AI-Powered Chatbot for Quick Access to Jenkins Resources

Google Summer of Code Program 2025 Project Final Report

Project Repository

https://github.com/jenkinsci/resources-ai-chatbot-plugin

Giovanni Vaccarino
giovannivaccarino03@gmail.com
https://github.com/giovanni-vaccarino

Project Abstract

Beginners often struggle to move the first steps into Jenkins' documentation and resources. Today, with the possibility of building state-of-the-art AI tools, this project aims to create a Jenkins plugin integrating an AI-powered chatbot. This chatbot will reduce the learning curve for newcomers and offer quick, intuitive support for all Jenkins users through a simple user interface.

Project Description

Introduction

This project focuses on the development of a Jenkins plugin that embeds a chatbot powered based on a Retrieval-Augmented Generation (RAG) architecture. The main goal is to enhance Jenkins' environment by integrating a plugin that embeds an assistant that can provide contextual and accurate responses based on project documentation, developer discussions, and related resources.

The report is organized into several key chapters, each highlighting a major stage of the work done:

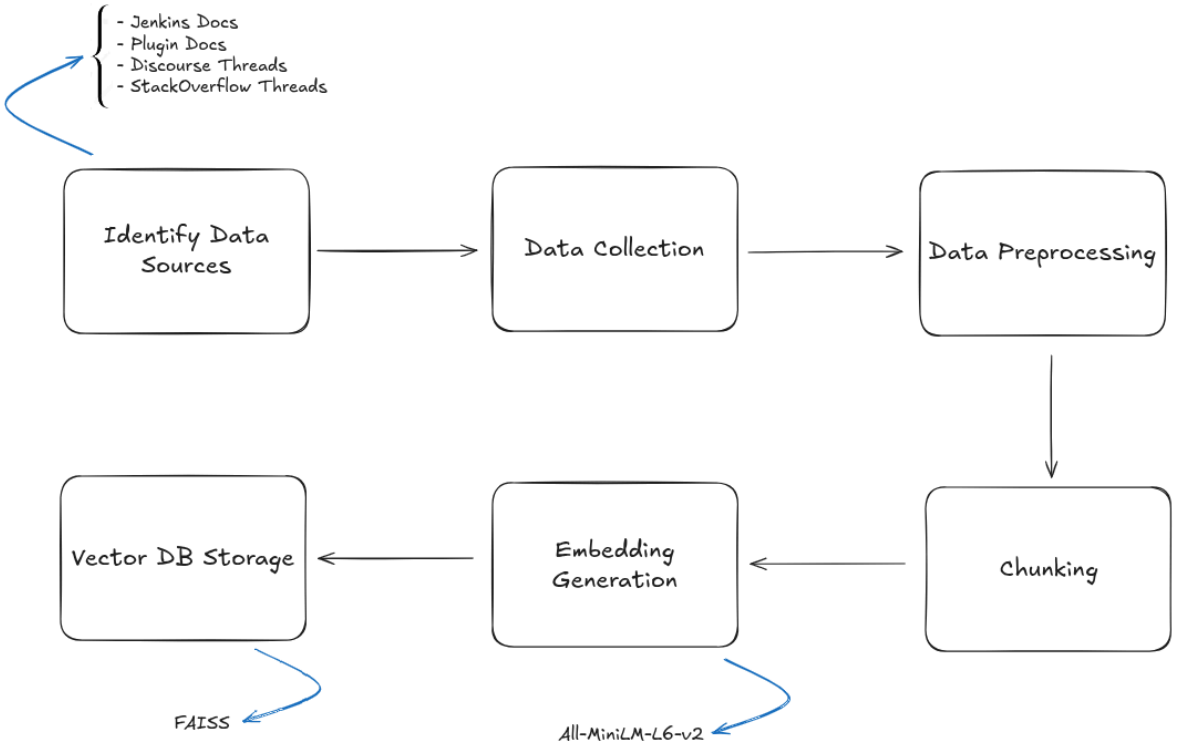
- **Data Pipeline:** The foundation of the chatbot lies in its data pipeline. This component is responsible for preparing the source data, splitting it into meaningful chunks, generating embeddings, and storing them in a vector database. This enables the

creation of a retriever capable of performing semantic search over the stored knowledge, ensuring that the chatbot can ground its answers in relevant context.

