

AI Agent for conversational Q&A over SaaS codebase using large language models

Olga Cherednichenko^{1,†}, Dmytro Sytnikov^{2,†}, Nazarii Romankiv^{3,*}, Nataliia Sharonova^{4,†}, and Polina Sytnikova^{5,†}

¹ Bratislava University of Economics and Management, Furdekova 16, Bratislava, Slovak Republic

² National University of Radio Electronics, 14, Nauki prospect, Kharkiv, 61166, Ukraine

³ National University of Radio Electronics, 14, Nauki prospect, Kharkiv, 61166, Ukraine

⁴ National Technical University, "Kharkiv Polytechnic Institute", 2, Kyrpychova str., Kharkiv, 61002, Ukraine

⁵ National University of Radio Electronics, 14, Nauki prospect, Kharkiv, 61166, Ukraine

Abstract

Software as a Service (SaaS) has become the dominant model in modern software, yet its sprawling, complex codebases pose daunting challenges for developers—especially as new engineers onboard to the project. Traditional keyword-based search and static documentation struggle to address this scale. Meanwhile, recent breakthroughs in Large Language Models (LLMs) offer powerful capabilities in natural language understanding and semantic search. By leveraging these models, it becomes possible to build conversational AI agents that let developers query a SaaS codebase in natural language. In doing so, the agent can surface contextually relevant snippets, streamline problem-solving, and accelerate the onboarding process. This article introduces an AI Agent for conversational Question Answering over SaaS code, using LLMs to streamline information retrieval, accelerate onboarding, and enhance overall productivity. The proposed approach leverages natural language interactions to deliver rapid, relevant answers directly from the codebase, spotlighting the transformative potential of LLM-based solutions in large-scale SaaS environments.

Keywords

AI Agent, LLMs, SaaS, Langchain, Python

1. Introduction

Modern technologies and the Internet have enabled Software as a Service (SaaS) [1] models to not only emerge but become the dominant paradigm in the global software landscape. As highlighted in prior reports, the SaaS market continues its robust expansion, underpinning critical digital infrastructure across industries. Developing and maintaining these sophisticated SaaS applications, however, presents significant engineering challenges, especially with the increasing scale and complexity of cloud-native [2] architectures. A key challenge within SaaS development is efficiently navigating and understanding the vast, intricate codebases that constitute these systems. Developers often face substantial challenges in locating specific information, comprehending legacy code, and effectively collaborating on large projects. SaaS applications, by their very nature, are often composed of millions of lines of code, distributed across numerous modules, microservices, and repositories. This architectural complexity creates a significant cognitive load for development teams.

For experienced developers, even those familiar with the system, keeping abreast of changes, understanding the interplay of different components, and efficiently debugging issues can be arduous. They grapple with understanding legacy code, tracing dependencies across services, and ensuring consistent behavior in a rapidly evolving environment. However, these challenges are

ISW-2025: Intelligent Systems Workshop at 9th International Conference on Computational Linguistics and Intelligent Systems (CoLInS-2025), May 15–16, 2025, Kharkiv, Ukraine

* Corresponding author.

† These authors contributed equally.

✉ olga.cherednichenko@vseba.sk (O. Cherednichenko); dmytro.sytnikov@nure.ua (D. Sytnikov); ayzyrian@gmail.com (N. Romankiv); nvsharonova@ukr.net (N. Sharonova); polina.sytnikova@nure.ua (P. Sytnikova)

ORCID 0000-0002-9391-5220 (O. Cherednichenko); 0000-0003-1240-7900 (D. Sytnikov); 0009-0004-9893-6823 (N. Romankiv); 0000-0002-8161-552X (N. Sharonova); 0000-0002-6688-4641 (P. Sytnikova)



© 2025 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

acutely amplified for developers who are new to a SaaS project. Onboarding into a large SaaS codebase is often a daunting experience. Newcomers face a steep learning curve as they attempt to grasp the system's architecture, business logic, coding conventions, and intricate interdependencies. They struggle to locate relevant documentation (which is often outdated or incomplete), identify subject matter experts, and quickly become productive contributors. This prolonged onboarding period directly impacts development velocity, team efficiency, and ultimately, the ability to innovate and respond to market demands.

Traditional methods for code exploration exacerbate these issues. Keyword-based code searches, while useful for simple tasks, fall short when developers need to understand the context and semantics of code. Static documentation, even when meticulously maintained, often lags behind the pace of change in agile SaaS development. Consequently, developers, especially newcomers, spend excessive amounts of time simply searching for information, deciphering existing code, and asking colleagues for clarification – time that could be better spent on feature development and innovation.

Concurrently, the field of conversational interfaces is rapidly advancing, driven by breakthroughs in Large Language Models (LLMs) [3]. LLMs are revolutionizing human-computer interaction, demonstrating remarkable capabilities in natural language understanding, contextual awareness, and information retrieval. The potential of LLMs extends significantly into software engineering, particularly in enhancing developer productivity and knowledge access within complex systems. Imagine the efficiency gains if developers could ask questions about the SaaS codebase in natural language and receive intelligent, contextually relevant answers directly from the code itself.

In this article, we explore the application of Large Language Models to address the critical challenge of knowledge access within SaaS codebases. We propose an AI Agent specifically designed for conversational Question Answering (Q&A) over SaaS code. This agent leverages the power of LLMs to provide developers with an intuitive conversational interface for querying the codebase, thereby streamlining information retrieval, accelerating problem-solving, and facilitating a deeper understanding of complex SaaS architectures.

