Query the Repository! Inspecting Github Repositories with Open-Source LLMs

Razvan Florian Vasile
University of Bologna
Department of Computer Science
razvanflorian.vasile@studio.unibo.it

Abstract

Large Language Models (LLMs) have gained significant popularity in recent years due to their remarkable question answering capabilities. However, when tackling a large codebase, the quality of the answers varies, largely due to the model's inability to focus on contextualized information. To address this challenge, this work presents a Retrieval Augmented Generation (RAG) pipeline designed to facilitate code retrieval and understanding with Github repositories. The approach empowers LLMs by combining non-parametric memory (retrieved code snippets) with parametric memory (pre-trained LLM weights) to generate insightful information.

The project emphasizes the engineering process, adhering to the agile methodology and documenting the development process. Notably, the system utilizes open-source technologies such as Ollama and Qdrant, enabling the utilization of various open LLMs through local indexing and retrieval of code snippets without reliance on proprietary services.

The work is meant to reduce the steep learning curve associated with understanding large codebases, and provide insightful explanations for complex coding concepts. The library is built in Scala, on top of the Langchain4j framework, and facilitates the integration with the LLM through interfaces built with Gradio and Scala.js. The library is available at https://github.com/atomwalk12/PPS-22-git-insp.

1. Introduction

Problem definition. Suppose a dedicated engineer had the habit of delving into unknown Github repositories, motivated by the desire to make a first contribution to a library, or wanting to learn better coding habits to apply to one's workflows or perhaps driven by the sheer curiosity to learn something new about a topic of interest. For all these reasons, tools that reduce the steep learning curve inherent to getting to know new codebases can facilitate the process and may provide useful insights into how repositories are structured, and how they are used.

Key ideas. In this work, I use a Retrieval Augmented Generation (RAG) workflow to answer questions about code stored in Github repositories. The task involves empowering Large Language Models (LLMs) through a hybrid approach that utilizes both pre-trained memory, as well as knowledge retrieval from Github repositories.

The project pays particular attention to the engineering process and follows an agile development methodology. Specifically, the process involves documenting the development process, creating a requirements document, setting up an architecture, creating an HTTP API with two interfaces (Gradio and Scala.js) and evaluating the system's performance to ensure it satisfies the requirements.

Defining success. The development process is documented and the successful completion of the project is verified by creating a substantial list of acceptance tests, each extensively documented in a traceability matrix. Moreover, a user questionnaire is used to assess the system's usability and effectiveness with a group of 5 users. An analysis is carried out to qualitatively assess the system's performance, which is done by visualizing the generated model's embeddings in a 2D space and observing pair-wise embedding similarities to identify clusters, offering insights into the inner workings of the process.

Paper organization. The paper is organized as follows: it begins with Section 2, where I describe the related work, introducing context into the core technologies. Sections 3.1 and 3.2 build up by introducing the general development process and key requirements, after which Sections 3.3 and 3.4 illustrate how the system is designed and implemented. The evaluation is presented in Section 4, followed by a brief conclusion in Section 5, with a reflection on the project's outcome and future work.

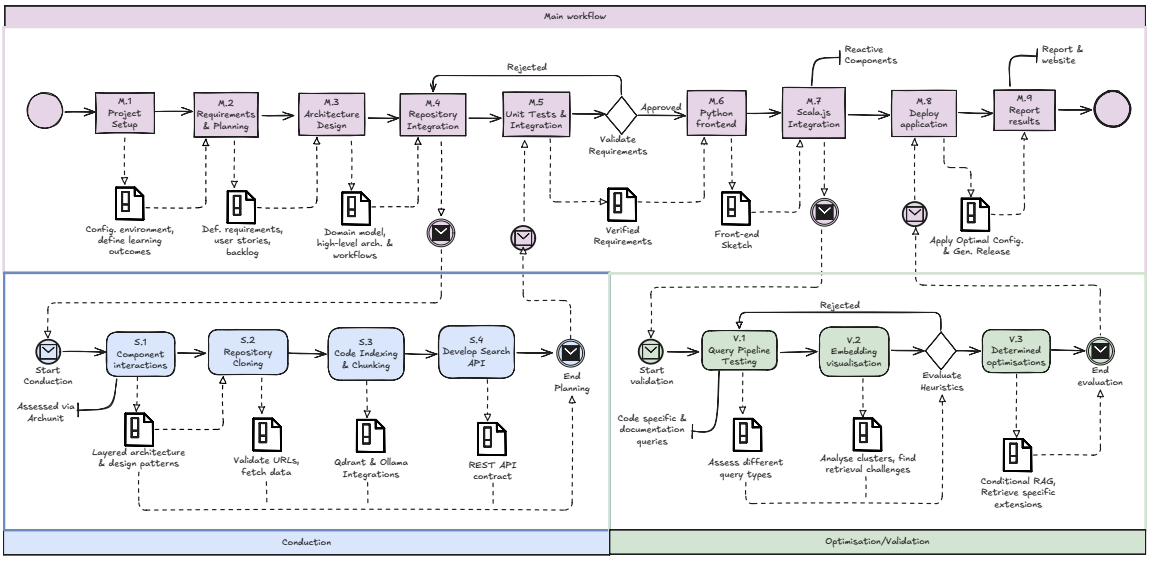


Figure 1. The main workflow highlighted in pale purple consists of four main phases: 1) initial planning (M.1-M.3), 2) development of the backend (M.4-M.5), 3) creation of the web interfaces and evaluation (M.6-M.8) and 4) reporting the results (M.9). The sub-processes highlighted using a blue background and a green background represent sub-activities.

2. Related work

Core Technologies. With respect to the core technologies, the frontend (web interface) is developed using two popular libraries, namely *Gradio* [1] and *Scala.js* [20]. The backend provides a RESTful API to facilitate client integration, and utilizes the *Langchain4j* [12] framework for seamless application of common agent workflows.

The Langchain framework provides primitives for building agents leveraging the *Ollama* [15] and *Qdrant* [16] backends, providing a flexible and modular architecture for the development of RAG pipelines. The Ollama module consists of a wrapper around the Ollama API, which in turn provides the necessary RESTful API for communicating with the LLM. The Qdrant module allows for efficient local indexing of a codebase, with the subsequent retrieval of code snippets that are relevant to the user's query.

Code Parser. The Langchain4j [12] framework is not as mature in terms of the available features as the Python Langchain [9] framework, the latter containing a superset of features present in the former. For instance, one challenge that arose during development was the lack of a built-in code parser that would allow to split the codebase into semantically meaningful chunks. This operation plays a crucial part in the effectiveness of the retrieval process, and since the existing parsers from the Langchain4j [12] library lack code splitters specialized for code, I ported the algorithm present in the Python Langchain [9] library to Scala.

This decision is fairly important, and as such some alternative options were considered. For instance, an alternative parsing algorithm was found by inspecting the *LlamaIndex* [7] source code, which implements a parsing mechanism that builds a concrete syntax tree for a source file using the tree-sitter library [27], which may offer improved performance[1]. However, I decided not to use it due to its heavy reliance on external dependencies. The Langchain alternative implements a recursive character splitter algorithm, which is a simpler to implement, and is still effective for the task at hand.

Agent framework. The *Open-AI-Scala* [5] library is an alternative to the Langchain4j library, however it was not used due to the following reasons. Firstly, the ability to decide at runtime whether to use a vector store is not present. Moreover, the functionality to filter the results based on specific metadata information (such as extension types) is also missing. The Git Inspector library utilizes all these features, which play an important role in the optimization process.

3. System Design and Implementation

3.1. Development Process

Development Process. A general overview of the development process is highlighted in Figure 1. The figure suggests an iterative development process, where the key phases are the following:

[1]For a more through discussion, see this blog post.