- **API Implementation:** To serve the chatbot's functionalities, I designed and implemented an API layer. This allows the system to handle user queries, interact with the retriever, and return responses.
- **User Interface:** A user-friendly UI was developed within Jenkins to provide seamless access to the chatbot. This interface is designed to be intuitive and responsive, ensuring that utilizers can easily interact with the assistant.
- **Testing:** The project includes a thorough testing phase that ensures the correctness of the chatbot.
- **Agentic Approach and Hybrid Search:** Beyond the classical RAG pipeline, we introduced an agentic approach to better handle complex user queries. This extended the chatbot's capabilities by enabling tool usage. We also integrated a hybrid search strategy, combining semantic and keyword-based retrieval for improved accuracy and robustness.
- **Documentation:** Alongside the development, documentation was produced to support both users and contributors. This includes usage guides, development instructions, and design rationales to ensure the project's sustainability.
- **Release Process:** Finally, there is the release process, detailing how the plugin was packaged, versioned, and prepared for deployment.

Data Pipeline

The data pipeline forms the backbone of the chatbot. Its goal is to collect relevant Jenkins-related resources, clean and normalize the content, transform it into a machine-readable form, and store it in a way that enables efficient semantic search. This section describes the main phases of the pipeline: from data acquisition to vector storage.



Source Identification

The first step consisted of identifying the most valuable information sources for Jenkins users. Four categories of resources were selected:

- **Official Jenkins documentation**: the primary reference for Jenkins features and architecture - https://www.jenkins.io/doc/
- **Plugin documentation**: Important for information about the ecosystem of Jenkins plugins, which form a large part of the user experience - https://plugins.jenkins.io/
- **Community discussions on Discourse**: a rich source of troubleshooting knowledge and best practices shared by contributors and users - https://community.jenkins.io/t/welcome-to-discourse/7
- **Stack Overflow threads**: widely used by developers for problem-solving, containing concrete issues and solutions.

Data Collection

Each data source required a tailored collection strategy:

- **Jenkins documentation**: Instead of relying on the GitHub repository of jenkins.io, a web crawl**er** was used. This choice makes it easier to automate future updates and ensures that the extracted content reflects the published website structure.
- **Plugin documentation**: A two-step approach was used. First, the list of official plugin names was fetched, and then a simple crawler retrieved the corresponding documentation pages for each plugin.
- **Discourse threads**: The Discourse API was leveraged to fetch relevant topics directly, taking advantage of its structured endpoints to collect discussions efficiently.
- **Stack Overflow threads**: Queries were executed through the StackExchange Data Explorer, which allowed precise filtering of Jenkins-related discussions, focusing on high-quality threads. In the repository you can find the query used to fetch the threads.

In the repository collection markdown file can be found the details about the collection phase.

Related PR: #1

Data Preprocessing

Raw data from the different sources required careful cleaning and filtering before being used downstream:

- **Jenkins documentation**: Pages were parsed to extract only the main content while discarding navigation menus, scripts, images, table of contents blocks, and HTML comments. A filtering step removed low-value entries, such as pages with fewer than 300 visible characters or with an excessive link-to-text ratio. This eliminated placeholders and index-like pages, while still retaining important developer and non-developer documentation.

- **Plugin documentation**: Since plugin pages often contained boilerplate, a preprocessing step removed unnecessary HTML elements and filtered out trivial descriptions with fewer than 60 characters. This step ensured that only meaningful plugin content was passed forward.

- **Discourse threads**: Cleaning was largely handled during collection, as the API already provided structured discussions.

- **Stack Overflow threads**: Threads were pre-filtered during collection to include only Jenkins-related questions with positively scored, accepted answers. No additional preprocessing was required at this stage.

In the repository preprocessing markdown file can be found the details about the collection phase.

Related PR: #3

Chunking

To enable efficient semantic search, large documents and threads had to be divided into chunks of manageable size. The chunking process was guided by the following logic:

- Preserve semantic coherence, avoiding mid-sentence splits.

