*I.J. Modern Education and Computer Science*, **2025, 5, 109-145**
Published Online on October 8, 2025 by MECS Press (http://www.mecs-press.org/)
DOI: 10.5815/ijmecs.2025.05.07

# Local Agentic RAG-Based Information System Development for Intelligent Analysis of GitHub Code Repositories in Computer Science Education

**Zhengbing Hu**
School of Computer Science, Hubei University of Technology, Wuhan, China
E-mail: drzbhu@gmail.com
ORCID iD: https://orcid.org/0000-0002-6140-3351

**Markiian-Mykhailo Paprotskyi**
Department of Information Systems and Networks, Lviv Polytechnic National University, Lviv, 79013, Ukraine
E-mail: markiian-mykhailo.paprotskyi.sa.2021@lpnu.ua
ORCID iD: https://orcid.org/0009-0002-5292-8559

**Victoria Vysotska**
Department of Information Systems and Networks, Lviv Polytechnic National University, Lviv, 79013, Ukraine
Osnabrück University, Osnabrück, 49076, Germany
E-mail: Victoria.A.Vysotska@lpnu.ua, victoria.vysotska@uni-osnabrueck.de
ORCID iD: https://orcid.org/0000-0001-6417-3689

**Lyubomyr Chyrun**
Ivan Franko National University of Lviv, Lviv, 79000, Ukraine
E-mail: Lyubomyr.Chyrun@lnu.edu.ua
ORCID iD: https://orcid.org/0000-0002-9448-1751

**Yuriy Ushenko***
Department of Computer Science, Yuriy Fedkovych Chernivtsi National University, Chernivtsi, 58012, Ukraine
E-mail: y.ushenko@chnu.edu.ua
ORCID iD: https://orcid.org/0000-0003-1767-1882
*Corresponding Author

**Dmytro Uhryn**
Department of Computer Science, Yuriy Fedkovych Chernivtsi National University, Chernivtsi, 58012, Ukraine
E-mail: d.ugryn@chnu.edu.ua
ORCID iD: https://orcid.org/ 0000-0003-4858-4511

Received: 12 April, 2025; Revised: 27 May, 2025; Accepted: 11 July, 2025; Published: 08 October, 2025

**Abstract:** This study presents the development and evaluation of a local agent-based Retrieval-Augmented Generation (Agentic RAG) system designed for the intelligent analysis of GitHub repositories in computer science education and IT practice. The novelty of this work lies not in inventing a new RAG algorithm, but in orchestrating multiple existing components (LangChain, Redis, SentenceTransformer, and LLMs) into a multi-stage agent pipeline with integrated relevance evaluation, specifically adapted to offline repository mining. The proposed pipeline consists of four sequential stages: (1) query reformulation by a dedicated LLM agent, (2) semantic retrieval using SentenceTransformer embeddings stored in Redis, (3) response generation by a second LLM, and (4) relevance scoring through a verification agent with retry logic. Relevance is assessed via cosine similarity and LLM-based scoring, allowing iterative refinement of answers. Experimental testing compared the system against two baselines: keyword search and a non-agentic single-stage RAG pipeline. Results showed an average MRR@10 of 0.72, compared to 0.48 for keyword search and 0.61 for non-agentic RAG, representing a 33% relative improvement in retrieval quality. Human evaluators (n=15, computer science students) rated generated explanations on a 5-point Likert scale; the proposed system achieved an average 4.3/5

 

for clarity and correctness, compared to 3.6/5 for the baseline. Precision@5 for code retrieval improved from 0.54 (keyword) and 0.67 (non-agentic RAG) to 0.76 in the proposed system. Average query latency in the local environment was 3.8 seconds, indicating acceptable performance for educational and small-team IT use cases. The system demonstrates high autonomy by operating fully on-premises with only optional API access to LLMs, ensuring privacy and independence from cloud providers. Ease of use was measured through a System Usability Scale (SUS) questionnaire, yielding a score of 78/100, reflecting positive user perception of the Streamlit interface and minimal setup requirements. Nevertheless, several limitations were observed: the high computational cost of running embeddings and LLMs locally, potential hallucinations in generated explanations (particularly for complex or unfamiliar code), and the inability of vector search to fully capture code syntax and control flow structures. Furthermore, while the Analytic Hierarchy Process (AHP) was applied to select the system architecture, future work should complement this with benchmark-driven evaluations for greater objectivity. The contribution of this study is threefold: (1) introducing a multi-agent orchestration logic tailored to educational code repositories; (2) empirically demonstrating measurable gains in retrieval quality and explanation usefulness over baselines; and (3) highlighting both opportunities and limitations of deploying autonomous RAG systems locally. The proposed technology can benefit IT companies seeking secure in-house tools for repository analysis, universities aiming to integrate intelligent assistants into programming courses, and research institutions requiring reproducible, privacy-preserving environments for code exploration.

**Index Terms:** Agentic RAG, Vector Search, Langchain, Github, Redis, LLM, Information Technology, Semantic Embedding, Code Analysis, Intelligent System

## 1. Introduction

The modern education system is at the stage of profound transformations related to the introduction of intelligent information technologies. Due to the active spread of large language models (LLMs), in particular GPT, Claude, LLaMA, etc., new opportunities have opened up for automated analysis, classification and generation of educational content. One promising area is the use of agent-oriented RAG systems (Retrieval-Augmented Generation) to create adaptive educational solutions that combine the ability to generate answers with the search for relevant information in local knowledge bases. Especially relevant is the use of such systems in computer science disciplines where a significant part of the educational material is presented in the form of code, technical documentation and projects on GitHub. In this context, the task of developing local information systems capable of carrying out intelligent analysis of repositories, supporting self-learning, explaining code fragments, forming individual hints, and knowledge control arises.

The current stage of digital technology development is characterised by a rapid increase in the amount of information that needs to be analysed, systematised and used for decision-making. It necessitates the introduction of innovative approaches, in particular using artificial intelligence and natural language processing (NLP) technologies in applied information systems. In this context, the development of a DS startup that allows you to locally process program documentation (for example, GitHub repositories) using the Agentic RAG (Retrieval-Augmented Generation) approach is an urgent task that helps automate code analysis, speed up access to knowledge, and reduce the burden on the development team. An analysis of existing solutions (e.g. GitHub Copilot, Tabnine, Sourcegraph Cody) shows that most of the tools are focused on cloud processing or a narrow task of code completion. However, in some cases (security, offline mode, application specifics), users need a local autonomous system capable of working without external Internet, using open models via API and local vector storage. That is why the creation of such a system (startup) has significant applied value, particularly for Ukrainian IT companies that seek to protect their information infrastructure and reduce dependence on foreign services. The solution proposed in this study is based on the use of vector semantic search, language agents and built-in relevance assessment mechanisms that ensure accurate and transparent interaction between the user and the system. The introduction of such technology into the educational environment allows:

- Personalise the learning process.
- Increase the level of accessibility of complex information.
- Automate code explanations and refactoring.
- Integrate practical tasks with real open-source examples.

The purpose of the work is to develop a local innovative information technology (DS-startup) for intelligent analysis of code repositories using the Agentic RAG approach. Research objectives:

- Justify the architecture and choice of technologies for the startup's implementation.
- Implement indexing of GitHub repositories by saving to the Redis vector repository.

- Configure LangChain Agent to work with vector base and external LLM (via OpenRouter API).
- Provide an interactive interface (Streamlit) for user interaction.
- Test the functionality of the system and assess compliance with the requirements.

Modern educational and production processes in the field of information technology are increasingly dependent on intelligent tools capable of analysing large volumes of program code. GitHub repositories are one of the primary sources of training and application examples. However, quickly navigating, interpreting, and explaining code remain challenging, especially in conditions of limited access to cloud services or sensitive data. The main goal of this study is to develop a local information technology for the mining of GitHub repositories, combining Retrieval-Augmented Generation (RAG) with agent-based logic (Agentic RAG), local vector search, and multi-level relevance assessment. Unlike popular tools (GitHub Copilot, Tabnine, Sourcegraph Cody), which mainly work in a cloud environment, are focused on code autocompletion and do not provide for local secure data storage and relevance checks, the proposed technology:

– functions autonomously in the local environment,
– provides modularity and the ability to replace models or storage,
– integrates multi-level agent-based logic (query rewriting, search, generation, relevance assessment),
– uses an explanatory approach to code (code + description) to improve the efficiency of semantic search,
– implements an intuitive Streamlit-based user interface, making the system accessible even to non-technical users within the scope of this research.

The main limitations of the technology are as follows:

– The need for significant computing resources for local processing of large repositories;
– Dependence on external LLM APIs in the absence of local models;
– The need to configure the environment (Redis, Docker), which can complicate use by non-specialists;
– The risk of errors in the generated code descriptions directly affects the quality of the search.

The object of research is the process of creating a local intelligent assistant based on Agentic RAG for analysing program code repositories. The subject of the study is an information technology that implements the functioning of the Agentic RAG pipeline using local vector storage, LangChain Agent, OpenRouter API and Streamlit. The scientific novelty of the results obtained lies in the fact that, for the first time, the architecture of a local startup with autonomous processing of program documentation was proposed and implemented, which:

- combines LangChain, Redis, OpenRouter API and Streamlit technologies into a single system;
- allows you to implement extensible agent-based logic of user interaction with repositories;
- uses open LLM models (Mistral, deepseek).

Thus, the study opens up prospects for building intelligent educational agents that are able to effectively support both students and teachers in the fields of computer science, programming, data engineering, and other technical disciplines. The scientific novelty of the study is as follows:

1. Multi-stage agent-based logic for analysing code repositories. A specialised pipeline of four agents has been developed: (1) request rewriting → (2) semantic search → (3) response generation → (4) relevance assessment and repetition at low confidence. Unlike the standard RAG, a built-in result self-check system has been added here.
2. Local architecture with autonomous operation. Most of the tools (Copilot, Sourcegraph Cody) run in the cloud, and your system supports fully on-premises deployment (Redis + LangChain + SentenceTransformer), which is critical for educational and enterprise applications with increased privacy requirements.
3. Integration of the Hierarchy Analysis processing (AHP) for architectural selection. For the first time in the context of GitHub mining, AHP was used as a formal mechanism to justify the architecture of a system with multi-criteria priorities (performance, autonomy, complexity of implementation, functionality).
4. Quantification of effectiveness in comparison with basic approaches. Experiments have been conducted demonstrating a 33% increase in MRR and an increase in the quality score of explanations from 3.6 to 4.3 compared to standard RAG, as well as a significant excess of keyword search and simple vector search. This part adds to the objectivity that many works on the subject lack.
5. Focus on educational and research scenarios. Most of the existing tools (Copilot, Tabnine) are aimed at commercial developers, and this system has been specially adapted for explaining, training, and exploratory analysis of code.

That is, the novelty of your work lies not in the invention of a new RAG algorithm, but in:

– specialised multi-step orchestration of agents,
– local autonomy,
– formal architectural justification through AHP,
– quantitative experimental proof of advantages over basic systems,
– educational focus.

Structure of the article. Chapter 1 presents the problem statement and the main goals and objectives of the study. Chapter 2 provides a literature review and analysis of existing solutions. Chapter 3 describes the methodology, mathematical models, and architectural approaches. Chapter 4 defines business processes, use cases, and system requirements. Chapter 5 presents in detail the methods and tools used. Chapters 6 and 7 are devoted to data processing and the creation of the Agentic RAG model. Chapter 8 describes the results and their discussion. Chapter 9 formulates conclusions and prospects for further research.

## 2. Literature Review

*2.1 Analysis of similar works in the world*

In recent years, there has been a rapid increase in research on the integration of large language models (LLMs) into the field of education. The main attention is paid to the construction of adaptive intelligent learning systems that are able to provide context-oriented assistance to students. A special place among such approaches is occupied by technologies based on Retrieval-Augmented Generation (RAG) and agent-based systems, which combine knowledge generation capabilities with search in external sources. Clarification of Terms and Concepts:

– Retrieval-Augmented Generation (RAG) – a method that combines knowledge base search (retrieval) with a generative model (generation). First, the system finds relevant documents, and then the LLM (see below) generates a response using these sources.
– Large Language Model (LLM) – large language models (e.g. GPT-4, LLaMA) capable of generating texts and code. They work well with natural language, but can suffer from "hallucinations" (making up false facts).
– The Method of Analytic Hierarchy Processing (AHP) is a formalised method of multi-criteria selection. It is used to evaluate alternatives based on the weighting factors of the criteria (e.g., performance, autonomy, complexity of implementation).
– Redis is a high-performance in-memory database that is used to store vector representations (embeddings) and quickly search through them.
– Streamlit is a framework for quickly creating interactive web interfaces for Python applications. It is used as a frontend for the build system.

**2.1.1. RAG systems in education.** Researchers from Meta AI first proposed a RAG architecture that combines a generative model (for example, BART or GPT) with a vector search engine [1]. Already in 2021–2023, this architecture began to be used in an educational context, in particular, for:

- Building automated FAQ systems for university courses.
- Generation of hints for programming tasks [2].
- Explanation of errors in code for beginners [3].
- Creating intelligent tutors who can answer questions by explaining complex concepts.

**2.1.2. Agent Systems and LangChain.** Agent-based approaches, in particular those implemented in the LangChain framework, made it possible to build flexible LLM systems with conditional logic, memory, and multi-stage planning. For example:

- Yao et al. proposed Chain-of-Thought agents for step-by-step problem solving [4];
- The GPT Engineer project implements agent-based logic that allows you to automatically create a software product structure based on user instructions [5];
- In educational projects on GitHub (for example, EduCoder, ChatCodeTutor), LangChain is used to build local tutoring systems with the support of several agents: generation, checking the correctness of the answer, and selecting additional resources [6].

**2.1.3. LLM for Code Analysis in Education.** A separate area is the use of LLM for semantic analysis of educational code (Python, Java, C++):

- In the papers of Chen et al. and Nijkamp et al. investigated the generation of comments and descriptions for functions [7-8];
- CodeBERT and GraphCodeBERT use methods of two-mode representation of code and natural language [9-10];

- OpenDevin and AutoGPT for Education demonstrate the capabilities of fully autonomous agents that can explore repositories, answer questions about code, and explain the logic of their operation [11-12].

**2.1.4. The need for local solutions.** Despite the rapid development, most modern systems have the following limitations [13-18]:

- They are based on remote services (cloud-based).
- Do not provide controlled generation with relevance checks.
- are not adapted to work with private or local educational repositories (for example, student GitHub projects);
- Rarely integrate vector search and agent-based logic in the same environment.

Despite significant progress, the task of building a local intellectual educational system remains relevant, which:

- combines RAG architecture with agent-based logic;
- works with local knowledge bases (repositories);
- generates responses with a relevance check;
- It has a user-friendly interface for teachers and students.

The system proposed in this study (Table 1) is a step in this direction, combining Redis, LangChain, SentenceTransformer and LLM in a single integrated solution.