ID	Description
BR1	Allow users to efficiently search and understand code within Git repositories.
BR2	Improve developer productivity by facilitating code search/understanding workflows.
FR1.1	As a user, I can specify a Git repository URL to inspect its code.
FR1.2	As a user, I can search for code using keywords or natural language queries.
FR1.3	As a user, I can view the search results with code snippets and links to the original files in the repository.
FR1.4	As a user, I can filter search results by programming language.
FR1.5	As a user, I can view the context around a code snippet in the search results.
FR1.6	As a user, I can ask code-related questions via chat, and the chat history is preserved.
FR2.1	As a developer, I need the system to fetch and clone Git repositories from provided URLs.
FR2.2	As a developer, I need the system to index the code of the fetched repositories, to generate fast responses.
FR2.3	As a developer, I need the system to use a vector database to store code embeddings for semantic search.
FR2.4	As a developer, I need the system to integrate with an LLM to process natural language queries.
NFR1	The system will index code for targeted repositories within 10 seconds on the specified hardware.
NFR2	The system should achieve a System Usability Scale (SUS) score of 70+ based on at least 5 target users.
NFR3	The system should sanitize user search query inputs to prevent Cross-Site Scripting (XSS) attacks.
NFR4	A 2D visualization tool will display code embeddings to help analyze and improve indexing and search.
IR1	The system should be implemented in Scala, following functional programming principles.
IR2	The system should use Qdrant as the vector database for code embeddings.
IR3	The system should integrate with Ollama for LLM functionalities.
IR4	The system should follow a layered architecture approach, ensuring better modularity.

Table 1. This table summarizes business, functional (user and system), non-functional, and implementation requirements. Full details are in the requirements specification document (link). Acceptance tests are available here, and the traceability matrix is here.

1. **Initial planning (M.1-M.3).** This phase includes setting up the repository baseline with code quality tools (Scalafmt [21], Wartremover [33], Scalafix [4], Trunk [28], conventional commit hooks [6]) to ensure consistent development practices. After setting up this foundation, the project requirements are defined.

2. **Backend development (M.4-M.5).** This step includes setting up the infrastructure for the backend, including the Ollama and the Qdrant modules, and the Langchain4j framework. Design patterns are also defined at this stage.

3. **Web interfaces and evaluation (M.6-M.8).** Web interfaces are created, and the system's performance is evaluated. Optimization techniques include query-based index retrieval and fetching code snippets for specific extensions.

4. **Reporting the results (M.9).** The documentation is generated including: the report, the website, the coverage report and the code documentation (see [32]).

Testing. To ensure continuous feedback and integration, features are incrementally developed by following the TDD approach. The paradigm ensures that code is continuously tested, providing a certain degree of resilience to regression bugs. This approach was used throughout development, however given the heavy reliance on external dependencies, in some specific instances it was necessary to utilize mock objects to isolate unit tests. This was an instructive learning experience which significantly influenced the general design of the system.

Coverage report. I would like to point out that the coverage report does not take into consideration the acceptance tests, and the unit tests were developed chiefly as a tool to guide the development process. As a result, I did not give high priority to reaching a very high coverage percentage, knowing that the actual coverage report would be significantly different if the end-to-end acceptance tests were considered. The impact analysis is presented in Appendix A.1.

Continuous Integration. The CI/CD pipeline is defined using Github Actions. There exist pre-commit hooks that automatically run the formatter, linter, the tests and ensure adherence to the conventional commits specification. Moreover, the Github Actions workflows consist in building and testing the project, generating the coverage report, deploying the website using Github Pages and setting up the release. The release notes are automatically generated from the conventional commits specification using the Semantic Release plugin [18].

3.2. Requirements and Specification

The requirements are summarized in Table 1, and the full requirements document can be found project's website.

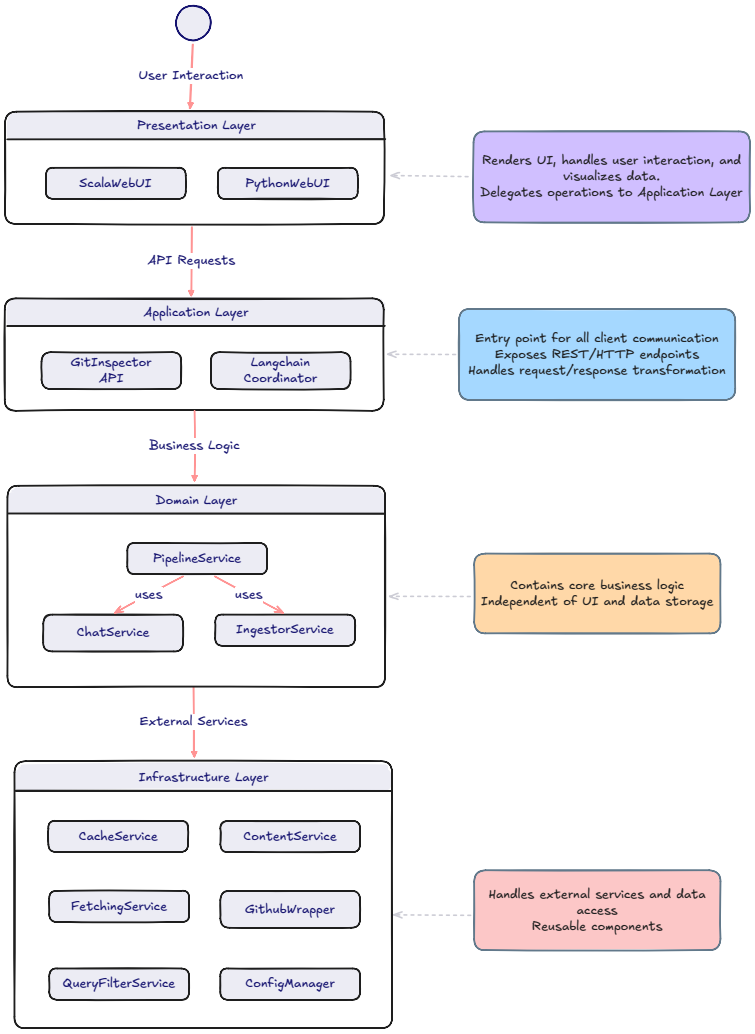


Figure 2. Unidirectional layered architecture. Information flows inwards starting from the presentation layer. The description of each module is available in Appendix A.2.

User stories. There are 5 types of requirements with each one's responsibilities being summarized below:

- **Business Requirements (BR):** define evaluation criteria by which to assess the success of the project.
- **User Functional Requirements (FR.1.x):** describe how the project's result is used by end users, detailing specific interactions with the software.
- **System Functional Requirements (FR.2.x):** describe how the system internally works and operates.
- **Non-Functional Requirements (NFR):** specify the general properties and quality attributes of the system.
- **Implementation Requirements (IR):** anticipate details of architecture, design, implementation, and the development process, acting as constraints.

3.3. Architectural Design

The codebase has two distinct Scala projects. The *backend* offers the RESTful API, and the *frontend* offers the web interface. A description of each module in the frontend is in Appendix A.3. These modules are separate, allowing flexible development, simplifying testing, and enabling each component to be developed independently.

Layered Architecture. The backend is organized around a strict layered architectural model, illustrated in Figure 2. This architectural style has layers that follow an unidirectional dependency flow, ensuring a direct and structured communication between layers. The semantic independence of the lower-level layers (*Domain* and *Infrastructure* layers) from the upper-level layers (*Presentation* and *Appli-*

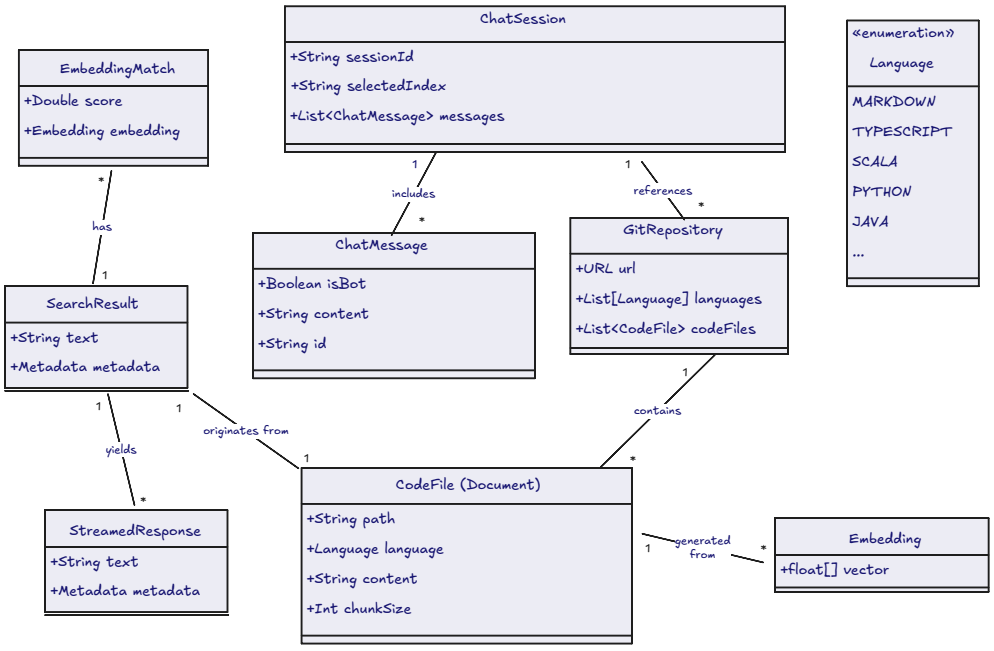


Figure 3. Class diagram illustrating the domain model.

cation layers) promotes reusability and separation of concerns, which ensure a more modular, testable and maintainable code (as suggested by [14]). Each layer is composed of a number of services, with a detailed description of each one's responsibilities being available in Appendix A.2.