2. Related works

In the research [4], authors explored *how* and *why* software engineering students use LLMs (e.g., Copilot, ChatGPT) when doing a non-trivial team project. LLMs can be a strong enabler in academic team projects—particularly for early-stage scaffolding, or for smaller “standard” tasks (e.g., a DFS, a specific data structure). Hence, the LLMs here were used mostly to generate the code, and all interaction with LLM happened via prompt engineering.

K. Tamberg and H. Bahsi in their work [5], shows that LLM-based vulnerability detection can indeed match or exceed certain static tools in terms of recall and F1, especially with sophisticated prompts. They have utilized prompting variants like self-refinement approaches, chain of thought, tree of thoughts etc. However, major disadvantages are its higher cost and time, plus potential for false positives or classification mistakes. For everyday code scanning at scale, tools like CodeQL or SpotBugs remain strong.

The work [6] demonstrates that LLM-based automated repair of Ansible scripts is both feasible and promising. Although still imperfect—only about 70% of fixes are labeled as helpful, and *identical patches* are rare— *the approach can significantly assist developers* in rectifying Ansible issues for Edge-Cloud infrastructures. LLM-based reviews might be valuable for deeper or broader security audits, where coverage of “unusual” vulnerabilities—and having a natural-language explanation—becomes important.

In another work [7], the authors built a web platform where developers can enter software requirements (in natural language). The service uses ChatGPT (GPT-3.5) to produce source code. The tool includes:

- a User Interface for controlling parameters (like temperature, max tokens);
- a Prompt Builder that wraps user instructions in a structured “prompt engineering” format to systematically guide ChatGPT;
- a Backend Service written in Java, supporting streaming calls to ChatGPT for multi-turn code generation.

The authors propose systematically adding specific instructions around the user's initial prompt. This includes setting a "role" for ChatGPT (e.g., "system role = code expert"), and including coding conventions, file structures, explicit requirements, examples, etc.

The paper [8] provides a structured introduction to LangChain's main features and typical usage patterns, highlighting how developers can build practical and robust LLM-based solutions quickly. LangChain drastically speeds up development of LLM apps by unifying prompt engineering, conversation memory, multi-step "chains," retrieval from user data, and agent-based logic in a single framework. LangChain supports a variety of end-to-end LLM-based applications. The authors outline scenarios such as:

- chatbots that maintain conversation context and personality;
- autonomous agents (like AutoGPT variants) that iterate steps to accomplish tasks;
- document Q&A: let the user load data (PDF, CSV, websites, etc.), then query it in plain language;
- extraction and summarization pipelines that read text from multiple files, chunk them, and produce structured output or short summaries.

The paper [9] delivers a structured analysis of how large language models (LLMs) can be adapted for code summarization with minimal labeled data. It focuses on Codex (a GPT-based model) to demonstrate the feasibility and advantages of few-shot approaches, especially for project-specific tasks. Unlike traditional methods that rely on extensive fine-tuning, the paper shows how prompting the model with only a handful of examples can deliver high-quality summaries of source code. The authors also emphasize the value of local, project-specific examples, leveraging domain vocabulary, naming conventions, and code idioms that a project uses. They note that while zero-shot or one-shot prompting significantly underperforms, transitioning to 10-shot quickly elevates BLEU scores to surpass heavily fine-tuned alternatives. This underscores LLMs' capacity for fast adaptation with minimal overhead. A major takeaway is that each software project has unique identifiers, domain terms, and patterns. A small amount of localized data helps the LLM align with these local conventions more effectively than broad, cross-project data alone. This minimal-sample training paradigm could be generalized to various software engineering tasks that likewise benefit from the synergy of local domain context and powerful generative models.

The paper [10] provides a structured investigation into whether conversational, in-IDE AI assistants can improve developers' ability to read, grasp, and modify unfamiliar code. Dubbed GILT (Generation-based Information-support with LLM Technology), the authors' prototype leverages GPT-3.5-turbo to produce on-demand, context-aware help – without requiring users to craft specialized prompts. Instead, GILT automatically includes highlighted code as context and offers pre-generated "buttons" for explaining code sections or giving API usage examples. GILT embeds the code snippet or the entire file directly into queries for the LLM. This approach means developers can easily select a portion of code to get a summary, an explanation of libraries or API calls, or usage examples – significantly reducing the friction of copy-pasting code into a separate web-based AI tool. Quantitative results show that participants using GILT correctly completed more sub-tasks, though they did not complete them significantly faster nor achieve significantly higher quiz scores. Interestingly, the biggest productivity boosts appeared among professional developers, whereas students did not benefit as strongly.

The paper [11] aims to improve few-shot prompting for code summarization by automatically augmenting prompt text with semantic information derived from static code analysis. While large language models (LLMs) already show impressive performance at many software-engineering tasks, most approaches still rely on providing raw code snippets (plus a handful of examples) in the prompt. This work proposes a more principled way to supply domain-relevant data—like data-flow graphs, tagged identifiers, and repository metadata—so that the LLM can generate higher-quality summaries of new, unseen code.

Therefore, recent research underscores the growing adoption of LLMs in software engineering activities—from generating standard code fragments [4], to detecting vulnerabilities [5], patching configuration scripts [6], and providing context-aware coding assistance [7,10]. Studies also reveal how framework-based approaches (e.g., LangChain) streamline the development of robust LLM-driven solutions [8], while minimal data "few-shot" strategies enable local, project-specific adaptation for tasks like code summarization [9,11]. However, none of these efforts fully explore an end-to-end "AI Agent" capable of natural, conversational Q&A specifically tailored to large and

evolving SaaS codebases. Hence, the goal of this article is to build an AI Agent that leverages LLM capabilities, to help engineers get answers with regards to the complex SaaS codebases, help them onboard quickly, troubleshoot problems, and facilitate deeper understanding of the codebase.