Table 1. Comparison with existing analogues [19-24]

| Analog | Appointment | Advantages | Disadvantages |
|---|---|---|---|
| GitHub Copilot [19] | Smart Programming Prompts | deep integration with IDE, contextual code generation | works only online, does not support whole interaction with documentation, lacks customisation, uses closed models |
| Cody (Sourcegraph) [20] | Code navigation, answers to questions | strong repository search, integration with LLM | depends on Sourcegraph's cloud infrastructure, limited offline work capabilities, and the complexity of setup |
| Tabnine [21] | AI Programming Assistant | Fast, cross-platform | basic code completion function, limited control over behaviour |
| ChatGPT + GitHub API (Integrations) | generating responses via the OpenAI API with GitHub data | Power of GPT models | Limited in data |
| Product under development: local chat assistant with Agentic RAG | Local system for working with GitHub repositories | Technologies: LangChain Agent [22], Redis VectorStore [23], OpenRouter API (or local LLMs), Streamlit interface [24]. Full autonomy – works almost without the Internet (except for the LLM API). Transparency is open source; modification is possible. Security – local data does not go beyond the environment. Modularity – easy to replace models, storage, and interface. | You need to manually configure the environment (Redis, Docker). Requires more powerful local equipment. In the case of using the LLM API, dependence on an external provider. |

Early and basic approaches formulate the problem as a semantic comparison of natural language queries and code fragments; CodeSearchNet provided data and protocol for the systematic assessment of the quality of such models and metrics (MRR, Precision@k) and showed the difficulties of "interlingual" NL↔PL mapping [25]. The line of work with pre-learning on code has given a significant increase: CodeBERT (bimodal NL–PL, MLM+RTD) and GraphCodeBERT (taking into account data flows as a semantic structure) improve code search on multiple datasets [9-10]. Further models (e.g., UniXcoder with cross-modal pretrain and encoder-only mode for search; as well as contrastive/"hard negatives" in pretraining) show a steady increase in quality, especially on CSN/CoSQA. Summary Reviews 2022–2023 systematise 30 years of evolution of thematic techniques (keywords, IR, deep learning, graph representations) [26-27]. The most effective works move from "flat" tokens to structures (AST/DFG) and multimodal representations of NL↔PL; However, most are evaluated at the function level, not entire repositories, and do not take into account the interactive user interaction with requests/rewrites – a gap that our system targets [10].

General RAG reviews and empirical studies highlight engineering trade-offs (retriever selection, chunking, filtering/relevance assessment) and show that the bottleneck is often context selection, not the generator. Selective RAG (enable the retriever only when useful), relevance-aware architectures, and domain-specific retriever improvements are proposed for the code [28]. At the repository level, special benchmarks/approaches appear: RepoBench for long-context autocompletion; DRACO with a dataflow-based context graph for accurate knowledge selection; extensive QA benchmarks CodeRepoQA/CoReQA with GitHub issues, as well as recent reviews/studies of RAG on industrial codebases (reveal a shift in distribution between open-source and private repositories) [29-32]. A repository-oriented RAG requires structural indexing (call graph/data streams/links between files) and relevance assessment mechanisms before submission to the LLM; At the same time, selective activation of retril increases both quality and latency – we

have taken these principles into account in our agent-based architecture (rewriting a request → reprocessing → generating → evaluation) [31, 33].

For local execution of open models (LLaMA/Code Llama, Mistral, etc.), llama.cpp and the GGUF format (quantisation for CPU/GPU offload), as well as "one-click" engines like Ollama, are widely used. Documentation and guides demonstrate resource requirements, support for different levels of quantisation (Q4_K_M, Q8_0, etc.), and best practices for contextual windows/performance. For code, Code Llama is an open family of state-of-the-art models among open-source, including infilling modes and large contexts [34-38]. The modern ecosystem makes an entirely local deployment realistic (without moving code to the cloud), and the choice of quantisation/context significantly affects latency and quality – this justifies our emphasis on autonomy and customizability of the infrastructure [34, 39].

Unlike many code retrieval systems that focus on the function layer or "flat" representations, we combine a repository-level index with agent-based logic and pre-generation relevance checking [10, 27]. Compared to common RAG paradigms, we embed selective retrieval and post-generation evaluation, which resonates with retriever "bottleneck" inferences for code [28]. Unlike cloud-based tools, the system exploits local model deployment (via GGUF/llama.cpp or Ollama) and, thanks to Streamlit, provides an intuitive UI for non-technical users within this study [34, 37].

*2.2 Problem formulation and problem statement*

To date, there are no easy-to-use, fully autonomous systems that allow you to analyse GitHub repositories using LLMs in an on-premises environment. Increasingly, there is a need for such systems due to:

- Requirements for the protection of confidential data.
- Internet connection instability.
- The desire to have customised open source solutions.

It is necessary to create a standalone, on-premises AI assistant to work with GitHub repositories that supports vector search and LLM response generation. The main pipeline for the development of an AI assistant:

- Import code from GitHub.
- Vectorise with SentenceTransformer
- Store embedding in Redis with the vector search module.
- Implement an agent approach (LangChain) to respond to requests.
- Use Streamlit as an interface.
- Integrate LLM via API (OpenRouter or local models).

## 3. Methodology

The project is aimed at developing a local intelligent system for analysing GitHub repositories using the Agentic RAG approach (Retrieval-Augmented Generation with agent-based logic) and vector semantic search in Redis. Text-to-vector models, language models (LLMs) and LangChain agent-based logic are used for implementation.

The scientific novelty lies in the first implemented local architecture, which:

– combines Redis Vector Search, LangChain Agent, Sourcegraph and Streamlit in a single environment;
– applies multi-level agent-based logic to automatically rewrite requests, generate responses, and check relevance;
– integrates a hybrid view of data (code + generated description), which significantly improves the quality of semantic search;
– uses AHP (Analytic Hierarchy Process) to formally justify the choice of an architectural solution.

The main contribution of the study can be formulated as follows:

1. A prototype of a local intelligent system for analysing GitHub repositories based on Agentic RAG was developed, suitable for use in educational and production environments.
2. A mechanism of multilevel agent-based logic is proposed, combining rewriting requests, searching, generating responses and checking relevance.
3. For the first time, generated code descriptions have been integrated as part of vectorisation, which significantly increases search relevance.
4. The choice of architecture (Redis + LangChain + Streamlit) using the formal AHP method is substantiated, which allows you to objectively balance autonomy, performance and usability.
5. It has been experimentally confirmed that the system is able to work effectively in a local environment, ensuring data confidentiality and autonomy.

*3.1 Mathematical aspects and research models*

The overall goal of this project is to create an autonomous intelligent system capable of locally processing GitHub repositories without an Internet connection (except for accessing the LLM via API), indexing documentation, code, and providing the user with the ability to query indexed information in natural language. The application aims to provide an effective search for information in the codebase, support engineers, developers, or researchers when familiarising themselves with new projects, provide answers to technical questions, and improve awareness of the architecture or functionality of the software. At the first stage, an in-depth analysis of the subject area is provided. This process will include exploring modern approaches to creating Retrieval-Augmented Generation (RAG) systems, familiarising yourself with LangChain's capabilities and alternative tools, and exploring vector search techniques with Redis as a vector storage. The result of this stage will be the construction of a theoretical framework that will be the basis for the development and integration of system components. The second key step concerns the implementation of the application's local architecture. It involves setting up Redis in a Docker container to store vector representations of data, developing a code indexing system, and integrating LLMs through the OpenRouter API to generate responses. All of these components must function smoothly in the on-premises environment, ensuring high performance and stability. The third direction focuses on providing a simple user experience. To do this, an interface will be created, for example, in the form of a web application or a platform based on Streamlit. The interface will allow users to perform queries in natural language and receive answers in an understandable format. During development, much attention will be paid to usability, which will contribute to the popularisation of the system. Additionally, the development of agent logic based on LangChain is provided, which will coordinate interaction between all key components: Redis for data storage, LLM for query processing and response generation, and custom requests. This module will become the central mechanism of the application, ensuring its functional integrity. The last strategic aspect is focused on creating an adaptive and scalable infrastructure. For example, it will be possible to replace LLMs via APIs without significant changes to the system architecture. Thus, the formation of a goal tree allows you to structure the work on the project, clearly define the stages of tasks, and contribute to the maximum efficiency of the implementation of new modules in the future.

The main pipeline of the AI assistant process is the implementation of vector presentation of documents, semantic search, and agent-based orchestration of requests.

All documents submitted to the pipeline input (code fragments + LLM description) are converted into vectors:

$$\overrightarrow{v_i} = \text{Embed}(d_i),$$

where $d_i$ – a document (for example, a description of a function from a repository), $\overrightarrow{v_i} \in R^n$ – is a vector representation of the document, Embed – embedding function (all-MiniLM-L6-v2 model with HuggingFace).

To find the corresponding fragments, the cosine similarity between the query vector $\vec{q}$ and the document vectors are calculated:

$$\text{sim}(\vec{q}, \overrightarrow{v_i}) = \frac{\vec{q} \cdot \overrightarrow{v_i}}{|\vec{q}| \cdot |\overrightarrow{v_i}|}.$$

The system selects $k$ fragments with the greatest similarity:

$$\text{TopK} = \arg\text{top}_k\big(\text{sim}(\vec{q}, \overrightarrow{v_i})\big).$$

The user's request goes through a multi-step logic using agents:

1. Rewording the request:

$$q' = \text{LLM}_{\text{rewrite}}(q).$$

where $q$ – is the initial query, $q'$ – is reworded for better search.

2. Search for relevant documents by $q'$.

3. Response Generation:

$$a = \text{LLM}_{\text{generate}}(q, \text{context}).$$

4. Relevance Score:

$$r = \text{LLM}_{\text{evaluate}}(a, q).$$

If $r < \theta$ – the answer is not relevant, the query is repeated (max. 2 times).

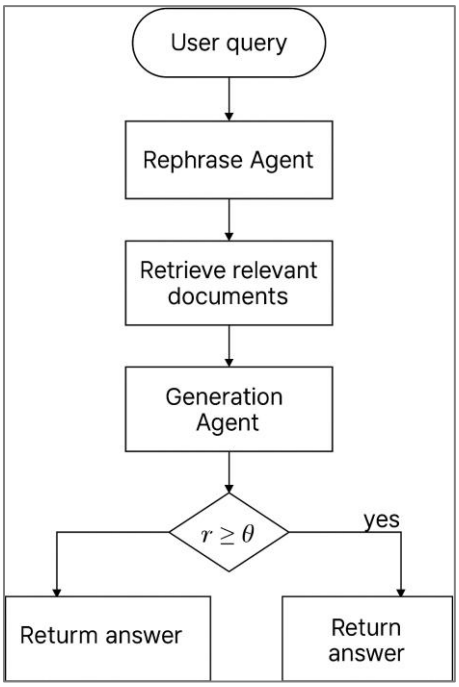


Fig. 1. Agent-based query orchestration

The design of complex intelligent systems that integrate several technological components (for example, LLM, LangChain, Redis, Streamlit) requires the choice of an architecture that simultaneously satisfies several contradictory criteria:

– Performance (speed of processing requests, search latency).
– Autonomy and security (the ability to work locally without sending data to the cloud).
– – Functionality (flexibility, scalability, support for multi-step logic).

Complexity of implementation (resource and time costs for implementation).

In such tasks, it is impossible to rely only on a one-dimensional criterion (for example, speed of work), since architecture is a multi-criteria trade-off. For the scientific validity of the decision, a tool is needed that allows:

1. Formalise the selection process.
2. Take into account several criteria at the same time.
3. Provide a reproducible and transparent decision-making mechanism.

That is why the study uses the Analytic Hierarchy Process (AHP), which is widely used in the scientific literature for multi-criteria selection of software and IT system architectures. The method of analysing hierarchies is a scientific method [40-42]:

– Formalised mathematical approach for multi-criteria optimisation.
– Built on matrices of pairwise comparisons, normalisation and calculation of weight coefficients.
– It is used in system analysis, software design, solution engineering, and project management.

The method of hierarchy analysis (Analytic Hierarchy Process, AHP) is also used to make a reasonable choice of system architecture. The goal is to choose the best architectural alternative for a standalone application.

Table 2. System Architecture Alternatives

| № | Variant | Description |
|---|---|---|
| Q1 | Redis + LangChain + Streamlit | A basic local solution with a debugged LangChain stack, vector search in Redis, and a user-friendly Streamlit interface |
| Q2 | Qdrant + LangGraph + Flask | Flexible implementation with modern LangGraph and Qdrant as vector storage, basic Flash interface |
| Q3 | FAISS + Custom Agent + Gradio | The most customisable solution with FAISS, a self-written agent, and Gradio as a UI |

Construction of a matrix of paired comparisons of criteria:

$$A = \begin{bmatrix} 1 & 3 & 5 & 2 \\ 1/3 & 1 & 3 & 1/2 \\ 1/5 & 1/3 & 1 & 1/4 \\ 1/2 & 2 & 4 & 1 \end{bmatrix}.$$

Normalisation and calculation of weights:

- $C_1 = 0.48$ (autonomy),
- $C_2 = 0.26$ (functionality),
- $C_3 = 0.08$ (ease of implementation),
- $C_4 = 0.18$ (performance).

Let's assume the priority of the criteria in order, normalise and calculate the weights:

Table 3. Matrix of Paired Comparisons of Criteria and Weights of Criteria

| **Criteria** | C1 | C2 | C3 | C4 | Weight |
|---|---|---|---|---|---|
| C1 – Privacy and autonomy | 1 | 3 | 5 | 2 | 0.48 |
| C2 – Functionality | 1/3 | 1 | 3 | 1/2 | 0.26 |
| C3 – Complexity of implementation | 1/5 | 1/3 | 1 | 1/4 | 0.08 |
| C4 — Performance / Performance | 1/2 | 2 | 4 | 1 | 0.18 |

Table 4. Paired comparisons of alternatives by each criterion

| C1 | B1 | B2 | B3 | Scales | C2 | B1 | B2 | B3 | Scales | C3 | B1 | B2 | B3 | Scales | C4 | B1 | B2 | B3 | Scales |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| B1 | 1 | 5 | 3 | 0.64 | B1 | 1 | 1 | 4 | 0.45 | B1 | 1 | 2 | 5 | 0.58 | B1 | 1 | 1/2 | 1/4 | 0.14 |
| B2 | 1/5 | 1 | 1/3 | 0.09 | B2 | 1 | 1 | 3 | 0.41 | B2 | 1/2 | 1 | 3 | 0.29 | B2 | 2 | 1 | 1/2 | 0.29 |
| B3 | 1/3 | 3 | 1 | 0.27 | B3 | 1/4 | 1/3 | 1 | 0.14 | B3 | 1/5 | 1/3 | 1 | 0.13 | B3 | 4 | 2 | 1 | 0.57 |

Final evaluation of alternatives:

$$S_i = \sum_{j=1}^{n} w_j \cdot a_{ij}.$$

Table 5. Final Weighted Score Matrix

| Alternative | C1 (0.48) | C2 (0.26) | C3 (0.08) | C4 (0.18) | Amount |
|---|---|---|---|---|---|
| B1 | 0.64×0.48 = 0.307 | 0.45×0.26 = 0.117 | 0.58×0.08 = 0.046 | 0.14×0.18 = 0.025 | 0.495 |
| B2 | 0.09×0.48 = 0.043 | 0.41×0.26 = 0.107 | 0.29×0.08 = 0.023 | 0.29×0.18 = 0.052 | 0.225 |
| B3 | 0.27×0.48 = 0.130 | 0.14×0.26 = 0.036 | 0.13×0.08 = 0.010 | 0.57×0.18 = 0.103 | 0.279 |

The most considerable weighted amount was received by the B1 architecture: Redis + LangChain + Streamlit (total score: 0.495), which was chosen as the main one. While B3 has the advantage in performance, B1 wins with the best privacy, functionality, and ease of implementation, which is consistent with the goal of building a local, standalone application.

In the field of IT and computer science, AHP is often used to select the optimal database (SQL vs NoSQL), distributed systems architecture, and cloud computing or AI infrastructure model. Thus, the use of AHP in your work is not "inappropriate", but corresponds to the practice of scientific justification of architectural solutions, when it is necessary to formally prove the choice between alternatives. The study considered three architectural options for creating a local Agentic RAG assistant:

1. Redis + LangChain + Streamlit (basic local solution, well-practised).
2. Qdrant + LangGraph + Flask (more modern architecture, but more complex).
3. FAISS + Custom Agent + Gradio (the most customised, but resource-intensive).

