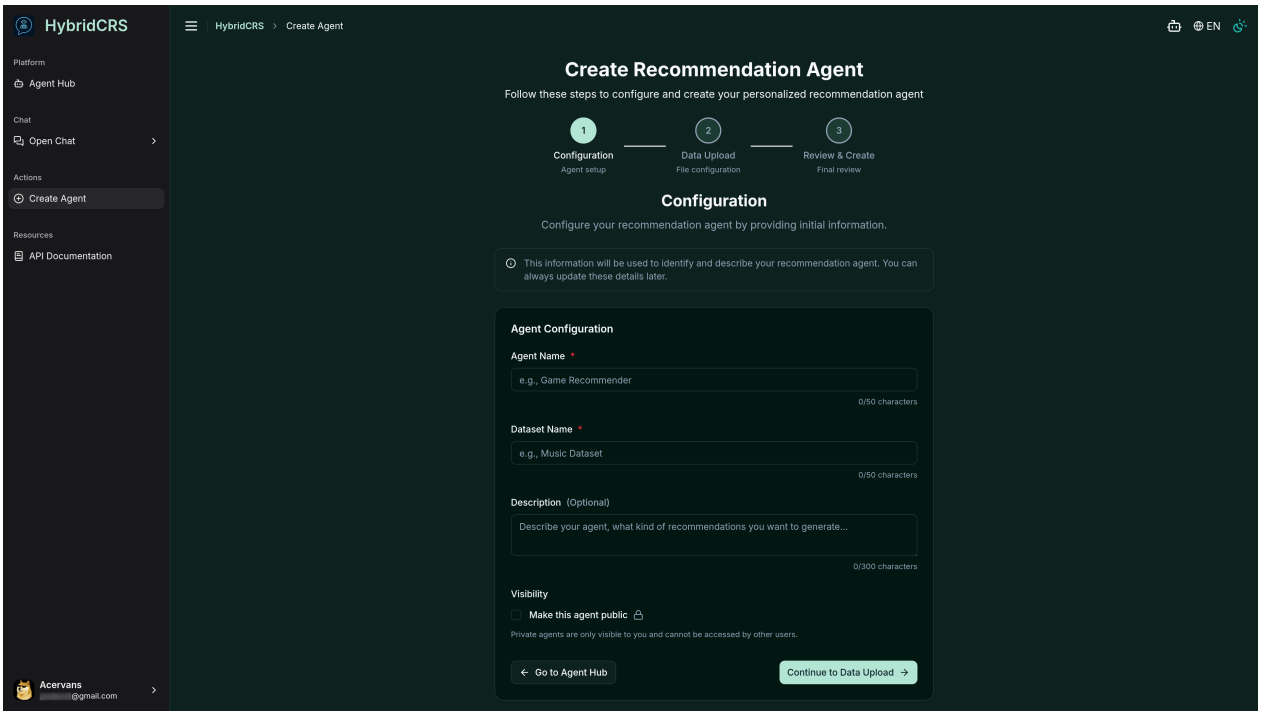


Figure 4.5: Step one of the agent creation view.

- **WorkflowContext:** Responsible for managing the state of an active conversation with a recommender agent, tracking the `workflowId`, handling the streaming of events from the backend, and managing user feedback on recommendations. It is the foremost context in the dashboard and wraps the `AssistantProvider`.
- **DataContext:** Handles the agent creation workflow and is responsible for tracking user-uploaded data files, the agent's configuration details (e.g., name, description), and the submission process.
- **ModelContext:** Holds the state for the currently selected LLM and the active recommender agent.
- **LocaleContext:** Manages the currently selected language for internationalization.

In addition to client-side state, the platform implements real-time state synchronization with the database. Through Supabase's real-time capabilities, the frontend subscribes to database changes in the `RecommenderAgent` table. This ensures that when an agent is created, edited, or deleted, the Agent Hub view updates immediately for all connected clients without the need for manual refreshes, creating a live and responsive environment.

4.2.3. API Integration

All communication between the frontend and the backend is handled through the RESTful API exposed by the FastAPI server. The frontend implements a set of client-side functions, primarily located in `src/lib/api.ts`, which encapsulate the logic for making *fetch* requests to the various backend

endpoints. Also, all the interactions with the Supabase database tables, whether to add, edit or delete records, are implemented in `src/lib/supabase/client.ts`, clearly separating backend from Supabase functionalities (mostly used for metadata).

Notably, the integration is found in the `AssistantProvider`. For the Open Chat view, it uses the `streamChat` function to send user messages to the backend's Ollama proxy endpoint, and receive a streamed response. For agent-specific conversations, the `startWorkflowApi` function is used to initiate a new session. The `WorkflowContext` then manages the real-time event stream from the server, parsing events to display streamed text or new recommendations.

The Agent Hub view also demonstrates this integration. When a user performs an action like deleting or retraining an agent, it calls the corresponding functions (`deleteAgent`, `retrainAgent`), which handle the authenticated requests to the backend asynchronously, providing a seamless user experience while the backend performs the necessary tasks.

4.2.4. Authentication and Authorization

Authentication and authorization are managed using Supabase's built-in authentication services, which provide a secure and robust solution for user management based on JSON Web Tokens (JWT). The platform supports user registration and login via both email/password credentials and an OAuth provider for Google, as shown in the authentication form in Figure 4.6.

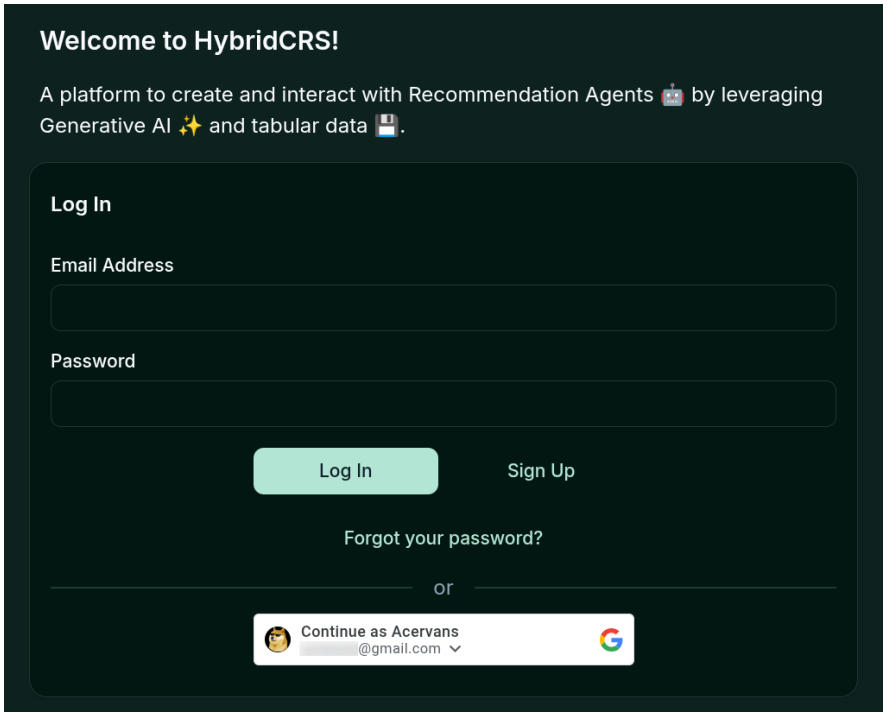


Figure 4.6: The user authentication form for logging in.

When a user logs in, the Supabase library handles the authentication flow server-side, for bet-

ter protection of sensitive data such as API keys and session tokens. Afterwards, the resulting JWT (containing the user's credentials) is securely stored in the browser as a cookie. For every subsequent request to the backend API, this JWT is retrieved client-side and included in the `Authorization` header as a bearer token.

The `SupabaseContext` is convenient in this process by providing the `getAccessToken` function. This function, used throughout the application before making an API call, ensures that a valid and fresh access token is always available. On the backend, a custom FastAPI middleware verifies the token, ensuring that all protected endpoints are only accessible by authenticated users.

4.3. Deployment

The deployment strategy for the platform is a hybrid approach designed to take advantage of the strengths of both cloud hosting and local infrastructure while managing resource constraints. The frontend application is deployed globally via a cloud platform optimized for web applications, while the resource-intensive backend services are run locally and securely exposed to the internet. This section details the containerization strategy, the deployment processes for the frontend and backend, and the approach to Continuous Integration and Continuous Deployment.

4.3.1. Docker Containerization

Containerization with Docker [40] is the cornerstone of the platform's deployment strategy, ensuring consistency, portability, and reproducibility across development and production environments. Docker also improves security and scalability by isolating each service in its own container, reducing dependency conflicts and limiting the spread of vulnerabilities. The entire backend stack is defined as a set of services in a `docker-compose.yaml` file, which orchestrates the FastAPI application, the Ollama LLM server, and the FalkorDB database instance (as well as the Next.js frontend application during development).

The backend's `Dockerfile` utilizes multi-stage builds to create an optimized and lightweight final image. A **deps** stage is used to install Python dependencies, and the final **runtime** stage copies only the necessary application code and installed packages. This practice leverages Docker's layer caching, which automatically triggers a rebuild of the image only when source code or dependencies are modified. Furthermore, remote service images like Ollama can be easily kept up-to-date by pulling the newest versions from image repositories (e.g., Docker Hub), ensuring the system runs on the latest stable versions.

4.3.2. Frontend Deployment

The Next.js frontend application is deployed on Vercel [23], a cloud platform specifically designed and optimized for hosting modern web applications and frontend frameworks. Vercel provides a seamless deployment experience by integrating directly with the project's GitHub repository. This integration facilitates a continuous deployment workflow, with every push to the main branch of the repository automatically triggering a new deployment on Vercel. The platform handles the entire build process, including installing dependencies, running the Next.js build, and deploying the resulting static and server-rendered assets to its global edge network. This ensures that the live application is always synchronized with the latest version of the code, providing quick iteration and delivery of new features and fixes.

4.3.3. Backend Tunneling

Due to the significant computational and memory requirements of the backend services (particularly the LLM, graph database and recommendation models) and the cost limitations of free-tier cloud hosting, the backend stack is run on local hardware. To make these local services accessible to the publicly deployed frontend on Vercel, a secure tunneling solution was implemented.

The open-source tool Zrok [41] was chosen for this purpose. Zrok provides a secure way to create a public URL for proxying requests to a local web service. It was preferred over alternatives like Ngrok due to its more generous free-tier resources. The process involves a few simple steps: first, enabling the Zrok environment with an account token; second, reserving a unique and persistent public name (and domain) for the backend service; and finally, initiating the share, which creates a secure tunnel from the public Zrok URL to the locally running FastAPI server at a specific port.

4.3.4. CI/CD Pipeline

The project adopts a pragmatic and differentiated approach to CI/CD for the frontend and backend.

For the **frontend**, a full CI/CD pipeline is implemented through its integration with Vercel. On every push to the main branch, Vercel's pipeline automatically lints, builds and deploys the application. As mentioned previously in Section 4.2, code quality is enforced locally before commits via a Husky pre-commit hook that runs ESLint and Prettier to ensure consistent code style and linting.

For the **backend**, a decision was made not to implement a fully automated CI/CD pipeline due to the overhead of configuring and maintaining such a pipeline outweighing its benefits. Instead, code quality and integration are ensured through robust pre-commit hooks. These hooks automatically format Python code with `black`, and run the entire test suite using `pytest`. This local CI process is sufficient for maintaining code quality. Furthermore, the backend's dependency on stateful services like Ollama makes automated testing in a standard CI runner environment prohibitively complex and costly.

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 domain logic, which encompasses data processing, recommendation generation, and conversational state management, was thoroughly evaluated with unit and integration tests to ensure its reliability, developed and tested using the `pytest` library [42]. Excerpts of each test are provided in Appendix D.

Unit Tests

The data processing pipeline, implemented in a utility module, was validated through `test_data_utils.py` (Code D.3). 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` (Codes D.4, D.5 & D.6). 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` (Codes D.7 & D.8) script tests the `UserProfile` class for managing user preferences mid-conversation, while `test_falkordb_chat_history.py` (Code D.9) ensures the reliable persistence of conversation logs in the graph database.

Integration Tests

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` (Codes D.1 & D.2), use the `pytest-mock` library to isolate the API from its downstream dependencies.

This mocking strategy is essential 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 token validation, allowing the focus to remain on the endpoint's logic.

5.2.2. 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 [43], 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 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 Laptop 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 [44], 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

Usability testing of the HybridCRS platform demonstrated strong performance in user experience, usability, and recommendation quality. Eleven participants, primarily male and technologically experienced, completed three core tasks involving agent creation and conversation-based recommendations. Tasks were deemed relatively easy (1.00–1.55 out of 5, where 1 is very easy and 5 is very hard), and feedback highlighted the platform's intuitive design, relevance of recommendations, and stability.

The system achieved a high overall usability score, with a System Usability Scale (SUS) [45] score of 92 out of 100 and an average general rating of 9.36 out of 10. Users appreciated the combination of structured and conversational agents, though suggested improvements include better context interpretation, clearer labeling of recommendation sources, and onboarding guidance.

For full details of the usability tests, including methodology, constructive feedback, and the designed questionnaire, please refer to the complementary research thesis [7].

5.5. Comparative Analysis

To contextualize the contributions of this thesis, this section provides a comparative analysis, positioning the developed platform against the state-of-the-art commercial and open-source systems. The comparison is structured around two main areas: the user-facing interface and the underlying backend architecture.

User Interface and Interaction Paradigm

The platform's frontend was designed to incorporate and synthesize several advanced paradigms observed in leading conversational platforms.

- **Core Chat Interface:** The fundamental UI, with its sidebar for chat history and a central message viewport, adopts the foundational design paradigm popularized by **ChatGPT** [2]. Like ChatGPT, it supports file uploads and advanced interaction beyond simple text.
- **Self-Hosted Philosophy:** Architecturally, the platform aligns closely with the ethos of **Open WebUI** [46]. By using Docker and supporting locally-hosted models via Ollama, it provides a self-hostable, privacy-preserving alternative to closed, proprietary systems.
- **User Control and Context Scoping:** The ability for a user to create and chat with distinct agents, each trained on a specific dataset, implements a core principle of user control. This is conceptually similar to **Perplexity AI**'s "Spaces" [47] or **Google Gemini**'s "Gems" feature, which allows users to create personalized agents with specific contexts, leading to more tailored results.
- **Collaborative Workspace:** Critically, the platform moves beyond a simple chat interface and successfully implements a Collaborative Workspace. It features a dedicated agent creation page that serves as a dataset management workspace, allowing users to upload and configure the data that powers their agents, before sharing them with others. This directly aligns with **Claude Artifacts**, which allows users to share ideas created with Claude.

Backend Architecture and Development Framework

The backend architecture and choice of development framework align with modern best practices for building LLM-powered applications.

- **Agent Framework:** The implementation of the conversational agent uses **LlamaIndex**, a state-of-the-art, data-centric framework that excels at creating knowledge-intensive applications. This is a more flexible and powerful approach for a platform-oriented project compared to using more rigid, specialized academic toolkits like CRSLab [48].
- **Modular Design:** The system's architecture is represented by the modular "Text-in, Text-out" philosophy seen in toolkits like **RecWizard** [49]. Instead of using a simple RAG pipeline, the project uses

a LlamaIndex workflow to orchestrate a hybrid recommendation strategy. The conversational agent does not retrieve recommendations directly; rather, it uses function calling to intelligently invoke external, specialized tools: a pre-trained EASER expert model managed by RecBole, and a real-time, explainable recommender powered by the **FalkorDB** graph database. This multi-component approach demonstrates an advanced and robust implementation of modern conversational AI concepts.

6 Conclusions and Future Work

6.1. Conclusions

This Master's Thesis set out to address the engineering challenges of integrating modern LLMs into a scalable and user-friendly platform for creating CRS agents. The primary contribution of this work is the successful design, implementation, and evaluation of a complete, end-to-end, full-stack platform that achieves this goal.

The project successfully met its core objectives. A modular, containerized architecture was designed and implemented, providing a clear separation between the Next.js frontend, the FastAPI backend, and the various AI and data services. The backend provides a robust and high-performance API that manages the platform's complex domain logic, while the frontend delivers a modern, responsive user experience that incorporates state-of-the-art UI frameworks and design patterns. An automated data processing pipeline was developed to handle the ingestion and preparation of user-provided datasets, and the entire system's portability and reproducibility are ensured through the use of Docker containers. Performance and unit testing have validated the platform's efficiency and reliability.

Ultimately, this project delivers a tangible and functional platform that serves as a strong foundation for both practical application and future research. It successfully joins the theoretical potential of conversational AI and the practical engineering required to build a deployable system, providing a valuable tool for anyone looking to create and interact with personalized, conversational recommender agents.