3. Methods and Materials

We selected Python as the primary programming language due to its extensive ecosystem of libraries and frameworks essential for Artificial Intelligence and Natural Language Processing (NLP). Python's readability and versatility facilitate rapid prototyping and integration of diverse components, including code parsing, data preprocessing, and interaction with Large Language Models.

Also, our choice of FastAPI for constructing the backend API of the AI Agent was driven by a confluence of factors crucial for building a high-performance, developer-friendly, and production-ready system. Specifically, FastAPI offers several key advantages:

- **Exceptional Performance.** FastAPI is built on top of Starlette and Pydantic, leveraging asynchronous Python capabilities to achieve remarkable speed and efficiency. This asynchronous nature is paramount for our AI Agent, as it needs to handle concurrent user queries, interact with LLMs (which are often accessed via asynchronous APIs), and perform vector database searches, all without becoming a bottleneck. FastAPI's performance ensures low latency in responses, leading to a more interactive and responsive user experience for developers querying the SaaS codebase.
- **Asynchronous Capabilities and Concurrency.** As highlighted earlier, FastAPI's asynchronous nature is critical. It natively supports asynchronous request handling, allowing the API to efficiently manage concurrent requests and perform non-blocking operations, such as calling external LLM APIs or querying the vector database. This is in stark contrast to traditional synchronous frameworks, which can become easily overwhelmed under load. For a SaaS codebase Q&A system that might be accessed by multiple developers simultaneously, FastAPI's concurrency handling is vital for maintaining responsiveness and scalability.

To streamline the development and orchestration of our AI Agent, we utilized Langchain. This framework provides crucial abstractions and tools for building applications powered by Large Language Models. Langchain simplifies tasks such as prompt management, model interaction, retrieval augmentation, and creating conversational agents, significantly accelerating our development process and enabling a modular architecture.

To manage asynchronous tasks, particularly the processing of codebase changes and updates, we integrated RabbitMQ as a message broker. RabbitMQ facilitates a decoupled architecture, allowing us to efficiently handle code ingestion, parsing, and indexing as background processes. This asynchronous processing ensures that the AI Agent remains responsive to user queries even during codebase updates and enhances the system's scalability and resilience by distributing workload across different components.

To efficiently manage and search the vector embeddings representing code snippets, we chose OpenSearch. We selected OpenSearch as our vector storage and search engine because it provides a comprehensive and highly scalable solution specifically tailored for the demands of semantic code search and retrieval in large SaaS codebases. OpenSearch offers a compelling set of features that make it ideally suited for our AI Agent:

- **Native Vector Database Functionality.** OpenSearch has robust native support for vector databases and similarity search. This is not just an add-on feature but deeply integrated into its architecture. It allows us to efficiently index and query the vector embeddings generated by text-embedding-3-large, enabling semantic search capabilities that go beyond simple keyword matching. OpenSearch supports various similarity metrics (like cosine similarity, which is commonly used for embeddings), allowing us to find code snippets that are semantically similar to user queries.
- **Scalability and Performance for Large Codebases.** SaaS codebases can be massive, containing millions of lines of code and numerous files and modules. OpenSearch is designed for scalability and high-performance search over large datasets. Its distributed architecture allows it to handle the indexing and querying of vast vector datasets efficiently. This scalability is crucial for our AI Agent to remain responsive and performant even when deployed on extensive SaaS codebases.

OpenSearch's ability to scale horizontally by adding more nodes ensures that the system can grow with the increasing size and complexity of the codebase it analyzes.

- **Full-Text Search Capabilities (Beyond Vector Search).** While vector search is central to our semantic code retrieval, OpenSearch is also a powerful full-text search engine. This allows us to combine vector search with traditional keyword-based search if needed. For example, we could potentially refine vector search results by also incorporating keyword matches or use full-text search for tasks like finding specific code patterns or variable names. This flexibility to leverage both semantic and keyword search within a single platform is advantageous.
- **Integration with Cloud Environments and Deployment Options.** OpenSearch is easily deployable in various cloud environments and offers flexible deployment options, including managed services from cloud providers. This simplifies the deployment and management of our AI Agent in real-world SaaS development settings, which often rely on cloud infrastructure. Its cloud-native design makes it a practical choice for integrating with modern SaaS development workflows.

We employed MongoDB as our primary database to store various application data, including processed code representations, user interactions, and conversational history. MongoDB's NoSQL nature and flexible schema are advantageous for handling semi-structured data and evolving data models common in application development. Its scalability and document-based structure offer efficient storage and retrieval for diverse data elements within our system.