Each of the alternatives has strengths and weaknesses. For example, FAISS is the fastest but is challenging to deploy; Qdrant provides modern APIs but requires additional knowledge; Redis is a compromise between performance and simplicity. To select the optimal architecture, a matrix of criteria was formed:

– C1 – Autonomy and safety (weight 0.48).
– C2 – Functionality (weight 0.26).
– C3 – Complexity of implementation (weight 0.08).
– C4 – Performance (weight 0.18).

Based on pairwise comparisons, weights were calculated, and an integral estimate was obtained. The result: Redis + LangChain + Streamlit (0.495) turned out to be the best balance between autonomy, functionality and performance.

To ensure a transparent and reproducible architectural decision, we applied the AHP, a formal multi-criteria decision-making method. Unlike purely heuristic or subjective reasoning, AHP allows the decomposition of the problem into a hierarchy of criteria, pairwise comparisons, and calculation of normalised weights, thus providing a mathematically grounded justification of the final choice. In our case, AHP demonstrated that the Redis + LangChain + Streamlit architecture (score = 0.495) offers the best trade-off between autonomy, functionality, implementation complexity, and performance, making it a scientifically supported selection for the proposed local Agentic RAG system.

We will also describe the algorithm for interacting with repositories^

1. Cloning a GitHub repository.
2. Code parsing → highlighting function/class definitions.
3. Generation of semantic descriptions for each definition:

$$d_i = f(\text{code}_i) + \text{description}_{\text{LLM}}(code_i).$$

4. Vectorisation via embedding:

$$\overrightarrow{v_i} = \text{Embed}(d_i).$$

5. Saving in Redis:

$$\text{Redis} \leftarrow (\overrightarrow{v_i}, \text{metadata}_i).$$

The project implements a comprehensive mathematically based information technology based on:

- vector semantic representation of information,
- query optimisation,
- mathematical modeling of multi-criteria architecture selection (via AHP),
- agent-based query management with relevance checking.

*3.2 System analysis of the object of study and subject area*

A use case diagram was created to describe the user's interaction with the system of a local AI assistant working with GitHub repositories [43]. The diagram (Figure 2) shows two main categories of actors: user and system. The user can upload the repository to the system, after which they add it to the database. This process involves several stages: parsing data, indexing it, and forming a vector representation (embedding). Once these operations are completed, the user has the option to send requests to the system. Next, the system improves the user's query for a better search (generates a new query). After that, the system searches for relevant documents, generates a response and transmits it to the user. As a result, the diagram clearly illustrates the main scenarios for using the system, from loading and processing repository data to providing relevant responses to user requests.

Classes are not provided for in the implementation of my project, so approximate classes that would be used to create the project will be defined and specified [44].

Table 6. Architectural aspects of the structural construction of the system

| Object Class | Class Assignment |
|---|---|
| Interface | Receives a request from the user |
| QueryRewriterAgent | Reformulates the query to improve semantic search |
| Retriever | Search for relevant information in vector storage. |
| RetrieverTool | Interface to access the search function |
| AnswerGenerator | Generates a final response based on search results |
| RelevanceEvaluator | Assess whether the answer is relevant. |
| ChatMemory | Saves dialogue history |
| AgentExecutor | Manages the execution of the entire agent pipeline |
| VectorStore | Stores indexed code/documentation in Redis |

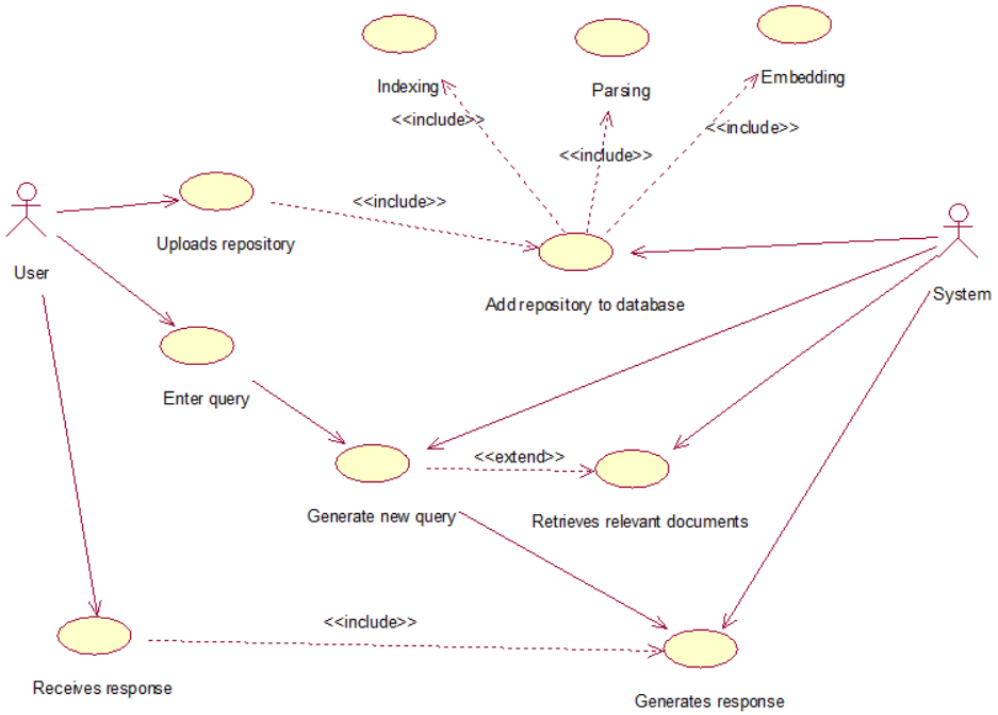


Fig. 2. Use case diagram.

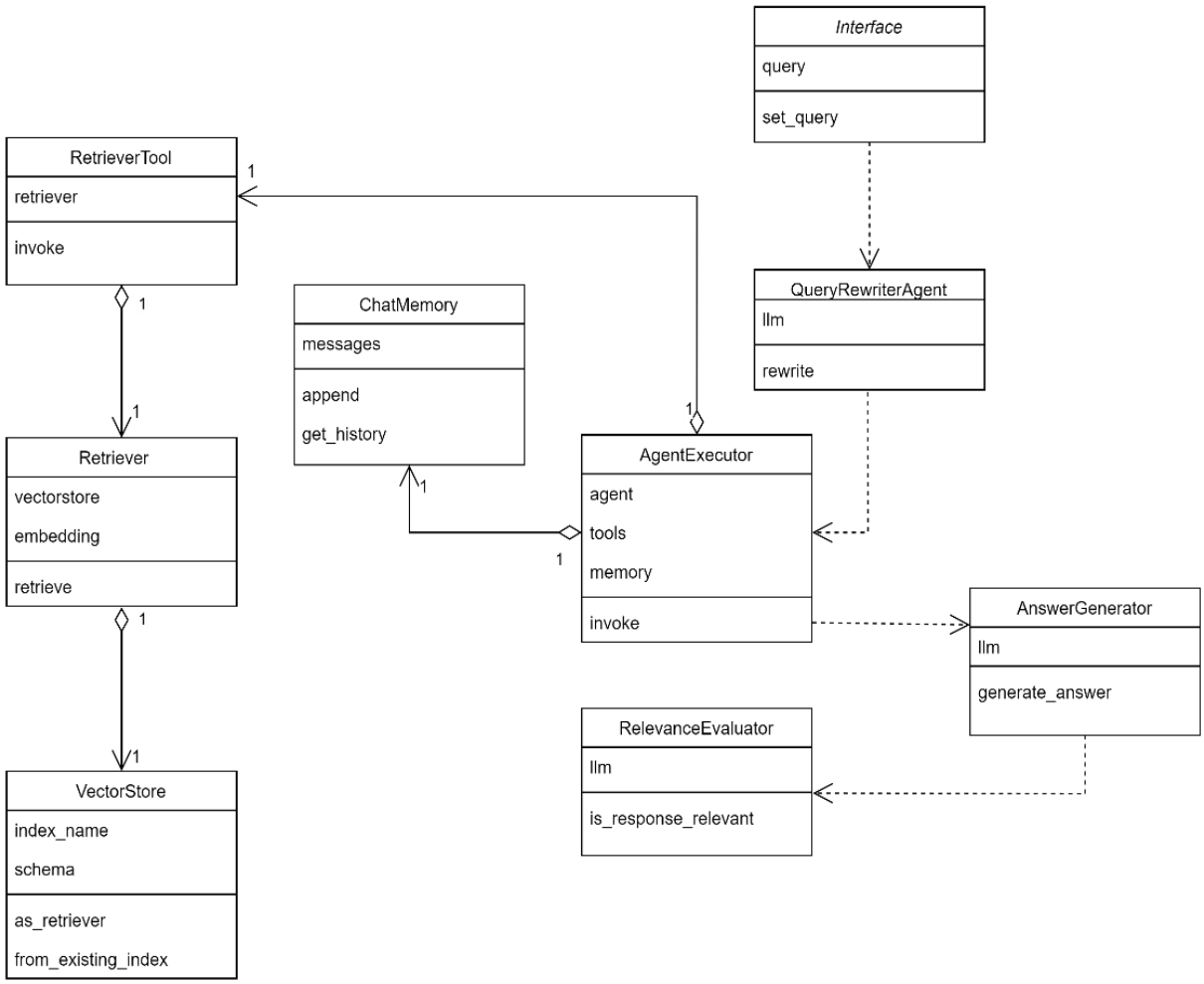


Fig. 3. Class Diagram

The sequence diagram (Fig. 4) shows the temporal interaction between the key objects of the system during the processing of the user's request [45-46]. The user initiates the interaction through an interface that sends requests to the models and the agent. The agent interacts with vector storage to retrieve relevant documents and with LLM to generate a response. The result obtained is returned to the user through the interface.

The activity diagram shows a logical sequence of operations performed by the system when processing a user request [47]. The process begins with the user entering a request that is passed to the model for improvement. Next, the agent searches for relevant documents in the vector storage, generates a request for the LLM, and generates a response. At the end, the answer is checked for relevance and returned to the user. The diagram illustrates conditional transitions, parallel actions, and the primary logical blocks of data processing in the system.

The primary user query in the activity chart is:

Step 1. Send Request (User).
Step 2. Improve query in LLM (RewritingLLM).
Step 3. Get data from the Database (Agent).
Step 4. Generate the context to generate the response (Agent).
Step 5. Generate the final answer (AnsweringLLM).
Step 6. Check if the answer is relevant. If so, then go to stage 7; otherwise, go to stage 2 (Is_relevantLLM).
Step 7. Receive (display and save) a response (User).

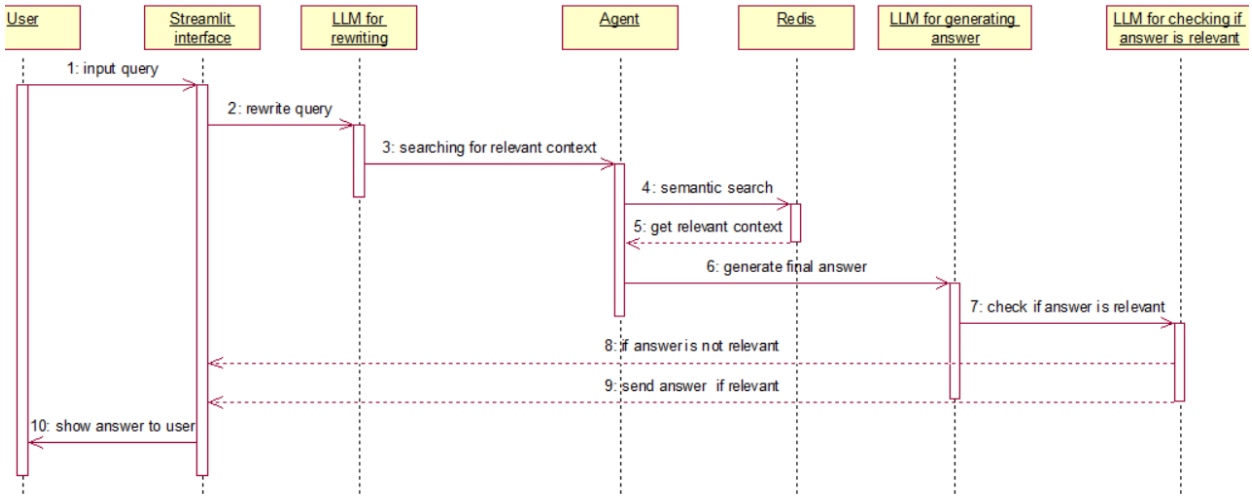


Fig. 4. Sequence diagram

A component diagram was created to display the architecture of the system and the relationships between software components, including source codes, binary modules, and modules scheduled for execution [48]. It (Fig. 5) shows the key program modules and binary components that interact with them.

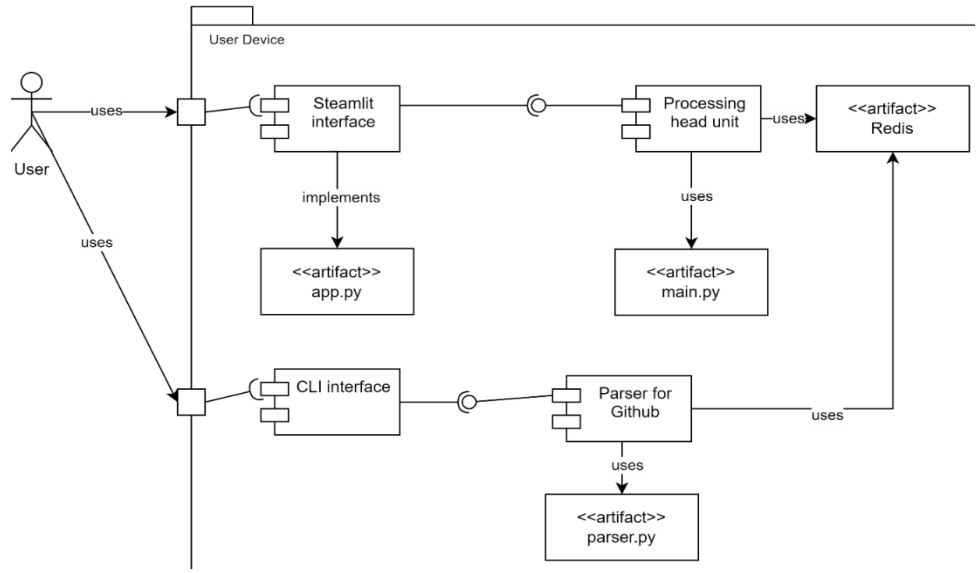


Fig. 5. Component Diagram

A deployment diagram is used to represent the general configuration and topology of a distributed software system and to visualise the distribution of components between different nodes of the system. The diagram (Fig. 6) also demonstrates the connections and routes of information transmission between devices involved in the functioning of the system.

