

AI-Powered Chatbot for Quick Access to Jenkins Resources



Google Summer of Code Program 2025 Project Proposal

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

Promoting time efficiency: Why does Jenkins need an AI chatbot?

I'd bet that most Jenkins developers, especially in their early days, have experienced a solid number of headaches trying to fix code bugs, builds that do not work, dependencies that conflict with each other. Often this requires spending hours on the code, reading the documentation and reading forums. As much as this is a great gym for our career experience, often it leads to hours of frustrations accompanied by coffees. And as far as I consider using forums one of the best ways to learn and go forward, often the issues we encounter would require just simple and good guidance with solid references to the documentation.

I have felt this in first person: I am a beginner in open-source code whose attention has been caught by the Jenkins project, and even though I am a MSc Computer Engineering student, I have experienced serious mental breakdowns in doing simple tasks like building the BlueOcean plugin. My experience with building that plugin lasted for quite a few days, and without spending a few hours, seeking on the Jenkins documentation and opening a discussion on discourse, I am not sure I would have arrived at the solution in such a “short” time. Have I enjoyed these happy days of trying to arrive at a solution? Absolutely. Do I think these days have helped me growing in the environment? Without any doubts! But I also believe that the journey could have been shorter and less painful, not just for me, but for many others. While exploring the forums and reading through Discourse threads, I’ve seen countless examples of beginners running into similar issues. This isn’t just a personal pain point, it’s a pattern.

An AI-powered chatbot is definitely a feature that would have helped me and other developers saving precious time and mental health, and without any doubts a tool Jenkins needs. Especially nowadays, that we live in a period where state-of-the-art AI chatbots have become such a common technology and this feature is no longer just fanta-scientific stuff, I find that having such a tool is mandatory.

Someone could argue that with the birth of super-fancy chatbots such as ChatGPT, Gemini, DeepSeek, we do not need any more chatbots, just ask one of those three and they will solve any of your issues. Well, that’s terribly wrong: again I can bring my own experience, but I am sure lots of Jenkins users can assert that. Even though those tools are really powerful, they are designed to be general-purpose models that lack specialization with respect to the Jenkins ecosystem. They often suggest optional run parameters that do not exist, and propose solutions that would only result in losing time. Hence, having such a specialized assistant tailored within the Jenkins environment is paramount in order to have a solid and specific support.

Until now I have raised a few points regarding this chatbot, especially for beginners to Jenkins, but I would like to point out its utility also for seasoned users and developers of the Jenkins ecosystem. Indeed, even those can benefit from this AI-Powered tool, since they are frequent in consulting documentation, exploring new plugins, or searching for advanced configuration examples. The chatbot, integrated directly into a Jenkins plugin, allows them to retrieve this information instantly without breaking focus or switching contexts. Additionally, as Jenkins evolves, staying updated with the latest features and best practices becomes crucial — and this chatbot can serve as a dynamic, always available resource for quickly browsing updates and community solutions, enhancing productivity and supporting continuous learning.

To summarize, this AI-powered chatbot will be a must-have tool for every user, whether their confidence or expertise with Jenkins and whether they are developers or simply users. In particular:

- For users, it will provide quick, reliable access to documentation, plugin information, and community resources without leaving the Jenkins environment, reducing time spent searching and allowing them to stay focused on their tasks
- For beginner developers and contributors, it will offer essential guidance on building, troubleshooting, and understanding Jenkins’ architecture, making the learning curve less steep and contribution efforts more approachable
- For advanced developers and long-time contributors will benefit from instant access to up-to-date references, advanced configuration examples, and emerging best practices, enabling them to remain productive and informed as the Jenkins ecosystem continues to evolve.

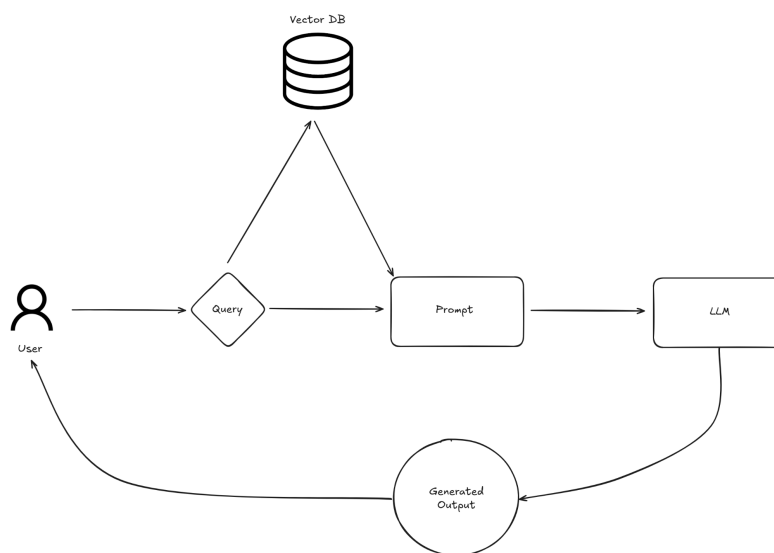
Existing solutions

While Jenkins provides extensive documentation, FAQs, and community chat channels and forums, these solutions are static or human-dependent and can often overwhelm new or intermediate users looking for quick answers. Exploring plugins I also found the Support Core plugin, that offers valuable infrastructure for generating detailed support bundles, which can assist with troubleshooting by collecting system and configuration data. However, this tool is focused on diagnostics for troubleshooting and does not provide interactive explanations or real-time guidance for users. No existing solution offers an AI-powered conversational assistant that can help users navigate Jenkins documentation, explain concepts in simple terms, suggest relevant plugins, or guide troubleshooting steps in real-time.

In other ecosystems, the success of AI-powered assistants has shown how useful it can be to have those tools, and in particular to have them integrated into our environment. For instance, GitHub Copilot is delivered as a plugin inside popular development environments, allowing developers to receive code suggestions and explanations without leaving their workflow. Another crucial factor is the specificity of the AI-tool. For example, in the Kubernetes ecosystem, tools like Botkube provide an ecosystem-specific assistant that allows users to query cluster status, receive alerts, and troubleshoot issues directly through chat interfaces, offering solid and specific guidance that understands the unique complexities of that platform.

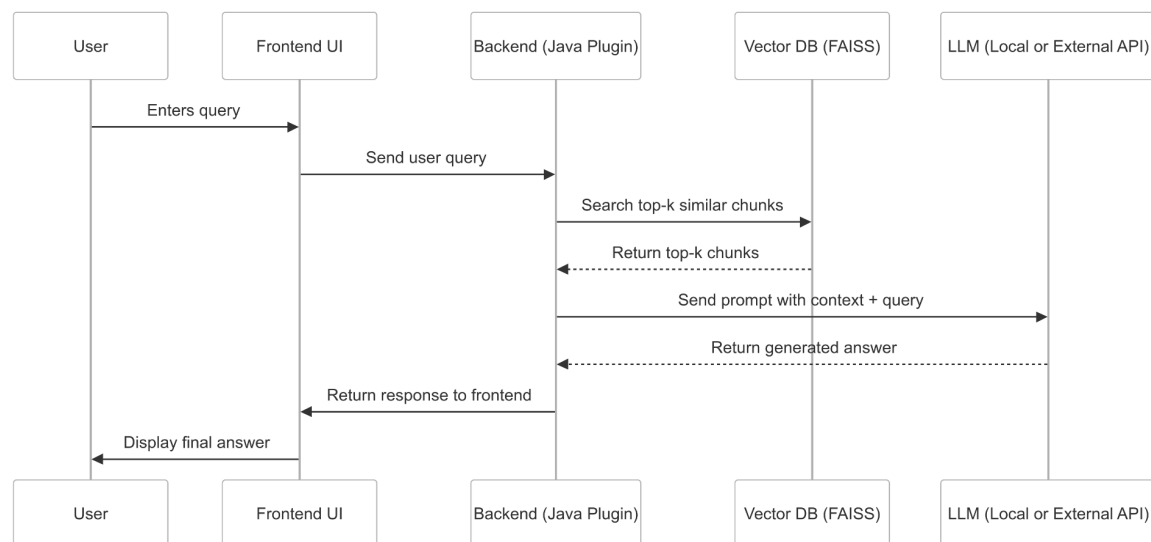
Inspired by these, my proposal aims to build an AI-powered chatbot as a Jenkins plugin — not just a general-purpose assistant, but one deeply integrated into Jenkins, designed to understand its terminology, workflows, and user needs. This chatbot will provide a faster, more focused and more intuitive support experience.

Core Design Approach: Retrieval-Augmented Generation (RAG) as the Foundation



The core design approach for the chatbot will be based on a Retrieval-Augmented Generation (RAG) model, which is currently considered a state-of-the-art approach for building domain-specific chatbots. RAG allows large language models to generate responses by leveraging external information sources, rather than relying on static knowledge embedded in the model weights. This approach ensures that the chatbot remains up-to-date with Jenkins documentation, plugins, and community resources without the need for constant re-training, relying instead on periodically updating the vector database with the latest content.

The process works as follows: when the user submits a query through the Jenkins chatbot interface, the backend first encodes this query into an embedding vector. Using this vector, the system searches a vector database containing pre-embedded chunks of Jenkins documentation, plugin descriptions, and community discussion summaries. The most relevant pieces of information are retrieved and combined into a structured prompt, along with the user's query and clear instructions for the language model. This prompt is then passed to an LLM, which generates our final answer. The chatbot then returns this response to the user. In the image below you can see the sequence of interactions that take place during the chatbot's response generation process, from the user query to the final answer:



To manage the prompt, I plan to use LangChain, which provides powerful abstractions for working with prompts and retrieval. The final prompt will combine four core components:

- 1) System instructions → high-level directives guiding the LLM's behavior
- 2) Retrieved context → relevant documentation, or user discussions retrieved from the vector DB
- 3) Chat history → A window of previous user and assistant messages to maintain conversational continuity and context
- 4) User question → the query the user typed

Below you can find an example of the prompt that will be constructed and sent to the LLM after the retrieval part:

```

1 from langchain.prompts import ChatPromptTemplate, MessagesPlaceholder
2 from langchain.schema import HumanMessage, AIMessage, SystemMessage
3
4 system_message = SystemMessage(
5     content="You are an expert AI assistant for Jenkins. Use the provided context and conversation
6         history to help answer user questions accurately and clearly."
7 )
8 prompt_template = ChatPromptTemplate.from_messages([
9     system_message,
10    MessagesPlaceholder(variable_name="chat_history"),
11    HumanMessage(content="Refer to the following context to answer: {retrieved_docs}"),
12    HumanMessage(content="{user_query}")
13 ])

```

Chat history will look something like this:

```

1 chat_history = [
2     HumanMessage(content="Hi, can you help me with Jenkins?"),
3     AIMessage(content="Of course! What part of Jenkins are you working on?"),
4     HumanMessage(content="I'm trying to configure a pipeline"),
5 ]

```

Starting with this basic RAG chatbot model is an ideal choice, as it aligns completely with the core objective of providing accurate and context-aware answers based on Jenkins documentation and community resources. Moreover, this approach establishes a solid and modular foundation that can be easily extended to incorporate more advanced features, such as specialized tool integrations and agents.

Local Model or External API call?

One of the first key design decisions for this project is whether to rely on an externally hosted LLM API or to run the model locally. While external APIs from providers like OpenAI can offer access to powerful models, in our case it is reasonable to choose a lightweight, locally hosted model. This supports portability and offline accessibility, allowing the chatbot to work in diverse environments without requiring an internet connection.

However, running a local model comes with its own set of limitations and challenges, including managing hardware resource constraints, optimizing inference latency, and selecting a model that provides a good balance between size, performance, and portability. Running a model locally requires that the full model weights fit into the machine's available RAM, or into VRAM if using a GPU for inference. This naturally pushes us toward choosing lighter models that can operate on common hardware without compromising too much on output quality.

Looking forward, I refer you to the Future Improvements section of this document for potential enhancements on this side, regarding the possibility of having a configurable approach to choose between using a local model or an external API call.

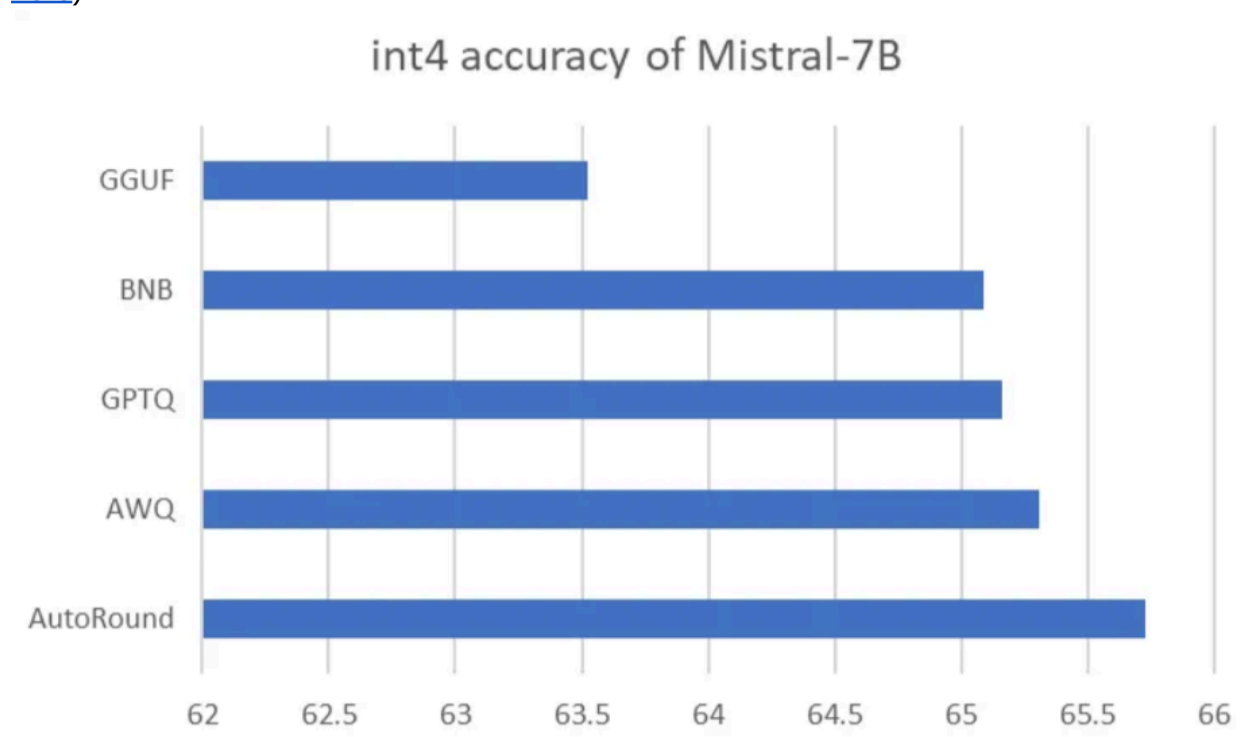
A solution for hardware resource constraints: Quantization

One practical solution is to use quantized models. Quantization is a technique where model weights are stored in lower precision (for example, 4-bit or 5-bit instead of the standard

16-bit or 32-bit floats). Obviously this reduces the memory occupied by the model and makes inference faster, with a limited impact on the response quality. With quantization, models that would require tens of gigabytes of memory can become portable enough to be run on the personal laptops of Jenkins users.

Before diving deeper into the quantization talk, I would like to clear out the doubt of whether it could be better to engage a larger quantized model or a smaller non quantized model. Although it is not an absolute truth, Meta has given some info about it in [this research paper](#). Meta researchers have demonstrated that in many cases, not only does the quantized model result in superior performance, but it also allows reduced latency and enhanced throughput. So not only do they result “better”, but also “faster”.

Going back to quantization, there are two techniques of obtaining quantized models. The first one is Post-Training Quantization(PTQ) that consists in converting the weights of an already trained model to a lower precision, without any retraining. It is straightforward and easy to implement, however PTQ might degrade the model’s performance due to the loss of precision in the value of the weights. The alternative to PTQ is Quantization-Aware Training(QAT) that integrates the weight conversion process during the training stage. This often leads to superior model performance, but it’s more computationally demanding. The gap in the performance is very clear in an article done by Intel(you can find the article [here](#)).



In this image taken from that article, we can see a comparison of five quantization algorithms on the Mistral-7B model. There are both PTQ algorithms like GPTQ and QAT algorithms like AutoRound. As you can see, [AutoRound](#) outperforms the other quantization methods. Moreover AutoRound can accommodate the majority of popular models, whereas other quantization algorithms have limitations. This is why AutoRound models are readily accessible.

On [hugging face](#) we can find many open-source models. In particular Intel has made its own

[leaderboard](#) to focus on quantized models. I plan to exploit this quantized version of [Mistral-7B](#), that occupies approximately 4GB. Assuming that most laptops have at least 8GB of RAM, I think its size is appropriate. These are readily available in formats compatible with local inference engines such as llama.cpp or Ollama. However, in case Mistral-7B (obtained with AutoRound) would result in being too computationally demanding, I already plan to explore the same model, but obtained with a PTQ quantization method that could lead to better execution time.