For our AI Agent, we strategically selected a suite of Large Language Models from Anthropic and Voyage AI, each chosen for its specialized strengths in different aspects of our system:

- **Claude 3.5 Haiku [12]** (for Code Summarization and Entity Extraction). We chose Claude 3.5 Haiku for the initial stage of code processing, specifically for its exceptional speed and efficiency in summarizing code files and extracting key domain entities. Haiku's rapid processing capabilities are crucial for quickly generating concise summaries of code modules, which are then used to augment the codebase representation. Its proficiency in identifying key entities within code (like class names, function descriptions, and module purposes) allows us to enrich the indexed data with semantic information, improving the relevance of search results and the LLM's contextual understanding. The focus on speed with Haiku is vital for efficient preprocessing of large SaaS codebases, ensuring a scalable and responsive ingestion pipeline.
- **Claude 3.7 Sonnet [13]** (for Conversational Question Answering). For the core conversational Q&A functionality, we selected Claude 3.7 Sonnet. Sonnet offers a superior balance of intelligence and speed compared to larger, more computationally intensive models, making it ideal for interactive applications. Its advanced natural language understanding and reasoning abilities enable it to effectively interpret complex developer queries about the codebase and generate contextually accurate and helpful answers. Claude 3.7 Sonnet's ability to maintain context over multi-turn conversations is essential for a fluid and productive developer experience, allowing for iterative question refinement and deeper codebase exploration.
- **To generate meaningful vector representations of our text data—including both raw code and the summaries produced by Claude 3.5 Haiku—we utilized the text-embedding-3-large[14] model.** This model is specifically designed for text embeddings and excels at capturing semantic nuances within written material. We opted for a vector size of 1024 to strike a balance between semantic richness and storage efficiency. Given the potentially massive volume of code in SaaS applications, keeping the vector dimension at 1024 significantly optimizes storage requirements within our OpenSearch vector database and accelerates similarity search operations. While smaller vector sizes could further reduce storage needs, they might sacrifice some fine-grained semantic detail compared to higher dimensions. Our experiments showed that a 1024-dimensional vector space effectively captures the essential semantic information for robust text retrieval and question answering in our SaaS context, offering a practical trade-off for scalability.

As for the testing ground we have selected the open source codebase for ERP/CRM [15]. It is a well-suited codebase for evaluating your AI Agent for several reasons:

- **Representative SaaS Codebase Architecture.** MERN stack foundation reflects the architecture of many modern SaaS applications. The use of Node.js for the backend, React.js for the frontend, and MongoDB for data storage is a common pattern in cloud-native development. This makes it a realistic testing ground for your AI Agent, as the challenges encountered

within the codebase are likely to be representative of those in real-world SaaS projects. The modular nature implied by the MERN stack and mention of Ant Design components provides a level of architectural complexity that is valuable for testing the agent's ability to navigate and understand component interdependencies.

- **Meaningful Business Domain Complexity.** ERP and CRM systems, inherently involve complex business logic and data models. Concepts like invoices, quotes, customers, accounts, and inventory are interconnected and have specific business rules governing their behavior. This domain complexity ensures that questions posed to the AI Agent will require more than just simple keyword searches; they will necessitate understanding the underlying business logic and relationships within the code. This complexity is essential for demonstrating the value of a conversational AI Agent that can reason about the codebase's semantics, not just find code snippets.

4. Experiment

The AI Agent architecture consists from tow major components, the code ingestion pipeline and the user interface to answer engineer's questions. Let's review the ingestion pipeline represented on the diagram (Fig. 1).

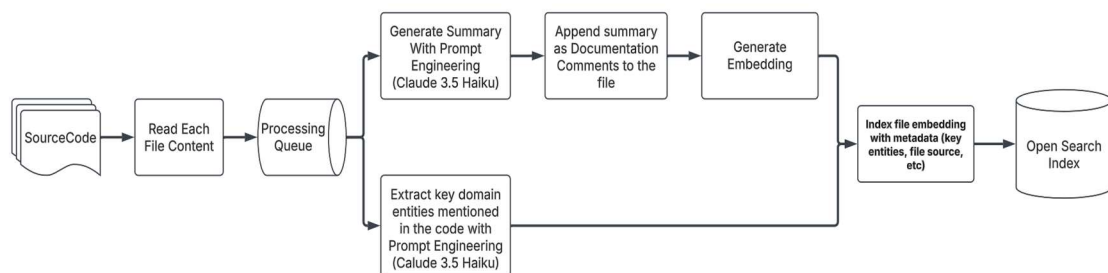


Figure 1: Code Ingestion Pipeline

- The pipeline reads each file one by one. It extracts raw elements (such as classes, functions, methods, or comments) and then relays them in a structured text format for further processing.
- All extracted content is funneled into a central queue. This enables subsequent operations—like summarization or entity extraction—to run in parallel or asynchronously as resources permit. Managing files through this queue architecture allows for seamless scaling when large sets of files need to be handled.
- Each file's contents are sent to a Large Language Model, "Claude 3.5 Haiku," using carefully crafted prompts. The LLM returns concise summaries capturing key functionality, etc. Those summaries are then prepared for integration into the source file.
- The generated summary is inserted into the original source code as comment blocks at the file's beginning. This step automatically enriches the code with human-readable explanations.
- Parallel to summary creation, another LLM-based approach isolates important domain-specific entities (e.g. specialized terminology, domain entities, etc.). This yields a structured set of terms, which is then associated with the file to assist in later stages of indexing and search.
- With documentation and domain entities in place, the system uses a specialized model text-embedding-3-large to create vector embeddings. These embeddings encode the semantic relationships within the code, placing similar or related files closer together in a high-dimensional vector space.
- Finally, the generated embeddings and accompanying metadata (file paths, identified entities, etc.) are stored in an OpenSearch index. This indexed information can be efficiently queried, using vector search.

The user interface architecture, as well as full integration with backend and AI agent infrastructure depicted on the following diagram (Fig. 2).