For those interested in the more research-focused aspects of the project, the complementary thesis [7] provides a detailed exploration of the conversational workflow, the underlying algorithms for recommendation and explanation, along with the usability evaluation's procedure and results.

The project repository can be found at `https://github.com/Acervans/Hybrid-CRS`. It contains all the code developed for the project, along with steps to set up the FastAPI backend, the Next.js frontend, and the necessary services. The web application is deployed at `https://hybrid-crs.vercel.app`, though it must be noted that, for the full functionality to be available, the backend must be running locally and accessible at a reserved domain through Zrok.

6.2. Future Work

While the current platform provides a robust and functional foundation, there is still much room for future development that could further enhance its capabilities, scalability, and user experience. The following points outline potential directions for future work, many of which are informed by the feedback gathered during usability testing.

The platform could be extended with more advanced administrative features to increase its flexibility. This could include:

- **Role-Based Access Control (RBAC):** Implementing a more granular role system with distinct permissions for an `Admin`, an `Agent Creator`, and a `Regular User`. An administrator could manage platform-wide settings, such as the available LLMs and expert recommender models.
- **Custom Recommender Configuration:** Allowing advanced users or administrators to directly edit the RecBole YAML configuration for an agent, providing fine-grained control over the expert model's hyperparameters.

The current implementation of the conversational workflow relies on an in-memory dictionary to map workflow instances, which constrains the backend to a single worker process. To support a larger number of concurrent users, the architecture could be evolved to:

- **Migrate to Kubernetes:** Deploying the services on a Kubernetes cluster to enable multi-node horizontal scaling and load balancing.
- **Stateful Connection Protocol:** To overcome the single-worker limitation, the communication for conversational workflows could be transitioned from standard HTTP requests to a persistent connection protocol like WebSockets. This would ensure that all messages within a single user session are routed to the same process, even in a multi-container environment.

Based on feedback from the usability study, several UI/UX improvements could be implemented:

- **Item Knowledge Confirmation:** Enhancing the feedback mechanism to include a "No opinion" option, or transitioning to a rating slider only after a user confirms they have seen an item.
- **Recommendation Tagging:** Adding tags or labels to recommendations to clearly indicate their source (e.g., from contextual preferences, collaborative filtering, or the expert model).
- **User Onboarding:** Implementing a brief tutorial or context-sensitive info tooltips, particularly for the multi-step agent creation process, to guide non-technical users.

These enhancements would significantly strengthen the platform's usability, scalability, and long-term adaptability. The platform's modular architecture, combined with the use of containerization solutions, positions it well for future expansion and integration with emerging technologies in the field of Conversational Recommender Systems.

Bibliography

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

[2] OpenAI, "ChatGPT." `https://www.chatgpt.com/`, 2023.

[3] Google DeepMind, "Gemini." `https://www.deepmind.com/research/`, 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." `https://www.anthropic.com/`, 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] Subapase contributors, "Supabase." https://github.com/supabase/supabase, 2025. An open-source Firebase alternative built on PostgreSQL.

[29] H. Steck, "Embarrassingly Shallow Autoencoders for Sparse Data," in *The World Wide Web Conference*, WWW '19, p. 3251–3257, ACM, May 2019.

[30] Meta and contributors, "React: A Javascript library for building user interfaces." https://reactjs.org/, 2025.

[31] A. Wathan, J. Reinink, D. Hemphill, and S. Schoger, "Tailwind CSS: A utility-first CSS framework." https://tailwindcss.com/, 2019.

[32] amannn, "next-intl: Internationalization (i18n) for Next.js." https://github.com/amannn/next-intl, 2025.

[33] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, pp. 8024–8035, Curran Associates, Inc., 2019.

[34] FireDucks Development Team, "FireDucks: Compiler-accelerated pandas-compatible DataFrame library for Python." https://fireducks-dev.github.io/, 2023.

[35] W. McKinney *et al.*, "Data structures for statistical computing in Python," in *Proceedings of the 9th Python in Science Conference*, vol. 445, pp. 51–56, Austin, TX, 2010.

[36] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, pp. 357–362, Sept. 2020.

[37] deedy5, "DDGS: Dux Distributed Global Search." https://github.com/deedy5/ddgs, 2025.

[38] I. Artifex Software, "PyMuPDF: High-performance Python bindings for MuPDF." https://github.com/pymupdf/PyMuPDF, 2025.

[39] C. McDonnell, "Zod: TypeScript-first schema declaration and validation library." https://www.npmjs.com/package/zod, 2025.

[40] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux journal*, vol. 2014, no. 239, p. 2, 2014.

[41] OpenZiti, "zrok: An open-source sharing solution." https://zrok.io/, 2025.

[42] H. Krekel, B. Oliveira, R. Pfannschmidt, F. Bruynooghe, B. Laugher, and F. Bruhin, "pytest 8.4." https://github.com/pytest-dev/pytest, 2004. Version 8.4. Contributors include Holger Krekel, Bruno Oliveira, Ronny Pfannschmidt, Floris Bruynooghe, Brianna Laugher, Florian Bruhin, and others.

[43] 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.

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

[45] J. B. Brooke, "SUS: A 'Quick and Dirty' Usability Scale," 1996.

[46] The Open WebUI Community, "Open WebUI: A user-friendly web interface for llms." https://github.com/open-webui/open-webui, 2024.

[47] Aravind Srinivas and Denis Yarats and Johnny Ho and Andy Konwinski, "Perplexity AI." https://www.perplexity.ai/, 2022. Accessed: 2025-07-13.

[48] K. Zhou, X. Wang, Y. Zhou, C. Shang, Y. Cheng, W. X. Zhao, Y. Li, and J.-R. Wen, "CRSLab: An open-source toolkit for building conversational recommender system," 2021.

[49] Z. Zhang, T. Laud, Z. He, X. Chen, X. Liu, Z. Xie, J. McAuley, and Z. He, "RecWizard: A toolkit for conversational recommendation with modular, portable models and interactive user interface," 2024.

Acronyms

AI Artificial Intelligence.

API Application Programming Interface.

CI/CD Continuous Integration and Continuous Deployment.

CPU Central Processing Unit.

CRS Conversational Recommender System.

CSS Cascading Style Sheets.

ER Entity-Relationship.

GPT Generative Pre-trained Transformer.

GPU Graphics Processing Unit.

HTML HyperText Markup Language.

JWT JSON Web Tokens.

LLM Large Language Model.

PWA Progressive Web Application.

RAG Retrieval-Augmented Generation.

RAM Random Access Memory.

RLS Row-Level Security.

RS Recommender System.

SUS System Usability Scale.

UI User Interface.

UX User Experience.

APPENDICES

A

API Schemas

Code A.1: Schemas used for defining requests of each API endpoint [1/2].

```python
"""Request schemas for api.py endpoints."""

from fastapi import UploadFile
from pydantic import BaseModel
from typing import List, Optional


class InferColumnRolesRequest(BaseModel):
    column_names: List[str]
    file_type: str


class InferFromSampleRequest(BaseModel):
    sample_values: List[str]


class Column(BaseModel):
    id: int
    name: str
    role: str
    data_type: str
    delimiter: Optional[str]
    original_name: str


class SniffResult(BaseModel):
    delimiter: str
    has_header: bool
    labels: List[str] | None
    newline_str: str
    quote_char: Optional[str]
    total_rows: int
```

Code A.2: Schemas used for defining requests in each API endpoint [2/2].

```
class DatasetFile(BaseModel):
    id: str
    name: str
    original_name: str
    file: Optional[UploadFile]
    file_type: str
    headers: Optional[List[str]]
    columns: List[Column]
    sniff_result: SniffResult


class AgentConfig(BaseModel):
    agent_name: str
    dataset_name: str
    description: Optional[str]
    public: bool


class AgentRequest(BaseModel):
    agent_id: int
    dataset_name: str
    user_id: str


class ChatHistoryRequest(BaseModel):
    chat_id: int
    user_id: str


class CreateChatHistoryRequest(ChatHistoryRequest):
    content: str


class AppendChatHistoryRequest(ChatHistoryRequest):
    new_message: str


class StartWorkflowRequest(AgentRequest):
    agent_name: str
    description: str


class SendUserResponseRequest(BaseModel):
    workflow_id: str
    user_response: str
```

B

Web Application Screenshots

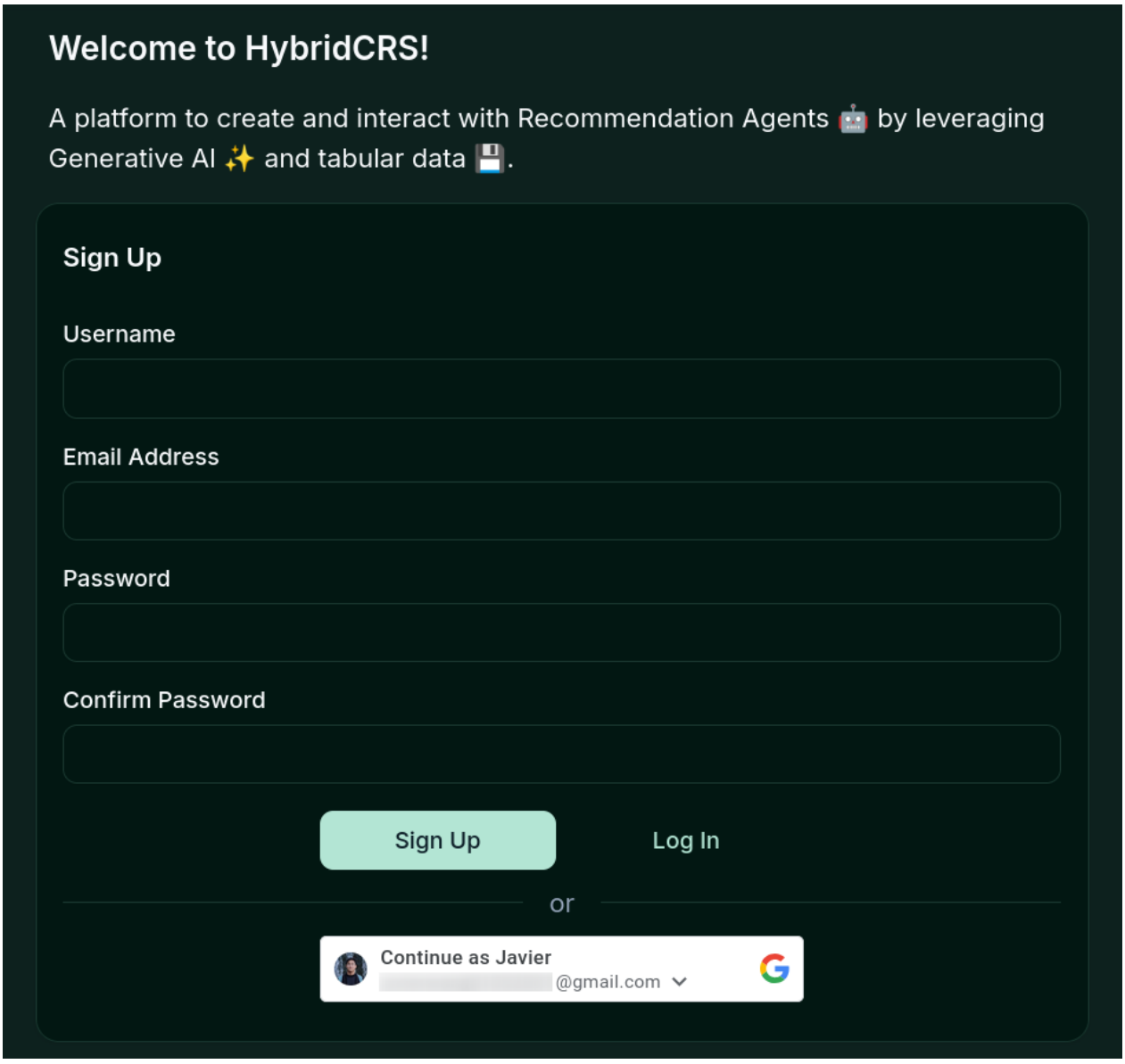


Figure B.1: Sign Up Form.

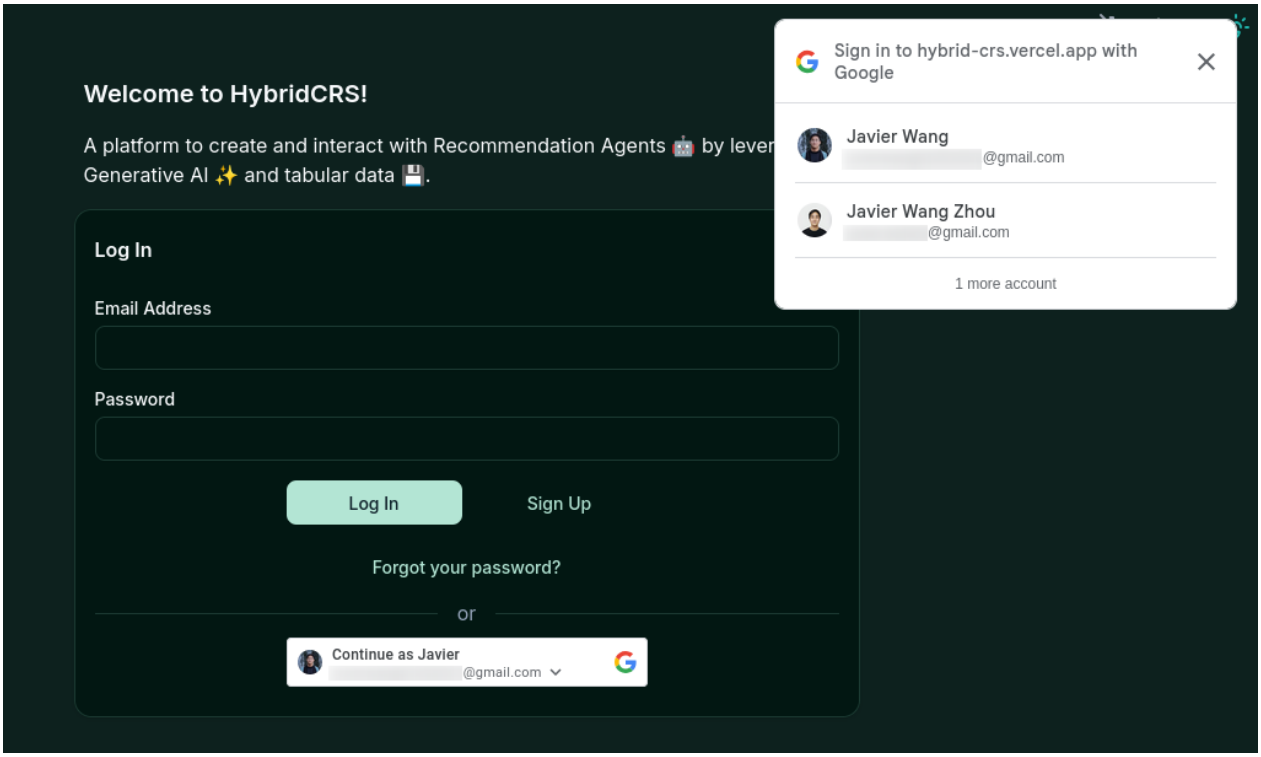


Figure B.2: Login with Google.