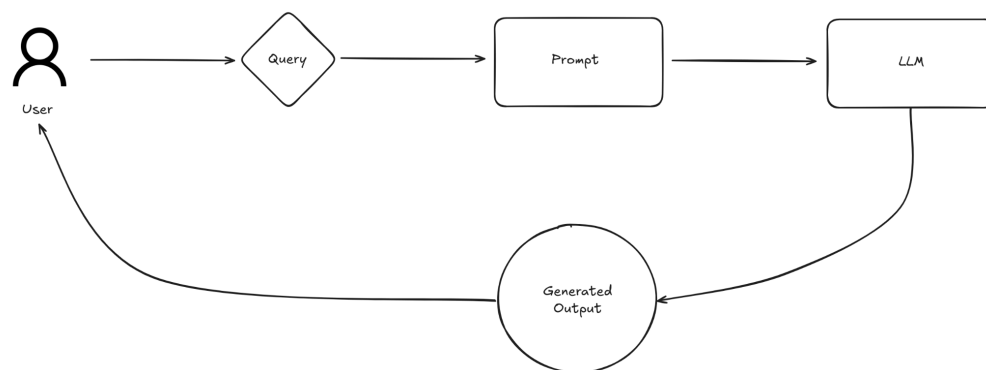
Retrieval over Fine-Tuning

Nowadays, fine-tuning is a popular approach for customizing large language models to specific domains or tasks. Given its visibility, it's important to clarify why I've chosen not to go in that direction for this project. Indeed, while fine-tuning can offer deeper model specialization, I don't think it aligns with the scope and constraints of this project. Fine-tuned models require significant computational resources and infrastructure, making them less suited for lightweight, user-installable tools designed to run in constrained environments. Moreover, fine-tuned models have their knowledge into the model weights, meaning that updating the assistant with newer Jenkins knowledge would require retraining or at least performing costly continual fine-tuning.

Instead, I've opted for a Retrieval-Augmented Generation (RAG) based approach. This allows the model to behave in a domain-specific way without embedding its knowledge in its internal weights.

I noticed there is another GSoC project proposed by Jenkins that explicitly focuses on building a fine-tuned, domain-specific LLM using ci.jenkins.io data. That project has a different scope and different goals from this one.

The simple diagram below illustrates a traditional fine-tuning-based architecture. In this setup, the user's query is passed directly into a prompt, which is then processed by a pre-trained and fine-tuned LLM. The LLM is solely responsible for generating a response, relying entirely on the knowledge embedded in its internal weights. Therefore, there is no external retrieval of updated information.



Prompt Engineering: Unlocking the true potential of the LLM

Although LLMs have impressive generative capabilities, in order to work at their full potential an efficient approach is needed: Prompt Engineering. What Prompt Engineering does is design the process of writing instructions enabling the AI model to provide the

desired answer. Without taking care of this part, even the best models can result in poor results. This makes Prompt Engineering a key part in the project.

The initial version of the assistant presented uses a RAG pipeline. This allows the model to base its answers on the data collected by:

- 1) Accepting the query of the user
- 2) Searching a vectorized index
- 3) Decorating a prompt with the retrieved context

This approach is valid, however it assumes a static logic. Indeed the retrieval will always happen first and the model will simply generate a response based on that context. What's missing is flexibility, making the model able to perform actions like validating its own response, ask a clarifying question or combine multiple reasoning strategies.

To give these flexibilities to the AI-Powered tool, what I plan to do is integrate structured reasoning flows through hains and exploit agents for a dynamic handling.

Chains

Chains are modular data pipelines that define a sequence of steps that the LLM will follow. Each chain will combine prompts, tools or models to execute a specific task. Some possible chain types include:

- Chain of Thought, that promotes a step-by-step logical thinking, by prompting the model to explain its reasoning before giving an answer. For example, if we ask how to set up a jenkins pipeline with kubernetes, the chatbot will walk us through a sequence of steps that lead to the final result. This is very common in many chatbots, like ChatGPT, where given a question like that, what it does is dividing in multiple steps, a sort of "divide et impera" approach.
- Chain of Verification, that adds a verification step after the initial answer. What the model does is rechecking its own answer.
- Rewriting Chain, that is principally used to summarize and transform the retrieved documents, enabling the chatbot to make more complex and technical things more clear and beginner-friendly

My idea is that each chain can be exposed to the system as callable modules thanks to tools.

Introducing the "Tools"

Tools can be considered an extension of the chain concept. They are callable interfaces that can be called(directly or by an agent-covered in the next section) to handle certain types of queries more efficiently or precisely. From an implementation standpoint, tools are simply functions exposed to the LLM.

Tools are particularly valuable for queries where dynamic, structured data retrieval or specific operations are needed. Some examples of these tools include fetching the latest Jenkins LTS version, suggesting some plugins for a given use case, or generating common pipeline configurations. Tools do not encapsulate only data retrieval, but also computations, reasoning and validation. Indeed, those may internally use chains to reason their task(for example chain of thoughts for the tool that suggests plugin), or simply fetch and return raw data(for instance fetching the last version of Jenkins from its website).

In practice, when the chatbot receives a query, in addition to the previous resources, it can also invoke one of these tools, integrating the output of the tools into the response. This

allows the chatbot to not only provide accurate information from documentation but also handle more dynamic, real-time queries. The use of tools also reduces the likelihood of hallucinations by the language model, as answers to fact-based queries can be returned directly from authoritative sources or APIs.

To sum up, I believe that integrating the chatbot with tools is a useful feature, that without any doubts I will include for all the reasons stated above.

Below you can find some examples of simple tools implemented using Langchain:

```
1 from langchain.tools import tool
2 import requests
3 from bs4 import BeautifulSoup
4
5 @tool("GetLatestJenkinsVersion")
6 def get_latest_jenkins_version() -> str:
7     """Fetches the latest Jenkins LTS version from the official Jenkins website
8     """
9     try:
10         response = requests.get("https://www.jenkins.io/changelog-stable/")
11         if response.ok:
12             soup = BeautifulSoup(response.text, "html.parser")
13             header = soup.find("h1", string=lambda text: text and "Latest LTS
14                 release" in text)
15             if header:
16                 version = header.text.split("Latest LTS release:")[1].strip()
17                 return f"The latest Jenkins LTS version is {version}."
18             else:
19                 return "Could not find the latest version on the page."
20         else:
21             return "Not able to fetch the latest Jenkins version."
22     except Exception as e:
23         return f"An error occurred: {e}"
```

```

1 @tool("GeneratePipelineSnippet")
2 def generate_pipeline_snippet(task: str) -> str:
3     """Generates a Jenkinsfile snippet for common tasks."""
4     snippets = {
5         "docker build": """
6         pipeline {
7             agent any
8             stages {
9                 stage('Build Docker Image') {
10                     steps {
11                         script {
12                             docker.build('my-app')
13                         }
14                     }
15                 }
16             }
17         }
18         """,
19         "maven build": """
20         pipeline {
21             agent any
22             stages {
23                 stage('Build with Maven') {
24                     steps {
25                         sh 'mvn clean install'
26                     }
27                 }
28             }
29         }
30         """
31     }
32     for key, snippet in snippets.items():
33         if key in task.lower():
34             return f"Here is a Jenkinsfile snippet for {key}: \n\n{snippet}"
35     return "I don't have a Jenkinsfile snippet for that task yet."

```

Agents

To obtain a dynamic way of handling a user query, I propose the introduction of agents. Agents are systems that enable the chatbot to dynamically decide how to handle a user's query. Instead of following a fixed path (e.g. always performing retrieval), an agent can evaluate the nature of the question and determine whether to:

- Perform a vector database search
- Call a tool
- Chain multiple actions (for example, search documentation and then summarize the findings)

So eventually the agent will dynamically determine what steps are needed to answer the query and which tools to call and in what order. Agents rely on the LLM's ability to reason step-by-step, choosing the appropriate actions based on the query and the available tools.

So what is going to happen is that when a user submits a query, the agent prompts the LLM with both the question and a list of available tools (including retrieval). The LLM then plans and executes a sequence of actions, such as searching the documentation, calling a specific tool, verifying the result, or combining steps. This allows the chatbot to handle a larger range of queries in a smarter and more contextual manner, with a less fixed logic.

A little note that could be not clear is the fact that a few lines before I mentioned that the retrieval part also becomes a tool. In LangChain if agents are involved, the standard approach of performing RAG is to wrap it as a tool, and provide it to the agent alongside the

other tools. To give more context here is a possible simple implementation:

```
1 from langchain.vectorstores import FAISS
2 from langchain.embeddings import HuggingFaceEmbeddings
3 from langchain.chains import RetrievalQA
4 from langchain.tools import Tool
5 from langchain.llms import Ollama
6
7 # Loading the vector store vectorstore and retriever ; More details on this in the next sections
8 embedding = HuggingFaceEmbeddings()
9 vectorstore = FAISS.load_local("jenkins_index", embedding)
10 retriever = vectorstore.as_retriever()
11
12 # Define the LLM
13 llm = Ollama(model="mistral:instruct")
14
15 # Create a RetrievalQA chain
16 rag_chain = RetrievalQA.from_chain_type(
17     llm=llm,
18     retriever=retriever,
19     return_source_documents=True
20 )
21
22 # Wrap the RAG into a Tool
23 retrieval_tool = Tool(
24     name="DataSearch",
25     func=rag_chain.run,
26     description=(
27         "Use this to answer technical questions about Jenkins. "
28         "It searches the documentation and community resources and returns a grounded response."
29     )
30 )
```

Now we can initialize the agent with this and the previous tools(the ones defined as examples).

```

1 from langchain.memory import ConversationBufferMemory
2 from langchain.agents import initialize_agent, AgentType
3 from langchain.prompts import SystemMessagePromptTemplate
4 from langchain.schema import SystemMessage
5 from chatbot_tools import retrieval_tool, get_latest_jenkins_version, generate_pipeline_snippet
6 from langchain.llms import Ollama
7
8
9 tools = [retrieval_tool, get_latest_jenkins_version, generate_pipeline_snippet]
10
11 llm = Ollama(model="mistral:instruct")
12
13 system_prompt = SystemMessage(
14     content=(
15         "You are an expert AI assistant for Jenkins. You can use external tools to look up
16         documentation, fetch plugin details, or generate Jenkins pipeline code. Be concise,
17         accurate, and helpful."
18     )
19 )
20
21 memory = ConversationBufferMemory(memory_key="chat_history", return_messages=True)
22
23 agent = initialize_agent(
24     tools=tools,
25     llm=llm,
26     agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION,
27     verbose=True,
28     agent_kwargs={
29         "system_message": system_prompt
30     },
31     memory=memory
32 )

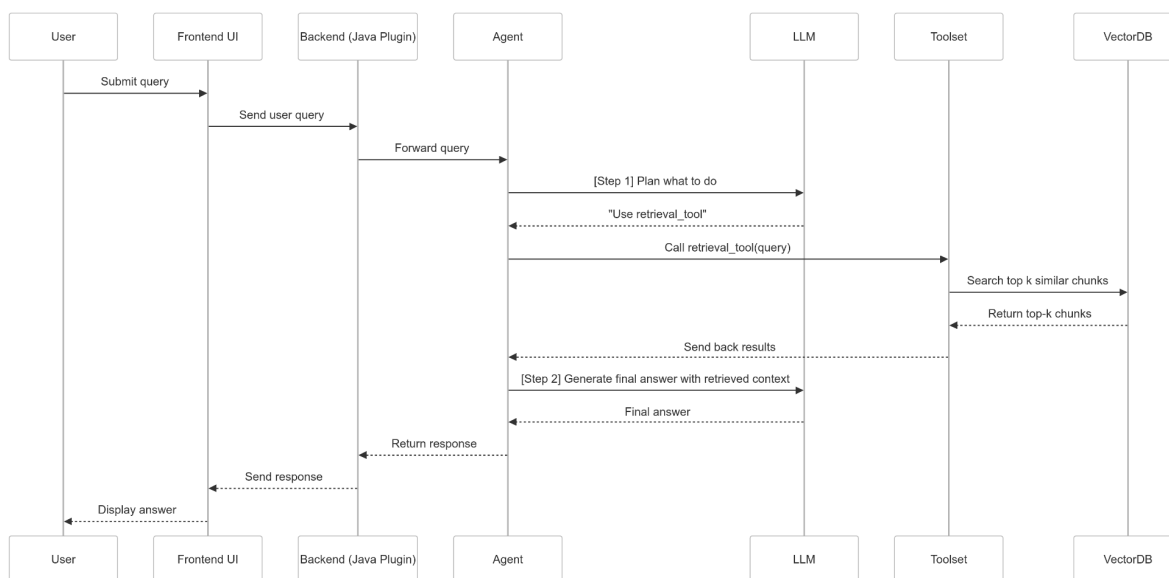
```

A detail I have not covered yet is the selection of the agent type. In LangChain, choosing the appropriate agent type is crucial to ensure that the assistant behaves in a goal-oriented and interpretable way. For this project, I will use the `initialize_agent` method with a ReAct-style agent (Reasoning and Acting), specifically `AgentType.CHAT_CONVERSATIONAL_REACT_DESCRIPTION`. This agent type enables the language model to reason step-by-step, dynamically choosing tools based on their descriptions without requiring fine-tuning or pre-defined examples. It is particularly well-suited for multi-turn interactions, as it is designed to maintain conversational context over time using memory.

However, agents have an important drawback: querying the model multiple times during the reasoning process increases the inference load, potentially leading to longer response times. Moreover, their autonomy can lead to unexpected behaviors or inefficient reasoning paths. Unlike fixed chains, it can happen that they deviate from the intended goal. Despite this, I still consider agents a key building block of my architecture. The reason stands in the acquired ability to dynamically plan actions, chain decisions and allow to handle well use cases that a fixed logic pipeline cannot handle.

In order to provide you a better understanding of the interactions, here you can see a possible example of the chatbot's flow now that agents are involved. In this scenario, the

agent evaluates the user's query and decides to invoke the retrieval tool to fetch relevant context before recalling the LLM to generate a final response. This showcases how the agent dynamically orchestrates the reasoning steps using available tools.



LangChain vs LangGraph

I've just described the agent assistant using LangChain's `initialize_agent` method with a ReAct-style agent. This approach is widely supported and remains well-suited for building modular, tool-using assistants like the one for this project.

That said, LangChain has recently introduced LangGraph, a more advanced framework designed for building agent workflows as composable, stateful graphs. This raises the question of whether to stick with the classic agent interface or adopt LangGraph. For now, I've chosen the LangChain agent API because it offers an excellent balance of simplicity, expressiveness, and aligns well with the current project's scope and timeline.

However, for future enhancements I think that exploring LangGraph could be a good option that can lead to more flexibility, particularly for supporting features like parallel tool usage and retry logics.

Query classification

I also explored the possibility of introducing query classification, which is a lightweight mechanism that categorizes user queries and routes them to the appropriate response method, whether that's document retrieval, a tool call, or a fallback response. Query classification can be considered a simpler, more computationally-efficient alternative to agent-based reasoning, allowing the system to handle predictable, well-defined queries quickly. However, this approach comes with its own issues. It can be rigid and may struggle with edge cases or more nuanced queries that fall outside clear categories. Additionally, maintaining and expanding classification rules over time can become complex as query variety grows.

While I do not plan to implement this as the deliverables of the project, a potential future

enhancement could involve combining both query classification and agents in a hybrid approach. In the future improvements part you can find a better description of the idea.

Details on the core workflow and indexing pipeline

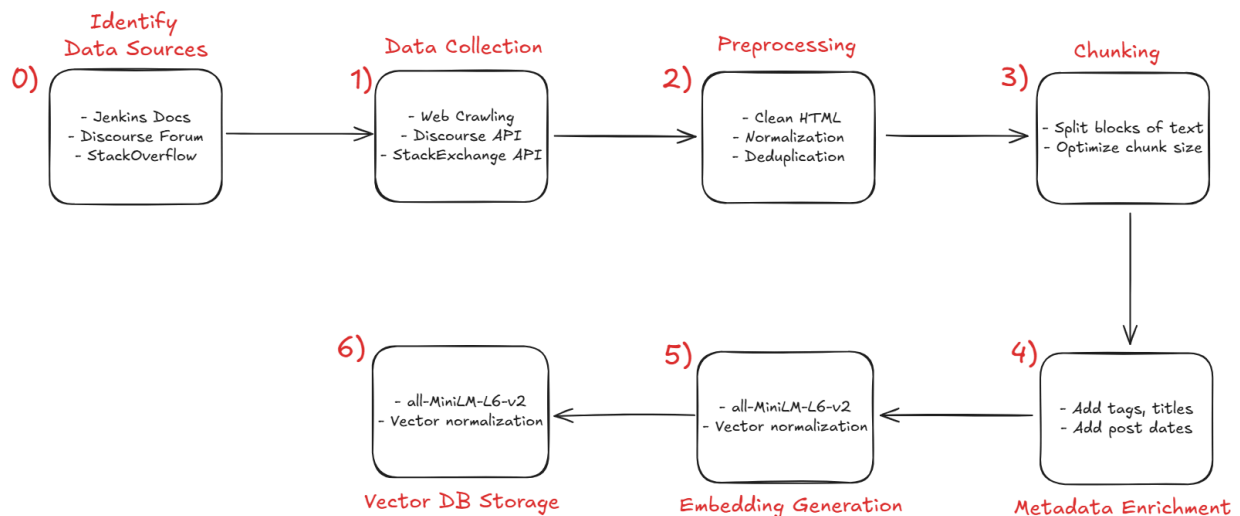
As already mentioned, RAG is the core component of this chatbot, since our main goal is to provide quick access to Jenkins documentation and community resources. Given the sizable and constantly evolving nature of these resources, it is not convenient to encode all relevant knowledge directly into model weights. This makes RAG the ideal approach, allowing the chatbot to dynamically retrieve up-to-date content and deliver context-aware responses without continuous model retraining.