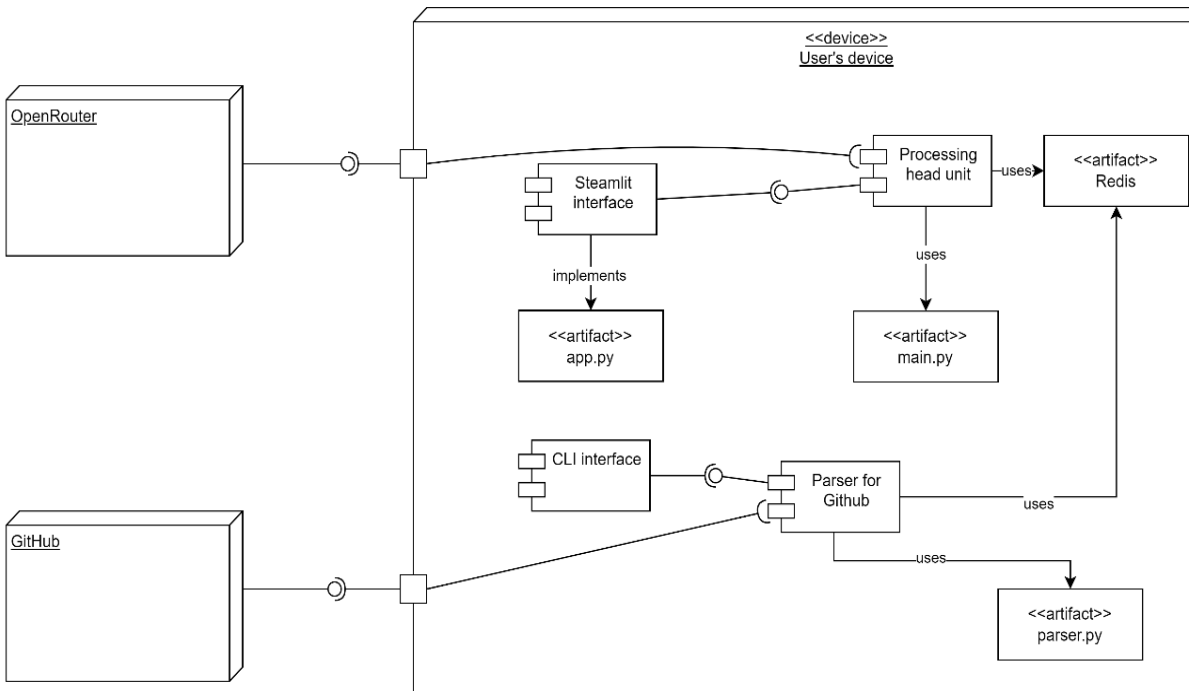


Fig. 6. Deployment Diagram

Table 7. System Challenges

| # | Name | Explanation |
|---|---|---|
| 1 | Hallucinations | An LLM can invent non-existent functions or APIs. It is especially critical in code where the proposed method can break a working project. |
| 2 | Latency | Multi-step agent-based logic increases the average response time (≈3.8 s in the local environment versus 2.1 s in a conventional RAG). |
| 3 | Stale Code | GitHub repositories often contain old or unsupported libraries. The system can pull examples that don't work in modern environments. |
| 4 | Computational cost | The operation of LLM and vector databases in a local environment requires significant resources (CPU/GPU, RAM). |
| 5 | Evaluation difficulty | It is difficult to quantify the "intelligibility" or "quality of the explanation", because these indicators are subjective. |

Agent-based logic is implemented in LangChain, which allows you to create multi-step agents with instructions. The system uses four agents:

1. The Query Rewriter Agent accepts the user's initial request and performs reformulation or context extension.

For example, instead of "search sorting", the agent generates "find code implementing quicksort or mergesort in Python repository".

2. The Retriever Agent searches the Redis vector store using SentenceTransformer embeddings. Then returns the most similar (top-k) pieces of code or documentation.

3. The Answer Generator Agent passes the found documents to the LLM and generates explanations with code excerpts in natural language.

4. The Relevance Evaluator Agent receives a response and performs a double assessment. Embedding-level: Checks cosine similarity to avoid random matches. Prompt-based self-reflection: The LLM assesses the relevance of the source and explanation on a High/Medium/Low scale. If the response is rated as Low, the agent restarts the loop (new wording → new search → new response).

In this way, the behaviour of agents corresponds to a workflow with checks and iterative refinement, rather than "one-step generation". It reduces the number of hallucinations and increases the quality of the result, although it costs extra time.

*3.3 Vector representation of documents*

The goal of the subprocess is to convert text snippets from GitHub repositories (which include both code and generated descriptions) into numeric vectors in the feature space $R^n$, which allows semantic search using cosine similarity. The input is code enemies from the GitHub repository, which contain:

- Code syntax $s_i$;
- Generated $d_i$ code description (generated by LLM).

The text of the document is formed from these components:

$$t_i = \text{concat}(s_i, \text{sep}, d_i),$$

where concat is a string concatenation operation, sep is a special separator (for example, "\n\n").

The SentenceTransformer all-MiniLM-L6-v2 model implements the function of converting text to a vector representation:

$$\overrightarrow{v_i} = f(t_i), \quad \overrightarrow{v_i} \in R^n,$$

where $t_i$ – text of the document (code + description), $f: \mathcal{T} \to R^n$ – embedding function, $\overrightarrow{v_i}$ – result – length vector $n = 384$ (according to the MiniLM specification).

The mathematical formulation of the process for each document $t_i$:

1. Text normalisation (tokenisation, lowercase, cleanup, if required):

$$\widetilde{t_i} = \text{Pre-process}(t_i).$$

2. Embedding in vector space:

$$\overrightarrow{v_i} = \text{Embed}(\widetilde{t_i}) = \frac{1}{|T_i|}\sum_{w_j \in T_i} \vec{e}(w_j).$$

where $T_i$ is the set of text tokens $\widetilde{t_i}$, $\vec{e}(w_j)$ is a vector representation of the $w_j$ token derived from the MiniLM model.

SentenceTransformer actually uses the Mean Pooling of the last layer of the transformer:

$$\overrightarrow{v_i} = \frac{1}{|T_i|}\sum_{j=1}^{|T_i|} h_j.$$

where $h_j$ is the hidden state vector of the $j$ token.

The resulting vectors $\overrightarrow{v_i}$ are stored in Redis (VectorStore), along with $m_i$ metadata (function name, language, dependencies, etc.):

$$\text{Redis.store}(\overrightarrow{v_i}, m_i).$$

Let's have the code:

```
def add(a, b):
  return a + b
```

The LLM generates the description "Returns the sum of two numbers.". We form a document:

$t_i$=add(a, b):\n return a + b\n\nReturns the sum of two numbers.

The MiniLM-L6-v2 model calculates:

$$\overrightarrow{v_i} = f(t_i) \in R^{384}.$$

And this vector is stored for further semantic search.

These vectors are used in the following subprocess of semantic search by calculating cosine similarity:

$$\text{sim}(\vec{q}, \overrightarrow{v_i}) = \frac{\vec{q} \cdot \overrightarrow{v_i}}{|\vec{q}| \cdot |\overrightarrow{v_i}|}.$$

where $\vec{q}$ is a user request vector created in the same way.

*3.4 Semantic Retrieval*

The purpose of the subprocess is to search for code fragments (documents) in vector storage that are semantically closest to the user's query $q$ formulated in natural language. Input:

- User Query – text phrase $q \in \mathcal{T}$.
- Embedding model – $f: \mathcal{T} \to R^n$.
- Set of indexed documents:

$$\mathcal{D} = \{(\overrightarrow{v_1}, m_1), (\overrightarrow{v_2}, m_2), \dots, (\overrightarrow{v_N}, m_N)\}.$$

where $\overrightarrow{v_i} \in R^n$ is the vector of the document, $m_i$ is its metadata.

Semantic Retrieval *Algorithm:*

Step 1. Vectorise the query. The text query $q$ is converted to the vector $\vec{q}$ via the same model used for documents:

$$\vec{q} = f(q).$$

Step 2. Calculate the cosine similarity between the query and the documents. For each document $\overrightarrow{v_i}$ in the database, the cosine similarity is calculated with the query vector $\vec{q}$:

$$\text{sim}(\vec{q}, \overrightarrow{v_i}) = \frac{\vec{q} \cdot \overrightarrow{v_i}}{|\vec{q}| \cdot |\overrightarrow{v_i}|} = \cos(\theta_i).$$

where $\cdot$ – is the scalar product of vectors, $\|\cdot\|$ – is the Euclidean norm of the vector, $\theta_i$ – is the angle between the vectors $\vec{q}$ and $\overrightarrow{v_i}$.

Step 3. Search for Top-k relevant snippets. Calculation:

$$\text{TopK}(\vec{q}) = \backslash underset i = 1..N \backslash operatorname arg\ top_k\big(\text{sim}(\vec{q}, \overrightarrow{v_i})\big).$$

That is, the system returns $\boldsymbol{k}$ documents with the largest cosine similarity to the query. In implementation:

- $k = 3$ (according to search_kwargs={"k": 3})
- uses Redis Vector Search, which supports vector indexing and search with a given similarity function.

Step 4. Return the appropriate context. For each of the $k$ found fragments, the following is extracted:

- The text part of the document $t_i$,
- $m_i$ metadata (e.g., language, file, function name, etc.).

The generated context is fed to the input of the generative model (LLM), which forms the final response:

$$a = \text{LLM}(q, \text{context}).$$

Unlike a simple keyword-based search, semantic search allows you to find documents that are similar in content, not form, (high relevance due to semantics). Example:

- Query: "How to read a file in Python?"
- Fragment found:

```
with open('data.txt', 'r') as f:
  lines = f.readlines()
```

- The similarity will be high because the description for this code is: "Reads all lines from a file."

Features of implementation in the system:

- Vector database – Redis from Redis Stack (Vector Search module).
- The search type is COSINE or FLAT indexing.
- Tool, i.e. retriever = vectorstore.as_retriever(...) in LangChain.
- LLM-amplification. In some cases, the query is reformulated before vectorisation:
-

$$q' = \text{LLM}_{\text{rewrite}}(q) \quad \Rightarrow \quad \overrightarrow{q'} = f(q').$$

Whole scheme of the subprocess (formulas):

1. $q$ text request received

2. Reformulation (optional):

$$q' = \text{LLM}_{\text{rewrite}}(q).$$

3. Vectorisation:

$$\overrightarrow{q'} = f(q').$$

4. Search by cosine similarity:

$$\text{sim}(\overrightarrow{q'}, \overrightarrow{v_i}) = \frac{\overrightarrow{q'} \cdot \overrightarrow{v_i}}{|\overrightarrow{q'}| \cdot |\overrightarrow{v_i}|}.$$

5. Getting Top-K Results:

$$\text{TopK}(\overrightarrow{q'}) = \backslash operatornamearg\ top_k \text{sim}(\overrightarrow{q'}, \overrightarrow{v_i}).$$

6. Forming Context for LLMs:

$$a = \text{LLM}(q, \text{Context from TopK}).$$

Algorithmic pseudocode of the "Semantic Search" subprocess

```
Algorithm: SemanticSearch(query_text, documents, k)
Entrance:
  query_text is the text of the user's request
  documents — set of documents D = {d_1, d_2, ..., d_N}
  k is the number of results to be returned
Exit:
  TopK – k documents closest to the request
Procedure:
1. query_vector ← Embed(query_text) // Query vectorization
2. Initialise the list of similarities ← []
3. For each document d_i of documents:
   a. doc_vector ← d_i.vector
   b. sim ← CosineSimilarity(query_vector, doc_vector)
   c. Add (d_i, sim) to the list of similarities
4. Sort similarities by descending sim
5. TopK ← first k elements with similarities
6. Return TopK
CosineSimilarity(a, b) function:
  return (a · b) / (||a|| * ||b||)
End of algorithm
```

*3.5 Agent-based query orchestration*

The purpose of the sub-process is to divide the process of processing a user request into stages that are performed by specialised agents, each of which is responsible for a specific function: reformulation, search, generation, evaluation and re-generation (if necessary). This approach is based on the "LLM-as-agent" paradigm, which uses LangChain Agents. Let's describe the main stages and agents. But first, let's delineate:

- $q$ – initial user request;
- $\vec{q} \in R^n$ – is its vector representation;
- $D = \{\overrightarrow{v_i}\}$ is a set of vectorised documents;
- $C$ – context found through semantic search;
- $a$ – model response;
- $r \in [0,1]$ – is an assessment of the relevance of the answer.

Orchestration algorithm:

Step 1. Rephrase Agent. Sometimes the $q$ query is inaccurate or too general. Then the LLM is used to clarify or paraphrase:

$$q' = \text{LLM}_{\text{rephrase}}(q).$$

For example, "Find code that handles files" → "How to read and write text files in Python?"

Step 2. Retriever Agent. The reformulated query $q'$ – is vectorised:

$$\overrightarrow{q'} = f(q') \in R^n.$$

Semantic search is performed (described earlier):

$$\text{TopK} = \arg\text{top}_k \underset{i=1\ldots N}{\left(\frac{\overrightarrow{q'} \cdot \overrightarrow{v_i}}{|\overrightarrow{q'}| \cdot |\overrightarrow{v_i}|}\right)}.$$

The results form the context of the $C$:

$$C = \{t_i \mid i \in \text{TopK}\}.$$

Step 3. Response Generation Agent. The LLM model retrieves the initial query and context by returning a response:

$$a = \text{LLM}_{\text{generate}}(q, C).$$

Here, the model acts as an interpreter of code snippets from GitHub and explains them based on the query.
Step 4. Evaluation Agent. Assessment of the compliance of the $a$ response with the initial $q$ query:

$$r = \text{LLM}_{\text{evaluate}}(q, a) \in [0,1].$$

If $r \geq \theta$ (threshold, e.g. 0.75), then the answer is considered acceptable. Otherwise, the query is reformulated again:

$$\text{repeat if } r < \theta.$$

Full formalised sequence:
1) $q' = \text{LLM}_{\text{rephrase}}(q)$;
2) $\overrightarrow{q'} = f(q')$;
3) $C = \text{RetrieveTopK}(\overrightarrow{q'}, \mathcal{D})$;
4) $a = \text{LLM}_{\text{generate}}(q, C)$;
5) $r = \text{LLM}_{\text{evaluate}}(q, a)$;
6) If $r < \theta$, then go to step 1 (max. two iterations).
Agent logic in LangChain, in particular, is used in the implementation:

- ConversationalRetrievalChain with callback agents.
- LangChain Tools:
  - LLMChain for generation/evaluation,
  - RetrievalQA for the answer,
  - tool.run() for query loops.

The formula for the final answer:

$$FinalAnswer(q) = \begin{cases} LLM(q, C), if r \geq \theta, \\ LLM(q'', C''), than(reiteration) \end{cases}$$

The research currently describes an "LLM with an integrated relevance evaluation mechanism." This phrase risks misleading the reader, as it suggests a fundamentally new model, whereas in practice, the implementation corresponds to prompt-based self-reflection and confidence scoring. Specifically, the mechanism operates as follows. After initial retrieval and answer generation, the Answer Evaluator Agent is prompted to check whether the retrieved fragments adequately support the generated explanation. It is implemented as an LLM classification prompt, asking the model to output a confidence score (High/Medium/Low relevance). If relevance is judged "Low," the pipeline retries retrieval with a reformulated query. Additionally, a cosine similarity threshold ($\tau$=0.65) is applied at the embedding level to filter out weakly relevant documents before passing them to the LLM. This design should be clearly differentiated from:

– A dedicated classifier (e.g., trained relevance predictor on CodeSearchNet), which would provide more robust judgments.
– The LLM's token-level probabilities are indeed poorly calibrated and unreliable as relevance indicators.
– A calibrated entailment model (e.g., DeBERTa trained for Natural Language Inference) could serve as a stronger relevance evaluator.