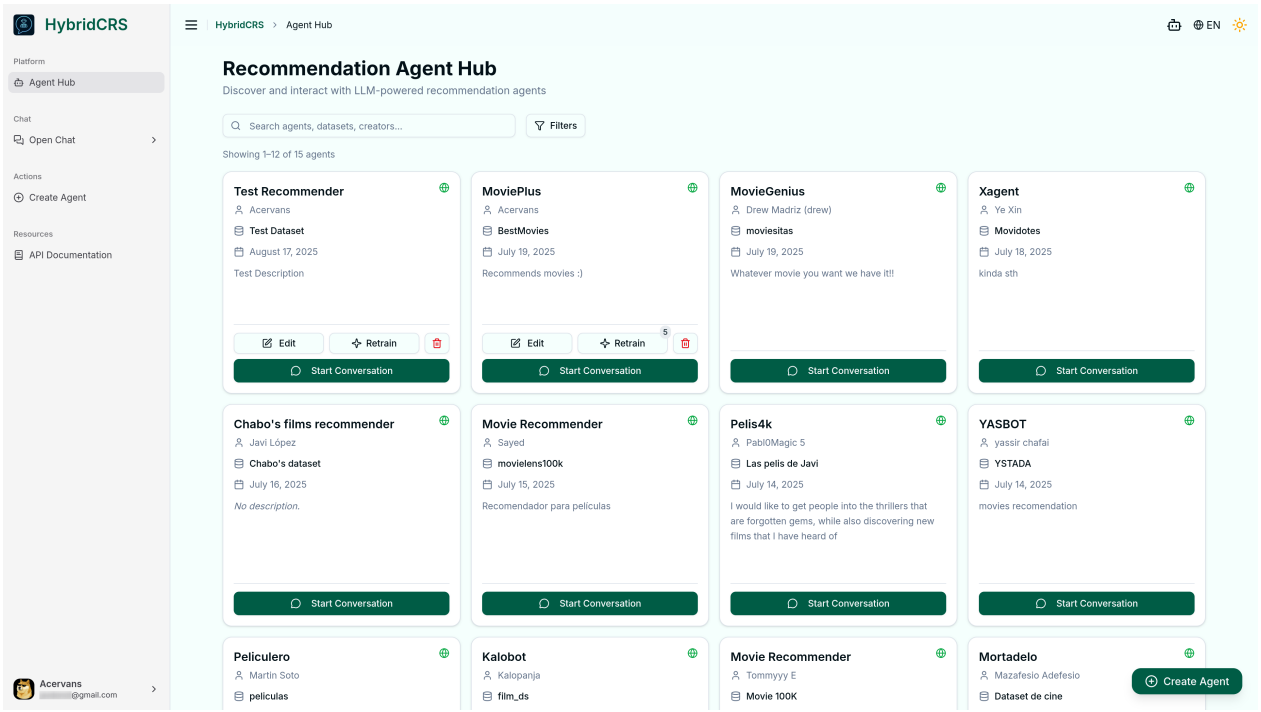


Figure B.3: Agent Hub – Light Theme.

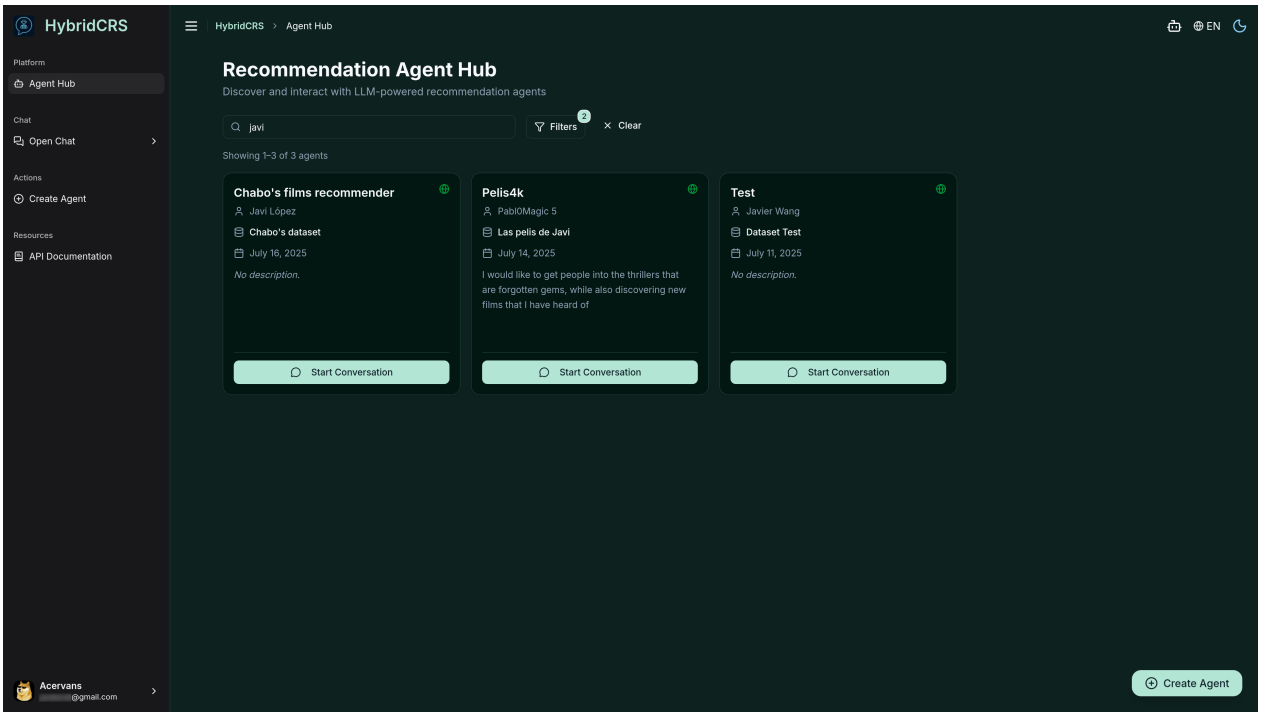


Figure B.4: Agent Hub – Filtered.

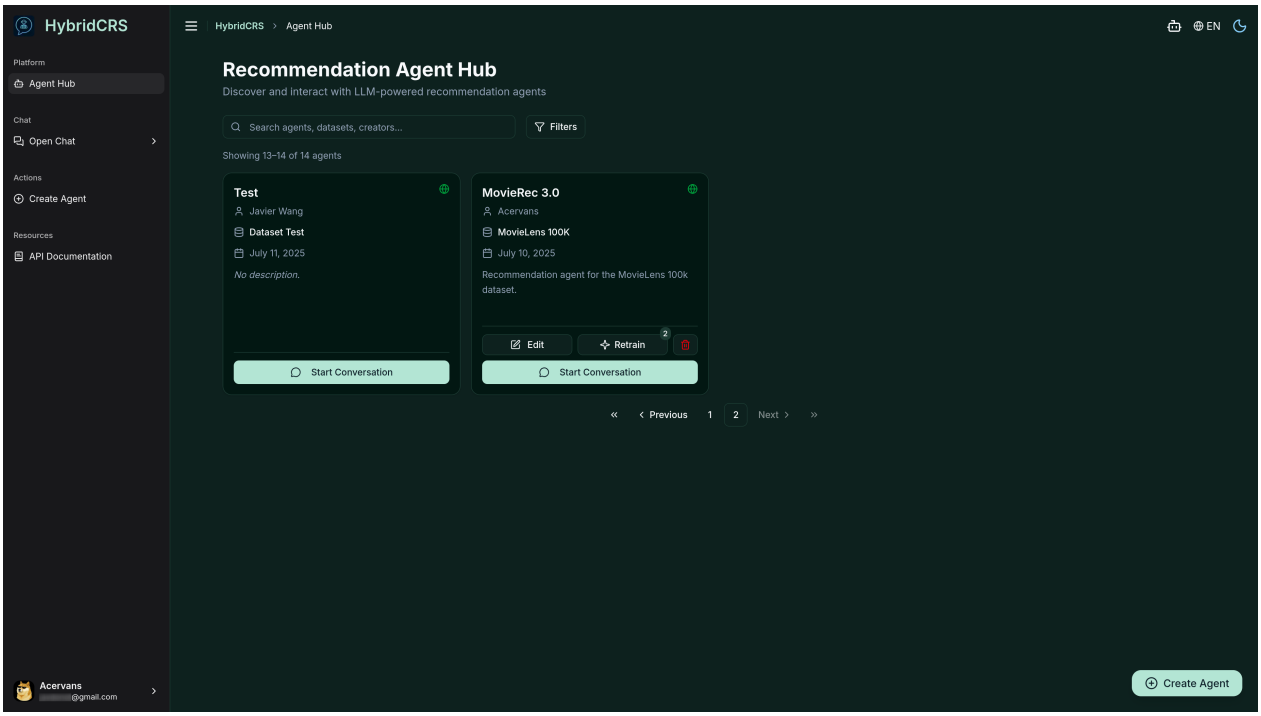


Figure B.5: Agent Hub – Pagination.

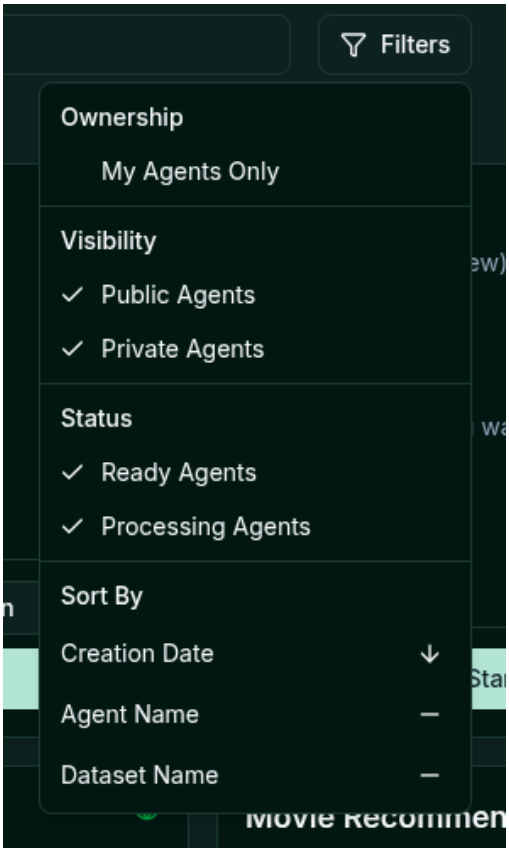


Figure B.6: Agent Hub – Filters.

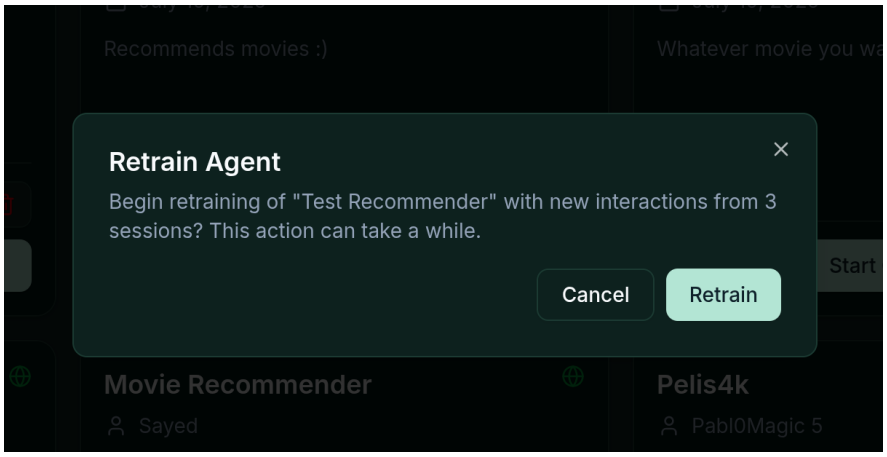


Figure B.7: Agent Hub – Retrain an Agent.

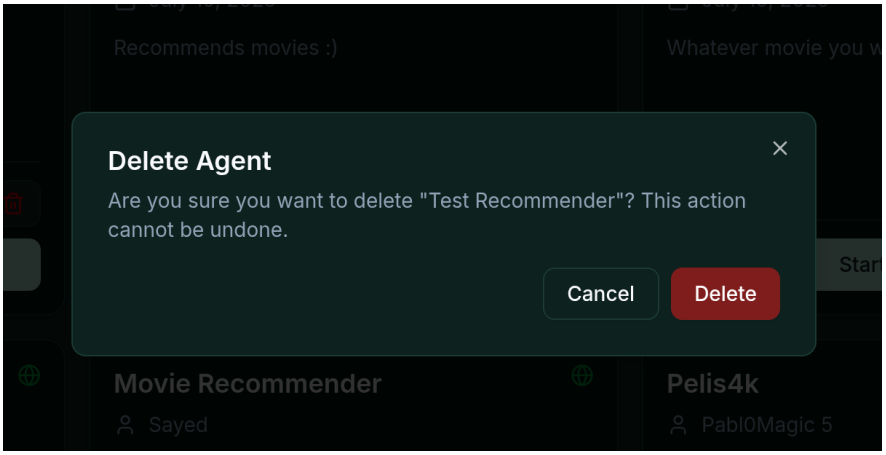


Figure B.8: Agent Hub – Delete an Agent.

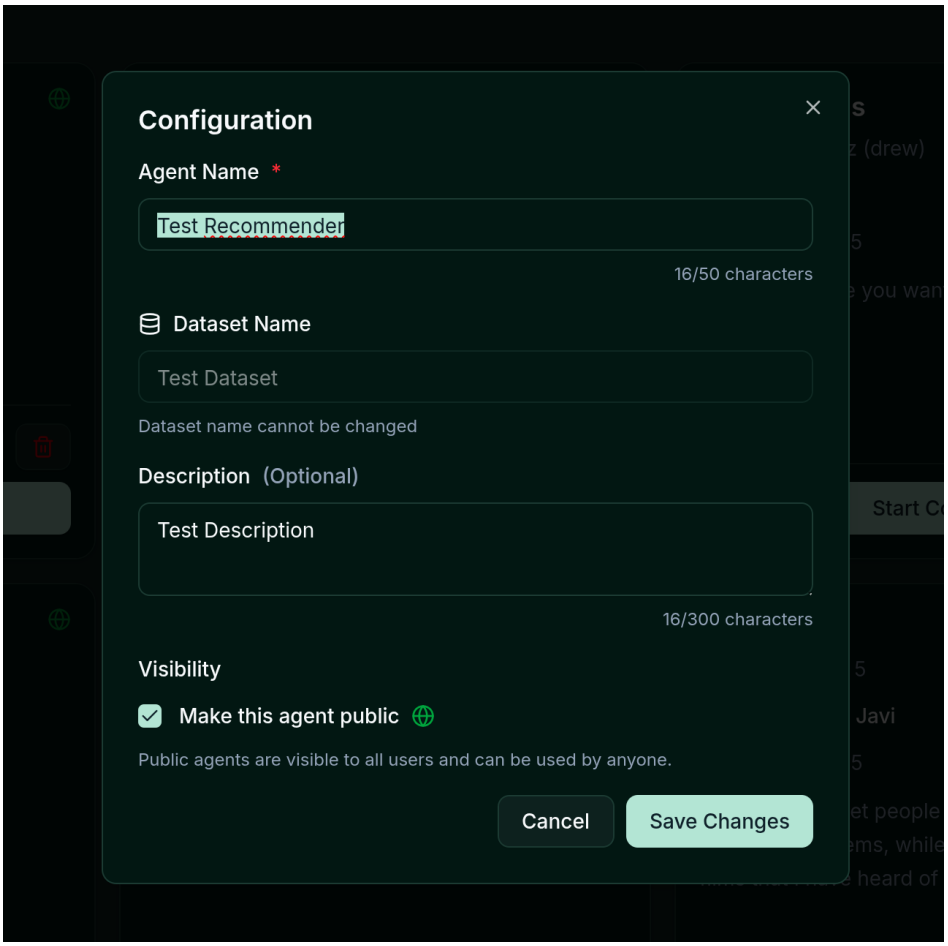


Figure B.9: Agent Hub – Edit an Agent.

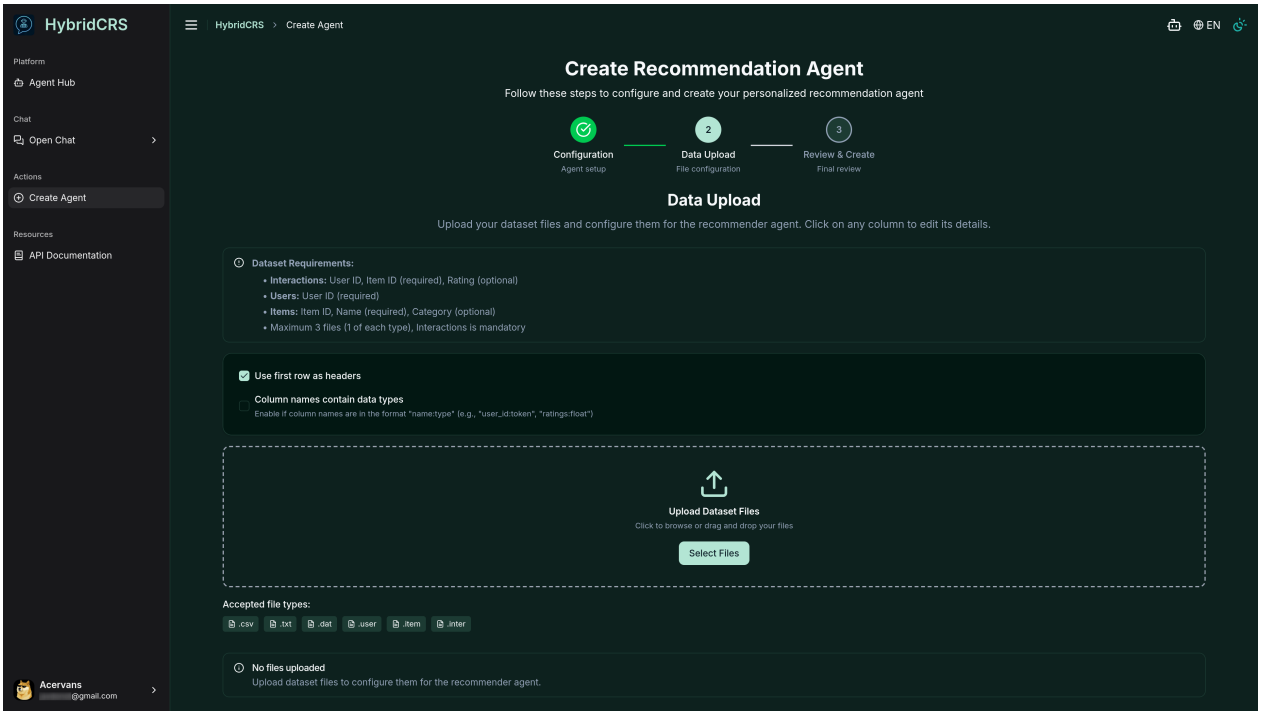


Figure B.10: Create Agent – Step 2.