ArchUnit Integration. To formally assess the structural dependencies between layers, the *ArchUnit* library is used. This analysis enforces the layered architecture constraints, checks package and class dependencies and provides feedback on any violations that may occur during the development of new features.

Technological Choices. The domain model, which is illustrated in Figure 3, complements the architectural design. Specifically, central domain model entities are the *GitRepository* and the *CodeFile* classes, which are built when fetching data from the Github API using the Uithub service [29].

The *ChatMessage* and *ChatSession* classes are predominantly relevant to the Scala.js framework, and are used to facilitate the communication between the client and the server.

The *Language* enumeration is used by the parser to accurately split the codebase into semantic chunks. The list of supported languages is retrieved from the *Langchain* [8] library. A detailed description of the parsing algorithm is provided in Section 3.4.3.

All remaining classes (*Embedding*, *StreamedResponse*, *SearchResult* and *EmbeddingMatch*) are used together with the Langchain4j framework, and are responsible for facilitating the retrieval process and the generation of the response.

3.4. Detailed Design

The agent's iterative response generation process can be described with a number of key components:

1. **Centralized Control:** All operations present in the library flow through the *PipelineService*, making the process easier to understand. The Pipeline designates specific sub-modules to handle specialized operations. These operations can include chatting instructions, initialization, the creation of an index or the ingestion of new documents.

2. **AIService:** represents the central wrapper that acts as a bridge between the *Langchain4j* framework and user code. It represents the module that allows to build complex queries that enable filtering index stores, managing metadata and chatting with the AI model.

Design patterns. The following design patterns and principles are used throughout the library:

- **Factory Pattern (Creational):** *RAGComponentFactory*, *ChatModelFactoryProvider*, and *IngestionStrat-*

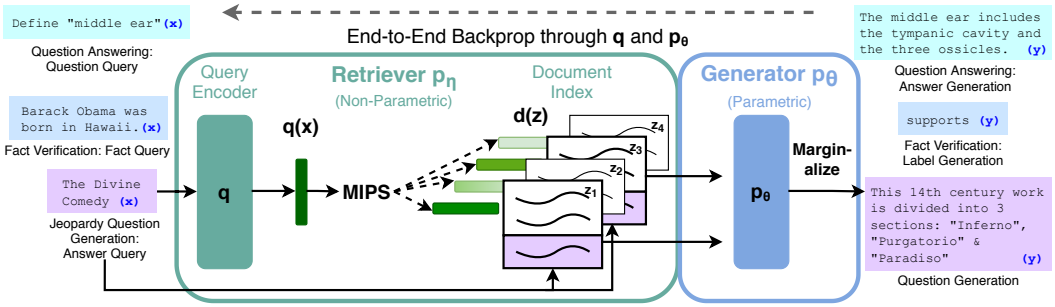


Figure 4. RAG Architecture taken from [13]. Note that the diagram slightly misrepresents the retriever component since the query encoder is actually an embedding instead of a model based on the BERT architecture.

egyFactory encapsulate object creation logic, allowing the system to create complex components without exposing instantiation details. The *ChatModelFactoryProvider* specifically enables configuration-based selection between different AI model implementations (Ollama, Gemini, Claude).

- **Strategy Pattern (Behavioral):** *QueryRoutingStrategy*, *QueryFilterStrategy*, *IngestionStrategy* and *QueryRoutingStrategy* enable algorithm selection at runtime, making the system adaptable to different requirements. For instance, *ConditionalQueryStrategy* and *DefaultQueryStrategy* provide specific strategies for the *QueryRoutingStrategy* interface.
- **Memoization:** The *CacheService* uses memoization to cache the results of expensive operations, such as the generation of an AI assistant with all its dependencies.
- **Dependency Injection:** Services like *CacheService* receive dependencies through constructors rather than creating them directly, promoting loose coupling and testability.
- **Data Access Abstraction:** The project separates higher-level business logic from details about how data is stored and retrieved, particularly for vector embeddings in *QdrantClient* interactions. For instance, *CacheService* separates data concerns from business logic through methods like *createCollection*, *deleteCollection*, *listCollections* and *listDocuments*.

3.4.1 Retrieval Augmented Generation

The following paragraphs provide an introduction to the concepts around the *Retrieval Augmented Generation (RAG)* engine. I start with the nomenclature, then introduce the retrieval component consisting of a vector store together with key details about the embedding models used to combine semantic node information for the subsequent retrieval of information.

Nomenclature. The key concepts are shown in Figure 4 and their significance is described below:

- **Tokens.** The original text is divided into substrings called tokens. The technique for dividing words into subparts generally depends on the language being used[2].
- **Retriever p_η.** Produces as output the embedded documents. This component has no learnable parameters hence is called non-parametric.
- **Query Encoder q.** In the original paper, q is defined as a BERT-based query encoder (a specific type of neural network architecture) that produces a query representation $q(x)$. Due to efficiency and simplicity concerns, instead of an encoder, our approach uses a simple embedding model. The embedding model is designed to convert text into a dense vector representation, which sits behind the Ollama API.
- **Document Index.** The index represents a vector knowledge base, from which data can be retrieved according to a similarity function such as the *Cosine Similarity*. In the figure, a document is denoted as z and its representation is $d(z)$.
- **The Generator p_θ.** Represents a pre-trained sequence-to-sequence transformer. It has learnable weights hence is called parametric. Any open large language model can be used, running them locally behind the Ollama backend.

3.4.2 The Retrieval Component

One vector store per data type. For organizational purposes, I decided to use one vector store for textual information and one for code embeddings. This decision allows to store the features representing the two information types separately, and query them independently. Because of this, it becomes impossible to mix-up entries that belong to the two different data modalities.

[2]For the English language syllables are usually a good choice.

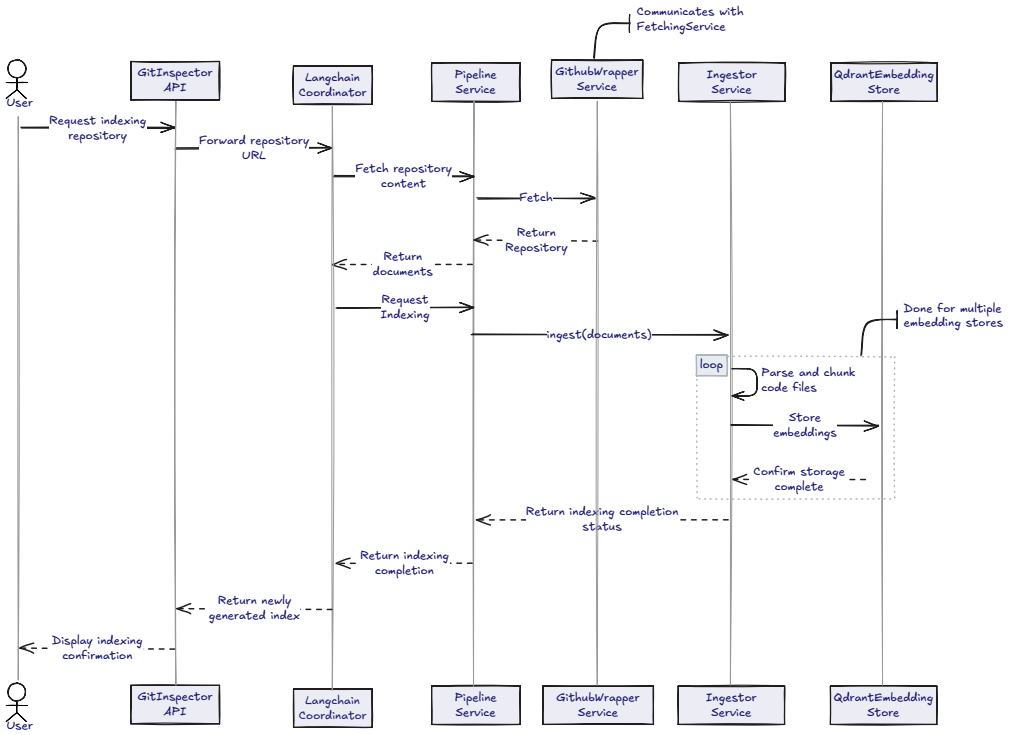


Figure 5. The process by which a vector store index is generated.