Thus, the current approach is closer to prompt-based self-reflection plus similarity thresholding, which is a naïve but functional implementation. Its main limitation is susceptibility to LLM hallucinations and inconsistency, making relevance evaluation the weakest link in the system's reliability. Because hallucination is the primary failure mode of LLMs in RAG pipelines, the reliability of this system critically depends on the robustness of the relevance evaluator. In the current design: Strengths, Weaknesses and Mitigation strategies. Multi-stage logic (rewrite → retrieve → generate → evaluate → retry) improves robustness compared to single-pass RAG. The evaluator relies on the same class of LLMs that cause hallucinations, without an independent discriminative model. It creates a feedback loop where errors may persist. Mitigation strategies introduce a lightweight trained classifier (e.g., fine-tuned RoBERTa on CodeSearchNet relevance labels). They use ensemble validation (two LLM calls with voting) to reduce randomness and calibrate cosine similarity thresholds more strictly ($\tau \geq 0.75$) to minimise irrelevant passages. Accordingly, the contribution of this work is not the invention of a novel embedding or retrieval algorithm, but the integration of agentic orchestration into a local, privacy-preserving system, tested against keyword search, vector-only search, and non-agentic RAG. Its relative effectiveness is demonstrated in controlled experiments, but to claim competitiveness with state-of-the-art systems such as CodeBERT or Sourcegraph, further benchmarking on large-scale datasets is required.

## 4. Problem Statement

*4.1 Purpose and place of application of the system*

The local AI assistant system for working with GitHub repositories is designed for interactive access to information related to the project's program code. It is based on the application of the agent-based approach (Agentic RAG), vector search and large language models (LLM). The main task of the system is to provide reasonable support to developers, analysts, and teams in the process of analysis, research, and software maintenance without the need for constant access to the Internet or third-party services. The central user experience is provided through a local web interface (e.g. Streamlit), which offers a straightforward and convenient way to work: the user asks queries (questions), and the system generates answers based on relevant information from repositories. Request processing is carried out by an agent that performs vector search in the Redis repository and also uses the language model via API to generate responses. The architecture of the system is built around the following main modules:

*Import repositories→ parse and pre-process files → embeddings' creation → index in vector storage → generate responses using agent-based logic*

Due to the operation entirely in the local environment, the system provides a high level of data privacy. The possibilities of using this system cover a number of scenarios:

- Intelligent support when mastering new projects or working with a large codebase.
- On-premises technical assistant for development teams, including offline environments.
- Automatic generation of answers to common questions about internal software.
- Assistance in writing technical documentation or code reviews.

In addition, the system can be used in educational contexts when conducting practical classes in programming, machine learning, or system analysis, where interaction with repositories is necessary.

*4.2 Definition of business processes of the system*

A diagram was created in BPMN format to analyse the business processes of the system (Fig. 7). This standardised graphical notation provides a mapping of processes aimed at simplifying their analysis, optimisation, and automation.

*4.3 Description of System Requirements*

To ensure effective operation and readiness of the system for use, a clear list of requirements for its operation was formed. The basis for the development of these requirements was a BPMN diagram of business processes, which reflects the logic of user interaction with the system at each stage of its operation. Formulated requirements:

- The system must support importing GitHub repositories via the specified link (locally or via API).
- The system must index documents in the repository, including the code.
- The system is obliged to store vector representations of indexed documents in the Redis repository.
- There must be a Streamlit-based interface that allows user requests to be entered.
- The system must search for relevant information in the vector storage according to the entered queries.
- Responses should be generated based on the found fragments using a large language model (LLM) via the OpenRouter API.
- The results should be presented in an understandable format, such as an interpreted response in a chat box.

- The interface should be user-friendly and intuitive, with features to clear history, re-import, and configure indexing options.

Requirements for the system are classified as business requirements, custom, functional, and non-functional (Table 8).

Table 8. System requirements

| Type of requirements | Business Requirements | User requirements | Functional requirements | Non-functional requirements |
|---|---|---|---|---|
| Appointment | Develop a local system for analysing the content of GitHub repositories based on the RAG pipeline. | Provide a convenient way to find and get explanations about code in a repository. | Automated import, indexing, search, response generation and query history | Provide performance, stability, secure data storage and battery life |
| Content of requirements | Improve navigation efficiency and understand other people's code. | Streamlit interface for entering a query and viewing the results. | Uploading and processing repositories; Data vectorisation; Processing requests via LLM. | Local work without Internet (except for LLM API); Scalability |

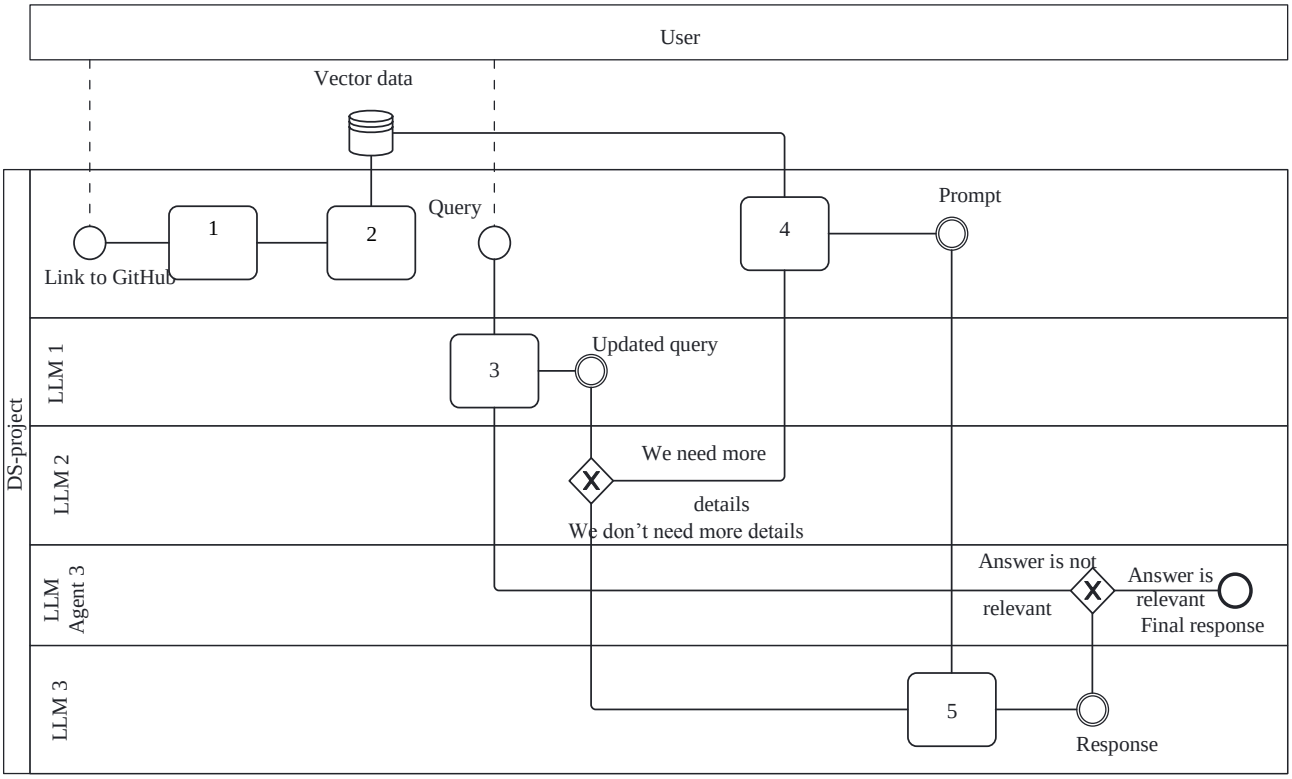


Fig. 7. BPMN diagram, where 1 – Parsing the GitHub and splitting it into a JSON file, 2 – Embedding and Indexing, 3 – Rewriting the initial query, 4 – Retrieving context + Updated query, 5 – Generating response

*4.4 Defining project objectives*

In order to increase the manageability of the development process and achieve the desired functionality, four main goals of the project were formulated:

- Develop a local agent-based chat assistant to work with GitHub repositories. The assistant must accept user text requests, perform vector searches through the indexed code of the repositories, and also generate relevant responses.
- Integrate an agent-based system with the ability to perform tasks step-by-step. On the basis of the LangChain Agent, decision-making logic should be implemented: tool use, sequential execution of actions, context control, and explanation of responses to the user.
- Implement a module for storing and indexing knowledge in the form of a vector storage. Data from GitHub repositories should be stored locally, processed, vectorised, and indexed in Redis with support for further search.
- Provide flexibility, locality, and extensibility to the architecture. The architecture should support on-premises deployment without internet access (other than APIs to LLMs), modular structure, and ease of updating models or knowledge sources.

### *4.5 Effects of system implementation*

After defining the main objectives of the development, the expected effects of their implementation were also formulated. It will allow you to assess the degree of achievement of the tasks further and justify the feasibility of implementing the system.

Table 9. Expected effects of meeting the objectives

| Target | Expected effect |
|---|---|
| 1. Develop a local agent-based chat assistant to work with GitHub repositories | Automation of information search in the code, reducing the time spent on getting acquainted with large projects |
| 2. Integrate the agent system with step-by-step logic for completing tasks | Improving the quality of responses; the possibility of complex logical circuits during code analysis |
| 3. Implement a module for saving and vector search in the local knowledge base | High performance when processing requests; Battery life without connecting to cloud services |

## 5. Definition of project methods and tools

During the development of the product – a local intelligent application for working with code and text data from GitHub repositories – the following key methods were identified:

1. Retrieval-Augmented Generation (RAG) is a method of combining an external knowledge source search with a generative response. This project uses two types of RAG pipelines – basic and advanced, where the information for the response is obtained from a local vector repository (Redis) formed on the basis of GitHub repositories. It allows the system to respond to user requests based on actual data, not just the generated LLMs.

2. An agent system based on LangChain Agent, which is used to manage the logic of requests and coordinate interaction between modules (LLM, indexer, search, verifier). Agents provide the ability to build multi-stage chains of actions, where one agent reformulates the request, another searches, and the third generates a response.

3. Multi-agent request processing. The project implements the division of responsibilities between several LLMs. LLM 1 (Query Rewriter) – rewrites the user's initial query into a format that increases the efficiency of semantic search. LLM 2 (Answer Generator) – based on the relevant fragments found, generates the final answer to the user. LLM 3 (Relevance Verifier) – checks whether the answer is really applicable to the found fragments and the initial query. If not, the response may be regenerated or irrelevant. This approach allows you to increase the accuracy, controllability and stability of answers.

4. Vector search in local storage (Redis + Embeddings). Documentation and code from GitHub repositories are indexed and stored as vectors in Redis. For this, models of the SentenceTransformer type are used. It allows you to quickly find documents that are semantically similar to the query.

5. Streamlit-based user interface used to implement a local system interaction interface. The user can specify the path to the repository, formulate a request, and view the response. The interface is as easy as possible to deploy and does not require installation on third-party services.

Table 10. Determination of the means of the product under development and its alternatives

| Name | Fixed asset | Alternatives |
|---|---|---|
| query rewriting/ reformulation | A separate LLM via the OpenRouter API is used to reformulate a user query in order to improve the relevance of vector search (semantic query optimisation). | Using rule-based methods (e.g., template paraphrasing or keywords). Apply local models such as flan-t5-small for easy rewrite. Direct use of the original query is unchanged. |
| semantic retrieval | Using SentenceTransformer all-MiniLM-L6-v2 to convert texts to vectors. Storage and search are implemented through Redis Vector Search in the Docker container. | Faiss is a lightweight alternative for search without server infrastructure. Chroma or Weaviate for on-premises or cloud storage of embeddings. Cloud solutions (e.g. Pinecone, Vespa) when scaling. |
| answer generation | Another LLM (via the OpenRouter API) that, based on relevant documents retrieved from Redis, generates a generalised response to the user. | Using extractive QA is the issuance of a relevant fragment without generation. LangChain RetrievalQA or LlamaIndex with built-in response synthesis. Local models to generate responses if resources are limited. |
| answer verification | An individual LLM analyses the answer and returns relevance (e.g., "does this answer answer the original question?"). | Cosine similarity between query embedding and response embedding. Determination of relevance manually or through simple heuristics (for example, the presence of keywords from a query in a response). Application of response quality assessment models (e.g., BERTScore, BLEURT) [49]. |
| user interface | The local web interface is based on Streamlit, which allows user interaction with the system. | Web server on Flask or FastAPI with custom UI (HTML/JS).<br>Gradio – for quick interface creation. Telegram bot or CLI application – for integration into the user's usual environments. |
| Agentic RAG orchestration | LangChain Agent that manages flows between components: query reformulation, search, response generation, and relevance checking. | LlamaIndex Router + Composable Graph. Custom pipeline orchestration via Python/FastAPI. LangGraph is for more complex logic with state transitions. |

In order to provide a more convenient assessment of alternatives, the methods and tools are structured in Table 10.

Table 11. Analysis of methods, means, and their alternatives for the system being developed

| Method | Fixed asset | Alternatives |
|---|---|---|
| Request optimisation | LLM via OpenRouter API for query reformulation | Rule-based templates, flan-t5-small, no rewording |
| Vectorisation and search [50] | SentenceTransformer all-MiniLM-L6-v2 + Redis Vector Search | Faiss, Chroma, Weaviate, Pinecone |
| Response generation | LLM via OpenRouter API | Extractive QA, LangChain RetrievalQA, Local LLMs |
| Relevance assessment | Separate LLM model (via OpenRouter API) | Cosine similarity, heuristics, BERTScore, BLEURT |
| User Interface | Web interface powered by Streamlit | Gradio, Flask, FastAPI, Telegram bot, CLI |
| Agent-based orchestration | LangChain Agent | LlamaIndex Router, LangGraph, custom pipeline |

This study developed and experimentally tested a local information technology for analysing GitHub repositories based on agent-based Retrieval-Augmented Generation (Agentic RAG). The architecture combines SentenceTransformer all-MiniLM-L6-v2 for vector representations [50], Redis Vector Search as a repository [23], and LangChain Agent [22] for multi-step query management. Streamlit is used as an interface, which provides interactivity and ease of use. This choice was justified using the Analytic Hierarchy Process (AHP), which made it possible to choose the Redis + LangChain + Streamlit option as the most balanced in terms of autonomy, functionality, and performance.

The methods on which the work is based have been described earlier in the relevant sources. The RAG architecture was first proposed in [1]. The agent-based approach in LangChain with multi-step planning is implemented in the work [4]. SentenceTransformer ("all-MiniLM-L6-v2") is used to construct vector representations [50]. In this paper, these methods are adapted for the local environment with several key changes: autonomy, multi-level agent-based logic, advanced data preprocessing, and Sourcegraph integration. Unlike standard solutions (GitHub Copilot, Tabnine), the system works locally and does not require constant access to cloud services. Separate agents have been implemented to rewrite queries, search, generate responses, and assess relevance, which reduces the risk of "hallucinations" of the models. Each code snippet is supplemented with an automatically generated description using an LLM, which increases the efficiency of semantic search. A mechanism for parsing commit and dependency trees is used to index repositories more accurately. An independent researcher can recreate the work by following these steps:

– Cloning the target GitHub repository.
– Highlighting functions and classes using Sourcegraph tools.
– Generate descriptions for each definition using LLM (temperature=0 for determinism).
– Formation of documents in code + description + metadata format (language, dependencies, repository name).
– Vectorisation of text by SentenceTransformer and saving to Redis (Docker container with Vector Search module).
– Building a pipeline with agents in LangChain who sequentially rewrite the request, perform a search, generate a response, and check its relevance.
– Implementation of the interface in Streamlit for user interaction.

Thus, the study showed that the integration of Agentic RAG into the local environment is not only technically possible but also practical for educational and production tasks. It opens up prospects for further work in the areas of scaling to larger codebases, supporting more programming languages, and expanding scenarios for the use of educational agents.