The standard flow will work as follows:

- 1) The user submits a query
- 2) The query is embedded using a sentence embedding model
- 3) A search is performed in the vector database using the query embedding obtained
- 4) The top k most relevant chunks of content are retrieved
- 5) These chunks are included in the prompt

Before this system can function, it is essential to carefully define the data sources to be indexed. These will include:

- Jenkins official documentation
- Jenkins community forum threads on Discourse
- Relevant StackOverflow threads



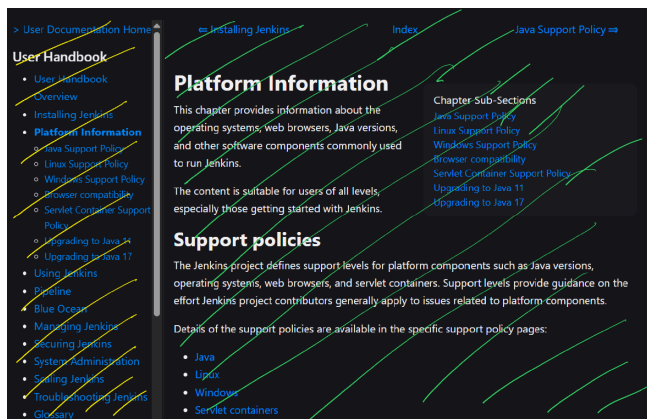
Once the data sources are identified, the next step is designing the indexing pipeline, a critical component of the system. The indexing phase is a crucial one, since it will determine how we will prepare the data for retrieval (you can find a summary of the indexing pipeline in the image above). The indexing phase will consist of:

- 1) **Data collection**, via automated crawling or API access.

For the Jenkins official documentation, I plan to implement a web crawler using

Python libraries such as BeautifulSoup or scrapy(I will probably opt for BeautifulSoup, since the number of pages is not massive- not in the order of millions. Scrapy would have been more appropriate for large-scale scenarios). The collected data will be stored in structured formats (like JSON) for subsequent preprocessing.

While exploring the Jenkins documentation structure and starting to implement some initial fetching scripts, I observed a consistent HTML pattern that can be leveraged for the content extraction. In particular, the main body of the documentation is always enclosed within a `<div>` with the class `container`, which also includes a left-hand sidebar navigation panel(the section with yellow lines in the image below). This sidebar, identified by `id="sidebar-menu"`, is not relevant to our use case and can be safely ignored. The actual documentation content of interest resides in the sibling `<div>` within the same container(the section with green lines in the image below). Based on this, I've already developed a simple script that targets the main container div, removes the sidebar, extracting this way only the relevant documentation part of that page.



For community content, since Jenkins uses Discourse, I will exploit the [Discourse API](#). The API provides access to thread titles, specific categories and tags. I will focus on collecting threads from key categories such as "Using Jenkins" and "Contributing". The data will be enriched with metadata fields like post dates and discussion links and tags, enabling the chatbot to point users directly to active community discussions when appropriate and to prioritize showing more recent threads. For example, rather than providing a generic answer, the chatbot could respond with: "This issue has been actively discussed in the Jenkins community forum. You can find more details and solutions in this thread: Pipeline trigger not working on Git push, posted on June 10, 2024." This little addition can increase the trust of the users in the AI-powered tool, which is paramount. Would you take advice from something you do not trust?

Here you can find a simple script to fetch some discussions(related to the categories specified above):

```

1 import requests
2 from bs4 import BeautifulSoup
3
4 BASE_URL = "https://community.jenkins.io"
5
6 #Fetches at most max_topics threads from a specific category("Using Jenkins" with id 7 ; "Contributing" with id 8)
7 def fetch_category_threads(category_id, max_topics=5):
8     url = f"{BASE_URL}/c/{category_id}.json"
9     response = requests.get(url)
10    if response.status_code != 200:
11        print(f"Failed to fetch category {category_id}")
12        return []
13
14    data = response.json()
15    topics = data.get("topic_list", {}).get("topics", [])[:max_topics]
16    results = []
17
18    for topic in topics:
19        thread = {
20            "title": topic["title"],
21            "slug": topic["slug"],
22            "id": topic["id"],
23            "link": f"{BASE_URL}/t/{topic['slug']}/{topic['id']}",
24            "views": topic["views"],
25            "posts_count": topic["posts_count"],
26            "created_at": topic["created_at"]
27        }
28
29        post_url = f"{BASE_URL}/t/{topic['id']}.json"
30        post_response = requests.get(post_url)
31
32        if post_response.status_code == 200:
33            post_data = post_response.json()
34            first_post = post_data.get("post_stream", {}).get("posts", [])[0]
35            cooked = first_post.get("cooked", "")
36            text = BeautifulSoup(cooked, "html.parser").get_text().strip()
37            thread["preview"] = text[:400] + "..." if len(text) > 400 else text
38        else:
39            thread["preview"] = "Error in the fetching of the preview"
40
41        results.append(thread)
42
43    return results

```

An output example is:

```

5. Failed to initialise k8s secret provider (12 views)
→ https://community.jenkins.io/t/failed-to-initialise-k8s-secret-provider/29332
Preview: I'm running the jenkins-operator in a Kubernetes cluster that sits behind a proxy.
I have a very vanilla CR to start new Jenkins instances. The only tweak my CR has is the JAVA_OPTS required for Jenkins to use the proxy.
http.proxyHost=my.proxy.com -Dhttp.proxyPort=443 -Dhttps.proxyHost=my.proxy.com...

```

To collect relevant StackOverflow discussions, I will use their [public API](#). StackOverflow contains a lot of content contributed by developers. Hence, there could be mistakes. To ensure the quality of the threads collected I will filter them. In particular I will filter for:

- Questions with a minimum number of upvotes
- Questions that have an accepted answer or answers with high vote counts

And obviously I will search only for Jenkins-related tags, to get only threads related to Jenkins. As for the data fetched from the Discourse API, I plan also here to enrich it with metadata for the same reasons. Here you can see a first script for fetching:


```

1 import requests
2 from bs4 import BeautifulSoup
3
4 def fetch_jenkins_questions(pagesize=5):
5     url = "https://api.stackexchange.com/2.3/questions"
6     params = {
7         "order": "desc",
8         "sort": "votes",
9         "tagged": "jenkins",
10        "site": "stackoverflow",
11        "filter": "withbody",
12        "pagesize": pagesize,
13        "accepted": True
14    }
15
16    response = requests.get(url, params=params)
17
18    if response.status_code != 200:
19        print(f"Error in fetching: {response.status_code}")
20        return []
21
22    data = response.json()
23    results = []
24
25    for item in data.get("items", []):
26        question = {
27            "title": item.get("title"),
28            "link": item.get("link"),
29            "score": item.get("score"),
30            "body": item.get("body")
31        }
32        results.append(question)
33
34    return results

```

This script queries the StackExchange API for Jenkins questions. The script filters for high-quality threads, in particular for those with accepted answers and high upvotes. Using BeautifulSoup, I also clean the body text and generate short previews to validate the retrieval. The collected data includes the title, link, vote score, and a cleaned preview of the question content. Example of output(to have a better output I've already cleaned the body text with BeautifulSoup):

```

1 1. How to restart Jenkins manually? (814 votes)
2 → https://stackoverflow.com/questions/8072700/how-to-restart-jenkins-manually
3 Preview: I've just started working with Jenkins and have run into a problem. After installing several plugins it said it needs to be restarted and went into a
   "shutting down" mode, but never restarts. How do I do a manual restart?...
4
5 2. Error - trustAnchors parameter must be non-empty (620 votes)
6 → https://stackoverflow.com/questions/6784463/error-trustanchors-parameter-must-be-non-empty
7 Preview: I'm trying to configure my e-mail on Jenkins/Hudson, and I constantly receive the error: java.security.InvalidAlgorithmParameterException: the trustAnchors
   parameter must be non-empty I've seen a good amount of information online about the error, but I have not gotten any to work. I'm using Sun...
8
9 3. Docker: Got permission denied while trying to connect to the Docker daemon socket at unix:///var/run/docker.sock (616 votes)
10 → https://stackoverflow.com/questions/47854463/docker-got-permission-denied-while-trying-to-connect-to-the-docker-daemon-socket
11 Preview: I am new to docker. I just tried to use docker in my local machine(Ubuntu 16.04) with Jenkins. I configured a new job with below pipeline script. node {
   stage('Build') { docker.image('maven:3.3.3').inside { sh 'mvn --version' } } But it fails with this error: Got perm...
12
13 4. Change email address in Git (460 votes)
14 → https://stackoverflow.com/questions/37805621/change-email-address-in-git
15 Preview: I have a project hosted in Git stash (now rebranded as Bitbucket Server). It is built using Jenkins. Now I made a typo while installing my Git locally. Like
   @ab.example instead of @abc.example After every build, Jenkins sends email notifications and it picks up my incorrect email address from Git co...

```

2) Text cleaning and other preprocessing techniques.

First, I will remove all HTML tags, unwanted markup and special characters. Next, I plan to remove repetitive headers, and other elements that often appear. Whitespaces and line breaks will be normalized in order to maintain consistent formatting.

While the initial plan is to clean the text by stripping HTML tags, removing navigation

elements, and normalizing whitespace, I realized that for Jenkins documentation, this approach could be not the best one. In particular, HTML headers such as `<h1>`, `<h2>` and `<h3>` define the semantic hierarchy of the documentation content. Instead of discarding them, I plan to leverage this structure to better preserve contextual boundaries and relationships within the text. For instance, I can extract each `<h2>` section as a higher-level parent and its associated content (including any `<h3>` subsections) as children, effectively creating a hierarchical representation of the document.

Moreover, for Jenkins documentation, preserving code block structure is essential, so I will ensure that code snippets are retained in their original format.

I also plan to implement deduplication to prevent storing multiple copies of almost the same content, especially for forum threads that may be repetitive(it's always easier to ask the same thing and not search for it). To achieve that there are traditional and more advanced techniques. The basic idea is that with traditional techniques like exact or fuzzy matching you have simpler algorithms that are not able to detect complex similarities. With advanced techniques such as machine learning algorithms you are instead able to capture hard ones. My idea is to go for the traditional techniques for two main reasons:

- The advanced ones add a significant level of complexity in the project pipeline. They can capture more complex patterns, but they require significant training and tuning, and I don't think it's worth it
- I am afraid that the model would accidentally erase relevant data. I prefer to have some more redundancy, but be sure to don't delete important data

Nevertheless, I consider exploring some advanced techniques a good enhancement that could be explored and therefore added in the future improvements section.

3) **Chunking**

Chunking is the process of dividing large documents into smaller sections of text, called chunks, before they are embedded and stored in the vector database. This process affects how precise and efficient the search will be.

Once the raw data is cleaned, I will split it into smaller, semantically meaningful chunks for embedding. An important decision in this phase is the chunk size, finding the right trade-off between context completeness and the retrieval granularity. Indeed smaller chunks allow for a more focused retrieval, while larger chunks provide more context in a single chunk. The chunk size also affects the performance, resulting in faster processing and retrieval time for smaller chunks, but with the drawback of increased storage requirements.

Given these trade-offs, I plan to experiment with different chunk sizes, since there is no one-size-fits-all answer. Indeed the optimal chunk size heavily depends on the specific case we're dealing with, and the best way is to test how it behaves with different sizes. To support this experimentation, I plan to leverage existing evaluation frameworks such as the LlamaIndex Response Evaluation module, which provides tools for systematically comparing different chunk sizes. What we will do is generate sample queries from a document set, then evaluate chatbot responses with multiple chunk sizes using metrics such as response time, faithfulness (absence of hallucination), and relevancy (how well the query is answered).

I plan to adapt different chunking behaviour based on the content type, so whether I'm dealing with Jenkins documentation or forum discussions. In addition I think it would be good to explore also sliding window techniques (to maintain context across chunk boundaries), considering more advanced approaches such as Sentence Window Retrieval, that combine the benefits of both small and large chunks.

I remind you of the future improvements section to eventually explore other advanced chunking optimization techniques, introduced by the recent research paper: ["Mix-of-Granularity: Optimize the Chunking Granularity for Retrieval-Augmented Generation"](#).

4) **Metadata enrichment**

This phase consists in enriching each chunk with the right fields to enable a better retrieval precision, before embedding the data and storing it in the vector database. For example each chunk can be annotated with:

- Title of the page or the discussion
- Content type (documentation, forum post, ...)
- Tags or categories(Pipeline, Installing Jenkins, ...)

This additional information will improve the effective retrieval. For instance if a user's query includes "latest plugin update", the system can bias on results with that tag.

While some metadata enrichment was already done during the data collection phase (such as post dates from the Discourse API), I will attach this metadata to each chunk with the specific purpose of improving the retrieval precision.

5) **Embedding generation**

After chunking and metadata enrichment, the next step is to convert the results into dense vector representations through embedding. These embeddings are numerical encodings that capture the semantic meaning of the chunk, enabling effective similarity search later on in the pipeline.

For this task, I plan to use a state-of-the-art sentence embedding model such as all-MiniLM-L6-v2, from the library sentence-transformers. This model offers an optimal balance between performance and compactness, making it ideal for the project. In particular, the embedding model will encode each chunk into a 384-dimensional dense vector. To ensure consistent similarity comparisons, I will apply L2 normalization on the embeddings. By doing this, we will improve cosine similarity computations and align with FAISS's default configuration.

Here you can see a simple snippet to show its usage:

```

from sentence_transformers import SentenceTransformer

model = SentenceTransformer('sentence-transformers/all-MiniLM-L6-v2')

chunks = [
    "To configure a pipeline in Jenkins, define a Jenkinsfile in your source control repository.",
    "Jenkins can use Kubernetes to dynamically spin up and down agents as needed.",
    "The Jenkins plugin manager allows you to install, update, and remove plugins from the UI.",
    "A stage in a pipeline can contain multiple steps, such as building and testing your code."
]

# Convert chunks to embeddings
embeddings = model.encode(chunks)

print(embeddings)

```

As previously said, each chunk is mapped to a dense vector:

```

[[ 0.00113924 -0.04320055 -0.05748817 ... -0.00857408  0.079579
 -0.07144964]
 [-0.03177112 -0.04164574 -0.01556977 ... -0.02408641  0.04656757
  0.01825408]
 [-0.03298785 -0.03553536  0.01904955 ... -0.02477288  0.1436178
  0.00482674]
 [-0.03037018  0.02068612 -0.01483179 ...  0.02984233  0.09409086
 -0.08533097]]

```

6) Storage into a Vector Database

Once each document chunk has been embedded into a dense vector, these embeddings need to be stored in a vector database that supports fast and accurate similarity search. For this project, I plan to use FAISS, which is an open-source library developed by Meta, designed for efficient similarity search on dense vectors. FAISS guarantees high performances-supporting both CPU and GPU-based indexing-, simplicity and portability, being ideal for local deployments, so in-line with our use case.

For the index an option could be to use a flat index which guarantees exact nearest-neighbor matches. It is easy and simple to use, however it does not scale well with the dataset size, due to its linear search time. A better option is to opt for an approximate index, such as IndexIVFFlat, which trades off a small amount of accuracy for better speed and memory improvements. What it basically does is to divide the data space into clusters, and then perform the searches within these clusters.

Below you can find an example implementation on how to create an Inverted File Index and then how to search:

```

1 import faiss
2 import numpy as np
3
4 # Assuming embeddings is a numpy array of shape (num_chunks, embedding_dim), where each row is an embedding
  vector representing a document chunk
5 embedding_dim = embeddings.shape[1]
6
7 # Define the number of clusters
8 nlist = 100
9
10 # Create a quantizer: a flat index used as the base for clustering
11 quantizer = faiss.IndexFlatL2(embedding_dim)
12
13 # Create the IndexIVFFlat index. It divides the vector space into 'nlist' clusters and performs search over
  the most relevant clusters
14 index = faiss.IndexIVFFlat(quantizer, embedding_dim, nlist, faiss.METRIC_L2)
15
16 # IndexIVFFlat requires training before use -> this step builds the cluster centroids
17 index.train(embeddings)
18
19 # Once trained, we can add all the embeddings to the index
20 index.add(embeddings)

```

```

1 # Perform a search for a single query vector. The query has to be a numpy array of shape (1,
  embedding_dim)
2 query_vector = np.array([query_embedding]).astype('float32')
3
4 # Search for the top 5 most similar chunks
5 distances, indices = index.search(query_vector, k=5)

```

In addition to the comments I provided in the code snippets, I would like to add that the parameter *nlist* can be tuned (usually however it stays between 50 and 200). In general, increasing *nlist* leads to better search time but requires more training data.