- Leverage HTML structure (e.g., headings, lists) where possible.

- Apply sliding windows to retain context overlap between adjacent chunks.

To perform the chunking we used the LangChain's Text Splitter, that give a quick implementation of splitters.

Related PR: #6

Embedding

After splitting the collected documents into smaller chunks, the next phase involves converting those chunks into numerical vectors. The purpose of the embedding phase is to map each chunk of text into a high-dimensional vector representation, enabling semantic similarity search.

- **Why embeddings?** Embeddings capture semantic meaning, allowing the retriever to find relevant passages even if the query does not use the exact same words.

- **How?** Each chunk was passed through an embedding model that outputs fixed-length vectors. For this project, a dense embedding model was chosen to balance accuracy and computational cost. In particular *all-minilm-l6-v2* has been chosen, that is a sentence transformer model.

Related PR: #9

Vector Storage

The final step was storing the embeddings into a vector database, which acts as the backbone of the retriever. The chosen database supports efficient nearest-neighbor search, ensuring that relevant chunks can be quickly retrieved. The vector DB used is FAISS.

All chunks, regardless of their origin (official docs, plugins, Discourse, or Stack Overflow), were stored together in a unified index. This design allows queries to seamlessly retrieve information across different sources.

At this stage, the retriever relies solely on semantic similarity. However, in later chapters, we will introduce hybrid search techniques that combine semantic embeddings with keyword-based approaches to further improve retrieval accuracy.

Related PR: #9

API Implementation

The API is the central interface that exposes the chatbot's functionality as a service. Its primary scope is to provide endpoints that manage chat sessions and provide responses by the chatbot.

Related PR: #10

Tech Stack

The implementation is built with FastAPI, a modern Python web framework chosen for its performance and ease of use. FastAPI also offers automatic OpenAPI documentation and strong typing through Pydantic models, simplifying development and future integration.

The server is powered by Uvicorn, an ASGI server optimized for fast responses and concurrency. This choice allows the API to efficiently handle multiple simultaneous chat requests.

For the underlying model execution, the API integrates with llama.cpp library, enabling efficient inference of large language models on local hardware. This decision allows contributors and users to experiment with the chatbot without relying on external APIs.

An extendible architecture

A core design goal was to make the API architecture modular and extensible. The implementation follows a clean separation of concerns:

- **Controller layer** (`api/routes/`): Handles FastAPI routes, request validation, and HTTP status codes.
- **Service layer** (`api/services/`): Encapsulates the chatbot logic, including conversation memory, retrieval, and response generation.
- **Model/schema definitions** (`api/models/`): Defines Pydantic classes and the abstract LLM provider interface.
- **Prompt builder** (`api/prompts/`): Ensures structured and consistent prompts across different queries.
- **Configuration** (`api/config/`): Manages environment settings and API parameters.

LLM Abstraction

The API introduces an LLMProvider abstraction layer to decouple chatbot logic from the underlying model. Currently, it uses the Mistral 7B Instruct (v0.2, GGUF Q4_K_M) model, run locally via llama.cpp. This choice provides a good balance between performance and quality for local development.

However, the abstraction allows future providers to be plugged in easily, such as OpenAI or Gemini Models via API keys**.** This extensibility ensures that users with limited compute resources can opt for hosted LLMs while others can run models locally, making it work even offline.

Available endpoints

The API currently exposes three main endpoints:

- `POST /api/chatbot/sessions` → Creates a new session and returns a `session_id`.
- `POST /api/chatbot/sessions/{session_id}/message` → Sends a message to the chatbot and returns its reply.
- `DELETE /api/chatbot/sessions/{session_id}` → Deletes an existing session.

These endpoints provide a minimal but sufficient interface for managing conversations, with the potential to expand (e.g., session persistence, advanced configuration).

Session Memory Management

Conversation context is maintained using LangChain's ConversationBufferMemory, stored in a dictionary keyed by session_id. This allows each session to preserve its chat history, ensuring more coherent and context-aware responses. Currently, the memory is managed in-memory.