Conditional generation. Due to this, it becomes necessary to dynamically choose whether to query a single vector store at a time, or whether to query them independently then fuse the results. Unless this conditional filtering of the two indexes is handled correctly, the modal risks being misled to analyze too much non-relevant information.

The *QueryRoutingStrategy*[3] class is responsible for this mechanism. It intercepts the user query and appropriately decides whether to use a single collection, or to query both, then combine the results.

Code index with multiple extensions. One issue about storing all code embeddings into a single vector store is that it may become difficult to pinpoint information when multiple extension types are present. The solution involves dynamically excluding file types that correspond to specific extensions. This is achieved through the *QueryFilterService*[4] module, which decides by inspecting the query whether to focus the retrieval on specific extension types.

Repository Ingestion. Upon startup, the application initializes by generating a unique *AIService* instance for each index in the vector store, as illustrated in Figure 9. After the initialization is complete, data is ingested to populate the indices with embeddings, as illustrated in Figure 5. The operation involves creating nodes based on their semantic representation, embedding each node into its numerical features and storing the results into into its corresponding index (or collection) behind the Qdrant vector database.

Below is a formal description of the process:

- **Github Repositories:** We have a number of repositories, each with a number of files, labeled as $r_1, r_2, \ldots, r_n$.
- **File Segmentation:** For each repository r_i we extract the code files. Let's call the k-th file of repository i as $f_{i,k}$.
- **Semantic Chunking:** For each file $f_{i,k}$, we extract the semantic chunks. Let's call the these chunks $c_{i,k,1}, c_{i,k,2}, \ldots, c_{i,k,m}$.
  At the time of writing, the Langchain4j library does not have a built-in code splitter, so I wrote the *RecursiveCharacterTextSplitter*[5] algorithm from the Python Langchain library [9] in Scala.
  We then use the embedding function f (as described

[3]The strategy is available here.
[4]The service is available here.
[5]The splitter is available here.

in [2]) to convert each chunk into a 768-dimensional vector: $e_{i,k,l} = f(c_{i,k,l})$. Here, $e_{i,k,l}$ is the embedding of the l-th chunk from the k-th file of the i-th repository.

- **Vector Database:** The numerical features are stored in the Qdrant vector database D, using [16].
- **User Query:** When a user submits a query q, we:
  1. Convert the query to an embedding using the same function: $e_q = f(q)$.
  2. For each retriever, use Cosine similarity to find top 50 closest matches in the vector database D.
  3. As described in [3], use a reranker to find the top 3 most relevant chunks from the top 100 matches (50 matches for each vector store).

Model Name	Use case	Dim.	Size	Score Function
jina-embeddings-v2-base-code [2]	querying	768	161M	Cosine Similarity [22]
jina-reranker-v2-base-multilingual [3]	reranker	N/A	278M	Relevance Score [17]

Table 2. Properties of the embedding models.

3.4.3 Parsing Algorithm

The parsing algorithm was implemented in Scala by inspecting the source code of the Python Langchain library [8]. Its pseudocode is shown in Algorithm 1 and described below:

1. Chunk Selection: It iterates through the provided separators list (which is ordered by priority). It finds the first separator in the list that actually exists within the current text chunk being processed. This becomes the separator for this level of splitting.

2. Initial Split: The text is split into preliminary splits using the chosen separator.

3. Chunk Processing: The *splits* are processed in order.

For small chunks, if the *split* is shorter than the limit (*chunkSize*), it is added to the *goodSplits* buffer.

For large chunks, if the *split* is longer than the limit (*chunkSize*), any accumulated chunks in the buffer are merged and added to the *finalChunks* list. The algorithm then checks if any separators are still pending to be processed, and recursively repeats the process until all separators are consumed.

4. Final Merge: After iterating through all initial splits, if there are any chunks left in the goodSplits buffer, they are merged using mergeSplits and added to finalChunks.

Algorithm 1 Recursive Text Splitting Algorithm [8].

```
function _SPLIT_TEXT(text, separators)
    finalChunks ← []
    goodSplits ← []

    // 1. Choose highest-priority separator
    // Loop over separators in-order, sekect first text correspondance
    sep ← selectSeparator(text, separators)
    remaining ← separators after sep

    // 2. Apply the chosen separator
    splits ← splitTextWithRegex(text, sep)

    // 3. Process each split
    for chunk ∈ splits do
        if len(chunk) < chunkSize then
            // Chunk is small enough, add to buffer for later merging
            goodSplits.add(chunk)
        else
            // a. Merge and add any previously buffered small chunks
            if goodSplits.nonEmpty then
                finalChunks ← finalChunks + mergeSplits(goodSplits)
                goodSplits.clear()
            end if
            // b. Process the large chunk. Recursive base case.
            if remaining.isEmpty then
                // Cannot split further, add the large chunk as is.
                finalChunks.add(chunk)
            else
                // Recursively split with remaining separators
                recursiveChunks ← _SPLIT_TEXT(chunk, remaining)
                finalChunks ← finalChunks + recursiveChunks
            end if
        end if
    end for

    // 4. Process any remaining small chunks in the buffer
    finalChunks ← finalChunks + mergeSplits(goodSplits)
    return finalChunks
end function
```

3.4.4 Chatbot Interactions

Embedding Models. The interaction between the user and the chatbot is illustrated in Figure 6. There exist two distinct phases: the semantic search for relevant information and the reranking of the retrieved chunks to produce the final answer. Table 2 shows the properties of each embedding model used in each phase.

Vector Store Generation. Phase 1 involves using the *jina-embeddings-v2-base-code* [2] embedding model to generate the vector store and to embed the user query. Based on the *BERT* architecture, the model was trained on code-related tasks with a significant amount of natural language included, making it suitable for both code and text retrieval tasks.

The model maps code snippets and docstrings to a 768 dimensional dense vector space and it is designed to excel at a particular type of search called symmetric bidirec-