A very useful feature of FAISS is its ability to save and load indices. Once we've trained and populated the FAISS index with the embeddings, rebuilding it every time would be a waste of time and resources. We can instead write it once:

```

1 faiss.write_index(index, "jenkins_index.faiss")

```

and then simply load it when needed (for example during the plugin runtime):

```

1 loaded_index = faiss.read_index("jenkins_index.faiss")

```

In case even IndexIVFFlat will still result in performance issues, IndexPQ could be leveraged. It ensures better time efficiency than IndexIVFFlat, but with the drawback of worse accuracy.

So to conclude, chunks will be stored with their metadata, allowing me to filter and format retrievals in a smart way. Frameworks like LangChain or LlamaIndex will

simplify the connection between vector search and prompt assembly.

Since Jenkins documentation and community discussions evolve, another aspect to consider is index maintenance and updates. About this I think that for future improvements this process of index maintenance could be automated. More on this in the future improvements section.

Testing Strategy (AI part)

To ensure that the chatbot performs effectively, it's crucial to perform some testing, both on the retrieval component and on the overall chatbot.

For the *retrieval evaluation* I will use a set of predefined queries with known relevant documents. Some queries could be like:

```
1 queries = {
2   "How to install Jenkins on Windows?": {"doc1", "doc7", "doc10"},
3   "What is a Jenkins pipeline?": {"doc2", "doc5", "doc8"},
4   "Does Jenkins support Java17?": {"doc3", "doc9"},
5 }
```

I will use metrics like Precision@k, Recall@k and Mean Reciprocal Rank to assess how well the retrieval system is able to catch relevant Jenkins documentation or community content. This part of the testing is also helpful to compare different configurations of the indexing pipeline(for example, experimenting with chunk size) to determine which one leads to the best response.

Assuming that a sample response looks like:

```
1 retrieval_results = {
2   "How to install Jenkins on Windows?": [
3     {"id": "doc1", "text": "You can install Jenkins on Windows by downloading the .msi installer"},
4     {"id": "doc4", "text": "Jenkins setup on Linux distributions..."},
5     {"id": "doc7", "text": "Troubleshooting Windows installation issues..."},
6   ],
7   "What is a Jenkins pipeline?": [
8     {"id": "doc5", "text": "Jenkins Pipelines are a way to define your build process using code"},
9     {"id": "doc6", "text": "Pipeline syntax and examples..."},
10    {"id": "doc2", "text": "A Jenkins pipeline consists of stages and steps..."},
11  ],
12  "Does Jenkins support Java17?": [
13    {"id": "doc3", "text": "Jenkins added official support for Java 17 in version 2.x..."},
14    {"id": "doc9", "text": "Supported Java versions for Jenkins..."},
15    {"id": "doc12", "text": "Using Java with Jenkins plugins..."},
16  ],
17 }
```

A simple implementation of a function to calculate these metrics can be:

```

1 def evaluate_retrieval(queries, results, k=3):
2     precision_scores = []
3     recall_scores = []
4     reciprocal_ranks = []
5
6     for query, relevant_docs in queries.items():
7         retrieved_docs = results.get(query, [])[:k]
8
9         retrieved_ids = [doc["id"] for doc in retrieved_docs]
10
11         relevant_found = [doc_id for doc_id in retrieved_ids if doc_id in relevant_docs]
12
13         precision = len(relevant_found) / k
14         recall = len(relevant_found) / len(relevant_docs)
15
16         rr = 0
17         for i, doc_id in enumerate(retrieved_ids):
18             if doc_id in relevant_docs:
19                 rr = 1 / (i + 1)
20                 break
21
22         precision_scores.append(precision)
23         recall_scores.append(recall)
24         reciprocal_ranks.append(rr)
25
26     avg_precision = sum(precision_scores) / len(precision_scores)
27     avg_recall = sum(recall_scores) / len(recall_scores)
28     mean_rr = sum(reciprocal_ranks) / len(reciprocal_ranks)
29
30     return {
31         "Average Precision@k": avg_precision,
32         "Average Recall@k": avg_recall,
33         "Mean Reciprocal Rank (MRR)": mean_rr,
34     }

```

The function takes as input the user queries, the corresponding retrieval results (including doc id and text), and a parameter k that specifies how many top documents to consider for evaluation.

To ensure the *overall quality of the chatbot response* I plan to focus on ensuring:

- Groundedness → Ensuring that the answer provided by the chatbot is based on the retrieved documents. This can be done manually by comparing the answer with the top-k retrieved text chunks and verifying that all factual claims are supported by the source content.
- Task Completion → Ensuring the AI-powered tool provides complete responses
- Hallucination Verification → Ensuring the chatbots do not generate unsupported information. This can be done by comparing the output obtained with the retrieved text
- Resource Monitoring → Given the fact that we are executing on a local machine, ensuring that the chatbot does not require too much time or computing resources is a key point in our use case. Hence, I will monitor its performance across various query types, measuring response time and memory/CPU usage.

LLM as a judge

To support the evaluation of response quality, especially for groundedness and hallucination detection, I plan to adopt LLM as a judge technique. This strategy involves prompting a second LLM to act as an evaluator, assessing whether the assistant's answer is accurate, grounded in retrieved content, and complete, based on clearly defined criteria.

A crucial point is that since the assistant is designed to run locally using a lightweight quantized model, I believe it is not the best candidate for our "judge". In some use cases, the LLM used for the judge can be the same as the one used for the generation. Therefore,

given our resource constraints on the LLM for the generation part, I plan to use a second, more powerful open source LLM as the evaluator. This “judge” model will only be used during development and testing, and will not be part of the deployed plugin, ensuring the plugin itself remains lightweight, efficient, and easy to run in local environments.

While powerful closed-source models like GPT-4 or Claude would ideally offer the best evaluation capabilities, they are accessed only via paid APIs, introducing billing constraints during the development process. To avoid this (and since I think it should be sufficient), I plan to run the evaluator using another open-source LLM that can realistically fit in environments like Google Colab or Kaggle. The evaluator model may have the same architecture (e.g., Mistral 7B) but in a more “uncompressed” version, making it more suitable for evaluation. For the expected usage, such as testing a curated set of prompts or comparing variations, I believe I can manage this process using free resources such as Google Colab or Kaggle.

This approach is particularly valuable in my use case, where responses are expected to be based on technical documentation (such as Jenkins docs). The judge model is presented with the user query, the retrieved context (top-k chunks), and the assistant's answer. It is then instructed—through specific prompts, to assess the consistency of the response, identify unsupported claims, or rate how helpful the response is for task completion.

Using this method aligns well with the project for several reasons. First, it enables a scalable way to review the assistant behavior across many interactions, which would be impractical to verify manually. This will be especially useful when refining some prompts or adjusting the tool's description. Second, because the assistant is designed to be grounded on retrievable documentation. Therefore the correctness of answers can be assessed more objectively.

In particular, here’s how I plan to apply the LLM as a judge approach:

1. Prompt design → I will define tasks like “Evaluate if this response is fully supported by these retrieved documents. Rate also the answer’s usefulness on a 1 to 5 scale”
2. Context Injection → The judge model will receive the query, the retrieved chunks from the vectorDB, and the generated answer in a well defined structured format
3. Usage of the output of the judge → These scores or any other types of evaluation by the judge will help me refine the details of my AI assistant

However, it's important to point out that these LLMs, even when used as evaluators, are not perfect, indeed they can carry biases, produce inconsistent judgments on edge cases, or generate plausible but incorrect assessments. Hence, I will use LLM as a judge as a complementary tool rather than a replacement for human review. I will validate its output also with manual checks, especially during the first development phases.

User Testing and Feedback

In addition to the testing strategies presented, I plan to collect manual feedback from real users, including Jenkins developers and contributors, and my mentors. The feedback I expect will focus on aspects like the clarity of the assistant’s responses and the perceived helpfulness of the latter. In particular, I will rely on more experienced users to identify cases where the chatbot may hallucinate or provide incorrect information, issues that less experienced developers might not recognize. Even with a small group of testers, I believe this process can significantly improve the assistant’s quality. After all, the user perception is one of the most important metrics, as these are the people the assistant is ultimately built

to support.

User interface and Plugin Integration

Dual Chat Interface

The user interface (UI) will play an important role in making the chatbot intuitive and easily accessible within the Jenkins environment. Since this project will be delivered as a Jenkins plugin, I plan to develop the frontend using React, that is in line with Jenkins' modern plugin development stack. React offers a modern development experience. Its component-based structure makes the codebase clean, reusable, and easy to extend. Additionally, I intend to use TypeScript in order to have better type safety, improve maintainability, and have a better development experience, since many runtime errors are caught as compile time errors during the build step introduced. This technology stack also aligns perfectly with my existing skill set.

I plan to integrate the chatbot into Jenkins by adding a dedicated button to the left sidebar, which will open the effective AI-powered assistant page. Additionally, I am also enabling a quick-access available on every page of Jenkins via a floating button in the footer. This dual-access approach offers flexibility: users can either engage with a lightweight assistant on the fly or switch to the full screen experience for more complex, multi-chat interactions. The quick-access version is designed to handle only a single chat session at a time, focusing on fast, transient interactions. While the full page interface supports managing multiple conversations, the footer chatbot will feature a simpler workflow, with a possible "clear chat" or "save and clear" mechanism to optionally save a conversation to the main chat list. I plan to share storage between the two interfaces where appropriate, but maintain this functional separation to simplify the user experience.

The chatbot interface will include:

- A chat window showing the conversation history
- A user input field
- Response bubbles displaying answers from the chatbot, including clickable source links when available
- Loading indicators for LLM inference steps, so users receive clear feedback during response generation
- Features like copying code blocks.

Component libraries

To accelerate the UI development process, I will explore leveraging existing open-source UI component libraries, such as [shadcn](#), which can help avoid building common already existing components. These libraries provide well-tested elements like chat bubbles, buttons, collapsible panels, spinners, and input fields, allowing me to focus more on backend integration and conversational flow. Of course I will use these components, as long as I find them enough customizable to match the desired design

Design overview

For a first idea of the design, look out to the "Initial Plugin Implementation and First Results" section.

Packaging into Jenkins Plugin

The frontend React application will be compiled and bundled using Vite, and placed into the

plugin as static resources under `/src/main/webapp/`. Jenkins automatically serves these static assets at predictable URLs using its Stapler routing mechanism. This allows the chatbot UI to be accessed directly within Jenkins, without requiring an external frontend server. I found more details on exposing bundled resources [here](#).

Approximate Code Structure

Here you can see an approximate visualization of the code structure:

```
chatbot-plugin/
├── frontend/
│   ├── public/
│   ├── src/
│   │   ├── components/
│   │   └── ...
│   ├── package.json
│   ├── vite.config.ts
│   └── ...
├── src/
│   ├── main/
│   │   ├── java/
│   │   │   └── io/.../chatbot/
│   │   │       ├── ChatbotRootAction.java
│   │   │       └── ...
│   │   ├── resources/
│   │   └── webapp/
│   └── test/
├── pom.xml
└── README.md
```

As you can see, as a standard plugin code structure, the Java backend code will reside in `src/main/java`, while the frontend will live inside a separate `frontend/` directory located at the root of the project. This way only the compiled frontend is included in the plugin's .hpi file, while the development environment remains isolated in the `frontend/` directory. Inside the `frontend/` directory lives a typical react application folder structure that contains a folder for components

Upon building the React application, the static assets (HTML, JS, CSS) will be output directly into the plugin's `src/main/webapp/` directory. Indeed the `vite.config.ts` file will look like this:

```

1  import { defineConfig } from 'vite'
2  import react from '@vitejs/plugin-react'
3
4  export default defineConfig({
5    plugins: [react()],
6    base: '',
7    build: {
8      outDir: 'dist',
9      assetsDir: 'assets',
10     rollupOptions: {
11       output: {
12         entryFileNames: 'assets/index.js',
13         assetFileNames: 'assets/index.css'
14       }
15     }
16   }
17 });

```

The plugin will be built using a standard build process (by running `mvn clean install`), which produces a `.hpi` file. Since the React application must be built first, I will add an `npm run build` step to the Maven lifecycle using the `exec-maven-plugin`. This way the compiled frontend assets are placed in `src/main/webapp/` before the `.hpi` file is generated.

Static analysis

To ensure maintainable and high-quality code throughout the project, I plan to use static analysis tools for the frontend, in order to catch potential issues earlier in the development, have a warning (and error) free codebase and to have a standardized coding style. In particular I will use ESLint and Prettier.

Accessing to the chatbot page

Once the plugin is installed the goal is to show a button on the left sidebar of the Jenkins UI, that will let you access the chatbot interface.

In order to do that we will exploit the java file `ChatbotRootAction.java`, that will expose the AI-powered tool under a `/chatbot` route in Jenkins.

Initial Plugin Implementation and First Use Cases

As a continuation of the work described in the previous section, I've started implementing the Jenkins plugin to better visualize what I will effectively build, and to practice working with Jenkins plugin development. To move the first steps, I referred to the [official Jenkins plugin tutorial](#), starting from a basic template to generate the initial project skeleton.

Sidebar integration

I first added a button to the sidebar by creating a `ChatbotRootAction.java` file. This class is annotated with `@Extension` and extends `RootAction`. I also created an `images/` folder under

/webapp/ to store the sidebar icon. To expose the chatbot page, I created an index.jelly file in resources/io/.../ChatbotRootAction/, which loads the UI when users navigate to /chatbot. Here you can see what ChatbotRootAction.java looks like:

```
1 package io.jenkins.plugins.chatbot;
2
3 import hudson.Extension;
4 import hudson.model.RootAction;
5
6 @Extension
7 public class ChatbotRootAction implements RootAction {
8     @Override
9     public String getIconFileName() {
10         return "/plugin/chatbot-plugin/images/chatbot-icon.png";
11     }
12
13     @Override
14     public String getDisplayName() {
15         return "Jenkins AI Assisant";
16     }
17
18     @Override
19     public String getUrlName() {
20         return "chatbot";
21     }
22 }
```

React injection

In the root of the project, I created a frontend/ directory for the React application. As previously mentioned I am using TypeScript for better type safety and Vite for an efficient and simple bundling. For development purposes, I added the following npm script to package.json:

```
"build:jenkins": "npm run build && cp -r dist/* ../src/main/webapp/static/"
```

This builds the React app (with Vite) and copies the output into the Jenkins plugin's static/ folder. Then, I modified index.jelly to inject the app like this:

```
src > main > resources > io > jenkins > plugins > chatbot > ChatbotRootAction > index.jelly
1 <?jelly escape-by-default='true'?>
2 <j:jelly xmlns:j="jelly:core" xmlns:l="/lib/layout">
3   <l:layout title="Jenkins AI Assistant">
4     <l:main-panel>
5       <div id="root"></div>
6       <script type="module" src="${rootURL}/plugin/chatbot-plugin/static/assets/index.js"></script>
7     </l:main-panel>
8   </l:layout>
9 </j:jelly>
```

And finally set up main.tsx like this:

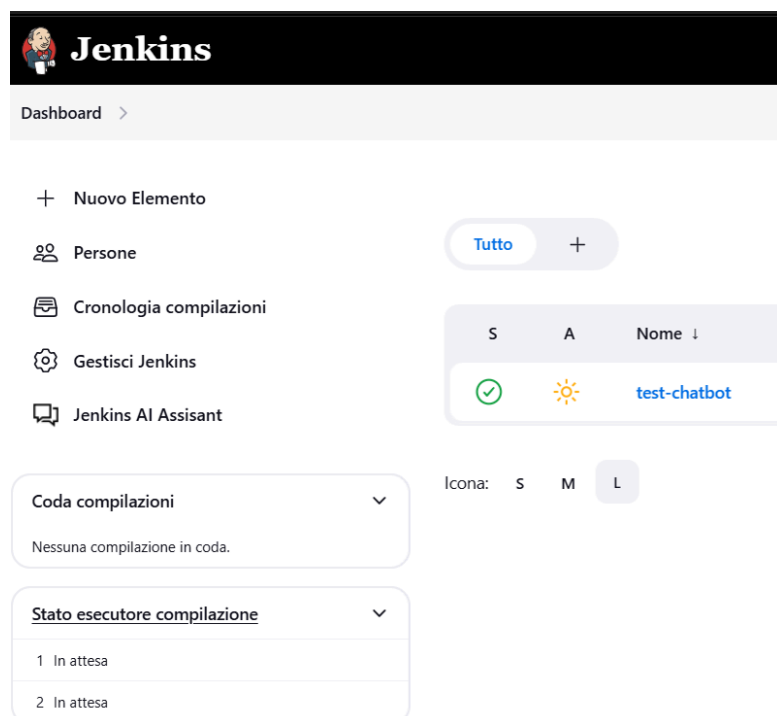
```

1  import { StrictMode } from 'react'
2  import { createRoot } from 'react-dom/client'
3  import './index.css'
4  import App from './App.tsx'
5
6  const rootElement = document.getElementById('root');
7  if (rootElement) {
8    createRoot(rootElement).render(
9      <StrictMode>
10     <App />
11   </StrictMode>
12 );
13

```

This way I'm able to successfully mount the React app into Jenkins. While the frontend is still minimal, it demonstrates that the integration works, and serves as the foundation for future UI development. You can find the source code [here](#)(for anyone reading before 8th April, I will make the repository public from the 8th April, once the GSoC submission deadline is over).