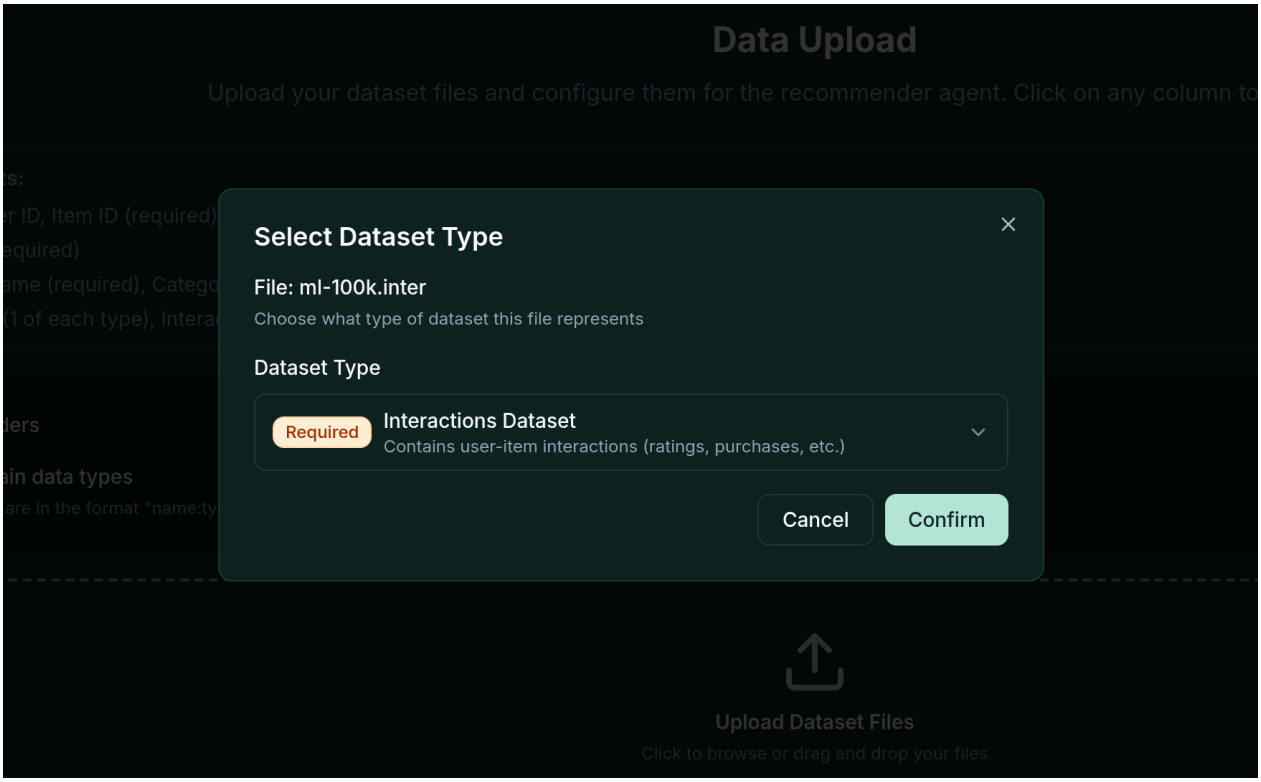


Figure B.11: Create Agent – Step 2 (Dataset Upload).

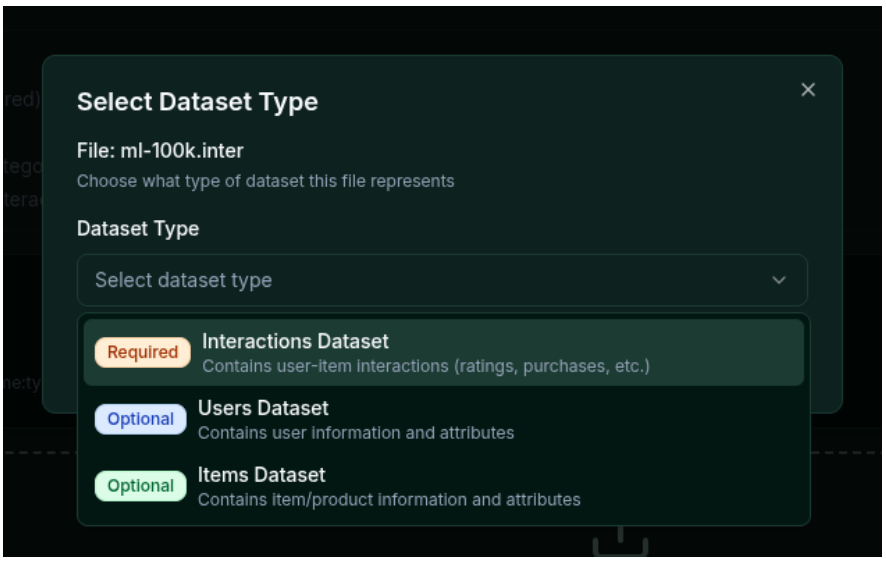


Figure B.12: Create Agent – Step 2 (Dataset Type Selector).

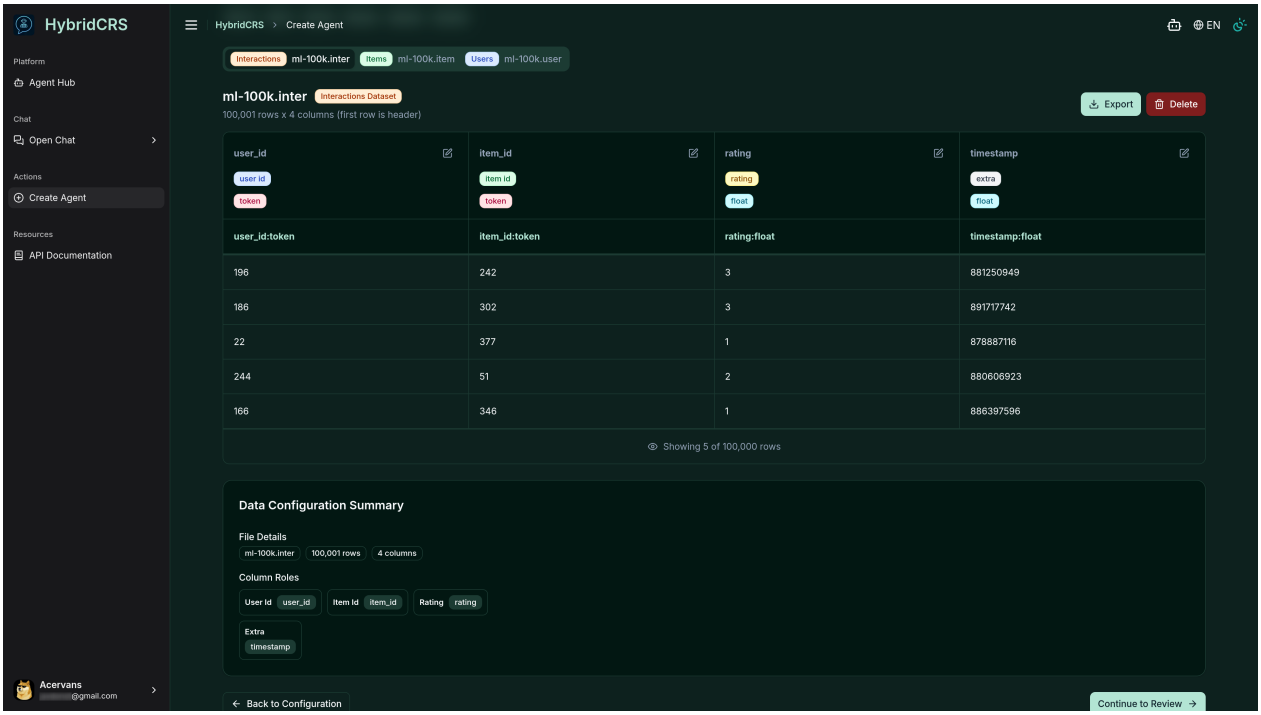


Figure B.13: Create Agent – Step 2 (Dataset Samples).

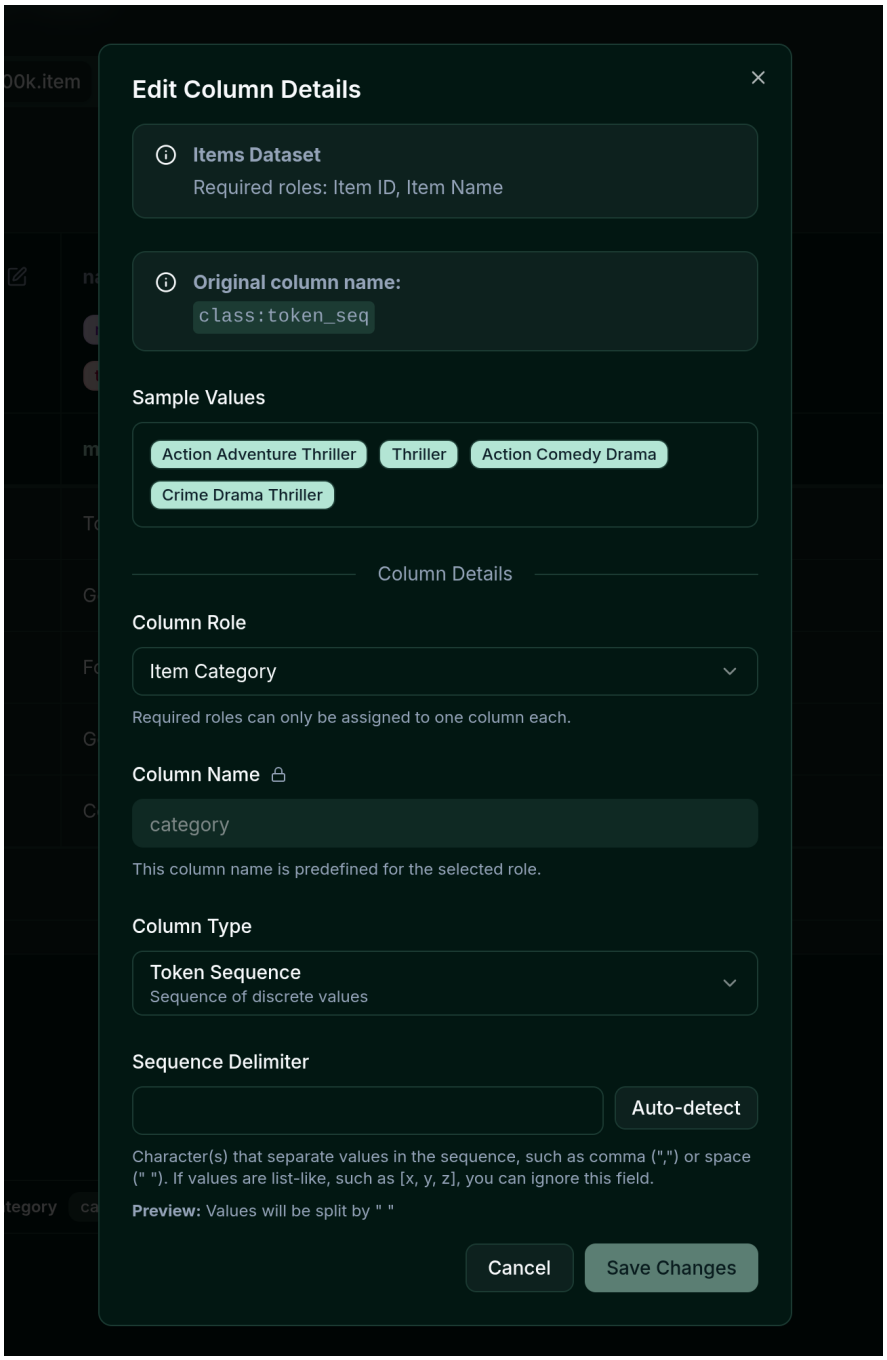


Figure B.14: Create Agent – Step 2 (Column Details Modal).

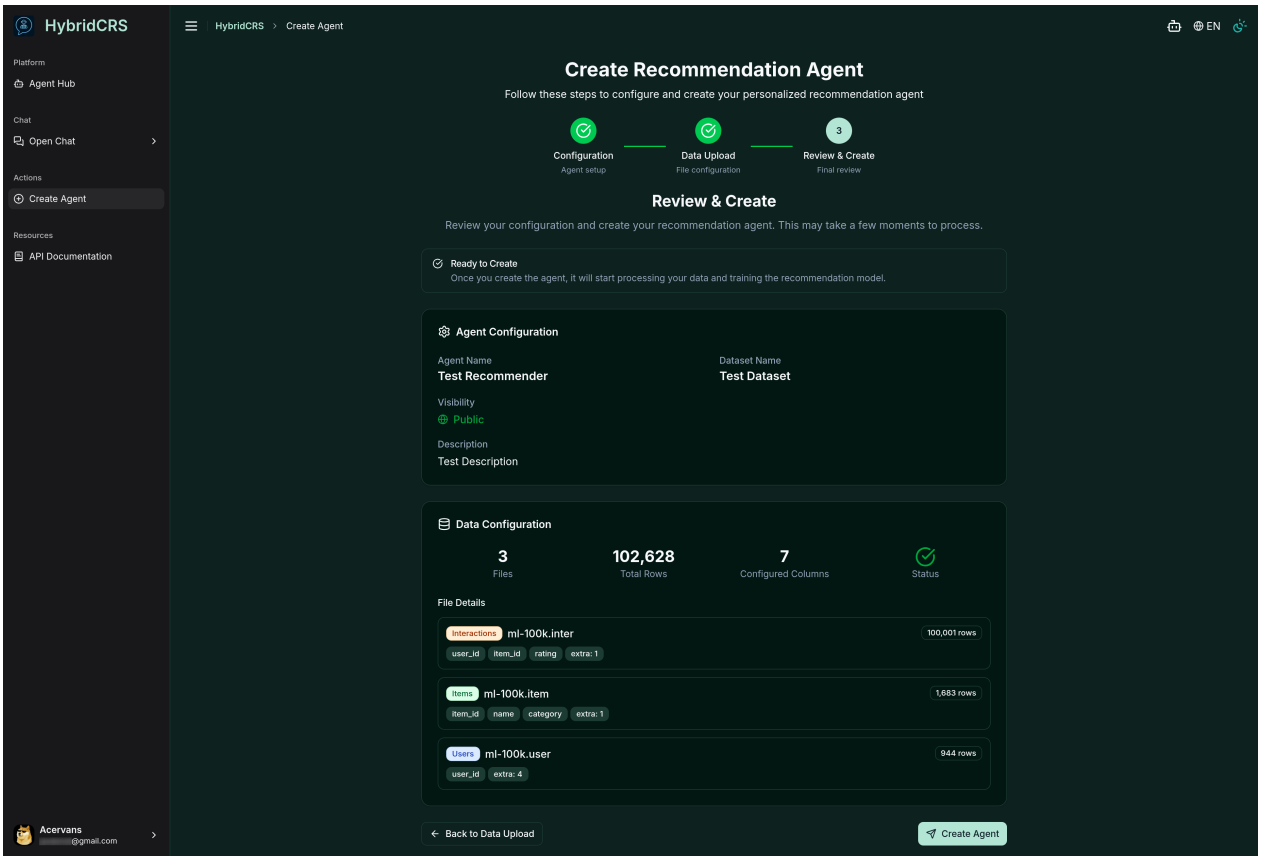


Figure B.15: Create Agent – Step 3.