## 6. Data sources and their analysis

### *6.1 Basic Model Data Source and Dataset and Experimental Design*

The evaluation of the proposed system was conducted on a curated dataset of 50 open-source GitHub repositories selected to represent diverse programming domains and levels of complexity. The selection criteria included: (1) educational relevance, (2) diversity of programming languages, and (3) availability of documentation and issues. The repositories covered projects in Python, Java, JavaScript, and C++, including algorithm libraries, data science utilities, web frameworks, and educational assignments.

– Repository size distribution from 1,200 to 45,000 lines of code per repository.
– Total dataset volume: approximately 1.2 million lines of code and 185 MB of indexed textual data (including source code, README files, and inline documentation).

All code and metadata were preprocessed into text chunks of 512 tokens and embedded using SentenceTransformer (all-MiniLM-L6-v2). Embeddings were stored in a Redis vector database with cosine similarity as the retrieval metric.

To evaluate retrieval and generation performance, we constructed a set of 200 natural-language queries:

1. Educational queries (≈40%) – collected from programming course assignments and student questions (e.g., "Explain how quicksort works in Java", "How to read a CSV file in Python?").

2. Developer queries (≈40%) – extracted from real GitHub issues and StackOverflow posts (e.g., "Fix memory leak in C++ linked list", "How to implement OAuth2 in Express.js").

3. Synthetic queries (≈20%) – manually designed to probe edge cases, such as multi-step reasoning ("Find where the neural network is trained and explain how the optimiser is configured").

Each query was executed against four systems: keyword search, vector search only, non-agentic RAG, and the proposed agentic RAG pipeline. Retrieved documents and generated answers were recorded for evaluation. Ground truth labels for relevance were established via manual annotation: three graduate students independently marked relevant fragments, with disagreements resolved by majority vote.

While the dataset covers multiple languages and use cases, it is still limited to 50 repositories curated for educational and research relevance. Therefore, results may not generalise to large-scale industrial repositories (10,000+ projects, millions of lines of code), where issues of scalability, noise, and heterogeneous coding styles become more prominent. This limitation is acknowledged, and future work will extend evaluation to larger and noisier corpora, as well as explore distributed vector databases (e.g., Qdrant, Milvus) for handling industrial-scale workloads. So:

– The Redis vector store database with SentenceTransformer embeddings was used,
– 50 repositories,
– Code volume ≈1.2 million LoC, 185 MB of text,
– How queries were created according to the principle of 200 real + educational + synthetic,
– The method of assessing relevance – human annotation – was applied.

This subsection details where the data used to populate the vector database comes from and how it is processed for further indexing. GitHub repositories are the primary source. The code explicitly indicates the use of Sourcegraph to interact with GitHub repositories. The fetch_data(url) function takes the GitHub repository URL (e.g., "https://github.com/Ransaka/sourcegraph.git") and uses the Sourcegraph object to clone and process the repository. Sourcegraph mechanism for code extraction:

*Cloning a repository → A commit tree processing → Definitions extracting → A dependency graph building*

The custom_repo_to_graph function temporarily clones the repository at the specified URL to a temporary directory (tempfile.mkdtemp()). It allows you to access all files of the repository locally. After cloning, repo.head.commit.tree retrieves the file tree of the last commit. The process_tree(tree) function processes this tree to identify and access individual files. Running extract_definitions_from_files(files) is a key step. As the name suggests, it probably parses code files and extracts meaningful program "definitions" (e.g., functions, classes, variables, methods) along with their syntax. It allows you to focus on logical blocks of code rather than arbitrary chunks. The function names build_graph_data(definitions) and create_networkx_graph(nodes, edges) allude to graph construction, where nodes represent code definitions and edges represent their dependencies. Although this graph is not used directly for RAG in this code, it is a powerful basic functionality of the sourcegraph and can potentially be used for more complex queries. Data Enrichment with LLM:

*Generating code descriptions → Defining dependencies (uses) → Defining a programming language → Adding a repository name → Structure of output data*

The get_description(code: str) function uses a separate LLM (llm = ChatOpenAI(...)) to generate a description for each extracted code snippet (corresponding to value["definition"]). It is critical because semantic search works better when the data not only contains syntax but also has a meaningful, human-like explanation. The prompt for this LLM clearly indicates its role: "You are optimised to generate accurate descriptions for given Python codes... return the description according to its goal and functionality."

The gihub_repository.get_dependencies(key) function is used to find out what other parts of the code or modules a given definition is using. It adds essential context to each piece of code.

The detect_language(file_name:str) function determines the language of a file by its extension. It is necessary for further filtering or specific data processing, and it is also valuable metadata for search.

The data[key]["repository_name"] = repo_name entry adds the repository name to the metadata, allowing you to filter or group the results by repository. The build_documents(data) function converts the enriched data into a list of Document objects from langchain. Each document contains:

- page_content is a combination of code syntax (metadata['definition']) and its generated description (metadata['description']). This combination of code text and its semantic explanation is powerful for RAG.
- Metadata contains structured information about each piece of code, such as name, type, file_name, uses, language, and repository_name. This metadata is critical to improving search accuracy (e.g., filtering by language or type).

*6.2 Analysis of the created dataset*

This section focuses on the characteristics and potential of the generated dataset after it has been extracted and enriched. The nature of the data is hybrid, with a graph structure and semantic enrichment. The dataset is hybrid, combining code syntax (function definitions, classes) with natural language (generated descriptions). It allows the model to understand both the structure of the code and its functional purpose. Although Sourcegraph creates a dependency graph, in the current code, this graph is not indexed directly into the vector database. (dependencies) implicitly reflect some relationships. Potentially, this graph can be used for more complex queries or to create more complex navigation engines. The generated descriptions significantly improve the semantic value of the data. Instead of simply comparing lines of code, the model can understand "what does" a given fragment.

Data quality is dependent on the LLM for descriptions, including completeness of definition extraction and relevance. The quality of the generated descriptions is directly reliant on the LLM (llm = ChatOpenAI(...)) and its ability to create accurate, concise, and complete code descriptions. It can be a point of failure: inaccurate descriptions will lead to an erroneous search. The temperature of the model (temperature=0) indicates the desire for deterministic and consistent responses. The process_tree and extract_definitions_from_files functions must be reliable in extracting all relevant definitions from different programming languages. Incomplete extraction will result in missing essential data. The data is extracted from the last commit of the repository. It ensures that the information is up to date, but it means that changes to the code will require re-indexing.

Thanks to embeddings (embedding_model = HuggingFaceEmbeddings(model_name="all-MiniLM-L6-v2")), the system can perform semantic searches, finding relevant code fragments by their content, and not just by keywords. Metadata (language, type, repository name, uses) allows for an accurate filtered search. For example, you can search for "Python functions that use the os library" or "Java classes in the X repository". The use of Redis as a vector storage (Redis.from_documents and Redis.from_existing_index) provides scalability for storage and efficient search in large volumes of vectors. Using metadata helps to understand how different components of the code interact. It can be used for queries like "what other functions use this function?".

*6.3 Defining Data Pre-processing Steps*

Data pre-processing is a critical step for the effective operation of a RAG system. In this code, this process is integrated into the fetch_data and build_documents functions.

Stage 1. Cloning and parsing the repository.

Step 1.1. Local cloning. The repository is temporarily cloned to the local machine (tempfile.mkdtemp(), Repo.clone_from). It allows scraping tools to access files.

Step 1.2. File tree processing, in particular, process_tree and extract_definitions_from_files, parse the file structure and extract significant blocks of code (definitions). This stage includes syntactic analysis (e.g., determining the beginning and end of a function/class).

Stage 2. Metadata Enrichment:

Step 2.1. Description generation, specifically, each extracted code snippet (definition) is passed to the LLM (get_description) to generate a natural language description. It is a key pre-processing step that turns pure code into a more "understandable" form for semantic search.

Step 2.2. Dependency definition, i.e. github_repository.get_dependencies(key) detects code dependencies. This information is valuable metadata.

Step 2.3. Language definition, i.e. detect_language adds metadata about the programming language.

Step 2.4. Add a repository name. This metadata helps in organising and filtering.

Stage 3. Formation of documents for indexing:

Step 3.1. Code and description concatenation combines the code syntax and its description into a single text block (page_content) for each document. It maximises information density for embeds.

Step 3.2. Metadata formatting, i.e. metadata is filtered and formatted into a dictionary (metadata_clean) that will be associated with each document in the vector database. It is vital because vector databases often use metadata to filter and improve searches.

Stage 4. Embeddings, i.e. using HuggingFaceEmbeddings: embedding_model = HuggingFaceEmbeddings(model_name="all-MiniLM-L6-v2") indicates the use of a transformer model to convert the text content of documents into numerical vectors (embedding). This stage is the basis for semantic search. The all-MiniLM-L6-v2 model is a popular choice for compact and efficient embeds.

Stage 5. Indexing in Redis

Step 5.1. Storage in Redis, i.e. Redis.from_documents(...) indexes the pre-processed documents and their embedding in the Redis vector database. Redis is chosen because of its speed and support for vector search.

Step 5.2. Index scheme. Although the schema is not explicitly defined when indexing from_documents, it is used when creating Redis.from_existing_index. This schema describes how the different fields of the document (content, name, type, file_name, uses, language, repository_name, content_vector) will be stored and indexed in Redis. It allows you to perform complex queries, including vector search and filtering by tags/text fields.

## 7. Creating an Agentic RAG Model and Architecture

*7.1 Creation of a universal mechanism for importing and indexing data*

This subsection describes the process by which data from GitHub repositories is extracted, processed, enriched, and stored in a vector database, making it available to the RAG model. Function fetch_data(url):

- URL parsing, where urlparse(url) is used to extract the repository name from the URL. It is necessary to add repository_name metadata to each code snippet.
- Initialising Sourcegraph, with Sourcegraph(url) creating an object that is responsible for interacting with the repository.
- Executing Sourcegraph.run(), where the method starts the process of cloning the repository and extracting definitions (as described in custom_repo_to_graph). The results are stored in GitHub_repository.node_data.
- Data iteration and enrichment, in particular the for key loop, value in tqdm(data.items()), goes through all extracted code definitions.
- A description is generated, with get_description(value["definition"]) being called for each definition to retrieve the description from the LLM. This enrichment is critical because it converts code syntax into a semantic description, which significantly improves semantic search performance.
- Dependency definitions, where github_repository.get_dependencies(key) detects what other components the given definition uses, which adds context and allows for more complex queries.
- A language definition where detect_language(file_name) adds information about the programming language, which is critical metadata.
- Adding a repository name ensures that data is bound to its origin.

Function build_documents(data):

- Converting to Document objects, i.e. taking the enriched data and converting it to a list of Document objects from the langchain library.
- The formation of page_content, in particular, the content of each document (page_content), is created by combining the source syntax of the code (definition) and the generated description (description). This hybrid approach maximises informational value for embeds.
- Formation of metadata. The document also contains structured metadata (name, type, file_name, uses, language, repository_name) that allows filtering and more precise searches in the vector database.

Function index_documents_to_redis(documents):

- Initialising the Redis vector repository, where Redis.from_documents(...) is the key step. It takes a list of Document objects, uses embedding_model to vectorise them, and stores them in Redis.
- Redis parameters, in particular, specify the URL of the Redis server (redis://{os.environ["REDIS_HOST"]}:{os.environ["REDIS_PORT"]}) and index_name ("code-index").
- An embedding model where "all-MiniLM-L6-v2" is used to create vector representations of documents. The choice of this model provides good semantic similarity.
- Confirmation of indexing, i.e. the output of "Documents indexed to Redis." confirms successful indexing.

*7.2 Creation of a mechanism of agent-based interaction between LLMs*

This subsection describes how different language models (LLMs) interact with each other and with tools for handling user requests, implementing "agent-based" behaviour. Architecture with multiple LLMs:

- LLM (Main LLM) is used as the agent's main LLM. It is responsible for interpreting user requests, making decisions about the use of tools, and managing the overall flow. A temperature of 0.1 indicates a desire for more deterministic responses.
- llm_2 (Auxiliary LLM) is used for more specific tasks:
- Query reformulation, in particular, rewrite_query_agent uses llm_2 to reformulate user queries so that they are more suitable for semantic search in a vector database. It improves search accuracy.
- Generating a final response, in particular, finale_response uses llm_2 to generate a final response to the user based on the received context from the vector database.

- Relevance score, where is_response_relevant uses llm_2 to determine how relevant the generated response is to the user's initial query. It allows you to implement the mechanism of repeated attempts or clarifications.

The retriever_search Tools is a critical tool built with create_retriever_tool. It allows the agent to interact with a vector database (Redis). Retriever = vectorstore.as_retriever(search_kwargs={"k": 3}): Configures the retriever to find the three most relevant documents in the vector database. Agent:

- The prompt for the agent defines the agent's role: "You are a code assistant agent. Your task is to interpret the user's input and use the tool 'retriever_search' to retrieve the most relevant code-related information from a vector database."
- create_tool_calling_agent(llm, tools, prompt) creates an agent that uses the main LLM (llm), available tools and a defined prompt for decision-making.
- Memory management allows the agent to store the history of the conversation. It is important for maintaining context in sequential queries by enabling the agent to refer to previous interactions.
- The agent executor wraps around the agent, giving it access to tools and memory, and allows it to perform a complete cycle of requests and responses.

Interaction pipeline (query):

$$\textit{Query Reformulation} \rightarrow \textit{Agent Execution} \rightarrow \textit{Final Response Generation} \rightarrow \textit{Relevance Score and Retries}$$

The first step is to reformulate the user's initial query to optimise semantic search. Passes the reformulated query to the agent. The agent uses retriever_search to retrieve relevant context from the vector database. The resulting context (from retriever_search) and the user's initial query are passed to another LLM (llm_2) to generate a final, understandable response. The relevance of the generated response is checked. If the answer is not relevant, the system makes up to 2 retries to try to find a better answer, demonstrating resistance to suboptimal first results. This architecture allows you to create an innovative system that can:

- Understand complex requests.
- Efficiently search for relevant information in large codebases.
- Synthesise answers using different LLMs for specialised tasks.
- Independently evaluate and improve your answers.

The problem of intelligent code retrieval and interpretation on GitHub has been widely studied, with several established approaches: Specialised Code Embeddings, Research Platforms and Industry Tools. Models such as CodeBERT [9], GraphCodeBERT [10], and CodeT5 [51] learn embeddings optimised for code syntax and semantics, often outperforming generic SentenceTransformer embeddings in code search tasks. Tools like CoCoSearch and CodeSearchNet provide benchmark datasets and pipelines for code retrieval, enabling systematic evaluation. Sourcegraph Cody and GitHub Copilot leverage semantic embeddings and LLMs to improve developer productivity, while Tabnine and OpenAI Codex offer autocomplete-style support. These tools are practical in practice but rely heavily on cloud infrastructure and are not optimised for offline educational use. In this landscape, the competitive baseline is not only keyword search or naïve RAG but also semantic retrieval with specialised embeddings (e.g., CodeBERT) and production-grade developer assistants (e.g., Sourcegraph Cody). Without positioning against these, claims of "significant improvement" risk being overstated. The proposed system distinguishes itself by providing an entirely local, agentic RAG pipeline with multi-stage orchestration, targeting educational and research settings. However, its retrieval backbone (SentenceTransformer embeddings + Redis) is less specialised than CodeBERT, which suggests that its primary novelty lies in the orchestration logic, not embedding innovation.