User Interface

The user interface (UI) of the chatbot was designed to integrate seamlessly into the Jenkins environment. Delivered as part of the plugin, the UI allows developers and operators to interact directly with the chatbot from within Jenkins without relying on external applications. The scope of this component is to provide an accessible, intuitive entry point to the assistant, ensuring that the functionality of the backend API and retriever pipeline is exposed in a user-friendly manner.

Related PRs: #7, #13

Tech Stack

The frontend is implemented using modern web technologies:

- **Framework**: React, chosen for its component-based architecture, flexibility, and strong community support.
- **Language**: TypeScript, providing a better type safety and maintainability for a growing codebase.
- **Build Tool**: Vite, enabling fast development builds and optimized production bundles.

These technologies were selected to balance performance, maintainability, and developer experience. The resulting frontend is bundled and served as part of the Jenkins plugin, ensuring a smooth installation process for users.

UI Integration in Jenkins

To make the chatbot accessible across the Jenkins interface, the plugin uses a global decorator extension. This injects the chatbot panel into every page of Jenkins, so users can interact with the assistant regardless of where they are in the interface.

Once installed, the chatbot can be accessed via a button located in the bottom-right corner of the Jenkins UI. This placement ensures minimal disruption to existing workflows while keeping the assistant readily available.

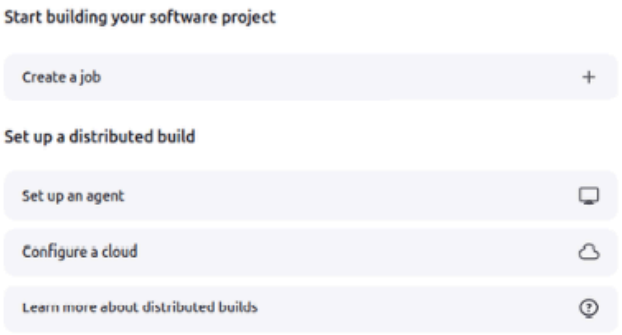
User Experience

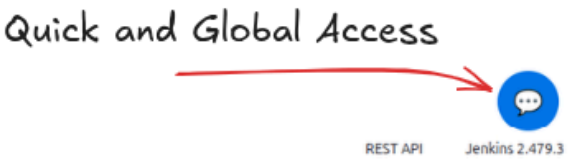
The chatbot UI is designed to be simple and intuitive:

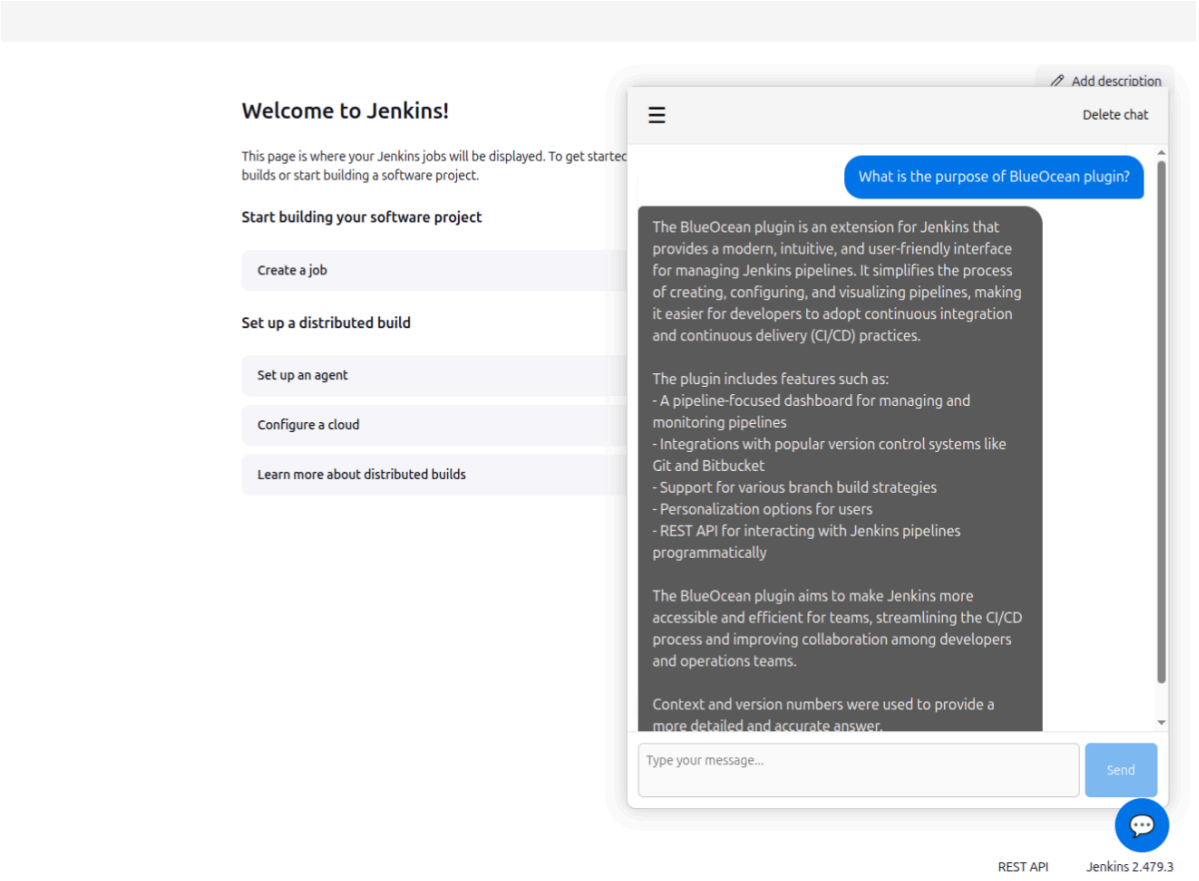
- Users can type or paste questions into a text box that supports both single-line and multi-line input.
- A **“**Clear Chat**”** button in the header resets the conversation.
- Sessions persist during page refresh, maintaining continuity within the same browser tab.