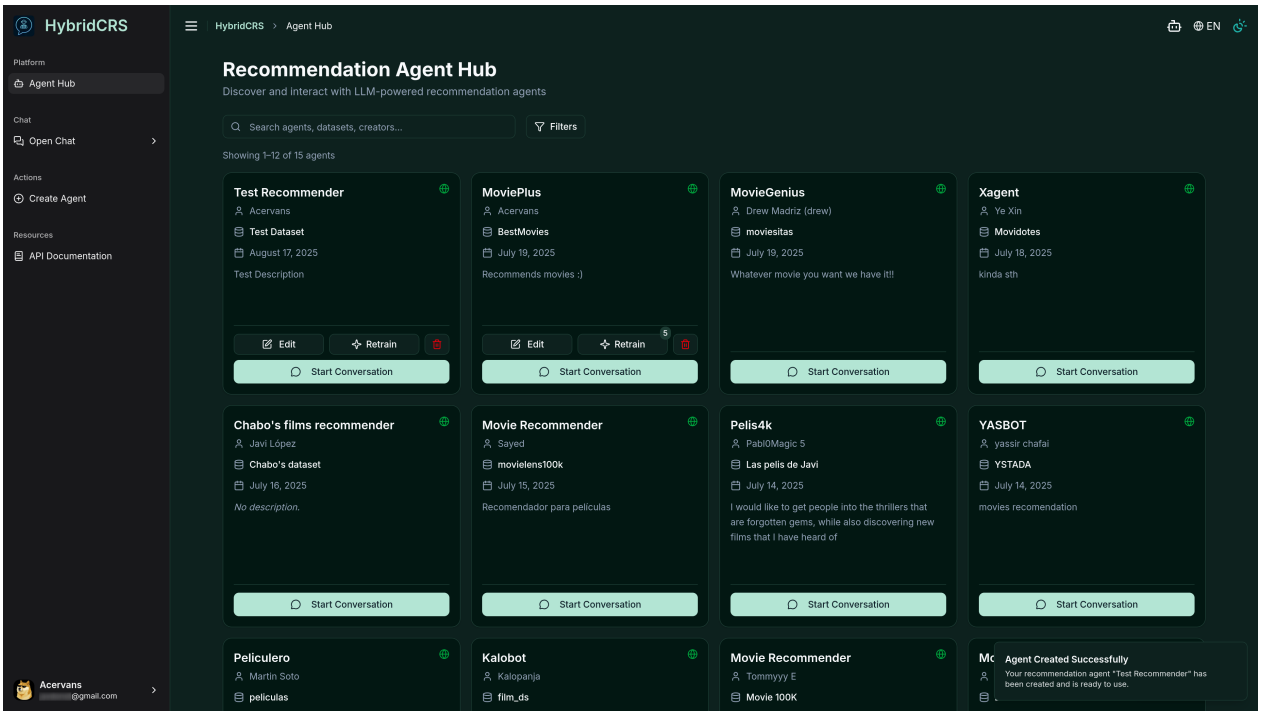


Figure B.16: Create Agent – Redirect to Agent Hub.

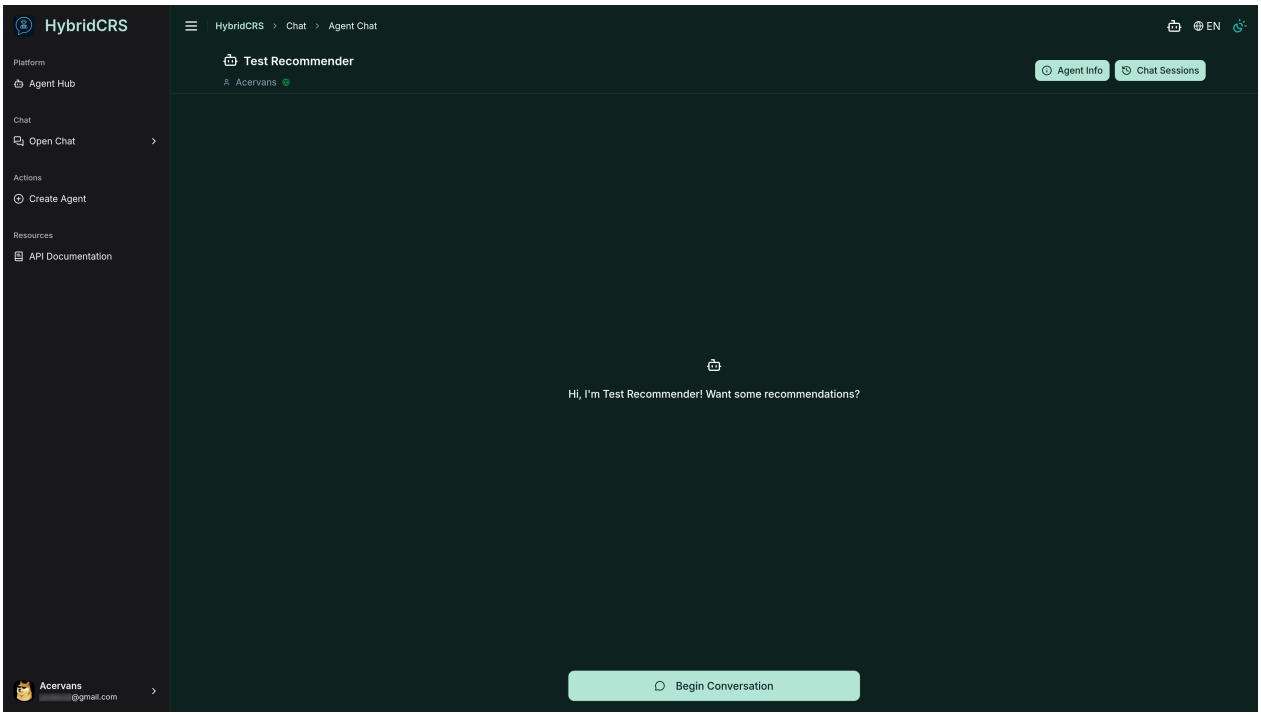


Figure B.17: Agent Chat – Initial state.

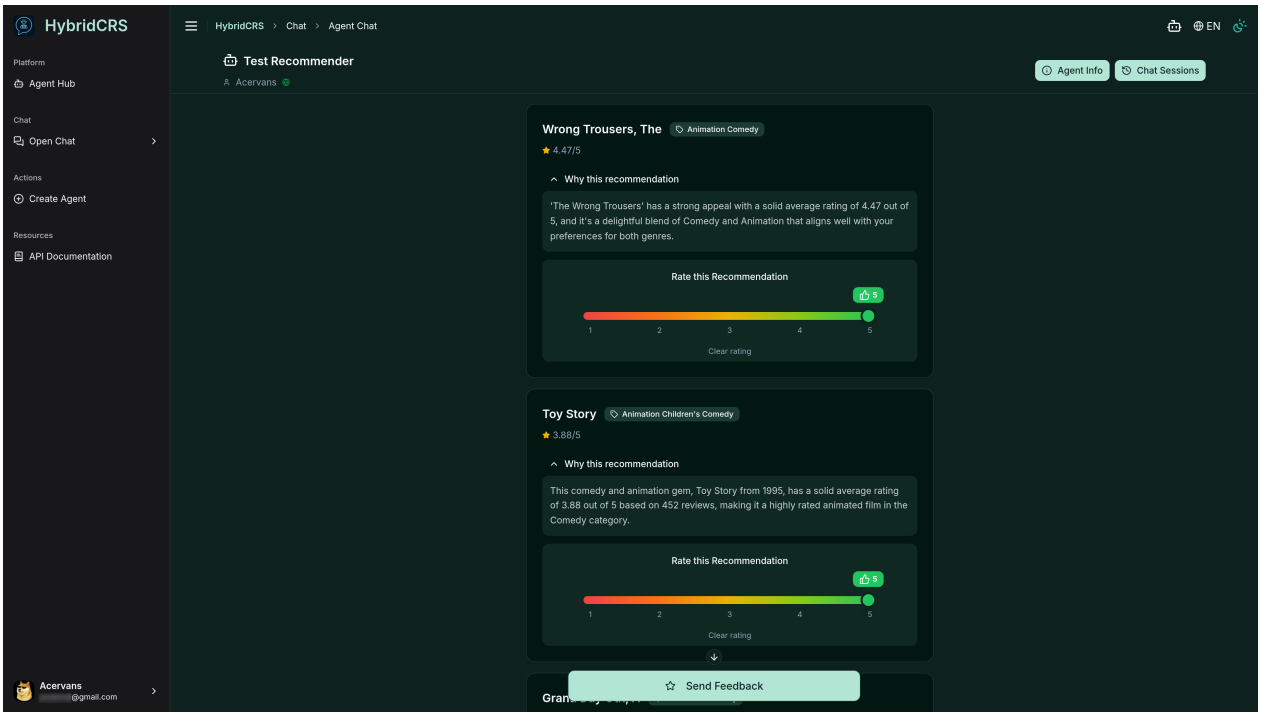


Figure B.18: Agent Chat – Rating recommendations.

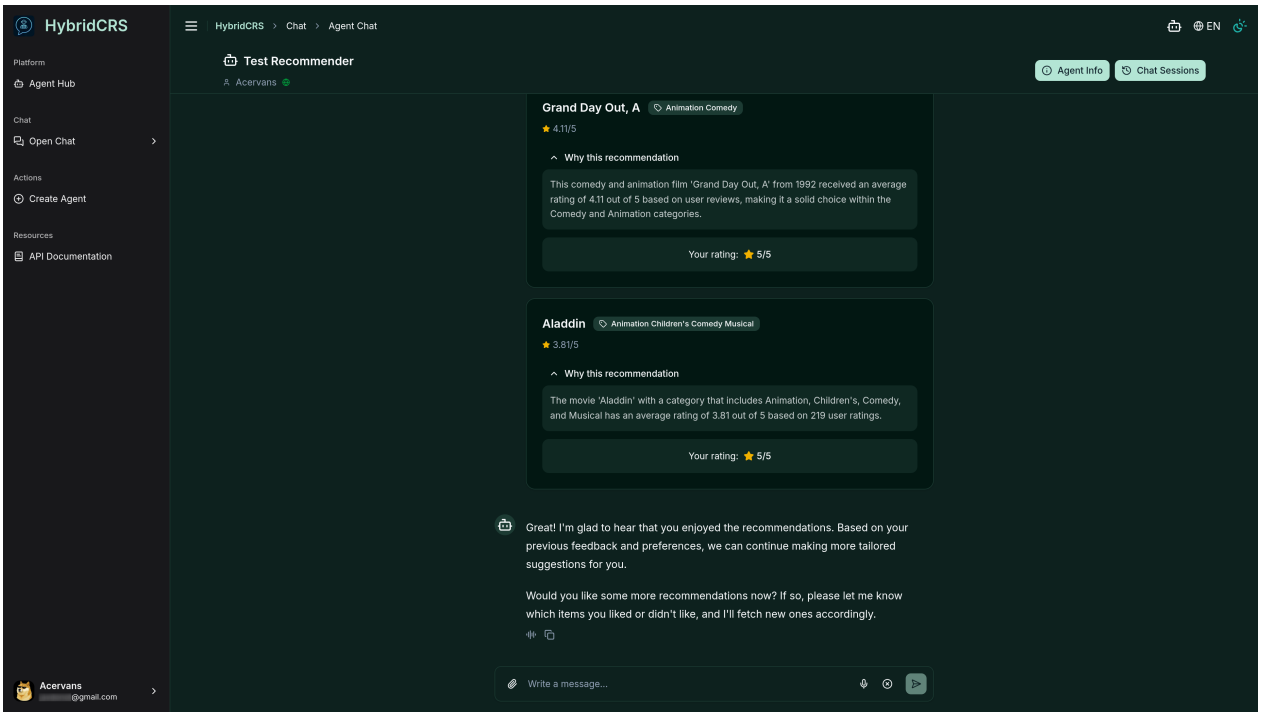


Figure B.19: Agent Chat – Feedback sent.

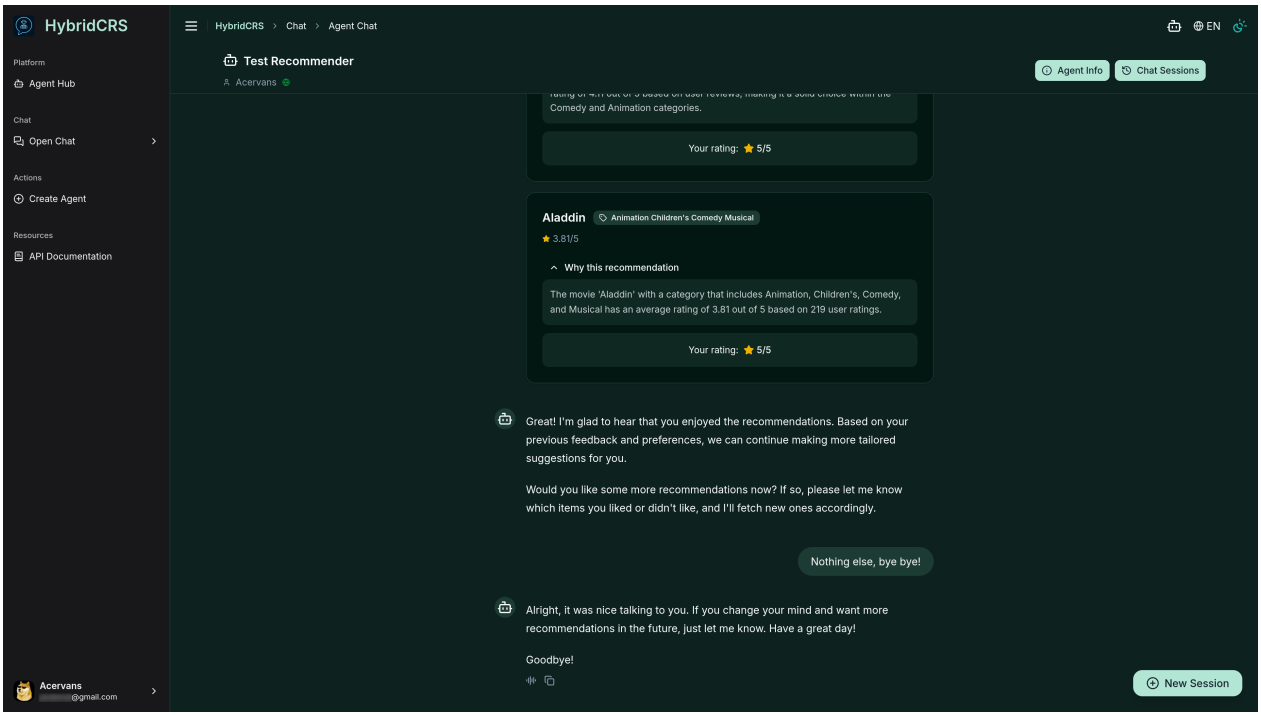


Figure B.20: Agent Chat – End session.

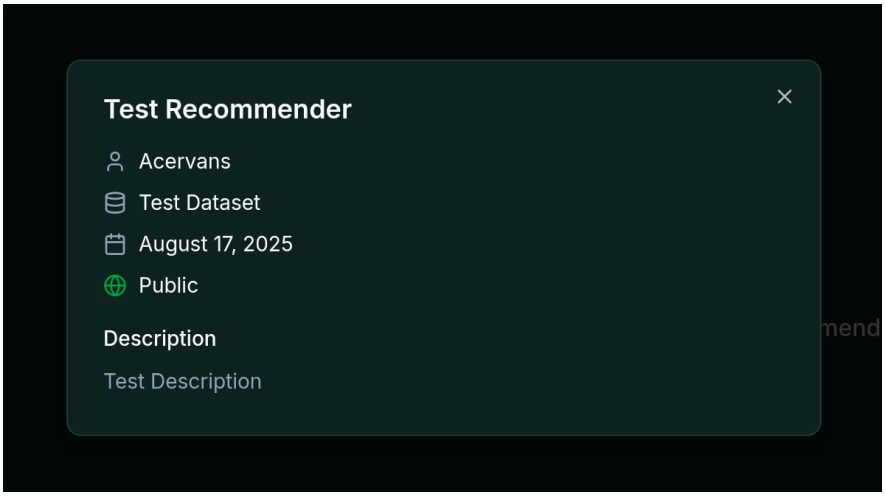


Figure B.21: Agent Chat – Agent info modal.