Here you can find some screenshots of how the accessibility from the sidebar and the /chatbot page looks:





The last screenshot shows only a preview. I will later show a more complete design of this page.

Providing a Quick Access

To provide a quick way to access the chatbot from any Jenkins page I also thought of adding a button in the footer that opens a panel where you can interrogate the AI assistant. To do this, I created another class: `ChatbotGlobalDecorator.java`. This class extends `PageDecorator`, which allows injecting content across all Jenkins pages via a footer or header. More about `PageDecorator` can be found in the [Javadoc](#).

```
1 package io.jenkins.plugins.chatbot;
2
3 import hudson.Extension;
4 import hudson.model.PageDecorator;
5
6 @Extension
7 public class ChatbotGlobalDecorator extends PageDecorator {
8     public ChatbotGlobalDecorator() {
9         super(ChatbotGlobalDecorator.class);
10        System.out.println(x:"---- ChatbotGlobalDecorator loaded");
11    }
12 }
```

And I also associated a `footer.jelly` file into `resources/io/.../ChatbotGlobalDecorator` that is:

```
1 <?jelly escape-by-default='true'?>
2 <j:jelly xmlns:j="jelly:core">
3   <div id="chatbot-root"></div>
4   <script type="module" src="${rootURL}/plugin/chatbot-plugin/static/assets/index.js"></script>
5 </j:jelly>
```

The React component I want to load in this case will be obviously different. So we have to handle that multiple `.jelly` files load React apps, even if they are part of the same bundle. To do that we have to edit the entry script `main.tsx` that will decide what component to mount depending on the page it's on (based on the unique ID in the `.jelly` files). So `main.tsx` will look like this:

```

1  import { StrictMode } from 'react'
2  import { createRoot } from 'react-dom/client'
3  import { ChatbotPage } from './components/ChatbotPage.tsx';
4  import { ChatbotFooter } from './components/ChatbotFooter.tsx';
5
6  const pageRoot = document.getElementById('root');
7  const footerRoot = document.getElementById('chatbot-root');
8
9
10 if (pageRoot) {
11   createRoot(pageRoot).render(
12     <StrictMode>
13     | <ChatbotPage />
14     </StrictMode>
15   );
16 } else if (footerRoot) {
17   createRoot(footerRoot).render(
18     <StrictMode>
19     | <ChatbotFooter />
20     </StrictMode>
21   );
22 }

```

Based on the page it's on, it will render the ChatbotPage component or the ChatbotFooter component. It's important to remark that *e/se* in main.tsx, since by writing the if else statement in that way we ensure that if we land on the chatbot page, the access button in the footer is not loaded.

So here you have the screenshots for the quick access:

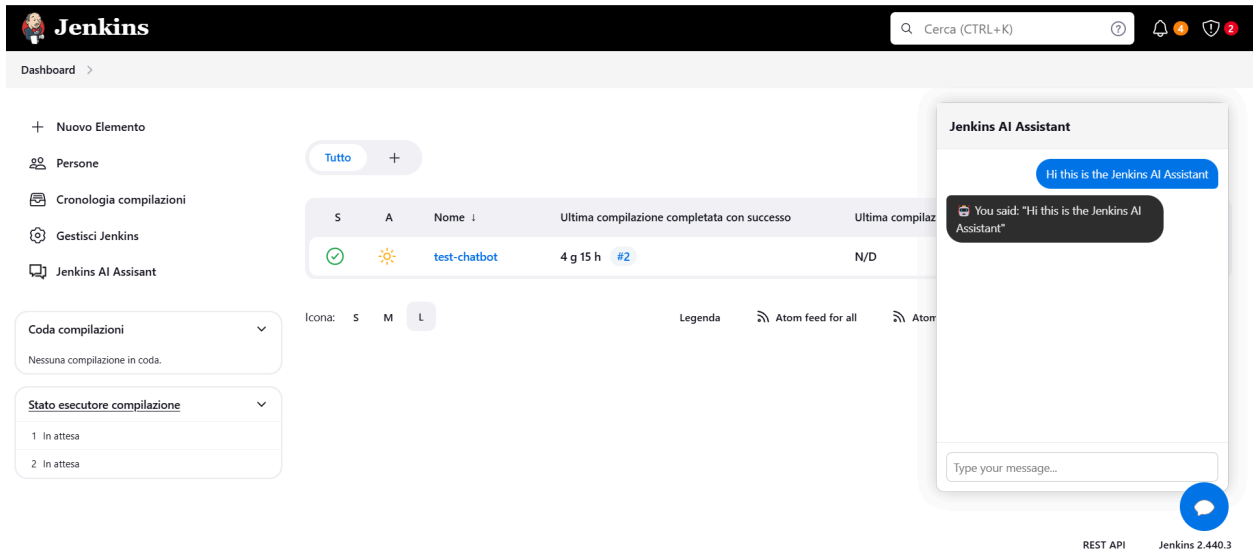
The screenshot shows the Jenkins Dashboard interface. At the top, there's a search bar and navigation icons. The main content area displays a table of build history for the 'test-chatbot' job. The table has columns for status (S), icon (A), name, and completion details. The first row shows a successful build (green checkmark) completed 4g 14h ago, with a duration of 68 ms. Below the table, there are sections for 'Coda compilazioni' (No builds in queue) and 'Stato esecutore compilazione' (2 executors in queue). The bottom right corner shows the Jenkins version 2.440.3 and REST API link.

S	A	Nome	Ultima compilazione completata con successo	Ultima compilazione non riuscita	Ultima durata
✓	☀	test-chatbot	4 g 14 h #2	N/D	68 ms

Icona: S M L

Legenda: Atom feed for all Atom feed for failures Atom feed for just latest builds

REST API Jenkins 2.440.3



This first drafted implementation shows the feasibility of my proposal. It introduces the entry points of the chatbot:

- 1) The sidebar, which redirects to the chatbot page that has a view ideal for a full-page, ChatGPT-style experience. Here the user will have the possibility to have multiple chats, to explore different topics without having to delete other chats.
- 2) The floating version, accessible from anywhere, that is useful for a quick help, without the need of leaving the page you're on, promoting quick accessibility

Serving the assistant service with an AI-backend

Until now, I haven't fully specified how the AI assistant service will be served or how the Python code integrates with the plugin structure. This section clarifies the serving strategy, technology stack, and project layout.

To provide access to the AI assistant, I plan to implement a lightweight HTTP server in Python using FastAPI (chosen for its speed, asynchronous support, and easy-to-document endpoints). Its endpoints will be consumed by the Jenkins plugin backend (written in Java) which acts as a proxy between the Jenkins UI (React frontend) and the AI backend (we'll see more on this proxy shortly).

I would like to make a little note on the fact that FastAPI is asynchronous. Indeed, while FastAPI supports asynchronous request handling, the current Jenkins plugin backend (written in Java) operates synchronously and acts as a proxy between the frontend and the AI backend. This introduces a potential bottleneck: even if the Python backend can handle concurrent requests, the Java layer processes them one at a time. A possible future improvement would be to make the Java proxy layer asynchronous as well. While this may not bring major benefits in the current setup, since the AI model runs locally and inference tasks would still compete for limited hardware, it becomes a lot more impactful when the plugin is connected to an external AI backend or hosted model. In that case, enabling true asynchronous handling across layers would support multiple concurrent chat sessions without blocking, improving the plugin's responsiveness.

Regarding the code structure the Python backend will reside in a dedicated `python-backend/` folder in the root of the plugin repository - alongside the `frontend/` directory. This keeps the

AI logic modular and cleanly separated from both the UI and the Jenkins core logic.

```
chatbot-plugin/  
├─ frontend/  
├─ ai-backend/  
├─ src/  
└─ pom.xml
```

Before the plugin UI and backend can interact with the AI service, the Python server must be up and running. To do this, I plan to configure the pom.xml to automatically launch the AI backend as part of the plugin's execution.

This can be done using the exec-maven-plugin, configured to run a script like this:


```
1 <plugin>  
2   <groupId>org.codehaus.mojo</groupId>  
3   <artifactId>exec-maven-plugin</artifactId>  
4   <executions>  
5     <execution>  
6       <phase>generate-resources</phase>  
7       <goals><goal>exec</goal></goals>  
8       <configuration>  
9         <executable>python3</executable>  
10        <arguments>  
11          <argument>ai-backend/main.py</argument>  
12        </arguments>  
13      </configuration>  
14    </execution>  
15  </executions>  
16 </plugin>
```

This setup works well for development, having the Python backend started manually or through Maven (like I've just showed). Ideally, I would like the deployment experience to be fully automated, where users can simply install the plugin and have everything working without additional configurations. However, achieving this level of automation is challenging, as Jenkins plugins cannot directly launch or manage external processes. Hence, I think that handling this aspect will be challenging. One idea to approach this is to support

Dockerization, allowing the AI backend to run alongside Jenkins with minimal manual steps. The plugin would include a configurable backend URL in Jenkins' global settings, so users can point to a local server or container started via a provided setup. The global setting would have a default value and a test connection feature, to ensure the python server is reachable.

First full page design implementation

In this short section I just want to show better my idea for the design of the full page. I've implemented it with React, and you can find it starting from the ChatbotPage component in the [repository](#). As already said, I'm not going to provide all the code snippets of the components to not overflow this document. Here you can find three screenshots, where the first one shows what you see when you first land in the page, while the other two show the visualization of a chat.



Jenkins AI Assistant


+ New Chat

Pipeline help

Docker build issue

Welcome to Jenkins AI Assistant

Create a new chat or select a chat from the sidebar to get started.



Jenkins AI Assistant

+ New Chat

Pipeline help

Docker build issue

Of course! Here's a basic Jenkinsfile you can use for a Node.js project:

```
...
groovy
pipeline {
  agent any
  stages {
    stage('Install') {
      steps {
        sh 'npm install'
      }
    }
    stage('Test') {
      steps {
        sh 'npm test'
      }
    }
  }
}
...
```


Hi! I need help creating a Jenkins pipeline for my Node.js project.

Thanks! How can I run this pipeline automatically when I push to GitHub?

Great question! To trigger the pipeline on GitHub pushes, you can set up a webhook in your GitHub repository that points to your Jenkins server. You'll also need the GitHub plugin installed in Jenkins.

Type your message...

Send



Jenkins AI Assistant

+ New Chat

Pipeline help

Docker build issue

```
}
}
}
...
```

Thanks! How can I run this pipeline automatically when I push to GitHub?

Great question! To trigger the pipeline on GitHub pushes, you can set up a webhook in your GitHub repository that points to your Jenkins server. You'll also need the GitHub plugin installed in Jenkins.

Would you like a step-by-step guide?

Yes, that would be helpful.

Sure! Here's a simplified guide:

1. In Jenkins, go to Manage Jenkins > Configure System.
 2. Under GitHub, add your credentials.
 3. In your GitHub repo, go to Settings > Webhooks and add your Jenkins URL:


```
...
http://your-jenkins-server/github-webhook/
...
```
 4. Make sure your Jenkins job is configured with 'GitHub hook trigger for GITScm polling'.
- Let me know if you want help configuring the job file.

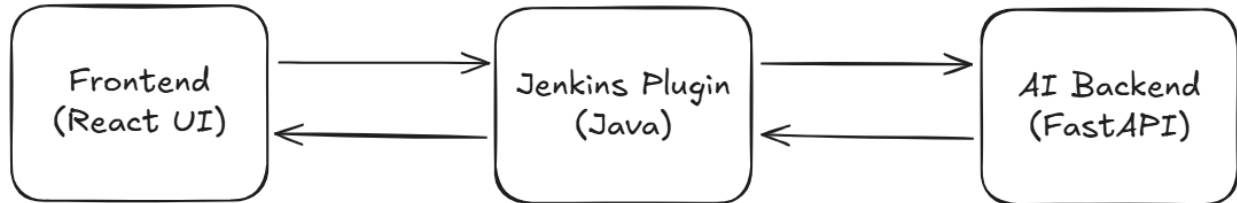
Type your message...

Send

Obviously this is a first draft of what the user interface will look like: many details need to be taken care of.

Plugin API layer

After an initial part on initializing the codebase, I have worked on the API, so on how from the frontend we are able to make a request to the backend, receiving the generated response of the chatbot.



The first thing to clarify is that to provide a smooth interaction between the React frontend and the AI backend (Python), I adopted a clean and structured API layer that has been implemented in the Jenkins plugin backend (Java). This layer acts as a proxy, receiving the request from React and forwarding to the AI backend — enforcing validation, session control, and efficient communication with the Python-based model.

Firstly I would like to highlight that my plan is to provide an interface where users can create and manage multiple chats with the AI assistant. However, I intentionally thought the system with a limit to the number of concurrently active chats.

This limitation serves an important purpose: since the AI backend is resource-intensive and runs directly on the user's machine or server, allowing too many open chats at once could lead to excessive memory usage, CPU load, and slower inference times. By limiting the number of active sessions, I can ensure that the assistant remains responsive and lightweight, even in our constrained environment. Nevertheless, if enabling more concurrent chats won't result in any performance decrease, changing this option will be trivial as changing a variable.

Java API Endpoints

The first thing I've done is implementing the API endpoints for our proxy in Java. To do so I've created a ChatbotAPI.java file that handles that:

```
10  /**
11   * REST API layer for the AI-powered chatbot plugin.
12   * This class exposes endpoints to manage chatbot sessions and handle user requests,
13   * acting as a bridge between the Jenkins frontend and the Python AI backend.
14   */
15  @Path("/chatbot/api")
16  public class ChatbotAPI {
```

The implemented endpoints are going to be:

1. *POST /start-session* → Starts a new chat session
2. *POST /message* → Sends a user message
3. *POST /delete-session* → Deletes an active chat session

When a user wants to start a new chat the start-session endpoint will register a new chat session, notifying the AI backend of the new chat and returning a unique ID of the chat session, that the user will use for the message endpoint to have a reference to the right chat. The AI backend will populate a hashmap, that associates a key(chat unique ID) to its chat history(you can see the example of chat history in the “Core Design Approach” section). An important detail is that this way we don’t have to include the chat history in the body of the HTTP requests we are going to do, which could become very long. Here you can find my implementation of this first endpoint:

```
18  /**
19   * Starts a new chat session.
20   * This initializes a session on the Python backend and returns a session ID
21   * that will be used for all future interactions in the conversation. The
22   * chat session ID is saved in the ChatSessionManager.
23   *
24   * @return HTTP 200 with JSON response containing the session_id
25   *       or HTTP 409 if chat session limit is reached
26   *       or HTTP 500 if the request fails
27   */
28  @POST
29  @Path("/start-session")
30  @Produces(MediaType.APPLICATION_JSON)
31  public Response startChatSession() {
32      try {
33          if (!ChatSessionManager.canStartNewSession()) {
34              return Response.status(409).entity("Too many active sessions").build();
35          }
36
37          HttpRequest request = HttpRequest.newBuilder()
38              .uri(URI.create(str: "http://localhost:8000/start-session"))
39              .POST(HttpRequest.BodyPublishers.noBody())
40              .build();
41
42          HttpClient client = HttpClient.newHttpClient();
43          HttpResponse<String> response = client.send(request, HttpResponse.BodyHandlers.ofString());
44
45          JSONObject json = new JSONObject(response.body());
46          String sessionId = json.getString("session_id");
47
48          ChatSessionManager.addSession(sessionId);
49
50          return Response.ok(new JSONObject().put("session_id", sessionId).toString()).build();
51      } catch (Exception e) {
52          e.printStackTrace();
53          return Response.status(500).entity("Failed to start a new chat session").build();
54      }
55  }
```

A detailed description of the *ChatSessionManager*, will be performed after the description of the three endpoints.