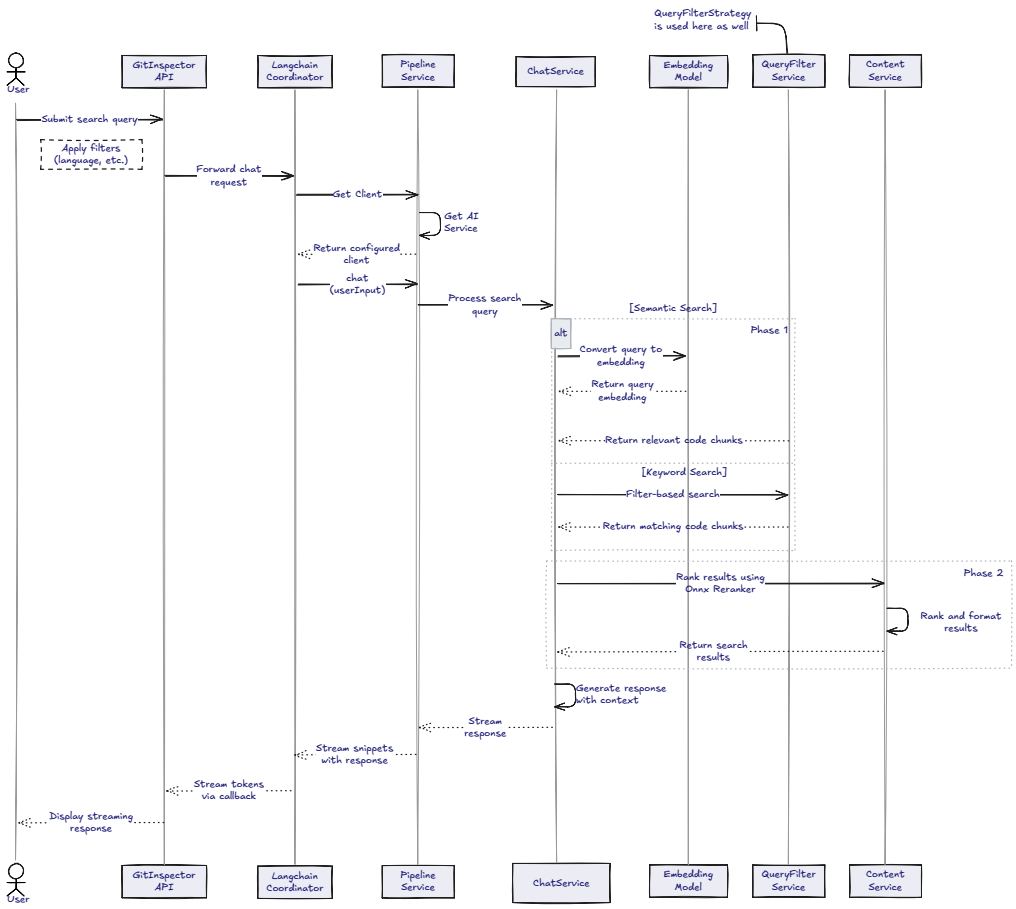


Figure 6. The key steps that enable the chatbot to answer questions about the codebase.

tional semantic search [24]. This constraints the usage of the same embedding model for both index generation and query embedding.

Reranking. Phase 2 involves using a reranker model such as *jina-reranker-v2-base-multilingual* [3] to reorganize the results produced in phase 1. This transformer-based model is trained to filter a large corpus of chunks based on their relevance to the user query. Both phases have hyperparameters that fine-tune the retrieval process, present in the configuration file. By default each index retrieves 50 chunks and the reranker selects the 3 most relevant items from the results.

Chatbot Interactions. Table 3 shows the chatbot's behavior for a number of specific queries. The responses include the file names, relevant code snippets the model builds its answer on, and the final response with the answer.

4. Analysis and Evaluation

Experiments are carried out to assess the effectiveness of the embedding model as well as the parsing algorithm.

Experimental setting. The experiments include 3 different codebases:

- **OpenAI-Scala-Client [5]**: A functional library alternative to Langchain4j, developed in Scala. The library contains a substantial amount of functional code.
- **Scipy [23]**: A scientific computing library that provides algorithms for mathematical operations including optimization, linear algebra, integration, and more.
- **Scala 3 [19]**: The Scala 3 compiler library, representing a complex codebase with a large number of files and adherence to functional programming principles.

These repositories were selected to provide a diverse set of codebases. While the Scipy codebase is notably different

Experiments using Qwen-2.5 Coder 32B

OpenAI Scala Client Functionalities

What functionalities does the openai-scala-client offer?

Source 1. File name: /service/OpenAIService.scala

/**
* Central service to access all public OpenAI WS endpoints [...].
* The following services are supported:
* - **Models**: listModels, and retrieveModel
* - **Completions**: createCompletion
* - **Chat Completions**: createChatCompletion, [...] createChatToolCompletion
* - **Edits**: createEdit (deprecated)
* - **Images**: createImage, createImageEdit, createImageVariation
* - **Embeddings**: createEmbeddings
* - **Batches**: createBatch, retrieveBatch, cancelBatch, and listBatches
* - **Audio**: createAudioTranscription, createAudioTranslation
* - **Files**: listFiles, uploadFile, deleteFile, retrieveFile, and retrieveFileContent
* - **Fine-tunes**: createFineTune, listFineTunes, [...], and deleteFineTuneModel
* - **Moderations**: createModeration
* - **Threads**: createThread, retrieveThread, modifyThread, and deleteThread
* - **Thread Messages**: createThreadMessage, retrieveThreadMessage, [...]
* - **Runs**: createRun, etc.
* - **Run Steps**: listRunSteps, etc.

Answer. The `openai-scala-client` offers a comprehensive set of functionalities [...]. Here's a breakdown:
1. **Model Operations**: List models [...], Retrieve details [...].
2. **Completions**: Generate text completions [...].
3. **Chat Completions**: Create chat completions [...]. With optional tool calls.
4. **Edits**: Create edits to text (deprecated).
5. **Images**: Generate images, Edit or vary existing images [...].
6. **Embeddings**: Create embeddings for texts [...].
8. **Audio**: Transcribe audio, Translate languages, Generate speech [...].
10. **Fine-tuning**: Create fine-tuned models, Manage the lifecycle of fine-tunes [...].
11. **Moderations**: Check text against content policy.
12. **Threads & Messages** (Beta): Create, retrieve, modify, delete threads.
13. **Runs**: Manage runs [...]. List steps [...].

The openai-scala-client provides a comprehensive Scala interface to OpenAI's API, supporting models, completions, chat, images, embeddings, audio, fine-tuning, moderation, threads, and run operations.

The Scala 3 Language

What are some key features of Scala 3?

Source 1. File name: /docs/_docs/pattern-matching.md

The implementation of pattern matching in Scala 3 was greatly simplified [...]. From a user perspective, this means that Scala 3 generated patterns are a lot easier to debug.

Source 2. File name: /scala2-library-cc/src/scala/language.scala

/**
* The language features are:
* - **Dynamics** [...] enables defining calls rewriting using the `Dynamic` trait
* - **Existentials** [...] enables writing existential types
* - **Implicit Conversions** [...] enables defining implicit methods and members
* - **Reflective Calls** [...] enables using structural types

Source 3. File name: /docs/_docs/reference/overview.md

To enable porting most uses of macros, we are experimenting with the advanced language constructs listed below. [...]
- **Match Types** [...] allow computation on types.
- **Inline** [...] provides a straightforward implementation of some simple macros [...].
- **Quotes and Splices** [...] provide a natural way to express macros and staging [...].
- **Type class derivation** [...] creates an in-language implementation of the `Gen` macro.
- **By-name parameters** [...] a more in-language implementation of the `Lazy` macro.

Answer. Scala 3 introduces several key features:
1. **Option-less Pattern Matching**: Scala 3 simplifies pattern matching, making it easier to debug by [...] ensuring all variables are visible in debug modes.
2. **Extractor Support**: Scala 3 supports a superset of Scala 2's extractor patterns.
3. **Language Feature Imports**: The scala.language object allows for selective import of language features [...], providing fine-grained control over feature selection.
4. **Advanced Language Constructs for Metaprogramming**:
- Match Types: Allow computation on types [...].
- Inline: Provides a straightforward way to implement macros [...].
- Quotes and Splices: Offer a unified set of abstractions to express staging [...].
- Type Class Derivation: Implements foundational libraries [...].
- By-name Context Parameters: An improved implementation of `Lazy` macro [...].

Scala 3 introduces significant language improvements including simplified pattern matching with better debugging, enhanced extractor support, selective language feature imports, and advanced metaprogramming constructs.

Table 3. **The sampled texts are describing the OpenAI Scala Client functionalities and Scala 3 language features.**. The myblue text represents the question asked by the user. The green background represents the documentation file from which information is retrieved. The pink background highlights specific details extracted from the source. The blue background is the synthesized final response.

in its focus and target audience, addressing more specialized operations, the other two repositories being written in Scala, share more similarities in terms of content.

4.1. Code Embedding Visualization

Qualitative analysis. The following paragraphs analyze Figures 7 and 8. For each modality (text and code), the diagrams show the data distributions through pairwise cosine similarity heatmaps and 2-dimensional projections using the *t-SNE* [31] and *PCA* [26] algorithms. The heatmaps display a similarity matrix of embedding pairs, with darker green indicating higher similarity. The 2-dimensional projections, generated by both *t-SNE* and *PCA*, reveal cluster formations in the embedding space, where each point is colored according to its originating codebase. The experiments are conducted using the *jina-embeddings-v2-base-code* [2] embedding model.