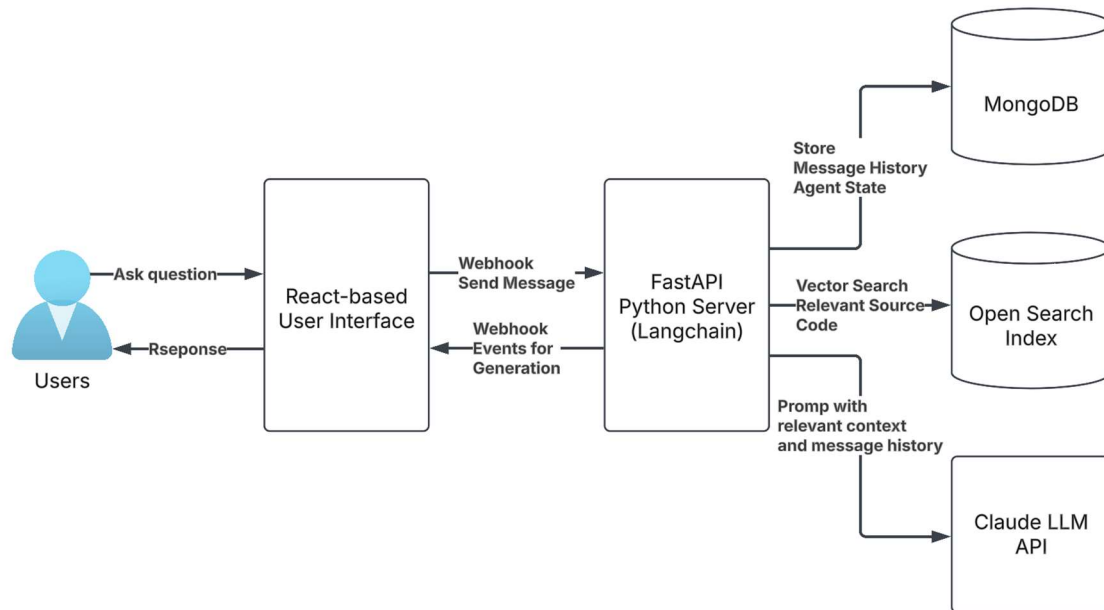


Figure 2: End-to-end integration of UI and backend with AI Agent infrastructure

This diagram (Fig. 2) illustrates the architecture of the Conversational AI Agent, showing how user questions are processed and answered.

The process starts when a User interacts with the system by submitting a question through a front-end interface component.

This interface then sends the user's question to a backend server component. This server component, which is the central processing unit, performs several actions.

First, it initiates a search for relevant code within a specialized data storage that holds indexed representations of the codebase. This search retrieves code snippets related to the user's question.

Second, the server component formulates a request for an external language model service. This request includes the user's question, the relevant code snippets found in the search, and potentially the history of the ongoing conversation.

The external language model service then generates a natural language response based on the provided information and sends it back to the server component.

The server component also stores the conversation history and internal state in a database for managing ongoing interactions.

Finally, the server component sends the generated response back to the front-end interface component, which then displays the answer to the User.

In essence, the system allows users to ask questions which are processed by a backend server to retrieve relevant code and then leverages an external AI model to generate a natural language answer, all managed through a user-friendly interface.

Next diagram (Fig. 3), represents cognitive architecture of our AI Agent. A cognitive architecture for an AI agent is a framework that outlines the fundamental components and control mechanisms of the agent's "mind." It defines how different cognitive functions, such as memory, reasoning, and action selection, are organized and interact to enable intelligent behavior.

Let's review each step in the cognitive architecture:

- **Rewrite User Question.** the agent first refines or reformulates the original question to optimize it for effective code or knowledge retrieval. This might involve adding context, clarifying ambiguities, or adjusting terminology to align with the indexing and search mechanisms. What's more important the search query will be rewritten taking into account the whole conversation context.
- **Search for Relevant Code Fragments.** Using the optimized question, the agent conducts a search against indexed code in OpenSearch index to locate potentially relevant snippets. This

involves vector-based semantic search, at this moment we generate an embedding of the user query via text-embedding-3-large and perform vector search.

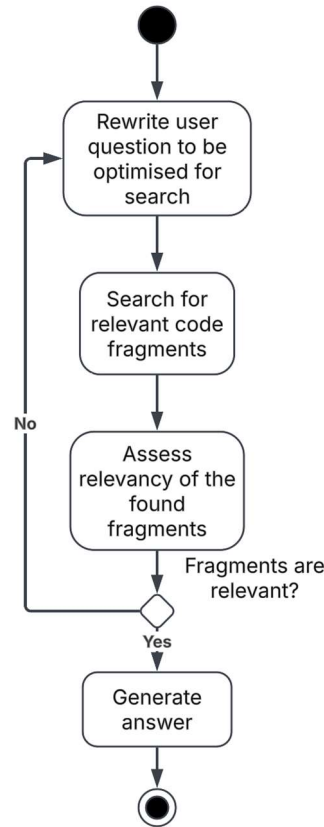


Figure 3: Cognitive architecture of AI Agent

- **Assess Relevancy of Found Fragments.** The retrieved snippets are evaluated to ensure they directly address the user’s query. This step filters out tangential or low-quality results, aiming to keep only the most pertinent information. If the fragments are not relevant the process loops back to Step 1, prompting a re-analysis or further refinement of the question. This feedback loop continues until sufficiently relevant fragments are located or the agent deems the query unsolvable.
- **Generate Answer.** Using the confirmed relevant code or knowledge fragments, the agent formulates a comprehensive answer. This response may be an explanation, code fix, or guidance, depending on the user’s original intent.