Now that I’m able to create a new chat, I want to send a message to the chatbot and receive an appropriate response. For this we rely on the *message* endpoint, that will take in the request the user’s query to deliver and the unique chat session ID, and answer with the generated response and the list of the sources that the chatbot used to generate the response. Here you can find my implementation for this second endpoint:

```

58  /**
59   * Sends a message from the user and forwards it to the Python AI backend.
60   * Ensures the session is active and not currently waiting for another response.
61   * The Python backend will use the session ID to retrieve context and return a response.
62   *
63   * @param request of type ChatRequest -> The user's message and session ID
64   * @return HTTP 200 with JSON containing the AI-generated reply
65   *         or HTTP 404 if the session ID is not found
66   *         or HTTP 409 if a response is already being processed for the session
67   *         or HTTP 500 if the request fails
68   */
69  @POST
70  @Path("/message")
71  @Consumes(MediaType.APPLICATION_JSON)
72  @Produces(MediaType.APPLICATION_JSON)
73  public Response handleMessage(ChatRequest request) {
74      try {
75          if (!ChatSessionManager.hasSession(request.session_id)) {
76              return Response.status(404).entity("Session not found").build();
77          }
78          if (ChatSessionManager.isWaiting(request.session_id)) {
79              return Response.status(409).entity("Already waiting for response").build();
80          }
81          ChatSessionManager.setWaiting(request.session_id, waiting:true);
82
83          String jsonPayload = new JSONObject()
84              .put("message", request.message)
85              .put("session_id", request.session_id)
86              .toString();
87
88          HttpRequest httpRequest = HttpRequest.newBuilder()
89              .uri(URI.create(str:"http://localhost:8000/generate"))
90              .header(name:"Content-Type", value:"application/json")
91              .POST(HttpRequest.BodyPublishers.ofString(jsonPayload))
92              .build();
93
94          HttpClient client = HttpClient.newHttpClient();
95          HttpResponse<String> httpResponse = client.send(httpRequest, HttpResponse.BodyHandlers.ofString());
96
97          ChatSessionManager.setWaiting(request.session_id, waiting:false);
98
99          JSONObject json = new JSONObject(httpResponse.body());
100
101          ChatResponse res = new ChatResponse();
102          res.response = json.getString("reply");
103          JSONArray sources = json.getJSONArray("sources");
104          for (int i = 0; i < sources.length(); i++) {
105              res.sources.add(sources.getString(i));
106          }
107
108          return Response.ok(res).build();
109      } catch (Exception e) {
110          e.printStackTrace();
111          return Response.status(500).entity("AI backend error").build();
112      }
113  }

```

I'm not commenting on some trivial model classes like ChatRequest. For more details you can always look at the github repository.

Finally when we want to delete an active chat we will call the *delete-session* endpoint. This endpoint will remove the chat from the active ones in the Java ChatSessionManager(which we'll see shortly) and also notify the AI backend, allowing him to delete the history associated with that chat. Here you can find my implementation for this third endpoint:

```

115  /**
116   * Deletes an existing chatbot session on the Python backend, clearing its context.
117   * It also removes the session from the ChatSessionManager.
118   *
119   * @param sessionIdJson JSON containing the session_id to delete
120   * @return HTTP 200 if deletion was successful
121   *         or HTTP 404 if the session ID does not exist
122   *         or HTTP 500 if the request fails
123   */
124  @POST
125  @Path("/delete-session")
126  @Consumes(MediaType.APPLICATION_JSON)
127  @Produces(MediaType.APPLICATION_JSON)
128  public Response deleteSession(String sessionIdJson) {
129      try {
130          if (!SessionManager.hasSession(sessionId)) {
131              return Response.status(404).entity("Chat Session not found").build();
132          }
133
134          HttpRequest httpRequest = HttpRequest.newBuilder()
135              .uri(URI.create(str:"http://localhost:8000/delete-session"))
136              .header(name:"Content-Type", value:"application/json")
137              .POST(HttpRequest.BodyPublishers.ofString(sessionIdJson))
138              .build();
139
140          HttpClient client = HttpClient.newHttpClient();
141          HttpResponse<String> httpResponse = client.send(httpRequest, HttpResponse.BodyHandlers.ofString());
142
143          ChatSessionManager.removeSession(sessionId);
144
145          return Response.ok(httpResponse.body()).build();
146      } catch (Exception e) {
147          e.printStackTrace();
148          return Response.status(500).entity("Failed to delete session").build();
149      }
150  }
151  }

```

In all the endpoints I've tried to be very accurate with the JavaDoc comments, trying to be as descriptive as possible, enabling any developer that will read my code to quickly understand what I've written. I've also aimed at being precise with the HTTP codes, following the REST standard.

Another aspect that has not been covered in the last snippets of the endpoints, is the Single Responsibility Principle. Even though it is not too obvious yet - due to the low amount of code - the endpoint functions should only handle the request part, acting as the controller. The eventual business logic should be delegated to other methods, leaving the endpoint functions light and promoting the S of the SOLID principles.

I would also like to point out that the hardcoded <http://localhost:8000> is just a temporary solution. In my future implementation I will obviously need to make it dynamically configurable through the plugin's settings, based on which port the AI backend will be started on. Hardcoding the backend address is not only bad practice, but also problematic: if the default port (e.g., 8000) is already in use, the AI backend may start on a different port, and the plugin would no longer be able to communicate with it.

The Session Manager

As previously mentioned, the ChatSessionManager is another important part in my implementation. It serves as a simple session manager for the active chats, having many useful methods that I can call in the endpoints. In its attributes we store the limit of concurrent active sessions. The class keeps track of the active chat sessions, using a ConcurrentHashMap, where the key is the unique chat session ID, and the value is the associated ChatSessionState. The latter is a model class that represents the state of an active session. At the moment the only attribute present is *isPendingReply* that is a boolean variable that marks if for that chat we are waiting for a response or not. Later this model

class can be extended with more useful attributes.

Here you can find my implementation of the ChatSessionManager:

```
1 package io.jenkins.plugins.chatbot;
2
3 import java.util.concurrent.ConcurrentHashMap;
4 import java.util.Map;
5 import io.jenkins.plugins.chatbot.model.ChatSessionState;
6
7 /**
8  * Manages the active chat sessions.
9  */
10 public class ChatSessionManager {
11     /**
12      * The maximum number of active chatbot sessions allowed at any time.
13      */
14     private static final int MAX_ACTIVE_SESSIONS = 10;
15
16     private static final Map<String, ChatSessionState> sessions = new ConcurrentHashMap<>();
17
18     public static boolean canStartNewSession() {
19         return sessions.size() < MAX_ACTIVE_SESSIONS;
20     }
21
22     public static void addSession(String sessionId) {
23         sessions.put(sessionId, new ChatSessionState());
24     }
25
26     public static boolean hasSession(String sessionId) {
27         return sessions.containsKey(sessionId);
28     }
29
30     public static void removeSession(String sessionId) {
31         sessions.remove(sessionId);
32     }
33
34     public static boolean isWaiting(String sessionId) {
35         return sessions.containsKey(sessionId) && sessions.get(sessionId).isWaiting;
36     }
37
38     public static void setWaiting(String sessionId, boolean waiting) {
39         if (sessions.containsKey(sessionId)) {
40             sessions.get(sessionId).isWaiting = waiting;
41         }
42     }
43 }
```

During the implementation I plan to bring MAX_ACTIVE_SESSIONS out of this class. An idea could be to make the user able to configure the maximum number of concurrent chats he can access, giving more flexibility to the plugin. To achieve that my idea is to let the user edit it in the "Manage Jenkins → Configure System" section. The only issue with that is that only admins can edit that. However in this way we ensure the persistence of the setting.

The React point of view

In the react application side we have to create the API functions to the Java Plugin backend. You can find the API call functions in the *chatService.ts* file. Inside it you can find the three functions that are "mapped" to the endpoints shown before.

```

4  ✓ /**
5    * Starts a new chat session.
6    * @returns session ID string
7    */
8  ✓ export const startSession = async (): Promise<string> => {
9  ✓    const res = await fetch(`${API_BASE}/start-session`, {
10     method: 'POST',
11   });
12
13  ✓    if (!res.ok) {
14     throw new Error('Failed to start chat session');
15   }
16
17     const data = await res.json();
18     return data.session_id;
19   };
20
21  ✓ /**
22    * Sends a user message to the backend and receives a reply.
23    * @param message - user message
24    * @param sessionId - active chat session ID
25    * @returns the assistant's reply and optional sources
26    */
27  ✓ export const sendMessage = async (
28    message: string,
29    sessionId: string
30  ): Promise<ChatResponse> => {
31  ✓    const res = await fetch(`${API_BASE}/message`, {
32     method: 'POST',
33  ✓    headers: {
34     'Content-Type': 'application/json',
35   },
36     body: JSON.stringify({ message, session_id: sessionId }),
37   });
38
39  ✓    if (!res.ok) {
40     const errorText = await res.text();
41     throw new Error(`Backend error: ${errorText}`);
42   }
43
44     return await res.json();
45   };

```



```

47  /**
48   * Deletes the current chat session.
49   * @param sessionId - session to delete
50   */
51  export const deleteSession = async (sessionId: string): Promise<void> => {
52    const res = await fetch(`${API_BASE}/delete-session`, {
53      method: 'POST',
54      headers: {
55        'Content-Type': 'application/json',
56      },
57      body: JSON.stringify({ session_id: sessionId }),
58    });
59
60    if (!res.ok) {
61      const errorText = await res.text();
62      throw new Error(`Failed to delete session: ${errorText}`);
63    }
64  };

```

Where `API_BASE` is imported from `config.ts`

```

frontend > src > TS config.ts > ...
1   export const API_BASE = '/jenkins/chatbot/api/';

```

Handling Chat Persistence

For the initial version of the plugin, I don't plan to support persistent storage beyond the current Jenkins session. The primary goal is to manage active conversations during usage, without relying on long-term storage across Jenkins restarts.

On the frontend, I plan to manage state using a combination of React local state and Redux, enabling access to chat histories. To retain state across browser reloads, I plan to use Redux Persist, which leverages local storage to persist the Redux store between sessions.

A potential future enhancement could involve storing chat history on the backend side, either in-memory or written to a file within Jenkins' home directory, to ensure persistence even across Jenkins restarts. By introducing simple GET endpoints, the frontend could fetch previous conversations when the user accesses the chatbot interface. This would significantly improve the user experience without needing too much implementation complexity.

First Use Cases

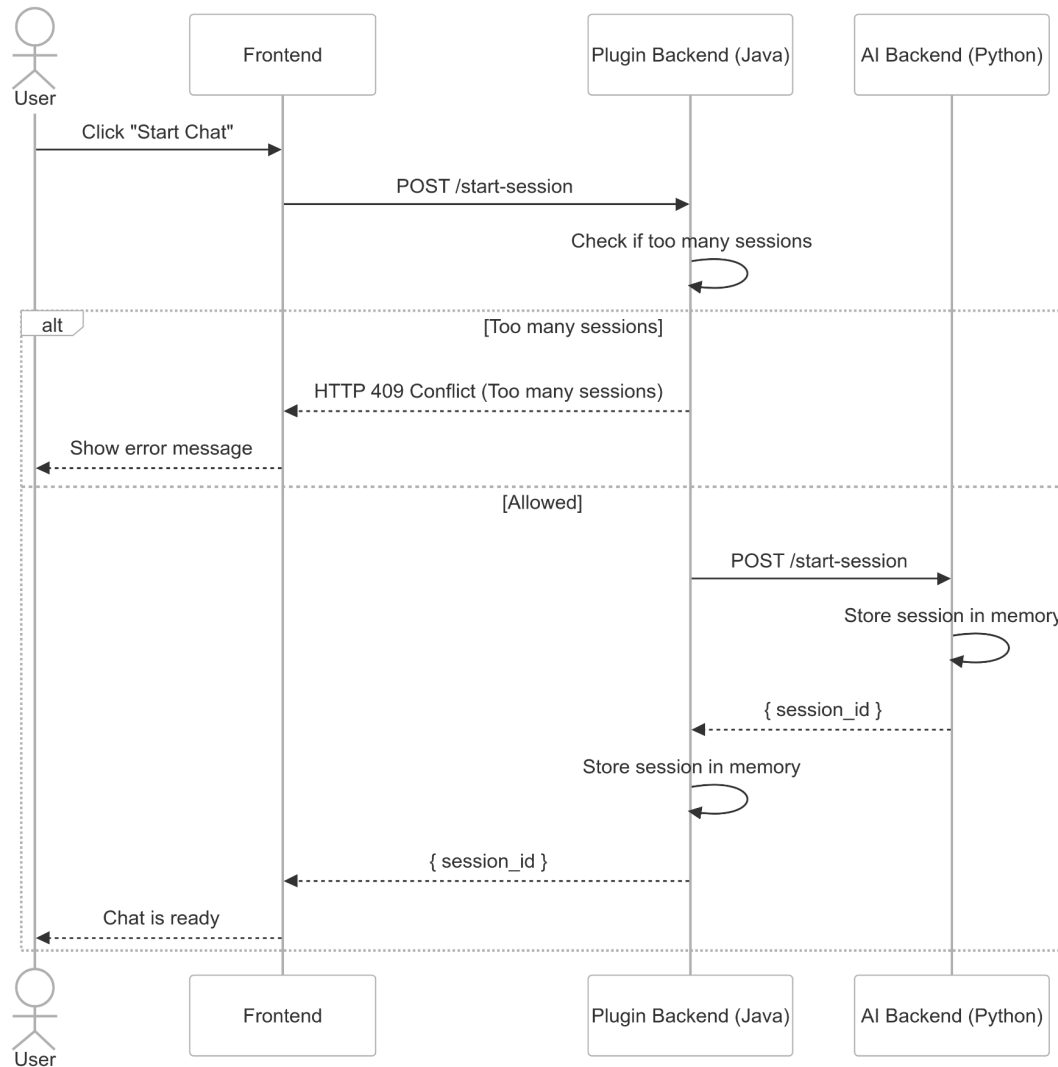
Here I would like to provide a more detailed view of the three main use cases:

1. The user starts a new chat
2. The user sends a message to the AI assistant in one of his active chats
3. The user deletes an active chat

Starting with the first use case, here you can find a more detailed table of its actions and the associated sequence diagram:

Actor	User
Entry conditions	The user is on the chatbot interface and intends to start a new conversation.
Event Flow	<ol style="list-style-type: none"> 1. The user clicks on "Start Chat". 2. The frontend sends a request to the plugin backend to initiate a new session. 3. The plugin backend checks if the number of active sessions is within the allowed limit. 4a. If the limit is exceeded, the plugin responds with an error. 4b. If allowed, the plugin contacts the Python backend to create a new session. 5. The Python backend responds with a new session ID. 6. The plugin backend stores the session and forwards the session ID to the frontend. 7. The frontend informs the user that the chat is ready.
Exit condition	A new session is successfully created and ready for use.
Exceptions	4a. Too many sessions active — request is denied.

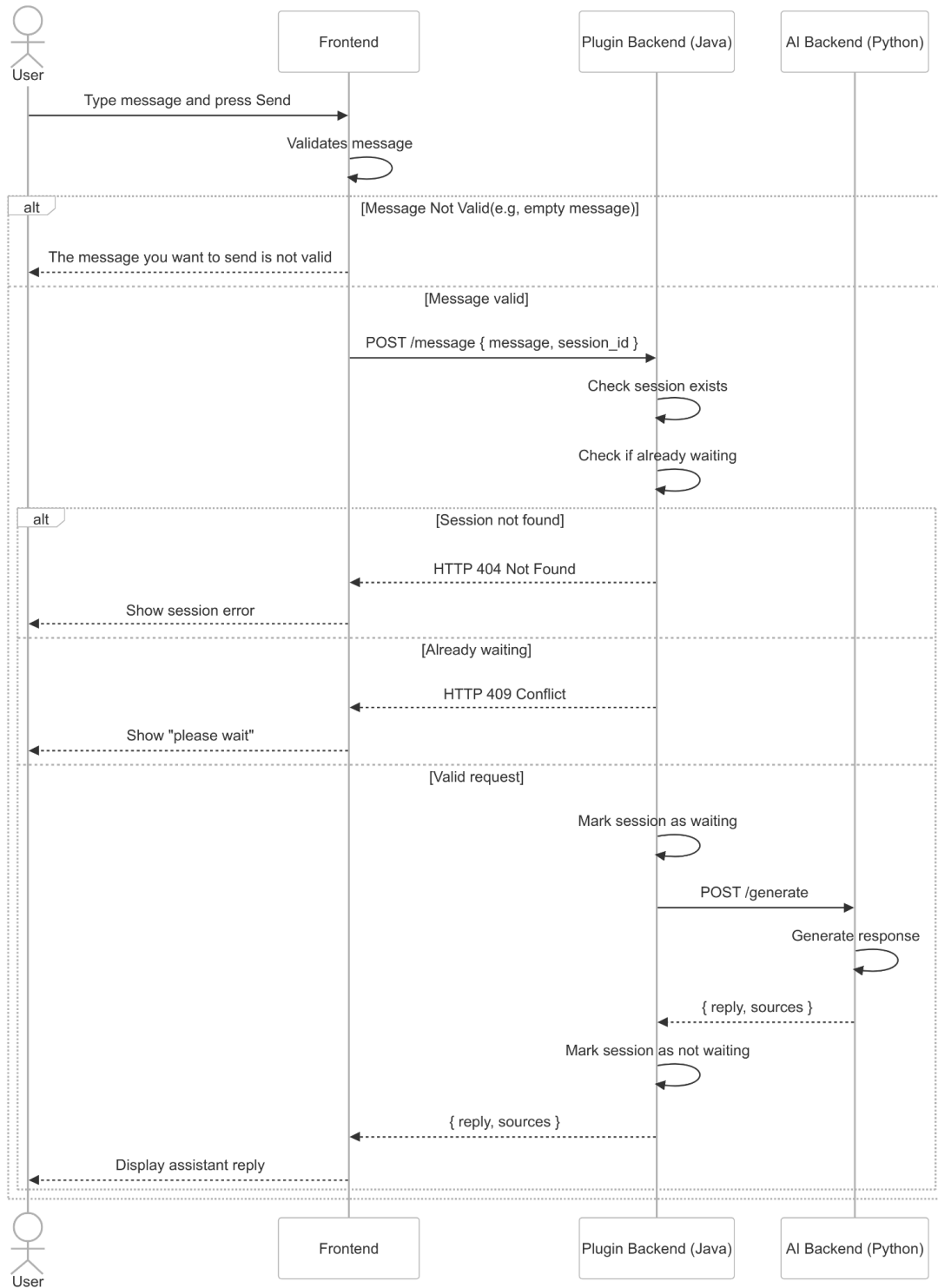
Table 1: User starts a new chatbot session



Continuing with the second use case here you can find a more detailed table of its actions and the associated sequence diagram:

Actor	User
Entry conditions	A chatbot session has already been started and is active.
Event Flow	<ol style="list-style-type: none"> 1. The user types a message and clicks "Send". 2. The frontend validates the message. 3a. If the message is valid, the frontend sends it to the plugin backend. 4. The plugin checks that the session exists and is not already waiting for a reply. 5a. If the session is valid and available, it marks it as waiting. 6. The plugin forwards the message to the Python backend. 7. The Python backend replies with the generated answer and optional sources. 8. The plugin clears the waiting status and returns the response to the frontend. 9. The frontend displays the assistant's reply.
Exit condition	The assistant's reply is displayed in the chat interface.
Exceptions	<ol style="list-style-type: none"> 3b. Message is invalid (e.g., empty). 4. Session does not exist or is already waiting.