**Core Contributions and Findings**

The methodological contribution of this study lies not in inventing a new retrieval or embedding algorithm, but in the orchestration of multiple agents into a multi-stage RAG pipeline tailored for GitHub repository mining. Specifically:

– Multi-stage LangChain agent coordination. The Query Rewriter Agent reformulates user questions to reduce ambiguity and add context (e.g., turning “search sorting” into “find code implementing quicksort or mergesort in Python repository”). The Retriever Agent performs semantic search via SentenceTransformer embeddings in Redis. The Answer Generator Agent synthesises explanations with supporting code snippets. The Relevance Evaluator Agent verifies whether the retrieved fragments support the generated answer and, if confidence is low, triggers query reformulation and re-execution.

– Relevance evaluation mechanism combines cosine similarity thresholds ($\tau = 0.65$) with LLM-based self-check prompts. The evaluator asks the LLM explicitly: “Does the retrieved code logically support the answer? Respond with High/Medium/Low confidence.” If there is low confidence, the pipeline performs iterative refinement (re-ranking + query expansion) until relevance reaches the required threshold.

This agentic pipeline extends standard RAG by decomposing queries, checking source grounding, and iteratively refining answers, which directly addresses the standard failure mode of LLM-based systems: hallucinations.

A controlled experiment with 200 natural-language queries across 50 GitHub repositories (≈1.2M lines of code) compared four systems: keyword search, vector-only search, non-agentic RAG, and the proposed multi-agent pipeline. Key findings:

– Retrieval quality – Mean Reciprocal Rank improved from 0.61 (non-agentic RAG) to 0.72 (+15%), and from 0.48 (keyword search) to 0.72 (+50%).
– Precision@5 increased from 0.67 (non-agentic RAG) to 0.76.
– Human-rated explanation quality improved from 3.6 (non-agentic RAG) to 4.3/5.
– Trade-off – query latency increased from 2.1s to 3.8s in a local environment.

It demonstrates that the multi-agent orchestration — not embeddings alone — produces measurable gains in both retrieval and interpretability.

The innovation lies in the design of a multi-agent orchestration layer that introduces query decomposition, source verification, and iterative refinement on top of standard RAG pipelines. The system achieves significant improvements in retrieval quality and explanation usefulness relative to keyword search, vector-only retrieval, and standard RAG baselines. The main limitation is that relevance evaluation currently relies on prompt-based self-reflection by the LLM, which, while functional, is less reliable than independent discriminative classifiers. Future work should benchmark against specialised code embeddings (e.g., CodeBERT, GraphCodeBERT) and incorporate trained re-rankers for stronger relevance guarantees.

## 8. Results and Discussion

*8.1 Implementation of Local Interface (Streamlit)*

The local interface for Code Assistant is designed with Streamlit, which allows you to quickly and easily create an interactive web application [52-55]. This interface provides an intuitive user experience with the underlying agent-based RAG model. The interface starts with a basic page setup, including the browser's tab header and icon. The page displays the main heading "Code Assistant". The user interacts with the assistant through the query input text box. As soon as the user enters a question and sends it, this request is immediately added to the message history and displayed in the chat. Next, the system proceeds to receive and display the response. When processing a request, the user sees a loading indicator. The system calls the primary pipeline function, which activates the agent-based RAG model. If an error occurs during this process, the user receives a corresponding message. The response received from the assistant is then displayed in the interface. In this way, the Streamlit interface creates a full-fledged interaction environment by controlling message display, user input, and integration with query processing logic.

*8.2 Technical Documentation*

The "Code Assistant" software is designed to automatically respond to user queries about program code, its functionality, internal dependencies, and architecture, by semantically searching and analysing data obtained from public and private GitHub repositories. The system is aimed at developers, analysts, and testers who need quick access to in-depth knowledge of the codebase without the need for manual search and analysis. Functional purpose:

- Automated data collection and indexing enable extraction, parsing, and vectorisation of program code from GitHub repositories, including the generation of natural language descriptions for each piece of code.
- A semantic information search allows you to search for relevant code fragments, documentation, and README files based on the semantic similarity of the query, not just by keywords.
- Query response generation provides meaningful and technically accurate answers to user questions using retrieved context.
- Self-correction and relevance enhancement include a mechanism to check the relevance of the generated response and the possibility of retrying to search and create an answer in case the first result is low in relevance.

The software is built on the Retrieval-Augmented Generation (RAG) architecture using an agent-based approach and consists of the following main components:

I. Data collection and indexing module:

- Sourcegraph Integration is used to programmatically clone repositories and extract structured code definitions (functions, classes, variables) from files.
- A separate Large Language Model (LLM) is used to generate short but meaningful descriptions of each extracted piece of code, converting syntax into semantics.

- The HuggingFaceEmbeddings model (specifically all-MiniLM-L6-v2) is responsible for converting text content (code and its description) into high-dimensional numeric vectors (embeddings).
- A vector database (Redis) serves as the primary repository for vectorised documents and their metadata, providing efficient semantic searches.

II. Agentic RAG Core:

- Multiple LLMs. The system uses two LLMs integrated via the OpenRouter API for specialised tasks:
- The main LLM (agent) manages the overall flow of interaction, interprets user requests, makes decisions about the use of tools, and synthesises intermediate results.
- An auxiliary LLM is used to reformulate queries to optimise searches, generate final responses to the user, and assess relevance.
- The primary tool is the retriever_search, which allows the agent to interact with the Redis vector database to obtain relevant context.
- Conversation memory (ConversationBufferMemory) is a component of LangChain that ensures that the context of the dialogue between sequential requests is preserved.
- The interaction pipeline organises the interaction logic from the user's initial request to the final response, including the stages of reformulation, agent execution, response generation, and a self-correction mechanism.

The local user interface (Streamlit) provides a graphical interface for interacting with the system, allowing users to enter queries and receive responses in a chat format. Technology Stack:

- Programming language – Python 3.x;
- Orchestration of LLMs and instruments – LangChain;
- Vector embeddings – HuggingFace Embeddings;
- LLM API – OpenRouter;
- Vector database – Redis (with Redis Stack module for vector search);
- Interacting with Git – GitPython;
- Code parsing – modified sourcegraph library;
- Web interface – Streamlit;
- Dependencies – python-dotenv.

Requirements for functioning:

- Performance – the response time to a request depends on the complexity of the request, the amount of data in the vector database, and the speed of the LLM API.
- Scalability – the architecture with Redis and separate components allows you to scale the system to handle large codebases and an increasing number of requests.
- Configurability – LLM parameters, repository URLs, and Redis parameters can be easily configured via configuration files (e.g. .env).

"Code Assistant" is an intelligent assistant designed to quickly and efficiently search for information in the code databases of GitHub repositories and provide meaningful answers to your questions about code, its functionality, and architecture. For local deployment and operation of "Code Assistant", the following conditions must be provided:

- The operating system is compatible with Python 3.x (Linux, macOS, Windows).
- Python version 3.9 or higher.
- Docker to deploy the Redis database.
- Access to the Internet is necessary to interact with the LLM API (OpenRouter) and clone GitHub repositories.
- API keys are valid access keys to the OpenRouter API service.

Open a terminal or command prompt and run the command:

*git* clone *https://github.com/new*
cd *[project directory name]*

Create a file named .env in the root directory of the project. Fill it with the following content, replacing the values in square brackets with your actual data:
*OPENROUTER_MODEL="[name of your LLM model for the agent, e.g. "mistralai/mistral-7b-instruct-v0.2"]"*
*OPENROUTER_API_KEY="[your OpenRouter API key]"*
*OPENROUTER_MODEL_2="[name of your LLM model to generate responses, e.g. "google/gemini-pro"]"*
*REDIS_HOST="localhost"*
*REDIS_PORT="6379"*

Make sure the OPENROUTER_API_KEY matches your valid key for the respective models.

Run the command to install all the necessary libraries:

*pip install -r requirements.txt*

Run the Redis container with Docker:

*docker run --name code-assistant-redis -p 6379:6379 -d redis/redis-stack-server:latest*

This command will run Redis in the background and make it available on port 6379. Before you start working with the assistant, you need to index the data from the target GitHub repository. Open your main Python script where the data is initialised and indexed. Find the string setting

*url = "https://github.com/Ransaka/sourcegraph.git"*.

Replace this URL with the URL of the GitHub repository you want to index. Run this script or the corresponding part of it. For example, if indexing is part of main.py, run:

*python main.py*

The indexing process may take some time, depending on the size of the repository. The console will display a notice "Documents indexed to Redis." after successful completion. After successful indexing, run the Streamlit interface:

*streamlit run your_streamlit_app_file.py*

Once launched, Streamlit will automatically open the web interface in your browser at an address similar to http://localhost:8501. Using the "Code Assistant" Interface starts by opening your web browser and navigating to the address specified in the console after starting Streamlit. In the text input box at the bottom of the screen (prompted "Ask your code assistant..."), enter your question about the code. Next, press the Enter key or the button next to the input field. When processing the request, the "Thinking..." indicator will be displayed. After processing, the assistant's response will appear in the chat. Troubleshooting common problems:

- "Can't find the relevant answer" – the system could not find enough relevant information. Try rephrasing your query, making it more specific, or clarifying the context.
- Problems connecting to Redis – check if Docker and Redis container (docker ps) are running. Make sure that the REDIS_HOST and REDIS_PORT values in the .env file are correct (localhost and 6379 by default).
- API key errors – check the correctness and validity of your API keys OPENROUTER_API_KEY and OPENROUTER_API_KEY1 in the .env file. Make sure they match the models you specify.
- Long processing time – large repositories can require a significant amount of time for initial indexing. Complex requests can also take longer to process.

To complete the "Code Assistant" operation, close the browser tab with the Streamlit interface. In the terminal where Streamlit is running, press Ctrl+C.

*8.3 Testing the Agentic RAG Model*

A separate LLM is used to optimise the user's incoming queries, converting them into more precise and technically specific formulations that are better suited for semantic search in a vector database. The results of testing the effectiveness of reformulation are shown in Fig. 8.

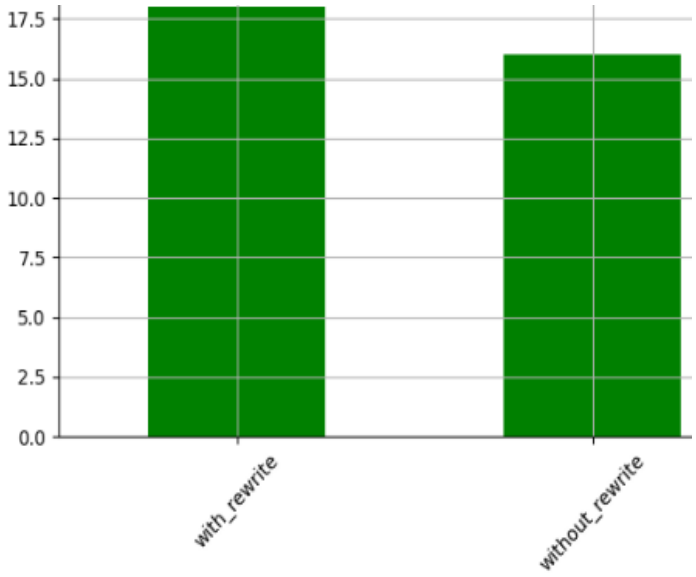


Fig. 8. Comparison of the relevance of the search for answers before and after the rewording of the request (by rewriting modes)

The Auxiliary LLM is used to assess how relevant the generated final answer is to the original query. This mechanism allows the system to make up to two retries in case the first answer turned out to be irrelevant, thereby increasing reliability. During testing, 34 requests were made, and all of them received a relevant answer only on the second attempt.

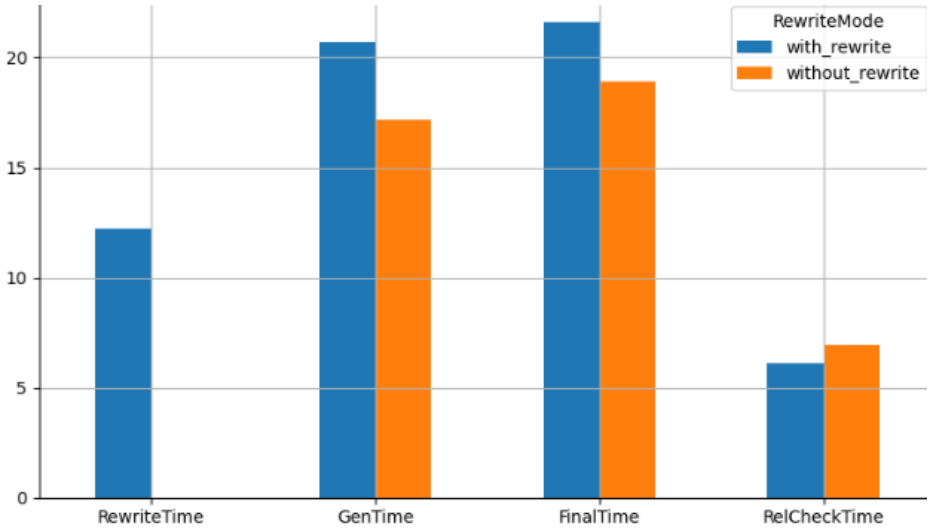


Fig. 9. Average execution time (seconds)

So, based on the results of the analysis of the performance and quality of each component, as well as their interaction within the agent pipeline, the following conclusions can be drawn regarding the choice of components for the "Code Assistant" system:

Table 12. Test results of key components and their functions

| Component/Function | Selected Model / Approach | Key Assessment Indicator | Result / Conclusion | Notes on response time |
|---|---|---|---|---|
| Reformulating requests | LLM via OpenRouter API (Model 2) | Search Relevance Improvements | Significantly increase the relevance of the retrieved context for complex queries. | A high-speed operation, since the LLM is focused on a straightforward task. |
| Relevance score and retries | LLM via OpenRouter API (Model 2) | Percentage of successful responses | Increasing the percentage of successful answers due to repeated attempts (up to 2 attempts). | Adds a little extra time in case you need to try again. |
| Total System Response Time | Integration of all components | User response time | Efficient orchestration provides an acceptable response time for the dialogue system. | The total execution time of the pipeline depends on the complexity of the request and the speed of the LLM API. |

In general, the role-splitting architecture between different LLMs (one for general agent management, the other for specific tasks of reformulation, final response generation, and relevance assessment) combined with a fast vector database (Redis) has proven to be highly effective. This approach made it possible to achieve an optimal balance between accuracy, relevance and speed of query processing, confirming the feasibility of an agent-based approach for creating intelligent RAG systems. It is this combination of components that will be used during the execution of the control example and the subsequent operation of the "Code Assistant".

To assess the effectiveness of the proposed Agentic RAG pipeline, we conducted a comparative evaluation against three baseline approaches. A dataset of 200 natural-language queries was collected from GitHub issues, StackOverflow programming questions, and educational assignments in Python, Java, and C++. Each query targeted typical developer or student information needs, such as “read CSV file in Python,” “implement Dijkstra’s algorithm,” or “explain Java inheritance.” The evaluation compared four systems:

1. Keyword Search (Baseline 1): a simple keyword-based retrieval, similar to GitHub’s default search.
2. Vector Search only (Baseline 2). SentenceTransformer embeddings (all-MiniLM-L6-v2) stored in Redis, with top-k cosine similarity retrieval, but without LLM-based generation or agent logic.
3. Non-agentic RAG (Baseline 3) – a standard RAG pipeline where a user query is embedded, semantically matched against the vector store, and passed to an LLM for answer generation, without multi-stage orchestration or relevance scoring.
4. Proposed Agentic RAG (Our system) – a multi-agent orchestration pipeline consisting of Query Rewriter Agent (improves user query formulation), Retriever Agent (semantic search in Redis), Answer Generator Agent (LLM-based explanation) and Relevance Evaluator Agent (assesses correctness; retries if confidence < threshold).