To evaluate the performance of the AI Agent, we have prepared a set of questions covering the business domain, how-tos, etc. related to the system.

5. Results

5.1. Ingestion Pipeline

First, we estimated the performance and cost of our ingestion pipeline, that takes all the files from a codebase, and indexes into a vector database, while generating summaries for each file, and extracting key domain entities. The table 1, contains key metrics per averaged per hundred files processed.

Indeed, the current processing times per hundred files is quite big and takes on average nine hundred thirty seconds. This is caused by long waiting times for LLMs to generate summaries and extract key entities for each file. Though, this can be significantly optimized, by parallelizing processing of each file, and scaling the consumer of the ingestion queue horizontally.

The cost of processing per hundred files is acceptable, considering that we selected cost-efficient LLM and embedding models. Albeit, on a huge scale if we ever need to process hundreds of thousands of files daily, this solution will require a self-hosting of an LLM to be economically viable.

Table 1
Statistics of ingestion pipeline

Metric	Value
Processing Time	930 seconds
Cost of Embedding	0,012\$
Cost of Summarization	0,4\$
Cost of Entity Extraction	0,13\$

5.2. AI Agent Interaction

The figure 4 shows the user interface of AI Chat, the interface uses a straightforward conversational layout, with messages stacked vertically. Each message block is clearly separated, making it easy for users to scan through the dialogue. Messages from the “AI Chat Assistant” and the user are visually differentiated. This reinforces who is speaking and enhances readability.

The user can see the history of the chat, each question and answer displayed as a separate bubble. What’s more important that the at the bottom of AI response, there’s a short list labeled “Sources” with references to various source code files (e.g., backend/controllers/invoiceController/index.js), which enables the engineer to jump to the relevant source code quickly.

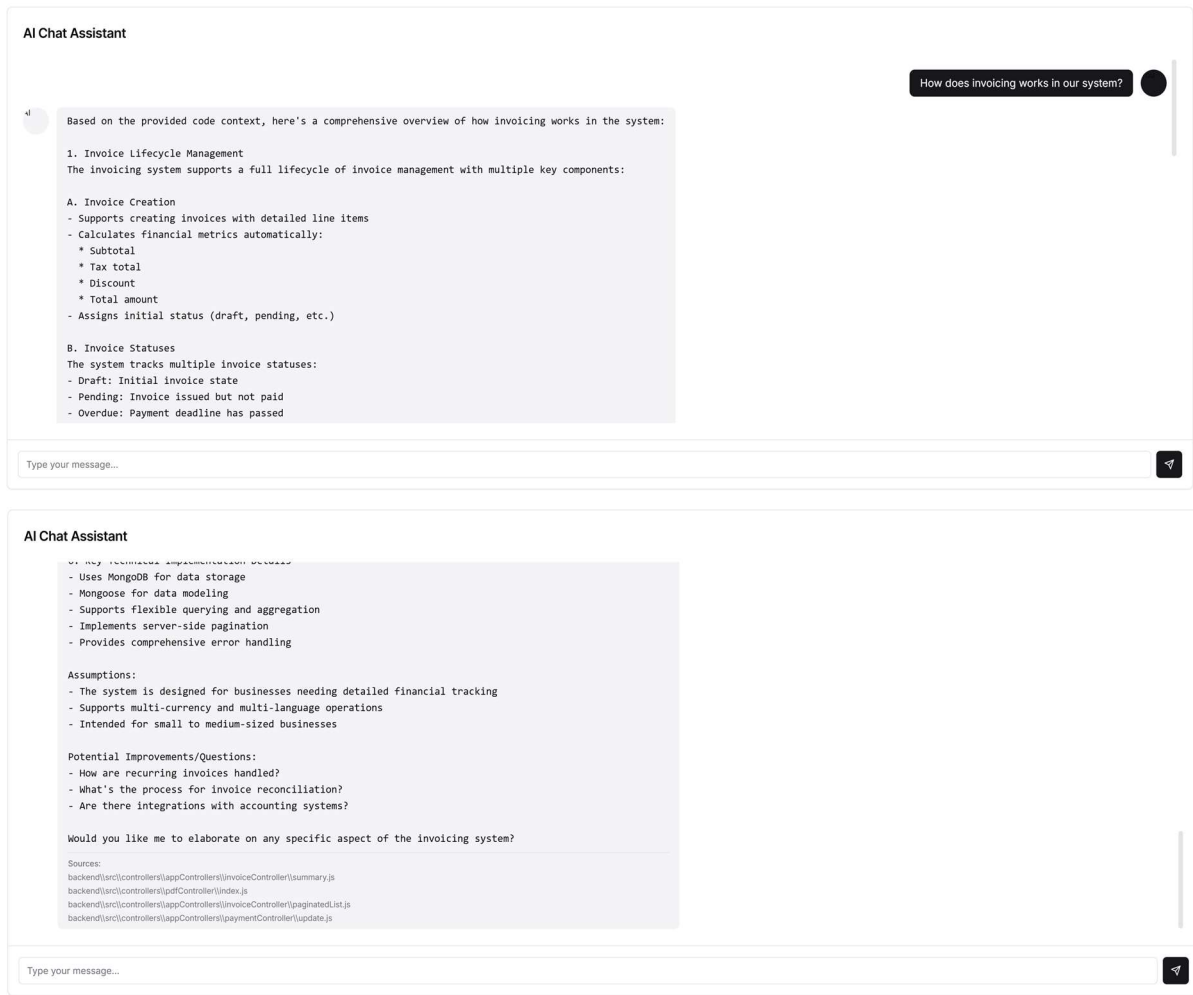


Figure 4: User Interface of AI Agent

After benchmarking our AI Agent, we have capture 2 key metrics in the table 2, namely average response time and cost per answer. Albeit, average response time seems high, taking into account that comparing it to a time it would take for an engineer to find out relevant information, this is a great result, for a cost of just 0.004\$ per answer.