Code embeddings interpretation. Below is a description of diagram 7.

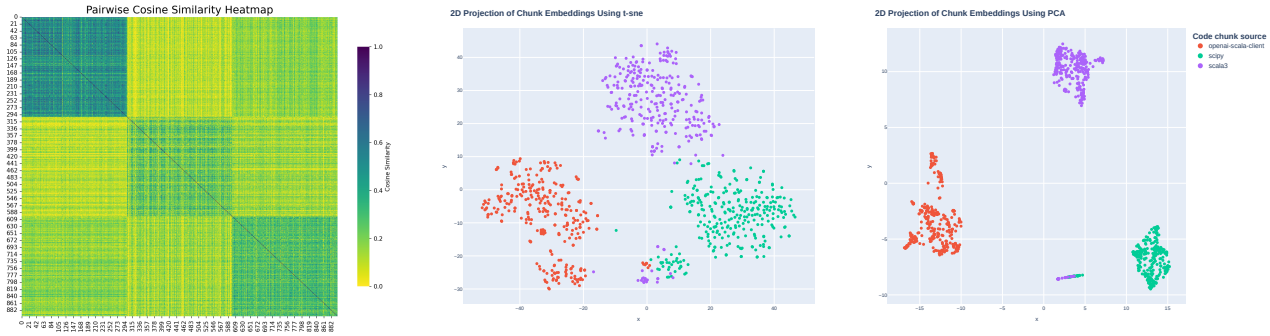
Similarity heatmap. The total number of embeddings in Figure 7a is 900, generated using stratified sampling [25]. It displays areas characterized by darker and lighter shades of green. The dark colors signify that there is a strong cosine similarity between the two fixed embeddings, while lighter colors suggest dissimilarity.

The diagram shows the formation of 3 clusters, which intuitively represent the 3 codebases. Specifically, the clusters are arranged the following way: *OpenAI-Scala-Client* (top-left), *Scipy* (middle), and *Scala 3* (bottom-right). The color bar on the right indicates the cosine similarity values. Yellow represents low similarity (close to 0) and darker colors (ranging through green to dark blue/purple) represent high similarity (close to 1).

The Scala 3 cluster suggests similarities with the OpenAI-Scala-Client codebase. This can be attributed to the fact that both codebases are written in Scala. The center cluster contains code written in Python which is characterized by lighter shades of green, suggesting dissimilarity with the other two repositories.

2-Dimensional projections. The projections from Figure 7b illustrate the distribution of the 768-dimensional data in a 2-dimensional space using dimensionality reduction algorithms.

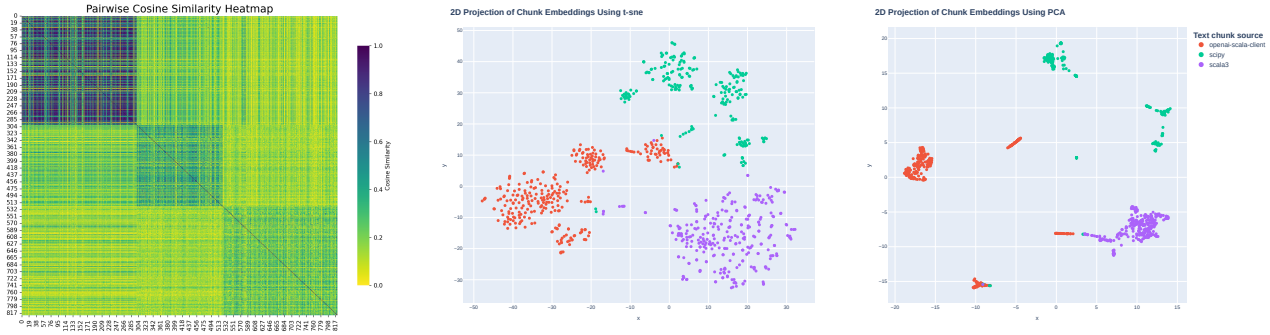
The PCA dimensionality reduction technique seeks to maximize variance and to preserve large pairwise distances.



(a) Cosine similarity heatmap

(b) 2-dimensional projections using t-SNE and PCA

Figure 7. Visualization of the code embeddings. OpenAI-Scala-Client (top-left), Scipy (middle), and Scala 3 (bottom-right).



(a) Cosine similarity heatmap

(b) 2-dimensional projections using t-SNE and PCA

Figure 8. Visualization of the text embeddings. OpenAI-Scala-Client (top-left), Scipy (middle), and Scala 3 (bottom-right).

As a result, points that are different from each other are kept far apart. The diagrams show the creation of well defined clusters, with a small number of outliers. This suggests that the embedding model is efficient at creating representations that are semantically meaningful.

The t-SNE algorithm differs from PCA by preserving only small pairwise distances or local similarities. This gives an intuition on how the data is arranged in a higher-dimensional space. Outliers are not visibly present, suggesting also in this case the effectiveness of the embedding model.

It is important to note that an index may contain files from multiple programming languages, however the dynamic filter strategy discussed in Section 3.4.2 allows to search for specific file extensions within the codebase which ensures robustness against outliers and helps pinpoint specific code more accurately.

Text embeddings interpretation. Below is a description of diagram 8.

Similarity heatmap. Figure 8a generally shows high intra-repository similarity with darker blocks along the diagonal. In contrast to Figure 7a, there are similarities detected between the Scipy and OpenAI-Scala-Client embeddings. These can be attributed to the markdown content having similar tutorial-style content on their respective usage. The model is able to discover relationships in the way the information is presented.

2-Dimensional projections. Figure 8b illustrates the creation of distinct clusters. There do exist more outliers than in Figure 7b, however the potential problems are minimized due to the separation of the index into two components, one for text and one for code. This decision ensures that the two modalities are kept separate, minimizing the potential mixture of data points belonging to different data types.

Usability Questionnaire. The System Usability Scale (SUS) questionnaire, shown in Appendix A.5, provides a quantitative assessment of the system. The survey complements the requirements document where it is heavily referenced. This numerical evaluation of the web interface gives developers valuable feedback to make focused improvements.

In this study, five participants were asked a total of 10 questions regarding the web interface, ranging from topics such as ease of use to overall satisfaction. The questionnaire reached a score of 85%, where the average score is

considered to be 68% [30]. However, it would be necessary to collect a larger number of responses to achieve a more accurate and representative assessment.

5. Retrospective

Future Work. Below are some potential ideas for future exploration and improvements:

- **External tools [11].** The library could be extended to provide access to a wide range of high quality external databases such as arXiv and Wikipedia.
- **MCP servers [10].** The MCP protocol provides interoperability between independent systems, ensuring minimal coupling between components[6]. This functionality would enable a wide range of applications.
- **Multi-agent workflows [34].** Allow users to interact with multiple agents through interactive conversations, where each agent takes on a specific role.

Closing remarks. I am particularly fond of the following aspects of the project:

- **Layered architecture.** The development of a strict layered architecture to organize the codebase, including the formal assessment of its correctness through automated tools (ArchUnit).
- **Unit testing.** Writing comprehensive unit tests was difficult, particularly due to the significant amount of external dependencies.
- **Functional programming.** By adhering to functional programming principles, the codebase is surprisingly easier to understand, however the learning curve is quite steep at the beginning.
- **Design patterns.** The implementation of various design patterns to ensure the codebase is easy to extend and modify.
- **Github workflows.** Due to their highly transferable nature, they become virtually applicable to any software engineering project.

By adhering to these principles, it was fairly simple at a later stage of the project to extend the library to support external providers such as Gemini or Anthropic. If I were to describe the single most important topic that I learnt from this project, is the ability to reason more easily about non trivial codebases by utilizing practices that are transferable to other topics and programming languages.