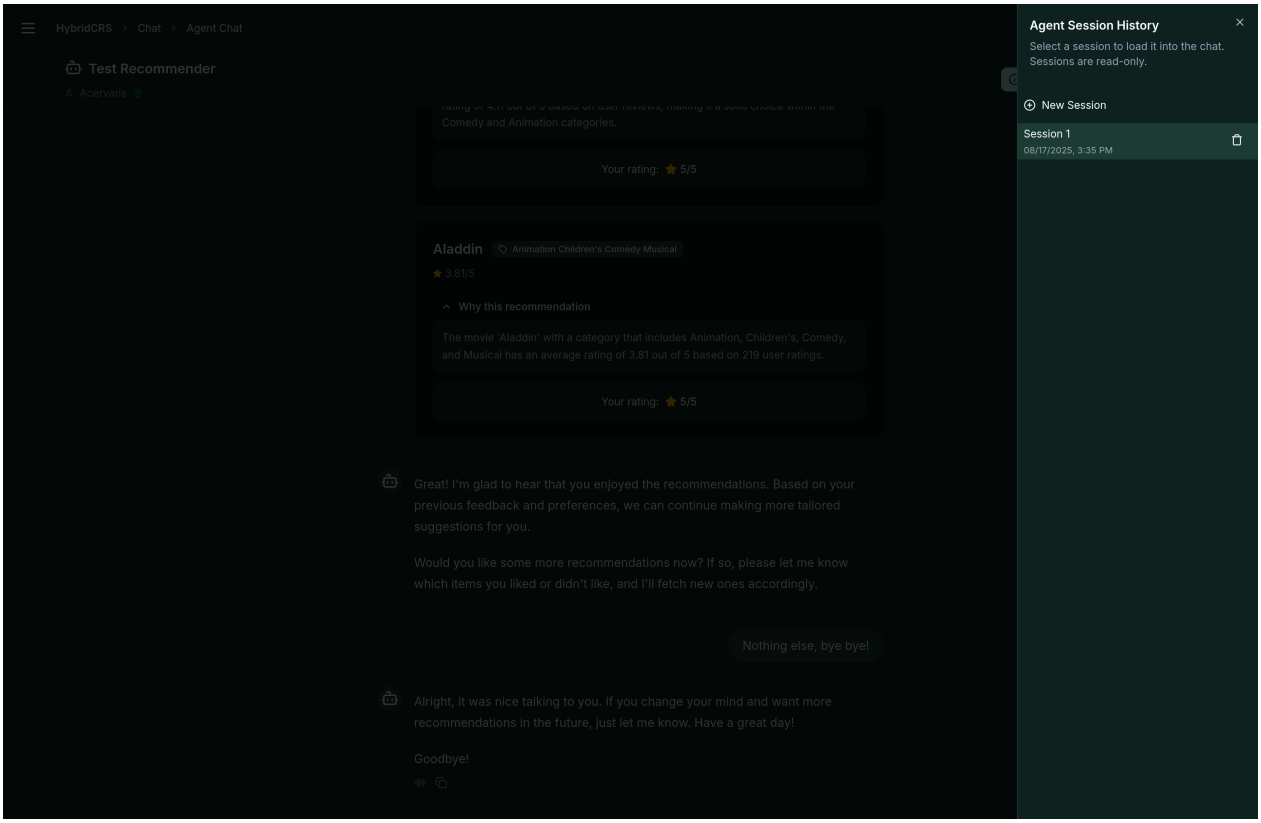


Figure B.22: Agent Chat – Sessions drawer.

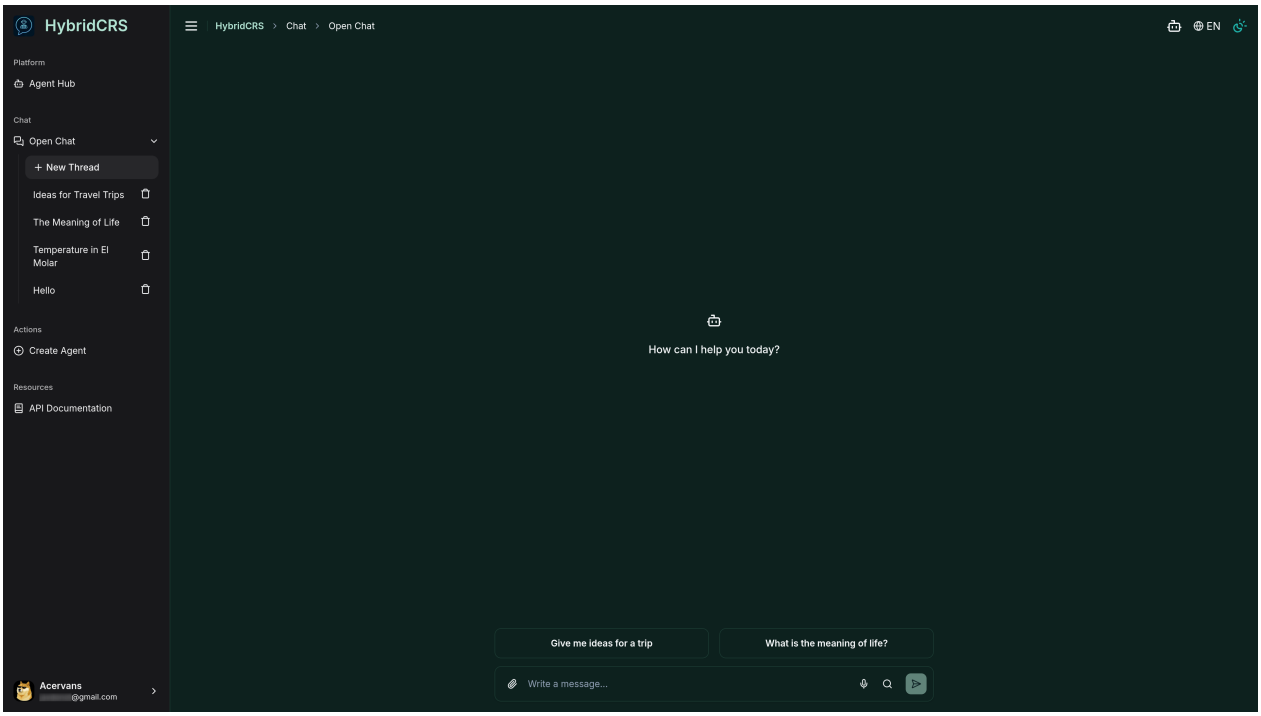


Figure B.23: Open Chat – Initial state.

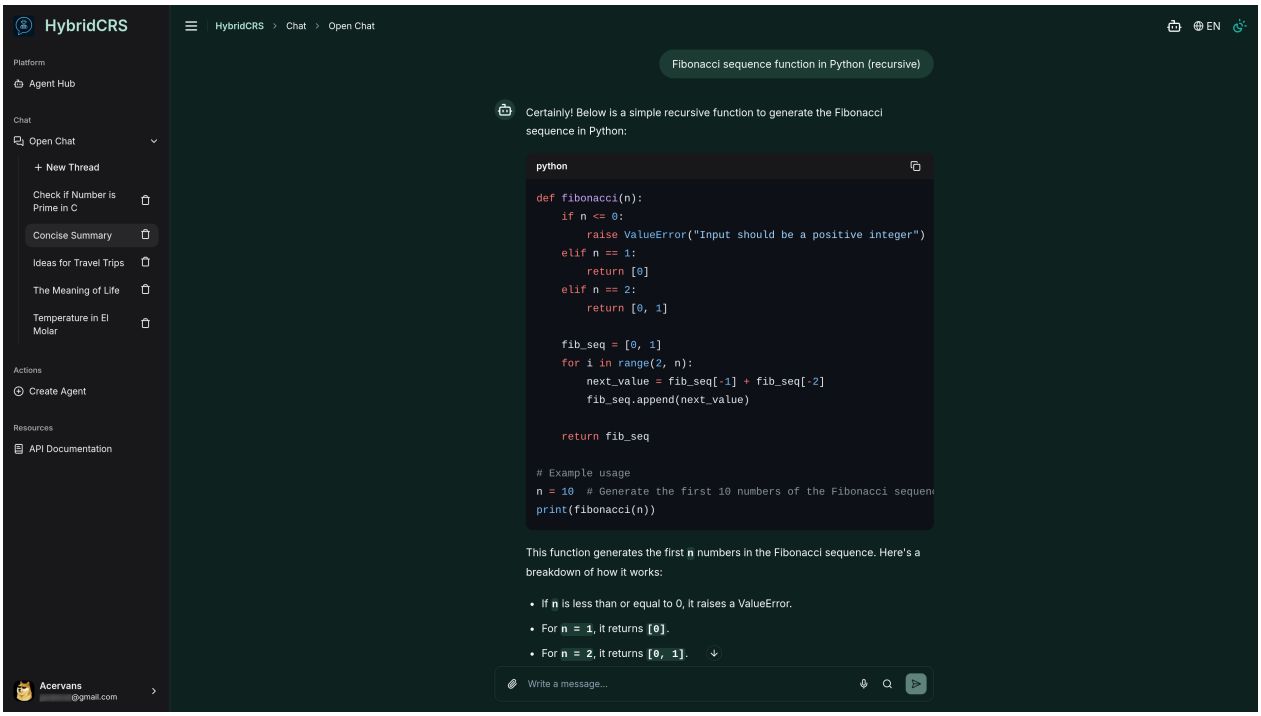


Figure B.24: Open Chat – Code display.

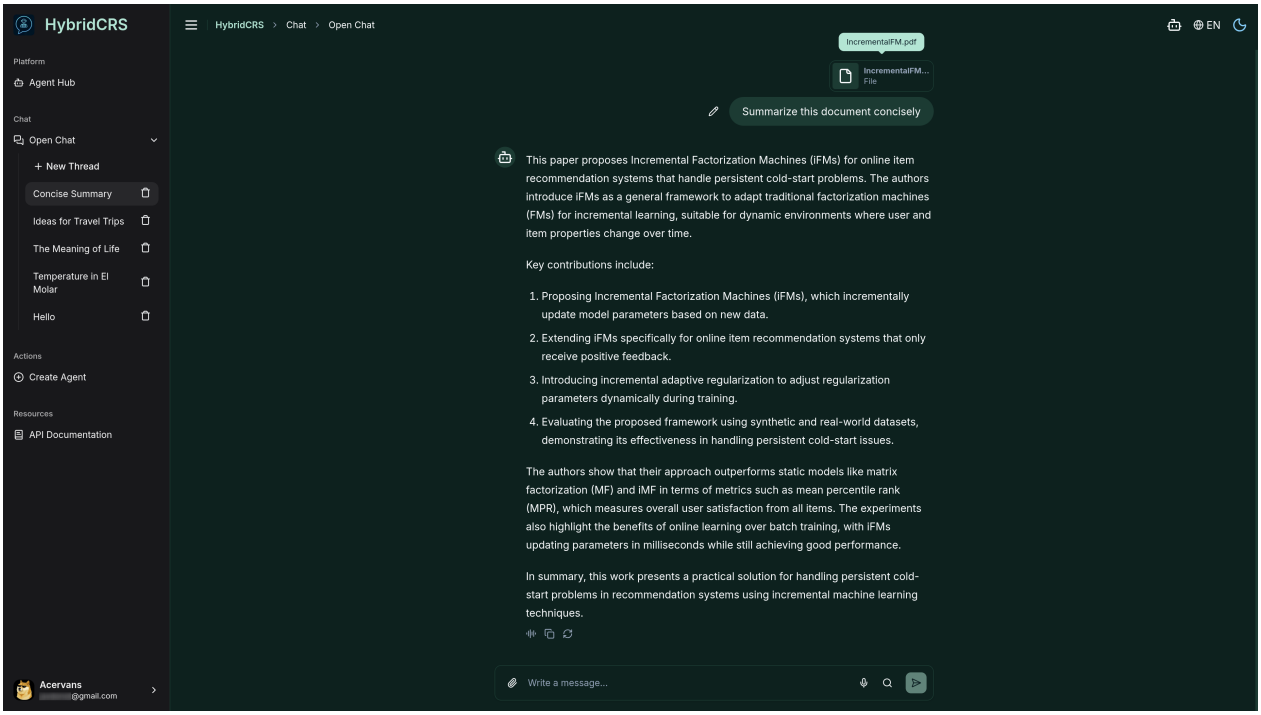


Figure B.25: Open Chat – PDF upload.

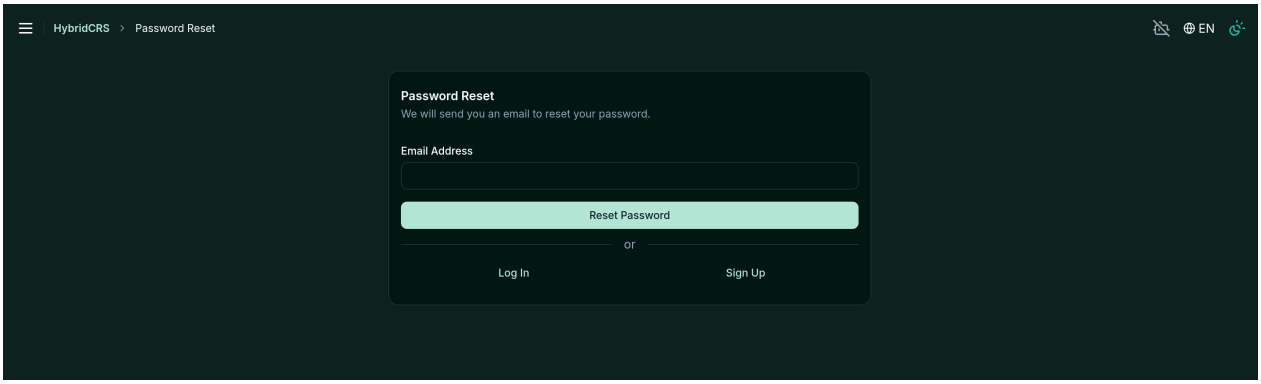


Figure B.26: Password Reset – Guest.

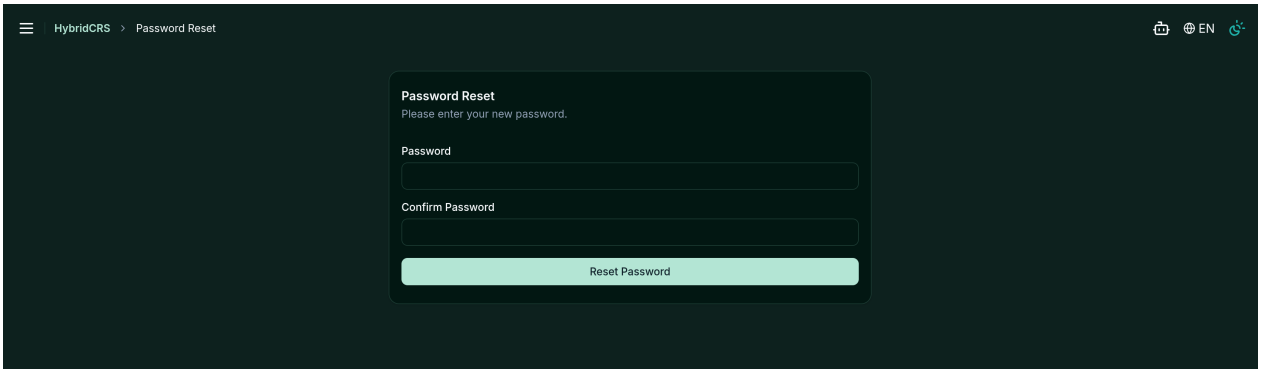


Figure B.27: Password Reset – Authenticated.

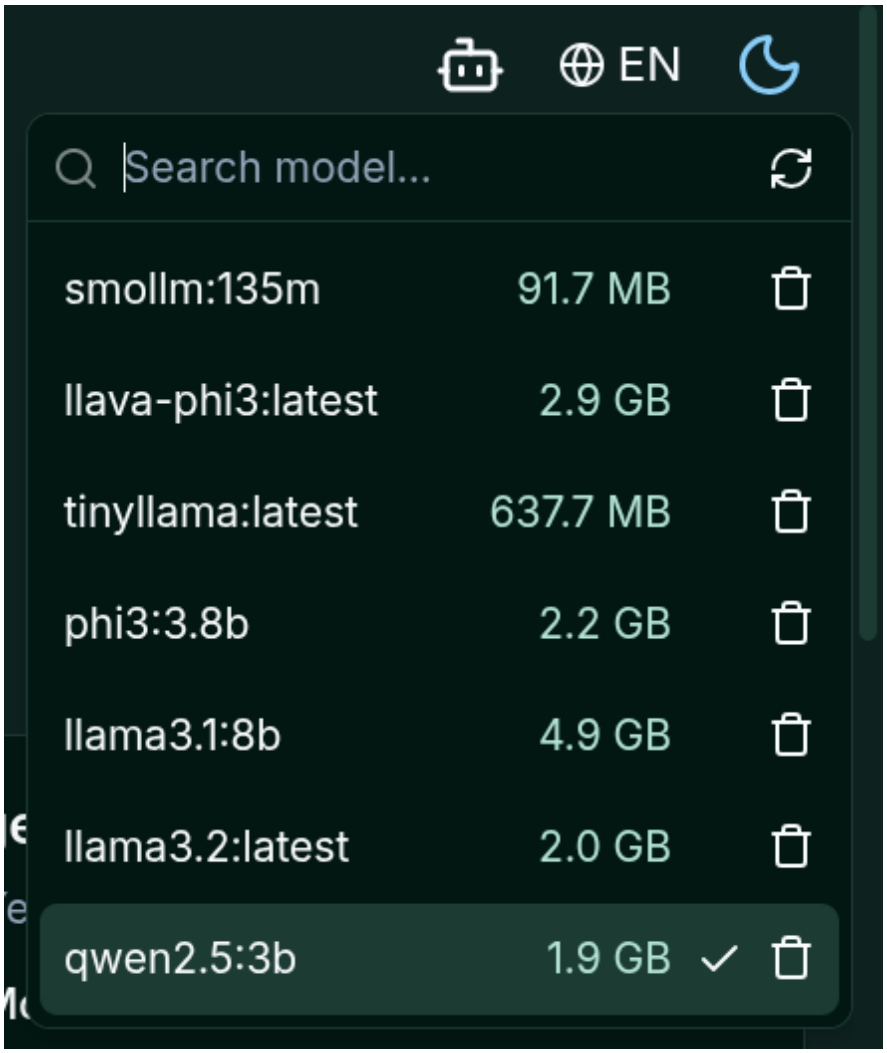


Figure B.28: Model Selector.