Table 2
Statistics of bot usage

Metric	Value
Average Response Time	28s
Average Cost of Answer	0.004\$

5.3. Example of answer

To showcase the capabilities of developed AI Agent, we provide an example of generated answer.

Question – How does invoicing works in our system?

Answer –

1. Invoice Lifecycle Management

The invoicing system supports a full lifecycle of invoice management with multiple key components:

A. Invoice Creation

- Supports creating invoices with detailed line items
- Calculates financial metrics automatically:
 - * Subtotal
 - * Tax total
 - * Discount
 - * Total amount
- Assigns initial status (draft, pending, etc.)

B. Invoice Statuses

The system tracks multiple invoice statuses:

- Draft: Initial invoice state
- Pending: Invoice issued but not paid
- Overdue: Payment deadline has passed
- Partially Paid: Some payment received
- Paid: Full payment completed
- Unpaid: No payment received

2. Financial Calculation Mechanics

- Automatic calculation of:
 - * Item-level totals (quantity * price)
 - * Subtotal
 - * Tax total (based on configurable tax rate)
 - * Discount application
- Precise financial calculations using a dedicated calculate helper

3. Invoice Management Features

A. Pagination Controller

- Supports flexible invoice list retrieval
- Features:
 - * Paginated results
 - * Sorting capabilities
 - * Search across multiple fields
 - * Filtering options

B. Invoice Summary Dashboard

- Generates comprehensive invoice metrics
- Tracks:
 - * Total invoice count and amount
 - * Status distribution percentages
 - * Payment performance
 - * Unpaid/partially paid amounts

4. PDF Generation

- Dynamically generates professional PDF invoices
- Supports multiple languages and currency formatting
- Uses Pug templates for consistent document design

5. Update Mechanism

- Allows modification of existing invoices
- Recalculates financial metrics on update
- Automatically adjusts payment status based on payments received

6. Key Technical Implementation Details

- Uses MongoDB for data storage
- Mongoose for data modeling
- Supports flexible querying and aggregation
- Implements server-side pagination
- Provides comprehensive error handling

Assumptions:

- The system is designed for businesses needing detailed financial tracking
- Supports multi-currency and multi-language operations
- Intended for small to medium-sized businesses

Potential Improvements/Questions:

- How are recurring invoices handled?
- What's the process for invoice reconciliation?
- Are there integrations with accounting systems?

Would you like me to elaborate on any specific aspect of the invoicing system?

6. Discussions

The presented AI Agent for conversational Q&A over SaaS codebases demonstrates tangible benefits in addressing one of the core difficulties in modern cloud-native software engineering: efficient knowledge retrieval and onboarding. By combining Large Language Models (LLMs) with vector-based semantic search, our system delivers contextually relevant information to developers in an intuitive dialogue format, significantly reducing the time spent on manual code exploration.

Our solution stands on the shoulders of recent advances in LLM-driven software engineering. Similar to prior research on code generation [4], vulnerability detection [5], and automated repair [6], we leverage LLMs to enhance developer productivity—though our focus differs by aiming for end-to-end conversational Q&A tailored to large SaaS codebases. As with framework-based approaches (e.g., LangChain [8]) and few-shot strategies [9,11], our system extends these concepts into a specialized domain: assisting developers in navigating large, complex, and rapidly evolving SaaS projects.

One direct advantage of this approach is the rapid onboarding of new engineers. Traditional knowledge transfer in SaaS environments is often slowed by outdated documentation and scattered institutional knowledge. Our AI Agent alleviates these bottlenecks by maintaining updated semantic indexes (via OpenSearch) and human-readable, LLM-generated summaries embedded within the code. This aligns with insights from [7,10], emphasizing how context-aware assistants integrated into developers' workflows reduce friction in comprehending unfamiliar code sections.

That said, several important considerations emerged:

- **Performance and Cost.** Our ingestion pipeline currently incurs both time (e.g., ~930 seconds per 100 files) and cost (summarization, entity extraction, vector embeddings). Although this is acceptable for moderately sized repositories, the pipeline must be parallelized and perhaps distributed across multiple workers to handle massive enterprise-scale codebases rapidly. As we observed, hosting self-managed LLMs may become more economical at high volumes, echoing concerns from [5] on the cost-effectiveness of large-scale LLM usage.
- **Potential for Hallucinations.** Like other LLM-centric systems, the AI Agent may generate confident yet inaccurate responses—an issue known as “hallucination”. While we mitigate this risk via retrieval-augmented prompting (i.e., tying responses to source code snippets),

users must remain vigilant and confirm critical information. This challenge underlines the need for continued advances in LLM alignment and validation techniques.