[6]The protocol provides access to external knowledge bases through well defined REST APIs

References

[1] Abubakar Abid, Ali Abdalla, Ali Abid, Dawood Khan, Abdulrahman Alfozan, and James Zou. Gradio: Hassle-free sharing and testing of ml models in the wild. *arXiv preprint arXiv:1906.02569*, 2019. URL: https://arxiv.org/abs/1906.02569. 2

[2] Jina AI. jinaai/jina-embeddings-v2-base-code · Hugging Face, January 2024. URL: https://huggingface.co/jinaai/jina-embeddings-v2-base-code. 8, 10

[3] Jina AI. jinaai/jina-reranker-v2-base-multilingual · Hugging Face, September 2024. URL: https://huggingface.co/jinaai/jina-reranker-v2-base-multilingual. 8, 9

[4] Scala Center. scalacenter/scalafix, April 2025. original-date: 2016-08-08T11:45:19Z. URL: https://github.com/scalacenter/scalafix. 3

[5] Cequence. cequence-io/openai-scala-client, April 2023. original-date: 2023-01-25T10:09:04Z. URL: https://github.com/cequence-io/openai-scala-client. 2, 9

[6] Conventional Commits. Conventional Commits. URL: https://www.conventionalcommits.org/en/v1.0.0/. 3

[7] Llama Index. GitHub - run-llama/llama_index: LlamaIndex is a data framework for your LLM applications — github.com. https://github.com/run-llama/llama_index. 2

[8] LangChain-AI. Langchain parsing algorithm. https://github.com/langchain-ai/langchain/blob/3072e4610aca919f7bda2ef26f54e60e43b9c617/libs/text-splitters/langchain_text_splitters/character.py#L77. 5, 8

[9] LangChain-AI. Build context-aware reasoning applications. https://github.com/langchain-ai/langchain, 2023. 2, 7

[10] LangChain4j. Model Context Protocol (MCP) | LangChain4j. URL: https://langchain4j.github.io/tutorials/mcp. 12

[11] LangChain4j. Tools (Function Calling) | LangChain4j. URL: https://langchain4j.github.io/tutorials/tools. 12

[12] LangChain4j. Supercharge your java application with the power of llms. https://github.com/langchain4j/langchain4j, 2023. 2

[13] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks, 2021. URL: https://arxiv.org/abs/2005.11401, arXiv:2005.11401. 6

[14] Diego Marmsoler, Alexander Malkis, and Jonas Eckhardt. A model of layered architectures. *Electronic Proceedings in Theoretical Computer Science*, 178:47–61, March 2015. URL: http://dx.doi.org/10.4204/EPTCS.178.5, doi:10.4204/eptcs.178.5. 5

[15] Ollama. Get up and running with llama 3.3, deepseek-r1, phi-4, gemma 3, mistral small 3.1 and other large language models. https://github.com/ollama/ollama, 2023. 2

[16] Qdrant. Official java client for qdrant. https://github.com/qdrant/java-client, 2023. 2, 8

[17] Revanth Gangi Reddy, Pradeep Dasigi, Md Arafat Sultan, Arman Cohan, Avirup Sil, Heng Ji, and Hannaneh Hajishirzi. Refit: Relevance feedback from a reranker during inference, 2024. URL: https://arxiv.org/abs/2305.11744, arXiv:2305.11744. 8

[18] Semantic Release. semantic-release/semantic-release, April 2025. original-date: 2014-09-15T16:18:28Z. URL: https://github.com/semantic-release/semantic-release. 3

[19] Scala. scala/scala3, April 2025. original-date: 2012-12-06T12:57:33Z. URL: https://github.com/scala/scala3. 9

[20] Scala-js. Harness the scala and javascript ecosystems together. develop robust apps for browsers, node.js, and serverless. https://www.scala-js.org/, 2020. 2

[21] scalameta. scalameta/scalafmt, April 2025. original-date: 2015-11-02T21:34:40Z. URL: https://github.com/scalameta/scalafmt. 3

[22] ScienceDirect. Cosine Similarity - an overview | ScienceDirect Topics. URL: https://www.sciencedirect.com/topics/computer-science/cosine-similarity. 8

[23] SciPy. scipy/scipy, April 2025. original-date: 2011-03-09T18:52:03Z. URL: https://github.com/scipy/scipy. 9

[24] SentenceTransformers. Semantic Search 2014; Sentence Transformers documentation — sbert.net. https://www.sbert.net/examples/applications/semantic-search/README.html. 9

[25] Ravindra Singh and Naurang Singh Mangat. Stratified Sampling. In Ravindra Singh and Naurang Singh Mangat, editors, *Elements of Survey Sampling*, pages 102–144. Springer Netherlands, Dordrecht, 1996. doi:10.1007/978-94-017-1404-4_5. 10

[26] Lindsay I Smith. A tutorial on Principal Components Analysis. *Journal of Machine Learning Research*, 2002. 10

[27] Tree-sitter. Introduction - Tree-sitter. URL: https://tree-sitter.github.io/tree-sitter/. 2

[28] Trunk. Overview — docs — docs.trunk.io. https://docs.trunk.io/code-quality/code-quality. [Accessed 29-04-2025]. 3

[29] Uithub. Easily Ask Your LLM Coding Questions. URL: https://uithub.com. 5

[30] UserFocus. Measuring Usability With The System Usability Scale (SUS). URL: https://www.userfocus.co.uk/articles/measuring-usability-with-the-SUS.html. 12

[31] Laurens van der Maaten and Geoffrey E. Hinton. Visualizing data using t-sne. *Journal of Machine Learning Research*, 9:2579–2605, 2008. URL: https://api.semanticscholar.org/CorpusID:5855042. 10

[32] Razvan Florian Vasile. Git inspector! inspecting github repositories with open-source llm. 2025. 3

[33] wartremover. wartremover/wartremover, April 2025. original-date: 2013-01-05T11:29:26Z. URL: https://github.com/wartremover/wartremover. 3

[34] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W White, Doug Burger, and Chi Wang. Autogen: Enabling next-gen llm applications via multi-agent conversation framework. In *AutoGen*, 2023. arXiv:2308.08155. 12

A. Appendix

A.1. Impact Analysis

As the project evolved, it became increasingly more complex to manage wide-ranging modification effects and to ensure consistency with the existing tests. As a result, the following methodology was adopted for impact analysis:

- **Pre-commit Hooks:** During the development process, each commit would automatically trigger the unit tests suite, ensuring that any breaking changes were immediately detected.
- **Continuous Integration (CI):** To speed up development, the pre-commit hook could be skipped, with the CI/CD pipeline automatically running tests for quick regression feedback.
- **Mocking and Isolation:** By using dependency injection and mocks, units of code were isolated, making it easier to pinpoint the impact of changes and avoid cascading failures in unrelated tests.
- **Selective Test Execution:** For tests that rely on external dependencies, I used test tagging and selective execution to ensure these are not included in the CI/CD pipeline.
- **Coverage Reports in Pull Requests:** Coverage differences were reported in each pull request, making it clear if a change increased, decreased, or maintained coverage.

A.2. Architecture Modules

The following describes each module's responsibilities present in Figure 2.

- **Presentation Layer:**
  - *ScalaWebUI:* Renders the user interface, handles user interactions, and visualizes data for the Scala frontend.
  - *PythonWebUI:* Provides an alternative user interface implementation using Python/Gradio for rapid prototyping.
- **Application Layer:**
  - *GitInspector API:* Entry point for client communication, exposes REST/HTTP endpoints.
  - *LangchainCoordinator:* Manages the workflow between different services, manages request/response operations.
- **Domain Layer:**
  - *PipelineService:* All interactions go through this service. Represents the core RAG pipeline.
  - *ChatService:* Handles chat interactions between users and the LLM.
  - *IngestorService:* Manages the processing and indexing of repository content for retrieval.
- **Infrastructure Layer:**
  - *CacheService:* Provides caching functionality to improve performance across operations.
  - *ContentService:* Transforms the content into a format to be displayed in the UI.
  - *FetchingService:* Handles external data fetching operations.
  - *GithubWrapperService:* Interacts with GitHub repositories to extract code.
  - *QueryFilterService:* Filters and processes queries for more accurate results.
  - *ConfigManager:* Manages application configuration settings.

The library follows a strict layered architecture. This ensures that each component has a well defined role with minimal dependencies on other components within the same layer and no dependencies on components in higher layers.