This design mirrors familiar chat interfaces, reducing the learning curve and making the chatbot feel natural to use.

Preview







Testing

Testing played a central role in the development of this project. Given that the chatbot integrates multiple components, from data processing to API endpoints and frontend interactions, it was crucial to ensure correctness, stability, and maintainability throughout the development cycle. Testing has been also integrated into the CI of the project, allowing the system to continuously verify that new changes did not break existing functionality.

Related PRs: #14, #17, #18, #21, #22, #23

Backend Testing

For the backend, testing was implemented using pytest, a widely adopted testing framework for Python. Unit tests covered individual modules such as data preprocessing utilities, API services, and memory management. Integration tests ensured that the full request–response cycle functioned correctly, validating interactions between the API, retriever, and language model. In the unit testing environment more mocks were involved, allowing to focus on the specific function's purpose, while for integration tests only the generation function was mocked.

Frontend Testing

On the frontend, testing was carried out using Jest and the React Testing Library. Unit tests validated isolated components such as buttons, input fields, and chat messages. This approach ensured that the user interface behaved as intended across different scenarios and that updates to the UI did not disrupt existing features.

Static Analysis

In addition to functional testing, static analysis tools were introduced to maintain a consistent and clean codebase. Linters and formatters were configured both for Python and TypeScript code, enforcing style conventions, catching common programming mistakes, and improving overall readability. This helped to ensure that the project remains accessible to future contributors and aligned with best practices.

Agentic approach

Motivation

Once the baseline RAG-based prototype was complete and product-ready, the next step was to improve the quality and adaptability of responses. A classical RAG pipeline relies on a fixed retriever and a single vector store, which can limit flexibility when dealing with diverse query types. To address this, the project evolved into an agentic architecture, where the language model is empowered to dynamically select the most suitable retrieval strategy and tools depending on the query.

Multi-tool Architecture

Instead of aggregating all data sources into a single vector database, the new design introduces separate vector databases per data source (e.g., Jenkins documentation, plugin documentation, Discourse threads, Stack Overflow discussions).

Each source is wrapped in a dedicated search tool. The LLM acts as an agent, choosing which tool to call and with what parameters. This modular approach ensures that queries are routed to the most relevant context while still allowing the system to combine information across sources when needed.