- **Quality of Summaries and Entity Extraction.** Although the Claude 3.5 Haiku model provides fast and coherent summaries, there is no guarantee of perfect accuracy for especially intricate modules or unconventional code structures. Ensuring robust domain adaptation—potentially via local few-shot examples [9,11]—remains a key step to improving summarization consistency. Similarly, entity extraction may overlook domain nuances unless carefully guided by specialized prompts or additional training data.
- **Scalability of Vector Indexing.** While OpenSearch natively supports high-volume indexing, maintaining real-time code coverage in dynamic SaaS projects can become computationally intensive. Large refactoring or frequent incremental updates may necessitate micro-batch ingestion or near-continuous indexing. Future work should explore incremental embedding strategies that minimize re-processing time when only certain files change.
- **Security and Privacy Implications.** Embedding sensitive code and shipping it to external LLM endpoints raises confidentiality concerns, especially for enterprise SaaS. Self-hosted or on-premise LLM deployments could mitigate data-exposure risks but introduce new complexities in model management and hardware costs.

7. Conclusions

Developing and maintaining a complex SaaS solution becomes ever-increasing challenging task, especially as you need to scale your engineering team and onboard new engineers and get them up to speed quickly. Therefore it's important to apply modern techniques to let the engineers have access to the latest information.

The article try to deal with this challenge, by means of introducing an AI Agent that is in context of your codebase and can find relevant code fragments based on the user question and the chat context, and what's more important this happens in a conversational manner, where an engineer can speak in a natural language. To achieve that we built an architecture that consists from two parts. The first one is ingestion pipeline that indexes the codebase in a scalable manner, using queues, to ensure that we can index huge SaaS codebases. The second part is user-interaction UI, where an engineer can interact with AI Agent in a conversational manner, and get answers to his questions.

To achieve best performance of our AI Agent, we have selected a combination of Claude LLMs, and text-embedding-3-large embedding model. As a vector storage we utilized OpenSearch, known for its scalability and native vector search support.

Though it is worth to mention that the prototype that we developed still operates on the basis of one “repository” and doesn't have integration with remote VCS system like GitHub or GitLab, to automatically re-index changes as they occur.

Apart from it, currently the AI Agent only has context of codebase, but not integrated into other ecosystem of tooling that typical IT project uses, like Jira, Confluence etc. There is huge potential in this work, to give the AI Agent access to knowledge from knowledge base, ticketing systems etc, which could bring the quality of his answers to a new level. These capabilities to be explored in future works.

Acknowledgements

The research study depicted in this paper is partially funded by the EU NextGenerationEU through the Recovery and Resilience Plan for Slovakia under project No. 09I03-03-V01-00078.

Declaration on Generative AI

The authors have not employed any Generative AI tools.

References

- [1] M. J. Kavis. Architecting the Cloud Design Decisions for Cloud Computing Service Models (SaaS, PaaS, and IaaS), Wiley, New Jersey, NJ, 2014
- [2] T. Erl, R. Puttini, Z. Mahmood. Cloud Computing, Concepts, Technology & Architecture 2nd. ed., Pearson, 2023
- [3] Large language model, URL: https://en.wikipedia.org/wiki/Large_language_model
- [4] S. Rasnayaka, G. Wang, R. Shariffdeen and G. N. Iyer, An Empirical Study on Usage and Perceptions of LLMs in a Software Engineering Project, 2024 IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code), Lisbon, Portugal, 2024, pp. 111-118
- [5] K. Tamberg and H. Bahsi, Harnessing Large Language Models for Software Vulnerability Detection: A Comprehensive Benchmarking Study, in *IEEE Access*, vol. 13, pp. 29698-29717, 2025, doi:10.1109/ACCESS.2025.3541146
- [6] S. Kwon, S. Lee, T. Kim, D. Ryu and J. Baik, Exploring LLM-Based Automated Repairing of Ansible Script in Edge-Cloud Infrastructures, in *Journal of Web Engineering*, vol. 22, no. 6, pp. 889-912, September 2023, doi:10.13052/jwe1540-9589.2263
- [7] Y. Li, J. Shi and Z. Zhang, An Approach for Rapid Source Code Development Based on ChatGPT and Prompt Engineering, in *IEEE Access*, vol. 12, pp. 53074-53087, 2024, doi:10.1109/access.2024.3385682
- [8] Topsakal, Oguzhan & Akinci, T. Cetin, Creating Large Language Model Applications Utilizing LangChain: A Primer on Developing LLM Apps Fast. International Conference on Applied Engineering and Natural Sciences. 1. 1050-1056. doi:10.59287/icaens.1127
- [9] Toufique Ahmed, Premkumar Devanbu, Few-shot training LLMs for project-specific code-summarization, 2022 doi:10.48550/arXiv:2207.04237
- [10] D. Nam, A. Macvean, V. Hellendoorn, B. Vasilescu and B. Myers, Using an LLM to Help with Code Understanding, 2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE), Lisbon, Portugal, 2024, pp. 1184-1196, doi: 10.1145/3597503.3639187
- [11] T. Ahmed, K. S. Pai, P. Devanbu and E. T. Barr, Automatic Semantic Augmentation of Language Model Prompts (for Code Summarization), 2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE), Lisbon, Portugal, 2024, pp. 2720-2732
- [12] Claude 3.5 Haiku \ Anthropic, URL: <https://www.anthropic.com/claude/haiku>
- [13] Claude 3.7 Sonnet \ Anthropic, URL: <https://www.anthropic.com/claude/sonnet>
- [14] Model - OpenAI API, URL: <https://platform.openai.com/docs/models/text-embedding-3-large>
- [15] Ayzrian/idurar-erp-crm: Free Open Source ERP CRM Accounting Invoicing Software | Node Js React, URL: <https://github.com/Ayzrian/idurar-erp-crm>