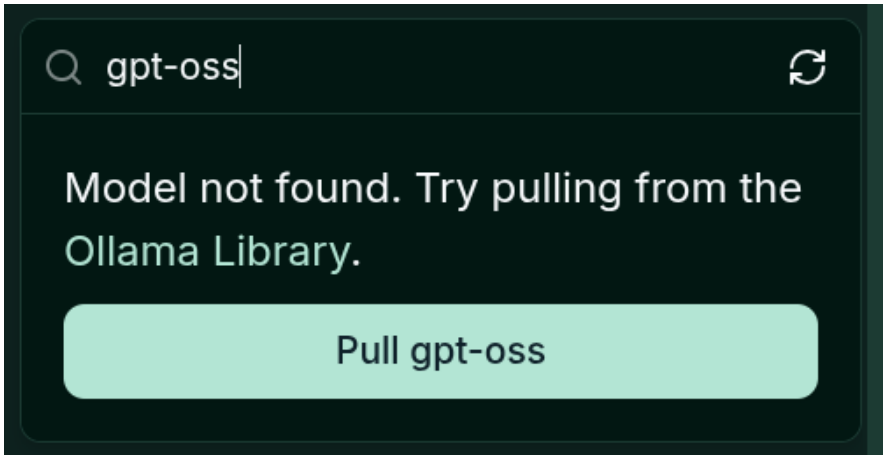


Figure B.29: Model Selector – Search model.

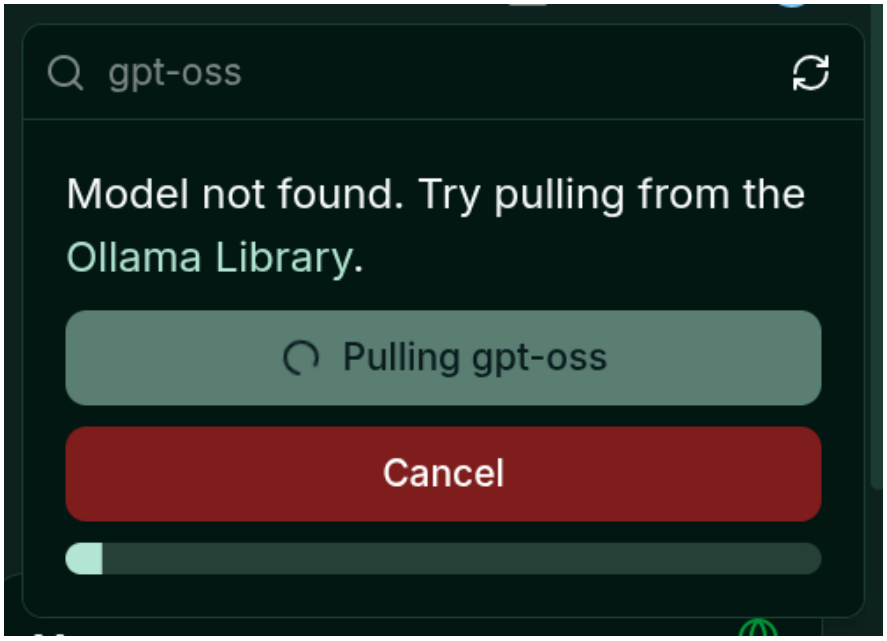


Figure B.30: Model Selector – Pull new model.

Figure B.31: Language Selector.

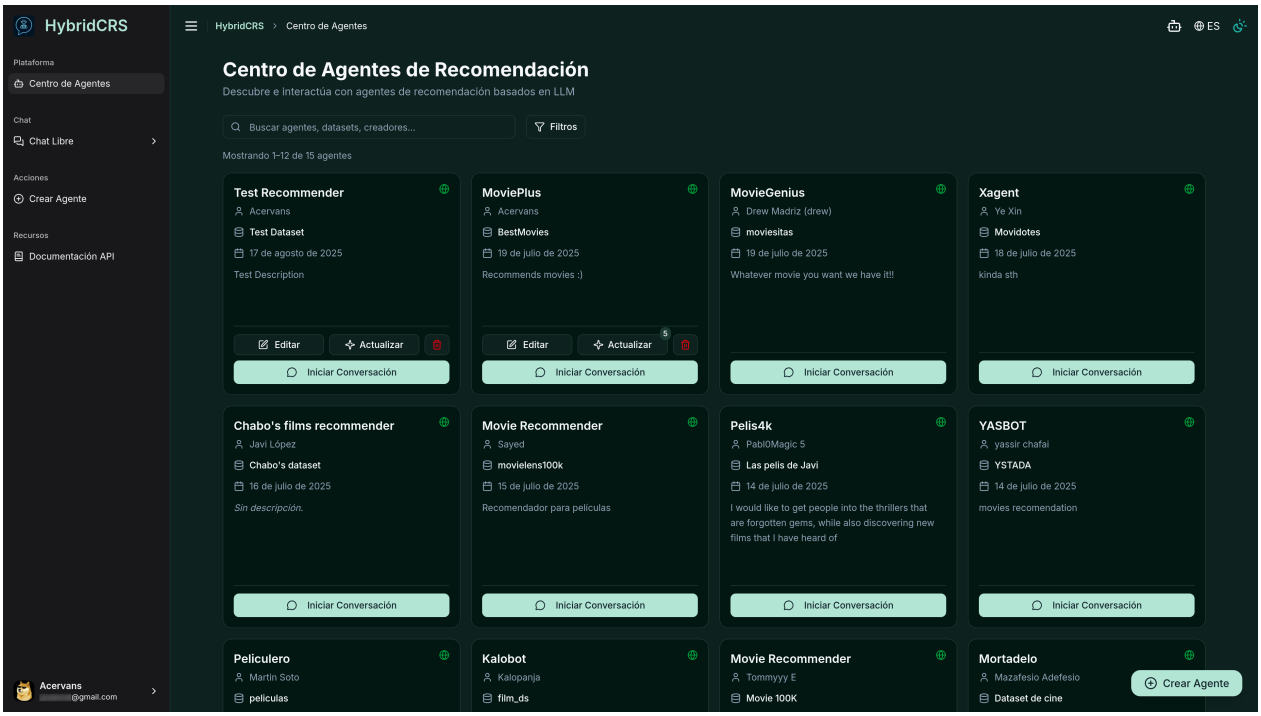


Figure B.32: Agent Hub – Spanish.

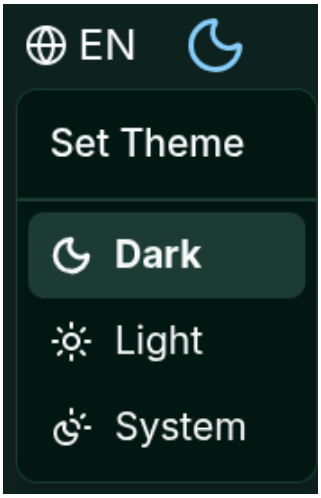


Figure B.33: Theme Selector.

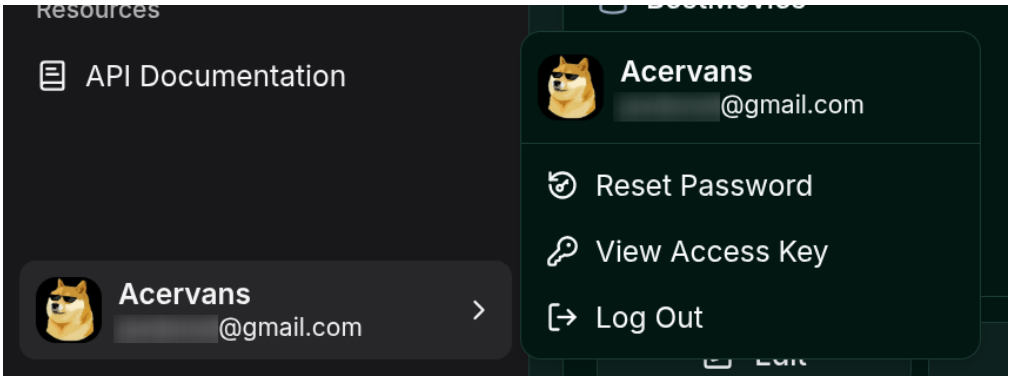


Figure B.34: User Authentication Actions.

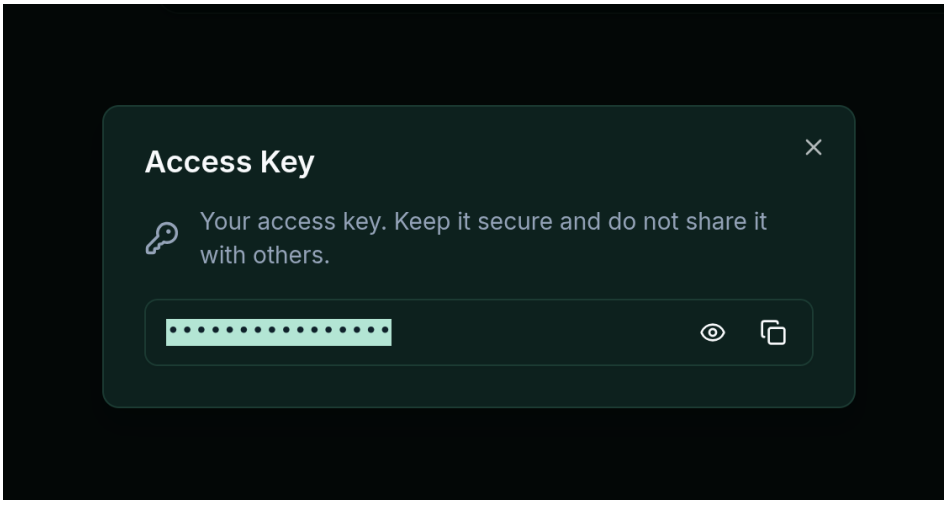
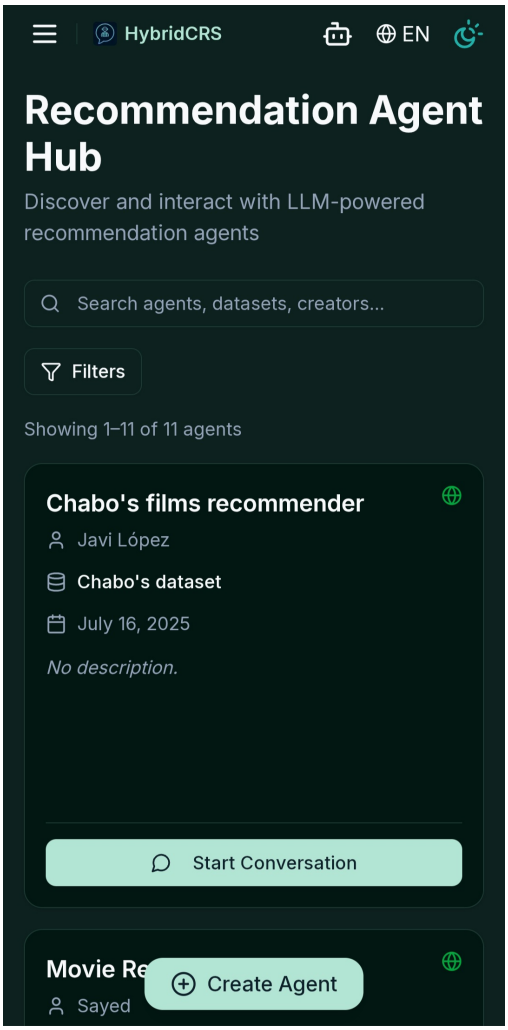


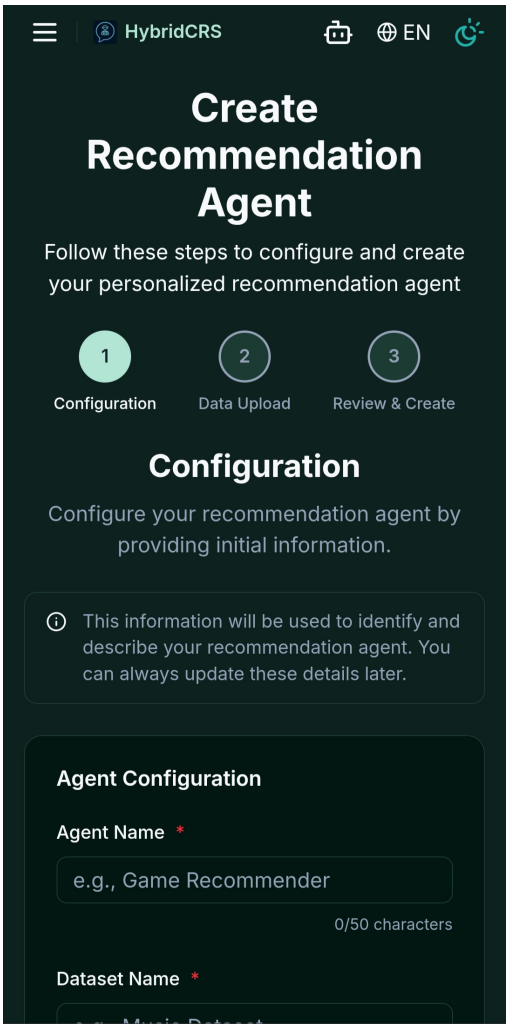
Figure B.35: Access Key Modal.