Hybrid Retrieval

To further improve retrieval quality, each search tool implements a hybrid strategy:

- **Semantic search**: powered by vector similarity between embeddings.
- **Keyword-based search**: implemented using the BM25 algorithm, integrated through the retriv library.

The combination of these two methods ensures both semantic relevance and lexical precision. By mixing the results, the chatbot can better handle both natural language queries and technical keyword-heavy queries (e.g., configuration parameters, plugin names).

Query Handling Flow

The attached diagram illustrates how queries are processed under this agentic approach:

1. **Query Classification**: The agent first classifies whether the query is simple or multi-part.

2. **Tool Selection**: For simple queries, the retriever agent selects the appropriate search tools (e.g., plugin docs, core docs, or threads).

3. **Context Retrieval**: The tool(s) performs both semantic and keyword-based retrieval, and the results are combined.

4. **Relevance Check**: Retrieved content is evaluated.

    - If highly relevant → passed directly to response generation.

    - If low relevance → the system avoids hallucination and responds with a fallback message.

5. **Multi-Question Queries**: if the original query contains multiple sub-questions, the agent splits them, processes each separately using the above steps, and then assembles a final combined response.

Related PRs: #35, #38, #39, #40

Documentation

Proper documentation was a key aspect of the project to ensure clarity, reproducibility, and maintainability. All project-related documentation is stored under the **`docs/`** directory in the repository.

The structure of the documentation mirrors the organization of the codebase, with multiple Markdown (`.md`) files explaining the scripts and the reasoning behind each process. For example, the section on data collection details how different sources such as official documentation, plugin pages, and community discussions are gathered and why each approach was chosen. Similarly, other parts of the documentation walk through preprocessing, chunking and embedding, the API layer, the frontend/UI, and the testing strategy. This way, the documentation not only describes what the code does but also provides context for the design decisions behind it, ensuring that future contributors can easily extend and adapt the project.

Moreover, a Makefile has been created to provide ready-to-run scripts that allows to perform the most important actions, such as running the API, or run the data pipeline procedure.

Related PRs: #25, #28

Release

To make the chatbot plugin available to the Jenkins community, a release process was carried out following the official Jenkins publishing guidelines. This ensured that the plugin could be distributed through the official Jenkins Artifactory and installed directly by users from within their Jenkins instance. The process followed the documentation provided by the Jenkins' Docs, available at https://www.jenkins.io/doc/developer/publishing/releasing-cd/.

Related PRs: #41, #42

What I did - Recap

To recap, over the course of the project I:

- Built a data pipeline to collect, clean, chunk, and embed Jenkins-related resources.
- Developed a retriever backed by FAISS for semantic and hybrid search.
- Implemented a FastAPI backend with modular architecture and LLM abstraction.
- Created a React UI integrated directly into the Jenkins environment.
- Added testing (backend with pytest, frontend with Jest/RTL) and static analysis tools.
- Produced user and developer documentation, and completed the release process.

Current State

The plugin is now fully functional and available for use within Jenkins. The system has been tested and documented, and a packaged release is available for the community. While the core objectives have been achieved, further improvements and features are outlined in the *Future Work* section.

Future Work

The 12 weeks of work resulted in a deployed plugin that delivers the core functionalities as expected. During the second half of the coding period, we also had planned to:

- Implement a Websocket connection to provide a more responsive, real time user experience.

- Carry out a comprehensive evaluation of the chatbot using an LLM-as-a-judge approach, to understand how well the overall system is behaving.

These two were not completed within the current timeline, and remain important points for future development. In addition, several were opened throughout the coding period. These include both feature proposals, such as allowing users to attach images or PDFs to the chatbot, and smaller improvements. All open issues can be found here.

Final Considerations

Overall, the experience has been extremely rewarding. I've learnt a lot, and the support and guidance from my mentors were invaluable. One of the most important lessons I take away is that plans inevitably change. At the start, I expected to define a path and follow it step by step, but in practice the project evolved continuously. Some tasks took less time than I anticipated, while others required much more. At times, I also discovered that certain ideas were not feasible and had to be rethought. This taught me to be more flexible, to adjust my approach as new information arises, and to set more realistic deadlines. Moving forward, I feel much more confident in planning, scheduling, and managing my work effectively.