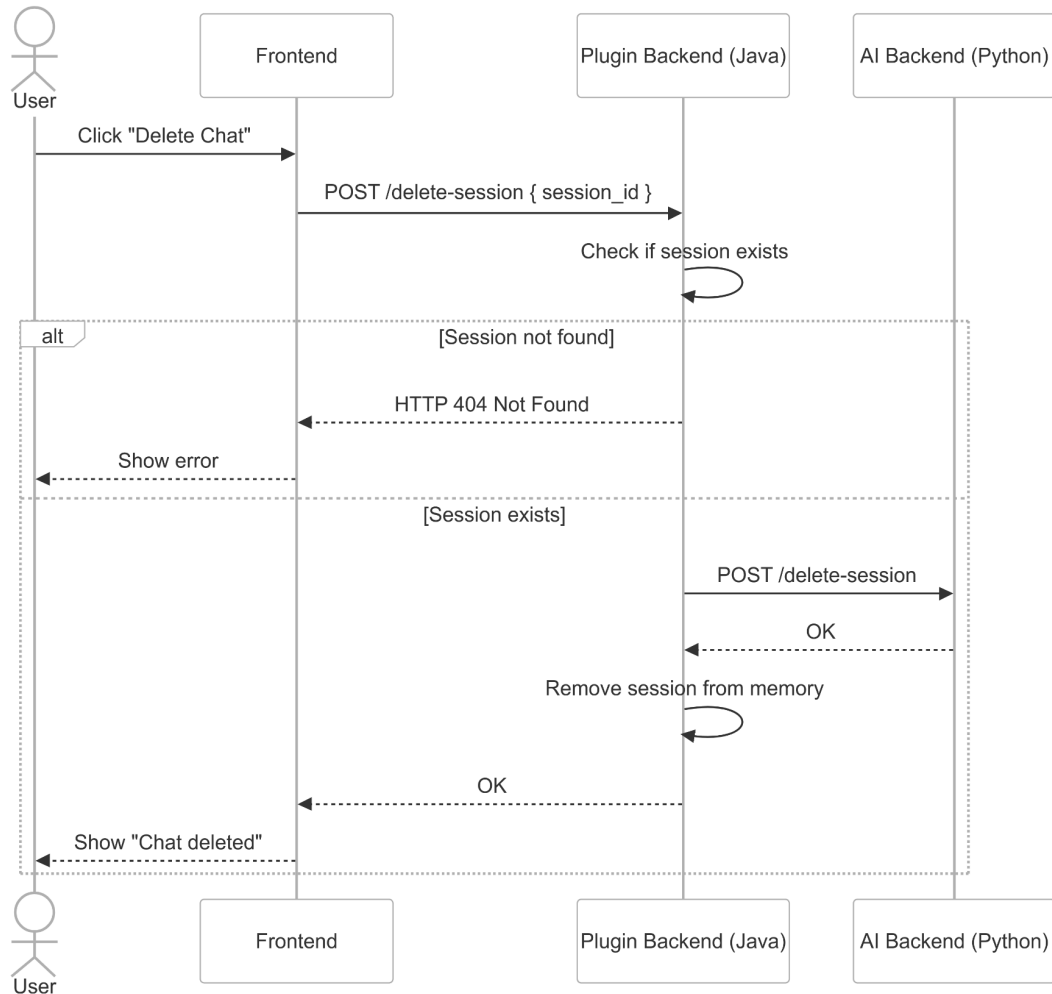
Table 1: User sends a message to the chatbot



And finally here you can find a more detailed table of the actions and the associated sequence diagram of the third use case:

Actor	User
Entry conditions	A chatbot session is currently active and the user chooses to delete it.
Event Flow	<ol style="list-style-type: none"> 1. The user clicks the "Delete Chat" button. 2. The frontend sends a delete request to the plugin backend. 3. The plugin checks if the session exists. 4a. If the session is found, it calls the Python backend to delete the session context. 5. The plugin removes the session from its internal store. 6. A success response is returned to the frontend. 7. The frontend informs the user that the chat was deleted.
Exit condition	The chat session is removed from both plugin and AI backend.
Exceptions	4b. Session does not exist — an error is returned.

Table 1: User deletes an active chatbot session



Testing Strategy

Without any doubts testing the overall project will be crucial to ensure its correctness, reliability and stability. I plan to implement a testing strategy that covers both frontend and backend components. I will follow the principles of the Test Pyramid and use a bottom-up strategy, focusing heavily on unit tests, followed by less dense integration tests.

At the base of the pyramid, *unit tests* will validate the smallest components of the system in isolation, without dependencies on external systems. These include:

- Frontend (React)
 - Message Input validation
 - Rendering of chat bubbles with content
 - UI behaviours like disabling input during the load of the response
 - More complex displaying features like links or code blocks
- Backend (Java)
 - ChatSessionManager logic
 - API endpoints business logic and request/response formatting
 - Error handling logic for failed requests

On the backend, I will use JUnit to test Java classes independently, such as API request handlers, response formatters, and error handling logic, while for the frontend I will use React Testing Library.

Integration tests will validate how different parts of the plugin work together, especially the plugin's integration into Jenkins. To do this, I will use JenkinsRule, which allows running up a lightweight Jenkins instance inside tests. I may also use JenkinsSessionRule, which with respect to JenkinsRule allows Jenkins to be restarted in the middle of a test. Instead, although RealJenkinsRule provides a more realistic environment, I do not plan to use it since I don't think it is necessary and it does not support utilities like LoggerRule which are useful for debugging logging outputs. In the javadoc there is mentioned an alternative for logging, but again, as long as I won't find out it gives lots of advantages, I don't have in my schedule to use it.

Some of the integration tests include:

- Plugin and Jenkins integration
 - Sidebar route availability → Assert that the /chatbot endpoint is correctly registered via RootAction, and verify that accessing this route loads the expected page
 - Footer button display → Ensure the floating quick-access button appears on every Jenkins page via PageDecorator
 - React injection into Jelly views → Verify that the React app is rendered in the index.jelly and footer.jelly views correctly
- Java ↔ Python Backend integration: testing whether the plugin correctly communicates with the Python backend via HTTP
 - Session creation → Verify that calling the /start-session Java API triggers a correct POST to /start-session on the Python backend, and validate returned session ID is stored in ChatSessionManager
 - Message sending → Verify that the /message Java endpoint accepts user input and session ID, sends the correct payload to /generate on the Python backend and correctly parses and returns the backend response (reply +

- sources)
 - Session deletion → Ensure the /delete-session Java endpoint triggers correct request to /delete-session on Python side and removes the session from ChatSessionManager
 - Edge cases → Backend not reachable → return proper 500 error ; Too many sessions → return 409 Conflict ; Already waiting → return 409 Conflict
- Frontend ↔ Backend Integration (React ↔ Java): tests that verify that the frontend and Java plugin are communicating correctly
 - Message submission → From React, send a message and assert the plugin receives it at /chatbot/api/message
 - Session management → Assert that the frontend can request a new session via /chatbot/api/start-session and assert chat session ID is received and stored client-side
 - Chat flow → Simulate user sending multiple messages and receiving responses and check that message content and sources render as expected in the UI
 -

Regarding end to end tests, while they simulate complete user flows in a real browser and can be valuable in certain types of applications, they are typically not used in Jenkins plugin development due to their complexity, fragility, and high maintenance cost. Setting up and running E2E tests often requires a full Jenkins controller, browser automation, and complex orchestration.

For this reason, I do not plan to implement end to end tests for this project. Instead, I will rely on a solid combination of unit and integration tests to ensure correctness, UI responsiveness, and reliable behavior across typical and edge-case scenarios. I will also conduct manual scenario testing in a real Jenkins instance, ensuring that the plugin behaves correctly under realistic user workflows.

For future improvements limited end to end testing could be performed, but at the moment I don't find them essential or justified, especially for the scope of the project.

Documentation

Since this project will be delivered as a Jenkins plugin, and (I hope) potentially extended, updated and maintained also by other Jenkins contributors, I plan to put a strong emphasis on writing a clear and complete documentation. The documentation will be for different types of users and contributors:

- For end users, I will provide help especially in the README, where there will be explained its purposes, how to interact with the chatbot, what it can do and a basic understanding of its basic architecture and the resources from which it takes the information
- For developers, I will document the architecture, the code structure, the indexing pipeline used and any plugin-related information like build process.
- For maintainers, I will include information regarding how to update the vector DB, so how to repeat the indexing pipeline I explained previously. I also plan to write how to add tools and eventually change the used LLM.

Why this project?

I chose this project because it brings together areas on which I am very interested about and align very well with my skills and capabilities. This project offers the opportunity to work on the core aspects of modern conversational AI — including NLP, LLMs, RAG, prompt engineering and designing efficient indexing pipelines. At the same time, it also involves the design and build of a user interface, and the need to integrate everything into a Jenkins plugin.

I am particularly interested in this combination of skills, as it allows me to apply theoretical knowledge in a practical, impactful setting while also learning more about Jenkins' ecosystem and plugin development. Most importantly, I truly believe that this chatbot would be a very useful feature for the Jenkins community, and I would be honored to carry on the project.

Project Deliverables

For each phase of the project, I've outlined the expected deliverables that will serve as key milestones to track progress and ensure alignment with the project goals.

- **May 8, 2025 - June 1 (Phase 1)**

- ☐ Interact with the mentors
- ☐ Define with mentors the project's details and timeline
- ☐ Continuous involvement with the community, including meetings and being active in discussions
- ☐ Finalized list of data sources
- ☐ List of tools to implement(they will be implemented in Phase 2)
- ☐ Complete setup of the development environment
- ☐ Finalize the code structure
- ☐ Finalize the high-level architectural plans for indexing pipeline and RAG

- **June 2 - July 13 (Phase 2)**

- **June 2 - June 22 (Sub-phase 1)**

- ☐ Complete data indexing pipeline
 - ☐ Quantized LLM final selection and performance evaluation locally
 - ☐ FastAPI backend

- **June 22 - July 13 (Sub-phase 2)**

- ☐ Implementation of the Java-based proxy API
 - ☐ First version of the full-page chatbot interface, supporting multi-chat, context display, and messaging
 - ☐ First version of the quick-access floating chat, embedded via PageDecorator in the Jenkins footer
 - ☐ Core UI logic shared across both views
 - ☐ Accessibility to the chatbot via the UI
 - ☐ Initial unit tests (frontend and backend)

- **July 14 - July 18 (Midterm Evaluations)**

By this moment, the AI core part (indexing pipeline + RAG) will be complete, with a quantized LLM. The Jenkins plugin will have its backend and initial version of the UI implemented and accessible. The chatbot will be able to respond to basic queries with relevant context. Initial unit tests will be in place. Midterm slides will be ready, showing the progress done.

- **July 14 - Aug 25 (Phase 3)**

- **July 14 - Aug 4 (Sub-phase 1)**

- ☐ Implementation of tools
- ☐ Integrated the agent approach
- ☐ UI enhancements
- ☐ Last Unit tests
- ☐ Initial Integration tests

- **Aug 4 - Aug 25 (Sub-phase 2)**

- ☐ Final UI refinements
- ☐ Last Integration Tests
- ☐ Documentation

- **Aug 25 (Final Evaluations)**

By this date, the plugin will be fully implemented, integrating tools and agents. The UI will be finished and with it also unit and integration tests. Moreover the documentation will be ready and available. Finally, the presentation slides will be ready for the final evaluation.

Proposed Schedule

March 24 - April 8, 2025 (Application Period)

In this period I am submitting my draft proposal to the mentors as soon as I have written a solid base. While I will wait for their feedback I plan to refine my proposal, working on details and adding more diagrams and code snippets to strengthen the document. Once I get reviewed the draft, I plan to read the suggestions given and apply them, completing my proposal and submitting it.

Once I've submitted the final proposal, I will continue working on the contributions. I have already got some issues on the radar to look at.

Finally, I will continue attending the meetings, especially the Platform ones.

April 9 - May 7 (Acceptance Waiting Period)

For what concerns the Jenkins environment, during the acceptance waiting period my idea is to continue interacting with the community, make myself more comfortable with some basic Jenkins stuff. Since an AI-powered tool is not already available, prepare for some of my questions on the Gitter channels :) . I also plan to study existing plugins more deeply to

gain a better understanding of plugin development patterns and clarify what I'll need to implement.

Additionally, I also intend to continue experimenting with crawling and vector db, and if relevant or useful I would like to share any progress openly on Discourse or Gitter, and eventually update the [github repo](#) on which I've done a first draft.

As time permits, I will also try my best to contribute to some more Jenkins issues. This will not only help the community but also allow me to sharpen my understanding of the Jenkins codebase and plugin development practices.

I'm not sure if this falls within the scope of the document, but in addition to my work on the Jenkins-related project, I will also be managing some university commitments, including taking exams and completing academic projects. Notably, during this period, I will be working on a project for my university's Natural Language Processing (NLP) course, which I believe will further prepare me and strengthen my background for this GSOC project.

May 8 - June 1 (Community Bonding Period)

During this time, I will get to know my mentors, learning more about them. Apart from hearing via email and chat, I hope there will be the possibility to also plan meetings with them and other members of the community, to have the chance to directly talk with them. I will discuss with my mentors the details about the project, figuring out with them if there are better solutions to some problems and defining what will be the final project pipeline. With them I plan to finalize decisions like the data sources that we will use for the project and a more detailed preview of the tools to implement.

My goal at the end of this period is to have a clear and detailed plan for my coding phase. I plan also to set up what is going to be my development environment, making myself 100% ready for the coding phase, preparing the project repo structure and having a complete high-level comprehension of what I will do in the following weeks.

I have already introduced myself to the community of Jenkins, joined all the channels on Gitter and introduced myself to the community. I have also participated in some meetings, including the platform one and the infrastructure one. In this period I will continue being active in the community and joining the meetings(especially those that I am more interested in).

June 2 - July 13 (Standard Coding Period)

I will divide this period in two sub-phases: one from June 2 to June 22, and the other from June 23 to July 13.

In the first sub-phase, I plan to focus entirely on building the AI backbone of the project. This includes completing the entire index pipeline, so crawling all the data sources, cleaning and preprocess them, chunk them in an effective way, embedding them, and then storing them in a FAISS vector database, curing the index. To serve the AI assistant, I will also implement a FastAPI-based backend, which will expose endpoints for query handling and inference that will act as the entry point for the Java plugin to interact with the AI logic. After the indexing pipeline is complete, my objective is to have picked and tested the quantized LLM. I plan to use Mistral-7B, but in case there will be issues with it, this few weeks will be the moment in which I will test and determine what the final LLM will be. In the last week on this sub-period I plan to assemble the first pieces, and have a system that given a query is able to generate relevant answers.

In the second sub-phase my focus will shift to what concerns the plugin development itself.

At this point, I will begin implementing the actual Jenkins plugin logic, connecting the different components I've already outlined. Specifically, this includes developing the Java-based backend that will serve as a proxy between the frontend (React) and the AI backend (FastAPI), routing requests and responses appropriately. Having already drafted a structure for this part, I will now flesh it out by implementing the endpoints, data handling, and serialization logic.