A.3. Components of the Web Interface

The entry point of the Scala frontend is the *Main.scala* file, which initializes all components and services. All dependencies are injected through the constructor, and the responsibilities of each entity are described below:

- **Components:**
  - *LinkViewer:* Component for viewing and indexing content from GitHub repositories
  - *ChatInterface:* Component for chatting with the LLM about indexed repositories, handling message display and submission.
  - *IndexSelector:* Component for choosing which repository index to query, with options to refresh or remove indices.
  - *StatusBar:* Simple component displaying status messages to users.
  - *TabContainer:* Component for switching between the Chat and Link Viewer tabs.
- **Services:**
  - *ContentService:* Service that communicates with the backend API to fetch content, generate indices, and handle chat interactions.
  - *HttpClient:* Low-level service handling HTTP requests and Server-Sent Events for streaming chat responses.
- **Models:**
  - *Models.scala:* Contains data models used throughout the application like ‘ChatMessage‘, ‘IndexOption‘, and various request/response models.
- **Utilities:**
  - *IDGenerator:* Utility for generating unique IDs for chat messages and other elements.
  - *Main.scala:* Entry point that initializes the application, sets up event listeners, and creates the UI components.
- **Python Interface:**
  - *main.py:* Implements an alternative frontend using Gradio with functionality for chatting with repositories, viewing content, and managing indices.
  - *style.css:* Provides styling for the Gradio interface.

A.4. Initialization of the pipeline

The following diagram illustrates the sequence diagram of the initialization of the application. This ensures that the AIServices are properly initialized to handle incoming requests. The process is summarized below:

1. The process loads necessary dependencies into the *CacheService*.
2. CacheService is used to list all available indices from the vector database.
3. Each index is used to allocate an AIService instance, which handles communication between user and model.
4. The default AIService produced at the end facilitates chat interactions when no specific index is selected.

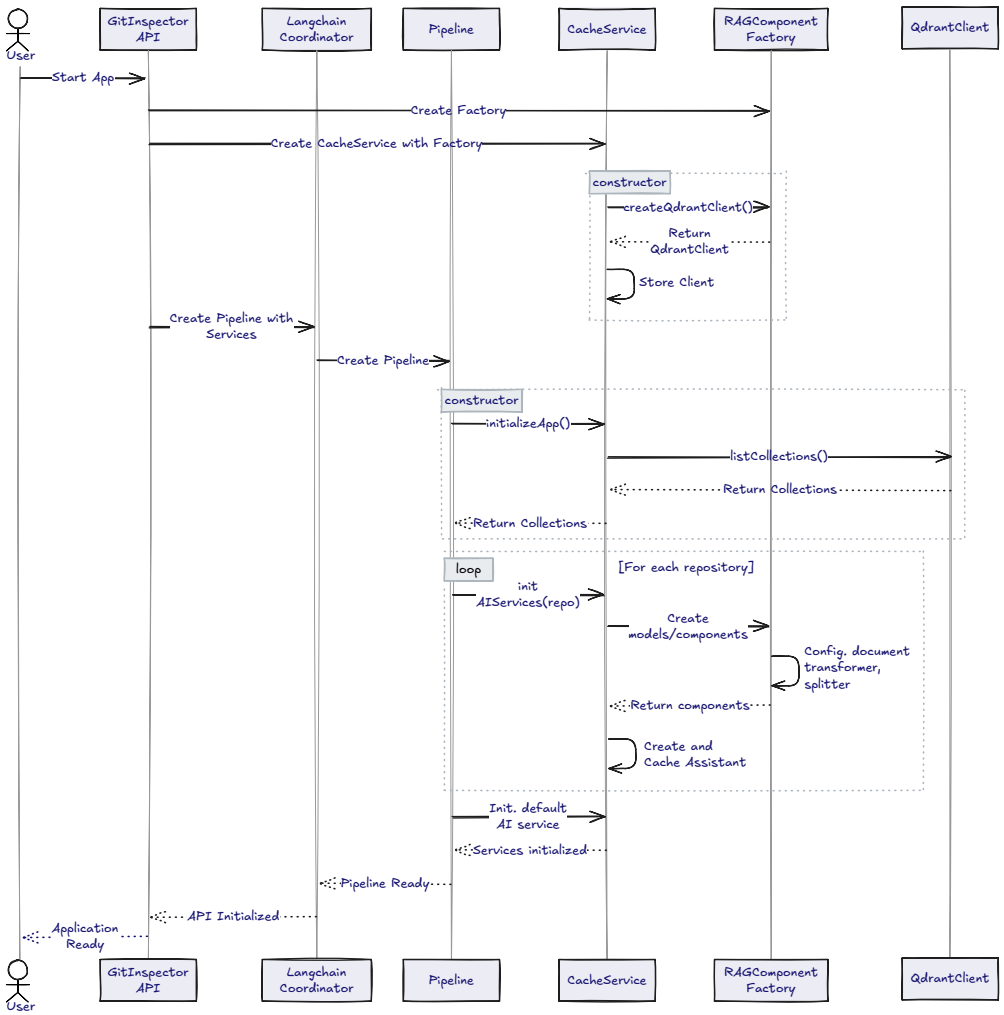


Figure 9. The sequence diagram of the initialization of the pipeline.

A.5. System Usability Scale (SUS) Results

The SUS consists of 10 questions, providing users with a scale of 1 to 5 to rate their experience with the system. The assessment was conducted with 5 participants, yielding an average score of 85% (calculation details). The questionnaire is available here.

	Forte Disaccordo	Disaccordo	Neutro	D'accordo	Fortemente D'accordo	ø	
1. Penso che mi piacerebbe usare questo sistema frequentemente.	0.00% 0	0.00% 0	0.00% 0	0.00% 0	100.00% 5	**5.00**	5
2. Ho trovato il sistema inutilmente complesso.	80.00% 4	20.00% 1	0.00% 0	0.00% 0	0.00% 0	**1.20**	5
3. Ho pensato che il sistema fosse facile da usare.	0.00% 0	0.00% 0	0.00% 0	60.00% 3	40.00% 2	**4.40**	5
4. Penso che avrei bisogno del supporto di un tecnico per poter usare questo sistema.	40.00% 2	0.00% 0	40.00% 2	20.00% 1	0.00% 0	**2.40**	5
5. Ho trovato che le varie funzioni di questo sistema fossero ben integrate.	0.00% 0	0.00% 0	0.00% 0	0.00% 0	100.00% 5	**5.00**	5
6. Ho pensato che ci fosse troppa incoerenza in questo sistema.	60.00% 3	40.00% 2	0.00% 0	0.00% 0	0.00% 0	**1.40**	5
7. Immagino che la maggior parte delle persone imparerebbe a usare questo sistema molto rapidamente.	0.00% 0	0.00% 0	20.00% 1	80.00% 4	0.00% 0	**3.80**	5
8. Ho trovato il sistema molto macchinoso da usare.	80.00% 4	20.00% 1	0.00% 0	0.00% 0	0.00% 0	**1.20**	5
9. Mi sono sentito molto sicuro nell'usare il sistema.	0.00% 0	0.00% 0	40.00% 2	0.00% 0	60.00% 3	**4.20**	5
10. Ho avuto bisogno di imparare molte cose prima di poter iniziare a usare questo sistema.	20.00% 1	60.00% 3	0.00% 0	20.00% 1	0.00% 0	**2.20**	5

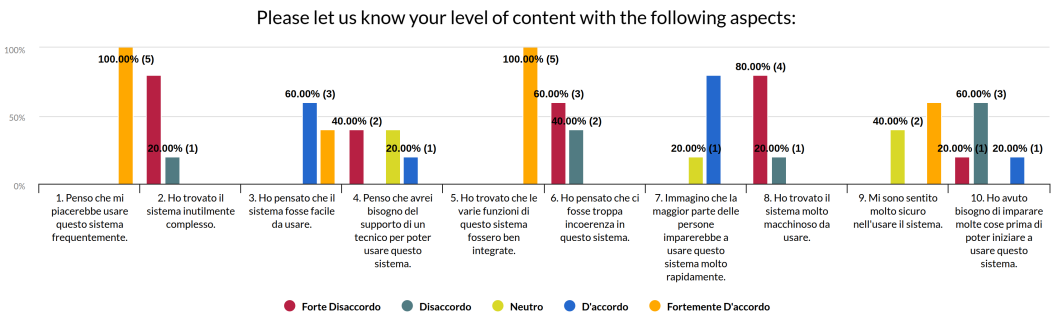


Figure 10. The diagrams illustrate the individual questionnaire responses and a vote distribution bar chart.