C

Mobile Screenshots

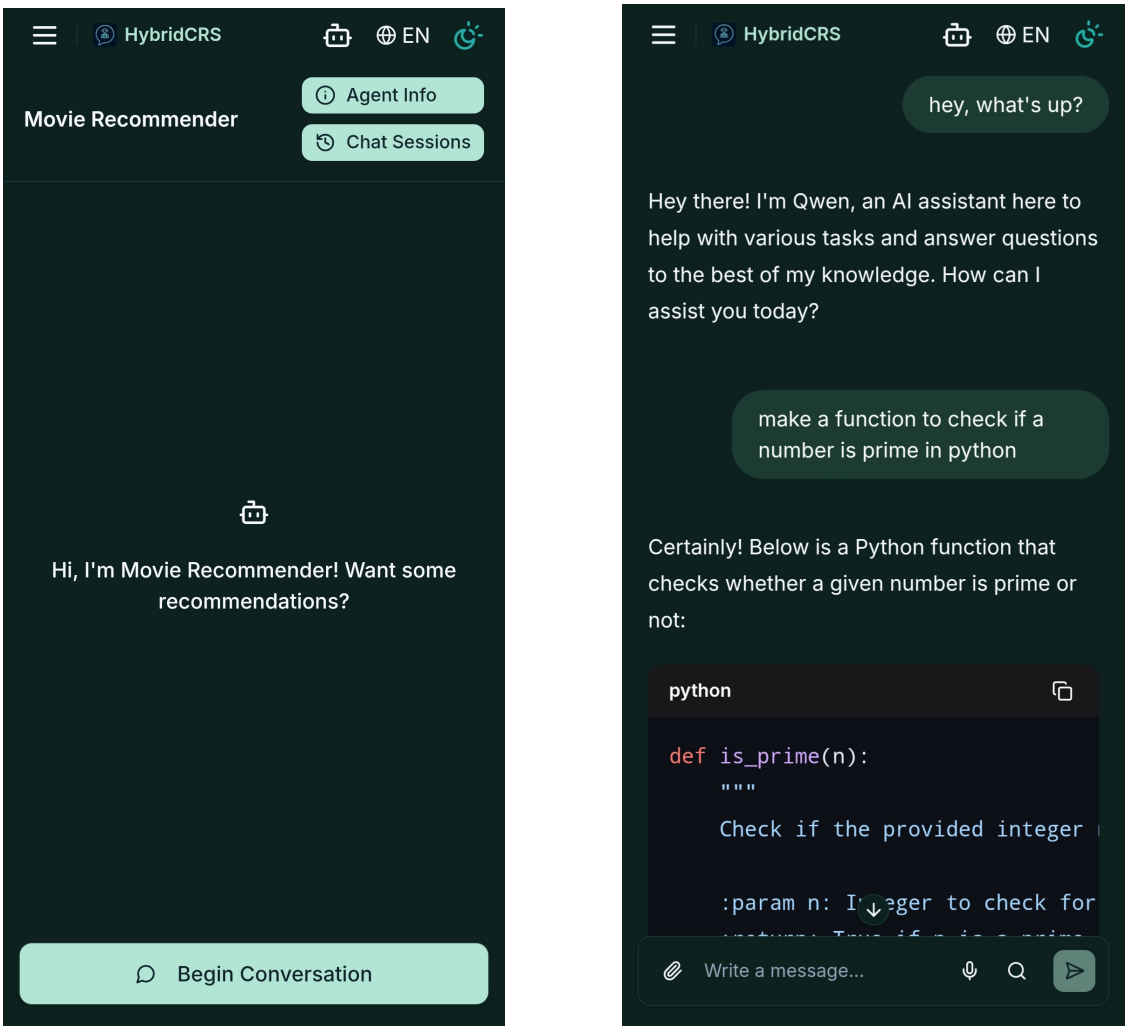


(a) Agent Hub



(b) Create Agent

Figure C.1: Mobile PWA Agent Hub & Create Agent



(a) Agent Chat

(b) Open Chat

Figure C.2: Mobile PWA Agent Chat & Open Chat

D

Backend Tests

Code D.1: Tests for the API module (`test_api.py`) [1/2].

```python
import io
import json
import pytest

from fastapi.testclient import TestClient
from api import app

client = TestClient(app)

JWT_SUB = "mock_user"
JWT_TOKEN = "Bearer test.jwt.token"
HEADERS = {"Authorization": JWT_TOKEN}

@pytest.fixture(autouse=True)
def mock_jwt_decode(mocker):
    mocker.patch("api.jwt.decode", return_value={"sub": "mock_user"})

def test_pdf_to_text():
    pdf_bytes = ...
    response = client.post(
        "/pdf-to-text",
        files={"file": ("test.pdf", io.BytesIO(pdf_bytes), "application/pdf")},
        headers=HEADERS,
    )
    assert response.status_code == 200
    assert response.text == "Example Content\n\n"

def test_infer_column_roles():
    ...

def test_infer_datatype():
    ...

def test_infer_delimiter():
    ...
```

Code D.2: Tests for the API module (`test_api.py`) [2/2].

```python
def test_create_agent(mocker):
    # Mock Supabase
    mocker.patch("api.supabase")

    # Mock recommender logic
    mocker.patch("api.FalkorDBRecommender")
    mocker.patch("api.train_expert_model", return_value=({}, {"score": 0.9}))
    mocker.patch("api.process_dataset")
    mocker.patch("api.os.remove")
    mocker.patch("api.shutil.rmtree")

    agent_config = {
        "agent_name": "TestAgent",
        "dataset_name": "TestSet",
        "description": "desc",
        "public": True,
    }
    dataset_file = {
        "file_type": "interactions",
        "columns": [
            {"name": "user_id", "data_type": "token", "role": "user"},
            {"name": "item_id", "data_type": "token", "role": "item"},
            {"name": "rating", "data_type": "float", "role": "rating"},
        ],
        "sniff_result": {
            "delimiter": ",",
            "newline_str": "\n",
            "has_header": True,
            "quote_char": '"',
        },
    }
    files = {
        "agent_id": (None, "123"),
        "agent_config": (None, json.dumps(agent_config)),
        "dataset_files": (None, json.dumps(dataset_file)),
        "upload_files": (
            "inter.csv",
            io.BytesIO(b"user_id,item_id,rating\n1,2,3\n"),
            "text/csv",
        ),
    }
    response = client.post("/create-agent", files=files, headers=HEADERS)
    assert response.status_code in (200, 500)

def test_chat_history_endpoints(mocker):
    ...
```

Code D.3: Tests for the data utilities module (`test_data_utils.py`).

```
from data_processing.data_utils import (
    get_item_headers,
    get_user_headers,
    get_inter_headers,
    get_datatype,
)

def test_get_item_headers():
    item_headers = get_item_headers(
        ["iid:token", "movie_title:token_seq", "genre:token", "release_year:float"]
    )
    assert item_headers.item_id_column == "iid:token"
    assert item_headers.name_column == "movie_title:token_seq"
    assert item_headers.category_column == "genre:token"

    assert (
        get_item_headers(["iid:token", "movie_title:token_seq"]).category_column == None
    )

def test_get_user_headers():
    user_headers = get_user_headers(
        ["USER_IDENTIFICATION:token", "USER_AGE:float", "USER_NAME:token"]
    )
    assert user_headers.user_id_column == "USER_IDENTIFICATION:token"

def test_get_inter_headers():
    inter_headers = get_inter_headers(
        ["user_id:token", "item_id:token", "rating_value:float", "timestamp:float"]
    )
    assert inter_headers.user_id_column == "user_id:token"
    assert inter_headers.item_id_column == "item_id:token"
    assert inter_headers.rating_column == "rating_value:float"

def test_get_datatype():
    assert get_datatype(["My name is Caesar", "The birth of a mother"]) in (
        "token",
        "token_seq",
    )
    assert (
        get_datatype(["Action Comedy", "Adventure", "Drama Animation"]) == "token_seq"
    )
    assert get_datatype(["Comedy", "Adventure", "Animation"]) == "token"
    assert get_datatype([1.12, 2.1, 3.1]) == "float"
    assert get_datatype([1, 2, 3]) in ("token", "float")
    assert get_datatype(["1 2 3", "2 3 3", "3 1 1"]) in ("float_seq", "token_seq")
    assert get_datatype(["[1 2 3]", "[2, 3, 3]", "(3 1 1)"]) in (
        "float_seq",
        "token_seq",
    )
```

Code D.4: Tests for the `FalkorDBRecommender` class (`test_falkordb_recommender.py`) [1/3].

```python
import pytest
import os
import shutil
import pandas as pd

from recsys.falkordb_recommender import FalkorDBRecommender

# Mock test dataset directory
@pytest.fixture(scope="module")
def test_data_dir():
    dir_path = "./tmp"
    os.makedirs(dir_path, exist_ok=True)

    users_df = pd.DataFrame({"user_id:token": ["1", "2", "3"], "age:int": [25, 30, 35]})
    items_df = pd.DataFrame(
        {
            "item_id:token": ["101", "102", "103", "104"],
            "name:string": [
                "Toy Story",
                "Jumanji",
                "Grumpier Old Men",
                "Waiting to Exhale",
            ],
            "category:string_seq": [
                "Animation Children Comedy",
                "Adventure Children Fantasy",
                "Comedy Romance",
                "Comedy Drama Romance",
            ],
        }
    )
    inters_df = pd.DataFrame(
        {
            "user_id:token": ["1", "1", "2", "2", "3", "3"],
            "item_id:token": ["101", "102", "101", "103", "102", "104"],
            "rating:float": [5.0, 3.0, 4.0, 5.0, 2.0, 4.0],
        }
    )

    users_df.to_csv(f"{dir_path}/test_rec.user", index=False, sep="\t")
    items_df.to_csv(f"{dir_path}/test_rec.item", index=False, sep="\t")
    inters_df.to_csv(f"{dir_path}/test_rec.inter", index=False, sep="\t")

    yield dir_path

    shutil.rmtree(dir_path)
```

Code D.5: Tests for the `FalkorDBRecommender` class (`test_falkordb_recommender.py`) [2/3].

```python
# Mock FalkorDBRecommender graph
@pytest.fixture(scope="module")
def recommender(test_data_dir):
    rec = FalkorDBRecommender(
        dataset_name="test_rec",
        dataset_dir=test_data_dir,
        graph_name="test-recommender-graph",
        clear=True,
    )
    yield rec

    rec.g.delete()

def test_initialization(recommender):
    assert recommender.graph_name == "test-recommender-graph"

    user_count = recommender.g.query("MATCH (n:User) RETURN count(n)").result_set[0][0]
    item_count = recommender.g.query("MATCH (n:Item) RETURN count(n)").result_set[0][0]
    rating_count = recommender.g.query(
        "MATCH ()-[r:RATED]->() RETURN count(r)"
    ).result_set[0][0]
    assert user_count == 3
    assert item_count == 4
    assert rating_count == 6
    assert recommender.global_avg is not None

def test_create_user(recommender):
    new_user_id = "4"
    created_id = recommender.create_user(new_user_id)
    assert created_id == new_user_id
    user_exists = recommender.g.query(
        "MATCH (u:User {user_id: $user_id}) RETURN u", params={"user_id": new_user_id}
    ).result_set
    assert len(user_exists) == 1

def test_add_user_properties(recommender):
    recommender.add_user_properties("1", {"country": "USA"})
    props = recommender.g.query(
        "MATCH (u:User {user_id: '1'}) RETURN u.country"
    ).result_set[0][0]
    assert props == "USA"

def test_add_user_interactions(recommender):
    user_id = "1"
    initial_interactions = recommender.get_items_by_user(user_id)
    assert "104" not in initial_interactions
    recommender.add_user_interactions(user_id, [("104", 4.5)])
    new_interactions = recommender.get_items_by_user(user_id)
    assert "104" in new_interactions
```

Code D.6: Tests for the `FalkorDBRecommender` class (`test_falkordb_recommender.py`) [3/3].

```python
def test_get_unique_feat_values(recommender):
    categories = recommender.get_unique_feat_values("Item", "category")
    expected_categories = ["Animation", "Children", "Comedy", "Adventure", "Fantasy", "Romance",
        "Drama"]
    assert all(cat in categories for cat in expected_categories)
    ages = recommender.get_unique_feat_values("User", "age")
    assert set(ages) == {25, 30, 35}

def test_get_items_by_user(recommender):
    items = recommender.get_items_by_user("2")
    assert set(items) == {"101", "103"}

def test_get_users_by_item(recommender):
    users = recommender.get_users_by_item("101")
    assert set(users) == {"1", "2"}

def test_recommend_contextual(recommender):
    recs = recommender.recommend_contextual(
        user_id="3", # User 3 has not seen item 101 (Comedy)
        item_props={"category": "Comedy"},
        top_n=5,
    )
    assert len(recs) > 0
    # User 3 has seen 102 and 104. Item 101 is a comedy they haven't seen.
    rec_ids = [r[0].properties["item_id"] for r in recs]
    assert "101" in rec_ids

def test_recommend_cf(recommender):
    # User 1 and 2 are similar (both rated item 101).
    # User 2 liked item 103, which user 1 has not seen.
    recs = recommender.recommend_cf(user_id="1", k=1, top_n=5)
    assert len(recs) > 0
    rec_ids = [r[0].properties["item_id"] for r in recs]
    assert "103" in rec_ids

def test_recommend_hybrid(recommender):
    recs = recommender.recommend_hybrid(
        user_id="1", item_props={"category": "Comedy"}, k=1, top_n=5
    )
    assert len(recs) > 0
    rec_ids = {r[0].properties["item_id"] for r in recs}
    assert "103" in rec_ids or "104" in rec_ids

def test_explain_blackbox_recs(recommender):
    explanations = recommender.explain_blackbox_recs(
        user_id="3", item_id="101", shared_props=["category"], min_rating=0
    )
    assert len(explanations) > 0
    explanation_text = " ".join(map(str, explanations))
    assert "You liked" in explanation_text and "similar category" in explanation_text
```

Code D.7: Tests for the `UserProfile` class (`test_user_profile.py`) [1/2].

```python
import pytest

from llm.user_profile import UserProfile, ContextPreference, ContextType

@pytest.fixture
def sample_user_profile():
    return UserProfile(
        user_id=1,
        context_prefs={
            "genre": ContextPreference(
                type=ContextType.DICT, data={"action": True, "horror": False}
            ),
            "min_rating": ContextPreference(type=ContextType.NUM, data=7.0),
        },
        item_prefs={101: 5.0, 102: 4.5},
    )

def test_add_context_def(sample_user_profile):
    new_context = ContextPreference(type=ContextType.BOOL, data=True)
    sample_user_profile.add_context_def("new_context", new_context)
    assert "new_context" in sample_user_profile.context_prefs

    with pytest.raises(ValueError):
        sample_user_profile.add_context_def("genre", new_context)

def test_remove_context_def(sample_user_profile):
    sample_user_profile.remove_context_def("genre")
    assert "genre" not in sample_user_profile.context_prefs

    with pytest.raises(ValueError):
        sample_user_profile.remove_context_def("nonexistent")

def test_update_context_preference_dict(sample_user_profile):
    sample_user_profile.update_context_preference("genre", {"comedy": True})
    updated = sample_user_profile.context_prefs["genre"].data
    assert updated == {"action": True, "horror": False, "comedy": True}

def test_update_context_preference_non_dict(sample_user_profile):
    sample_user_profile.update_context_preference("min_rating", 9.0)
    assert sample_user_profile.context_prefs["min_rating"].data == 9.0

    with pytest.raises(ValueError):
        sample_user_profile.update_context_preference("unknown", 1)

def test_remove_context_preference_keys(sample_user_profile):
    sample_user_profile.remove_context_preference("genre", ["action"])
    assert "action" not in sample_user_profile.context_prefs["genre"].data
```

Code D.8: Tests for the `UserProfile` class (`test_user_profile.py`) [2/2].

```
49  def test_remove_context_preference_all(sample_user_profile):
50      sample_user_profile.remove_context_preference("genre", None)
51      assert sample_user_profile.context_prefs["genre"].data == {}
52
53      sample_user_profile.remove_context_preference("min_rating", None)
54      assert sample_user_profile.context_prefs["min_rating"].data is None
55
56  def test_add_item_preferences(sample_user_profile):
57      sample_user_profile.add_item_preferences([103, 104], [3.0, 2.5])
58      assert sample_user_profile.item_prefs[103] == 3.0
59      assert sample_user_profile.item_prefs[104] == 2.5
60
61  def test_add_item_preferences_mismatched_lengths(sample_user_profile):
62      sample_user_profile.add_item_preferences([105], [1.0]) # OK
63      assert sample_user_profile.item_prefs[105] == 1.0
64
65  def test_remove_item_preferences(sample_user_profile, capsys):
66      sample_user_profile.remove_item_preferences([101])
67      assert 101 not in sample_user_profile.item_prefs
68
69      sample_user_profile.remove_item_preferences([999]) # Not in prefs
70      captured = capsys.readouterr()
71      assert "Item 999 not found" in captured.out
```

Code D.9: Tests for the `FalkorDBChatHistory` class (`test_falkordb_chat_history.py`).

```python
import pytest

from llm.falkordb_chat_history import FalkorDBChatHistory

@pytest.fixture(scope="module")
def chat_history():
    ch = FalkorDBChatHistory(graph_name="test-chat-history")
    yield ch
    ch.g.delete()

def test_store_chat(chat_history):
    chat_id = 1
    chat = [
        {"role": "user", "content": "Hello"},
        {"role": "assistant", "content": "Hi there!"},
    ]
    chat_history.store_chat(chat_id, chat)
    stored = chat_history.get_chat(chat_id)
    assert stored == chat

def test_append_message(chat_history):
    chat_id = 2
    initial = [{"role": "user", "content": "Start"}]
    new_message = {"role": "assistant", "content": "Go on"}
    chat_history.store_chat(chat_id, initial)
    chat_history.append_message(chat_id, new_message)
    updated = chat_history.get_chat(chat_id)
    assert updated is not None
    assert len(updated) == 2
    assert updated[-1] == new_message

def test_get_chat_not_found(chat_history):
    assert chat_history.get_chat(9999) is None

def test_list_chats(chat_history):
    chat_id = 3
    chat = [{"role": "user", "content": "Ping"}]
    chat_history.store_chat(chat_id, chat)
    all_chats = chat_history.list_chats()
    assert any(c["id"] == chat_id for c in all_chats)
    assert all(isinstance(c["messages"], list) for c in all_chats)

def test_delete_chat(chat_history):
    chat_id = 4
    chat = [{"role": "user", "content": "Bye"}]
    chat_history.store_chat(chat_id, chat)
    chat_history.delete_chat(chat_id)
    assert chat_history.get_chat(chat_id) is None
```


UÁM
Universidad Autónoma
de Madrid