Alongside the backend, I will begin building the user interface using React and TypeScript. The frontend will feature a full-page assistant interface, as well as a quick-access floating button available across all Jenkins pages. This will allow users to either initiate an in-depth multi-chat session or quickly ask questions on the fly.

During this sub-phase, I will also implement unit tests for the frontend and backend logic, ensuring UI stability and request correctness.

In parallel, I will begin preparing my midterm evaluation presentation slides. This will include a clear overview of what I've built so far, showing also a breakdown of what has been achieved with respect to the original plan made.

July 14 - August 25 (Work Period for Standard Route)

I will divide this period into two sub-phases: from July 14 to August 4, and from August 5 to August 25.

In the first sub-phase, my focus will be firstly on refining and expanding the chatbot's core capabilities. This will include implementing the planned tools and working on prompt-engineering. A key milestone in this phase will be implementing agent-based reasoning, allowing the chatbot to dynamically decide how to handle a query. The second important task in this sub-phase is to continue improving the user interface, enhancing the user experience with loading indicators, clickable source links, and code block display. Moreover I plan to start checking edge cases and continue writing unit and integration tests to cover both new features and the backend/frontend interactions.

In the second sub-phase, I will shift toward refining, optimizing, and wrapping up the entire project. This will include things like making the last UI adjustments, completing the last round of integration tests, and addressing any feedback provided by my mentors. At this point, the plugin should be functionally complete. I will also finalize the documentation for all targets: user-facing README, developer documentation for building and maintaining the plugin, and maintainer instructions for updating the vector DB or changing models. I will also make sure the entire codebase is clean and well-commented, and that the plugin can be built and used from source easily via the standard Jenkins plugin system.

As for the first coding phase, I will start preparing my presentation slides throughout the period, giving them the final adjustments in the last days. I will also prepare a blog post and a final report that explains the project, the problem it solves, and how it can be used or extended by others in the Jenkins community.

In conclusion, at this point the project will be complete and ready for usage.

September 1 - November 9 (Extended Coding Period)

If everything will go as expected, in this period I would like to evaluate with my mentors if some of the future improvements I've listed could be actually impactful, and in case of a positive feedback working on it. Personally, I think that the option of choosing between a

local and external API could be a very good candidate for a bonus implementation.

Future Improvements

Here, you can find listed some future improvements proposed. These are things I would personally want to pursue after the GSoC program timeline or as Bonus implementation during the program itself if things go better than expected.

- A valuable future enhancement would be to support a configurable backend option that allows the user to choose between running a local model or using an external LLM API (such as OpenAI). This addition would give more flexibility, make the plugin more adaptable to different environments — making available high-performance external calls, while still supporting an offline and lighter version through the local model.
- Exploring advanced deduplication techniques → while the initial implementation will rely on traditional approaches (like exact and fuzzy matching), more advanced deduplication strategies using machine learning or embedding-based similarity could be explored in the future. These methods could improve the quality of the index by detecting semantically similar documents or posts, but require additional training, tuning, and care to avoid removing useful content.
- Further enhancing chunking optimization it would be interesting to explore dynamic chunking based on the Mix-of-Granularity (MoG) framework, which uses a learned router to select the optimal chunk size per query. Also the part on graph-based pre-processing (MoGG) would be another interesting part to explore. More on the paper [here](#).
- Since Jenkins documentation and community content are constantly evolving, it would be great to ensure that the chatbot remains correct and updated over time, without the need of manual intervention. My idea is to create a scheduled Jenkins job on a dedicated external server that periodically runs a pipeline responsible for crawling the updated content, going again through the text cleaning and preprocessing phase, chunking and embedding it, and rebuilding the FAISS index. This index could then be either pushed directly into the plugin's code repository. So with this sort of central indexing approach we ensure that plugin users always receive the latest information without needing to rebuild the index locally
- A potential future enhancement could involve combining both query classification and agents in a hybrid approach. The idea is to first classify the query received with a lightweight classifier, and handle straightforward queries (e.g., version checks) using predefined tools or retrieval paths. However, if the confidence score of the classifier results to be too low, the system could "delegate" the query to the agent that will handle it dynamically. The advantage of this hybrid approach is that simple queries are treated in a more efficient way, while maintaining the more complex approach for harder queries
- Currently, the Java layer of the plugin, that is responsible for acting as a proxy between the Jenkins UI and the AI backend, handles requests synchronously. Although the FastAPI backend already supports asynchronous handling, this benefit is limited by the blocking behavior of the Java proxy. A future enhancement could involve refactoring the Java layer to support asynchronous communication. This would allow the plugin to handle multiple requests concurrently, which is particularly useful when supporting multiple simultaneous chat sessions. While this may not provide significant benefits in the current setup with a locally hosted backend (where hardware constraints already limit parallel inference), it becomes much more relevant when integrating with external or cloud-hosted LLMs.

Continued Involvement

I believe that my journey with Jenkins will not end with this GSoC project. I see it instead as a starting point in becoming a stable contributor and active member of the Jenkins community. As previously mentioned, in the short term I would like to keep working on the chatbot plugin even after GSoC, implementing some of the future improvements I've written(if mentors agree on that).

Talking about long term, by December 2025 I expect to have completed all my exams, and starting from January 2026, alongside from writing my thesis and applying for internships in the USA, I would love to stay engaged with the Jenkins ecosystem through consistent contributions. I'm especially interested in learning more about Jenkins plugin development and would love to eventually become a maintainer of a plugin, maybe adopting the chatbot one to keep growing it. I also look forward to participating more in community discussions and SIGs, particularly those focused on UX and Platform, which align well with my interests.

Looking very further ahead, I would love to return as a mentor for future GSoC contributors, giving back some of the support, respect(even if I know very little) and guidance I've received(and hope to continue receiving) from the community. Being part of an open-source community like Jenkins is incredibly motivating, and I'm genuinely excited about the opportunity to learn from this experience and to contribute not just through code, but by supporting others as well.

Conflict of Interests or Commitment(s)

I don't have any conflicts that will limit myself in correctly pursuing the project

Major Challenges Foreseen

- **Designing an effective tool invocation logic and controlling the agent correctly**

With the introduction of agents and tools, I believe that a big challenge will be balancing the agent's autonomy. Indeed, if not cared, agents might misuse tools, or chain unnecessary steps, leading to longer execution times and confusion in the answers provided. It's important to avoid "overthinking" behaviours from the agent. This will require some prompt tuning on the tool descriptions, the stop conditions and the agent type selection.

- **Managing LLM performance given the resource constraints**

Although I plan to use a quantized model to meet the resource constraints, I think there could be some issues yet, especially with speed performance. This will involve firstly testing how it handles different types of questions, especially those that require several reasoning steps. To handle that a way could be to exploit the future improvement I wrote about query classification.

- **Achieving a fully automated deployment experience**

One of the main challenges will be ensuring that, once the plugin is installed,

everything "just works" without requiring users to manually start the AI backend or make extra configurations. Since Jenkins plugins cannot launch or manage external processes, the backend has to be run separately. I think that automating this step in a user-friendly way will not be an easy thing.

References

- <https://www.capellasolutions.com/blog/streamline-data-deduplication-advanced-matching-techniques>
- <https://arxiv.org/abs/2406.00456>
- <https://medium.com/intel-analytics-software/low-bit-quantized-open-llm-leaderboard-748169e6a004>
- <https://arxiv.org/abs/2309.05516>
- <https://www.tensorops.ai/post/what-are-quantized-llms>
- <https://python.langchain.com/docs/introduction/>
- <https://docs.llamaindex.ai/en/stable/#introduction>
- <https://www.jenkins.io/doc/developer/book/>
- <https://www.tensorops.ai/post/prompt-engineering-techniques-practical-guide>
- <https://docs.discourse.org/>
- <https://arxiv.org/abs/2005.11401>
- <https://www.promptingguide.ai/>
- <https://www.zenrows.com/blog/scrapy-vs-beautifulsoup#which-is-best>
- <https://api.stackexchange.com/docs>
- <https://javadoc.jenkins.io/hudson/model/PageDecorator.html>
- <https://javadoc.jenkins.io/hudson/model/package-summary.html>

Relevant Background Experience

I have been contributing to Jenkins in the last month, and even though March has been a very demanding month with the university, I managed to make my first PRs and issues in a very short time.

Moreover, I have been involved in many research projects which concerned transformers, machine learning, and deep learning. These projects required me to read a lot of research papers, growing my ability to read and understand complex concepts and be able to understand and exploit state-of-the-art technologies. I believe this background will be particularly valuable when developing the core AI component of this project.

Additionally, I'm currently working on a university NLP course project that involves key techniques such as Retrieval-Augmented Generation (RAG), prompt engineering, large language models (LLMs), and data preprocessing. This experience is proving to be directly relevant to the GSoC project, reinforcing my understanding of the technical knowledge and giving me additional confidence in implementing those AI-powered systems.

Additionally, since the first year of my bachelor's degree, I've worked as a self-employed software developer, building websites, mobile apps, web apps and cloud-native solutions for many customers in industries like fitness, hospitality and digital startups. I also completed an internship as a full stack engineer, which gave me the opportunity to enhance my skills in

writing clean and maintainable code, get used to reading and understanding code written by others, collaborate closely with colleagues, and contribute to the writing of the documentation.

On top of this, I want to highlight my experience with Java, including writing tests using JUnit. A concrete example is my university project available on my [GitHub](#): "Development of a client-server board game (Codex Naturalis), covering all phases of the software lifecycle, from the design to the testing." The project includes advanced features like chat support, disconnection resilience, and parallel multi-game support, using both RMI and socket communication. This experience demonstrates my ability to handle backend logic, robust architecture, and test coverage — all of which will be essential when implementing the plugin's Java part.

These experiences have given me the building blocks to be able to handle entire projects on my own, and become confident with technologies that I will also exploit for this GSoC project, like React for the user interface part and Java for the plugin development. Additionally, thanks to my background on research projects, I believe I'm well prepared to approach the AI component as a proper research problem, by carefully collecting state-of-the-art tools and methods, critically evaluating them with respect to the specific needs of the project, and designing the ideal solution.

Personal

Hello everyone, I'm Giovanni Vaccarino, a 21-year-old computer engineering student. I hold a Bachelor of Science in Computer Engineering from Politecnico di Milano, and I'm currently pursuing a double degree Master of Science in Computer Science and Engineering between Politecnico di Milano and the University of Illinois Chicago.

I'm originally from Messina, a city in southern Italy -in Sicily- but at the age of 17, I moved to Milan to pursue my studies at Politecnico di Milano, one of the top technical universities in the country. This decision was driven by my determination to study in the most stimulating academic environment and to challenge myself alongside some of the brightest students in Italy. Starting in September 2025, I will continue my Master's journey in Chicago, where I will complete my coursework, work on my thesis, and hopefully explore career opportunities in the United States.

Since the very beginning of my academic journey, I've always been driven by a strong desire to learn by doing. During my first year of Bachelor's, I started working as a self-employed software developer, designing and building software solutions for a variety of clients. This hands-on experience taught me how to work independently, manage real-world challenges, and deliver results under pressure. Last summer, I also completed a full stack engineering internship, which offered a fresh perspective by immersing me in a collaborative, team-based development environment. I'm constantly looking for new opportunities and meaningful projects to broaden my skills and grow as a developer. Recently, I've started exploring the open-source world more seriously and I think it's a great way to contribute to impactful projects, deepen my understanding of real-world codebases, and adopt best practices.

When it comes to technical interests, I genuinely enjoy exploring a wide range of topics. I'm especially passionate about Artificial Intelligence, particularly areas like deep learning and natural language processing, which is one of the key reasons this project resonates with me. At the same time, I'm equally enthusiastic about software engineering, whether it's web

development, cloud-native technologies, or algorithms and data structures. More recently, I've been diving deeper into advanced computer architecture through my Master's courses, studying superscalar pipelines, memory hierarchies, and GPU execution models. I love connecting concepts across domains to have a better global understanding, and I'm always eager to expand my knowledge wherever I can.

Over the past month, I've been actively engaging with the Jenkins community. Even though it's only been a short time, I've learnt a lot of things that go from technical skills like writing clean, well-tested and maintainable code, to communication and collaboration skills. One of the most important things I've learned is the value of asking questions. At first, I hesitated to reach out on forums, but I quickly realized how helpful and responsive the community is. This shift in mindset has been incredibly valuable and not something I take for granted.

Moreover, attending some of the regular meetings have helped me understand in which part of the Jenkins community I would like to focus and contribute more. Even without understanding everything that has been discussed, I feel that listening to these has not been useless at all. Additionally, by joining these meetings I have had the opportunity to interact with Jenkins admins and active contributors, asking them questions and feeling more at home in the community.

From what I've explored so far, Jenkins looks like a very flexible and powerful tool. As a beginner, I'm impressed by how extensible it is. Indeed, if you want something just search for the plugin that does it and if you can't find it, you can create it yourself. The plugin ecosystem makes it incredibly adaptable to different use cases.

I first heard about Jenkins when I was doing a project for my software engineering course last year, and I was exploring tools to build a CI/CD pipeline. At the time, my understanding was quite high-level. However I recently discovered that Jenkins was promoting some projects for the GSoC and decided to dive deeper. Since then, I started exploring its documentation, looking at the forums, installing Jenkins locally, experimented with plugins and familiarizing myself with the codebase.

A few reasons why I want to participate in this GSoC project:

- I know this experience will help me grow — technically and personally
- I believe this project has the potential to make a significant impact on the Jenkins users
- It's a great opportunity to step-up to a higher level, getting myself used to write better, cleaner and well-documented code
- The Jenkins community has been very welcoming and available, and I see it as a great environment to grow in
- In the future I would love to become an active contributor or even an active member of an open source project, and this would be a great first step to achieve this
- Being mentored by experienced developers is a big opportunity to learn and level up

Why Me?

I believe I'm a strong fit for this project because I combine technical readiness with high motivation and a genuine passion for the problem we're trying to solve. I'm confident in my skills with modern web development (React, TypeScript), AI/NLP tools (LangChain, RAG, embedding models), and Java (including building and testing plugins using JUnit). My GitHub reflects my experience, from university projects to self-driven experiments like the

AI assistant plugin.

But more than that, I truly believe in this project's importance. Helping Jenkins users navigate its vast documentation through an intelligent, responsive chatbot would have a real, tangible impact, and I want to be the one building that. I'm deeply motivated to push my limits, improve my skills, and work hard on something meaningful. I don't just want to participate in GSoC, I want to deliver a tool that Jenkins users will actually benefit from.

Hence, I have the confidence to believe I'm a strong candidate for this GSoC project, and I'm ready to give to this project my full commitment and energy, and I'd be thrilled to bring this idea to life alongside such an amazing community!

Availability and commitments

During the GSoC timeframe I will not be involved in any internships or vacations, and my university lecture will be over at that time. I will only have exams to do in June(exact dates are not yet published, but they should be only in the second half of June), however they won't be interfering with my availability. Indeed, even during that short period of exams(approximately 2 weeks), I am sure to assert at least 20 hours per week. I have no issues in giving my time also on weekends, so time availability will definitely not be an issue. Especially in July and August I will be available 30/35 hours per week most of the weeks.

Experience

Open Source Contributions:

Merged Pull Requests:

- <https://github.com/jenkinsci/htmlpublisher-plugin/pull/334>

Opened Pull Requests:

- <https://github.com/jenkinsci/blueocean-plugin/pull/2607>

Reported Issues:

- <https://issues.jenkins.io/browse/JENKINS-75392>

Skill Set

Languages:

- Java (4/5)
- Python (4/5)
- Javascript, Typescript (4/5)
- C# (3/5)
- C (3/5)

Frameworks and Tools:

- React (4/5)
- SQL (4/5)
- JUnit (3/5)
- Git (4/5)
- ASP.Net Core (3/5)
- NestJs (3/5)

- *TensorFlow (3/5)*
- *PyTorch (3/5)*

Related Research and Work Experience:

- Since my first year of bachelor degree I've worked as a self-employed software developer, building websites, mobile apps, web apps and cloud-native solutions for 15+ customers. The solutions has been designed, implemented and tested with React and AWS serverless backend
- I've done a four-month summer internship as a Full Stack Engineer at Innova et Bella in Milan- last year, where I've worked on both frontend and backend tasks, using TypeScript, React, NestJs and AWS. I've also worked on SEO optimization to enhance web visibility and boost performances.
- I've worked at a research project, conducting a research study on the fault tolerance of Vision Transformers and Convolutional Neural Networks, by analyzing two framework injection campaigns (github repo [here](#))
- I am currently working on a research project with the Autonomous Driving team MOVE of Politecnico di Milano, in collaboration with Fiat and Maserati. We're working at 3D Object detection focusing on camera-based approaches, and in particular on query-based methods
- I am currently working on a research project with the NECSTLab of Politecnico di Milano, where the goal is to develop a distributed system that exploits Federated Learning on edge devices for real-time health monitoring

Personal Reference Links

[Linkedin](#)