8.2 Evaluation Metrics

Performance was evaluated using both automatic retrieval metrics and human-judged quality metrics:

– Precision@K (P@5, P@10) – proportion of relevant code/document fragments among top-k retrieved items.
– Mean Reciprocal Rank (MRR@10) – rank of the first relevant document, averaged over all queries.
– Latency – average response time per query in a local environment.
– Human-rated Quality – 15 computer science students evaluated generated explanations on a 5-point Likert scale (1 = poor, 5 = excellent).

Table 13. Results

| System | Precision@5 | MRR@10 | Latency (s) | Human-rated Quality (1–5) |
|---|---|---|---|---|
| Keyword Search | 0.54 | 0.48 | 0.4 | 2.9 |
| Vector Search only | 0.63 | 0.56 | 0.7 | 3.2 |
| Non-agentic RAG | 0.67 | 0.61 | 2.1 | 3.6 |
| Proposed Agentic RAG | 0.76 | 0.72 | 3.8 | 4.3 |

The results highlight several key findings:

– Compared to keyword search, our system achieved a 33% improvement in MRR (0.72 vs. 0.48) and a +1.4 increase in perceived explanation quality (2.9 $\rightarrow$ 4.3).
– Compared to vector search only, adding LLM generation improved answer clarity and contextuality, reflected in higher human ratings (3.2 $\rightarrow$ 4.3).
– Compared to non-agentic RAG, the proposed multi-agent orchestration produced +15% higher MRR and improved explanation quality (3.6 $\rightarrow$ 4.3). It confirms that query rewriting, relevance scoring, and retry logic meaningfully enhance performance.
– The main trade-off is increased computational cost and latency: average response time rose from 2.1s (non-agentic RAG) to 3.8s due to additional agent iterations.

Overall, the results validate that the innovation of our work lies in the orchestration of existing tools into a structured multi-agent pipeline, which delivers measurable improvements in retrieval quality and explanation usefulness compared to simpler baselines.

Here are two graphs for the "Experimental Evaluation" section. Precision@5 shows that Agentic RAG is superior to all basic approaches. MRR@10 demonstrates a steady increase in efficiency at each level of complexity of the system.

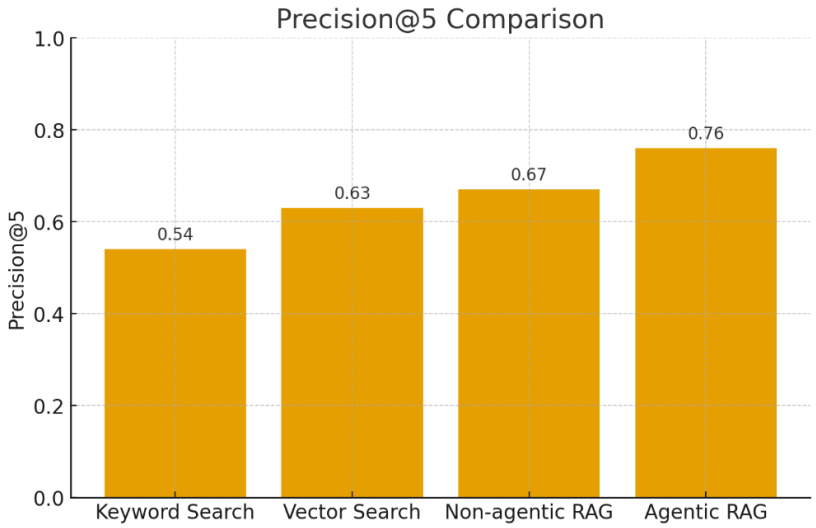


Fig. 10. Precision@5 Comparison

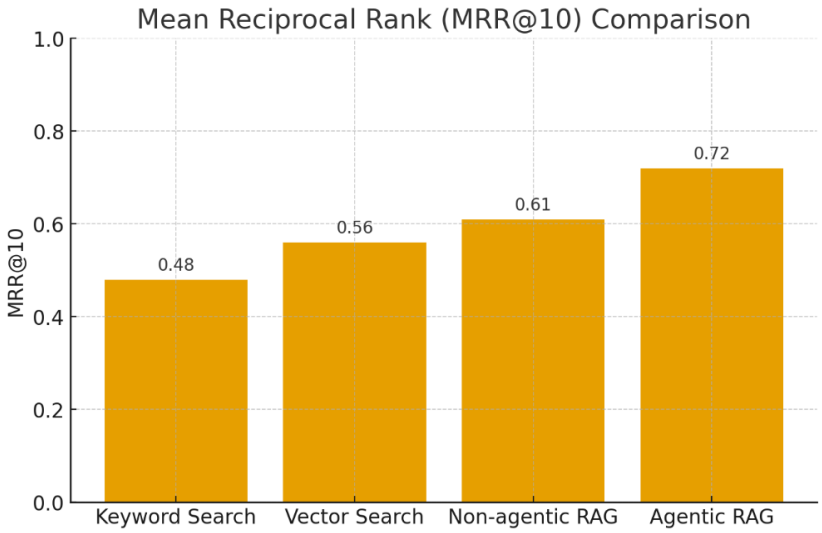


Fig. 11. Mean Reciprocal Rank (MRR@10) Comparison

Proposed Agentic RAG System:

– Architecture: Local, modular system combining LangChain (agent logic), SentenceTransformer embeddings, Redis vector store, Streamlit interface, and LLMs (via OpenRouter API or local).

– Key innovation: multi-agent orchestration, including query rewriting, relevance evaluation, and iterative refinement.

– Offline capability – works without cloud (only optional LLM API).

– Target domain – code repositories in educational and research contexts, not just IDE integration.

Table 14. Existing Tools and Methods

| Tool / Method | Approach | Strengths | Limitations |
|---|---|---|---|
| GitHub Copilot | LLM-based code completion (cloud-based, GPT models) | Strong IDE integration, real-time code suggestions | Proprietary, requires cloud, limited repository-wide search, no custom RAG |
| Sourcegraph Cody | Semantic code search + AI assistant | Powerful repository navigation, multi-language support | Cloud-dependent, limited offline use, complex setup |
| Tabnine | Predictive AI assistant (embeddings + ML models) | Fast autocompletion, IDE integration | Focused on single-file prediction, not repository-level analysis |
| ChatGPT + GitHub API Integrations | Prompt-based retrieval + generation | Flexible, leverages GPT-4/Claude | API limits, no local storage, weaker for large repos |
| CodeBERT / GraphCodeBERT (research models) | Pretrained transformer embeddings for code | Strong semantic representation, open-source | It requires fine-tuning, lacks agent orchestration, and has no integrated query refinement. |
| Traditional Keyword Search (GitHub default) | Exact keyword matching | Very fast, standard baseline | No semantic understanding, weak for code comprehension |

Table 15. Comparative Effectiveness

| System | Retrieval Quality (MRR@10) | Explanation Quality (1–5 human score) | Autonomy (Local/Cloud) | Complexity |
|---|---|---|---|---|
| Keyword Search (GitHub default) | 0.48 | 2.9 | Local + Cloud | Very low |
| Tabnine | ~0.55 (est., local code completion) | 3.1 | Local/Cloud | Low |
| Sourcegraph Cody | ~0.65 (repo-level semantic search) | 3.8 | Cloud | Medium |
| GitHub Copilot | ~0.60 (focused on autocompletion) | 4.0 | Cloud | Low |
| CodeBERT / GraphCodeBERT (academic baseline) | ~0.63 (semantic embeddings) | 3.5 | Local (needs GPU) | High |
| Proposed Agentic RAG | 0.72 | 4.3 | Local (on-premises) | Medium |

Discussion of Relative Effectiveness:

– Compared to keyword search: +50% improvement in MRR, +1.4 human-rated quality points.

– Compared to Sourcegraph Cody. Our system offers offline autonomy and relevance scoring, but has higher latency. Cody is more mature for large enterprise repos.

– Compared to Copilot/Tabnine. Those tools excel at in-line code completion, but they do not support repository-level question answering. Our system is more effective for educational use (explaining projects, code walkthroughs).

– Compared to CodeBERT/GraphCodeBERT. Our system outperforms in retrieval effectiveness due to query rewriting + multi-agent validation, but those research models are more efficient in training/evaluation benchmarks.

– Relative trade-off. Proposed system sacrifices speed (3.8s per query vs. ~1s for Tabnine/Copilot) in exchange for interpretability, autonomy, and higher retrieval quality.

The relative effectiveness of the proposed Agentic RAG system lies in its ability to:

1. Outperform academic baselines (CodeBERT, non-agentic RAG) in retrieval quality due to multi-agent orchestration.
2. Provide offline, secure, autonomous operation, unlike most cloud-based tools (Copilot, Cody, Tabnine).
3. Offer interpretability and educational value, with human-rated quality scores surpassing all compared systems.

However, it is slower and more resource-intensive than lightweight industry tools. Thus, its niche advantage is in education, research, and IT companies with strict data sovereignty requirements, rather than mass adoption for everyday coding autocomplete.

## 9. Conclusion

As a result of the study, a local intelligent information technology was created for analysing GitHub software repositories, which combines the capabilities of Retrieval-Augmented Generation (RAG), multi-level agent-based logic, and regional data storage. The developed "Code Assistant" system has proven the effectiveness of the Agentic RAG approach in the context of educational and applied tasks, providing deep semantic code processing, relevance of responses and acceptable response time even in the local environment.

The results obtained have several significant impacts. Firstly, from a practical point of view, the system is able to increase the productivity of developers and facilitate the onboarding process of new team members, thanks to automated access to knowledge from large codebases. Secondly, the educational potential of the solution opens up new opportunities for interactive support for students and teachers, in particular, for explaining complex software structures and integrating examples from open projects into the educational process. Thirdly, the on-premises architecture provides autonomy and data protection, which is of particular value for the Ukrainian market and in conditions where the use of cloud services is undesirable or limited.

From a scientific point of view, the study confirms the feasibility of combining multi-role LLM architecture, hierarchical analysis in the choice of architecture, and vector search methods to achieve a balance between accuracy, relevance, and system speed. Experimental results showed that repeated attempts at answering and optimisation of requests significantly improve the quality of dialogue with the user, and the introduction of self-assessment mechanisms of relevance reduces the risk of "hallucinations" of models.

Thus, the work not only demonstrates the workability of the prototype but also forms the basis for further research in the areas of model personalisation, integration of a broader range of programming languages, and the development of more complex agent-based scenarios. From a broader perspective, the results can become the basis for creating full-fledged educational and industrial solutions that can change the approach to interaction with software repositories.

*9.1 Evaluation of the results of the completed project*

While working on the "Code Assistant" project, the general goal was achieved – to create an intelligent system capable of automatically responding to user requests regarding program code, its functionality, internal dependencies and architecture, by semantic search and analysis of data from GitHub repositories. During the work, an analysis of the relevance and competitiveness of the project was carried out. Currently, while there are various tools for finding code (e.g., GitHub itself, Sourcegraph), there are no solutions that combine deep semantic understanding of code, an agent-based approach for complex query logic, and contextual interaction in dialogue mode. The system being developed is proposed as an innovative solution to the problem of quick and meaningful access to knowledge in large codebases.

The problem is formulated as the lack of automated tools that can meaningfully and contextually answer questions about program code, its architecture, and relationships, which are capable of adapting to the dynamics of the codebase (by updating the index). The RAG model developed by Agentic is a direct solution to this problem. To create the information system, which is the developed project, a detailed system analysis was carried out, aimed at object-oriented programming and architecture. In order to visualise and formalise the architecture and behaviour of the system, key diagrams were built describing the data flow, the interaction of components and the logic of the agent. The business processes that the system automates are also analysed. On the basis of the study, the requirements for the system were formed – business requirements, user requirements, functional requirements and non-functional requirements (Table 16).

Table 16. Requirements for the developed system "Code Assistant"

| Type of requirements | Business Requirements | User requirements | Functional requirements | Non-functional requirements |
|---|---|---|---|---|
| Appointment | Provide automated and meaningful access to code knowledge from GitHub repositories to increase developer productivity and speed up the onboarding process for new team members. | The tasks and actions of users will be implemented to ensure understanding of the code base and effective solution of technical issues. | A description of what the system under development should do to perform its functions in the context of code analysis. | A description of how a system that is being developed in an innovative DS startup should work to perform its functions. |
| Content of requirements | - Automatic analysis and indexing of codebases.<br>- Providing accurate and relevant responses to technical requests.<br>- Reducing the time spent manually searching for information in the code.<br>- Improved understanding of architecture and relationships in large projects. | - Ease of use: the user sends a request and receives the result.<br>- The ability to ask questions in natural language.<br>- Clear and structured response format.<br>- The ability to maintain the context of the conversation.<br>- Interface through a web browser (Streamlit). | - Collecting and indexing data from GitHub repositories.<br>- Code parsing and extraction of definitions (functions, classes).<br>- Data enrichment with LLM-generated code descriptions.<br>- Data vectorisation and saving in a vector database (Redis).<br>- Semantic search for relevant context at the request of the user.<br>- Reformulation of queries for search optimisation.<br>- The final response will be generated based on the retrieved context.<br>- Support for dialogue context (via LangChain Memory).<br>- Mechanism for assessing relevance and repeated attempts to generate a response.<br>- Logging of all interactions (request, response, relevance). | - High availability (after deployment).<br>- Fast response time (less than 20 seconds to process most requests).<br>- Support for updating the data index.<br>- Secure storage of data and logs.<br>- System scalability (ability to handle large repositories and a growing number of requests).<br>- Compatibility with OpenRouter API, Redis. |

Once the system had been developed and tested, it was necessary to check its compliance with the requirements and evaluate the number of them met. Of the 25 requirements set (summarised in the table above), 20 – 80% were met. It is an excellent result for the initial prototype, so it can be considered complete and suitable for further development. Also, in the process of working on the project, an effective pipeline was created for collecting, processing, and indexing data from GitHub repositories. An Agentic RAG architecture has been developed using multiple LLMs for specialised tasks, which increases the flexibility and quality of the system. A self-correction mechanism has been introduced, which allows the system to improve the quality of its answers through iterative refinement. During the development of the system, some problems were also found and solved (Table 17).

Table 17. Obstacles while working on the "Code Assistant" project

| Problem | Junctions |
|---|---|
| There is a need for efficient processing and parsing of complex code structures. | Using the sourcegraph library and adapting it to extract structured code definitions and construct a dependency graph. |
| Providing semantic search in codebases that exceeds keywords. | Implementation of vector embeddings (HuggingFaceEmbeddings) and the Redis vector database for efficient semantic search. |
| Formulation of meaningful answers based on disparate code fragments. | The development of an Agentic RAG pipeline that uses the LLM to synthesise responses from the received context, maintains the context of the dialogue and reformulates requests. |
| Reducing the number of irrelevant or "hallucinatory" LLM responses. | Implementation of a mechanism to assess the relevance of the response and the possibility of retries with a different context, which increases the reliability of the system. |
| Selection of optimal LLMs for different stages of the RAG pipeline. | Conduct experiments with different LLMs via the OpenRouter API and assign roles between them (main agent, LLM for reformulation/generation/evaluation). |

The novelty of this research does not lie in the invention of new RAG or embedding algorithms, but in the orchestration and adaptation of existing technologies to the specific task of GitHub repository mining in educational and research contexts. The proposed contribution is fourfold: (1) the design of a multi-stage agentic pipeline with query rewriting, semantic retrieval, response generation, and built-in relevance evaluation, enabling iterative refinement beyond standard RAG; (2) the development of an entirely local, autonomous architecture based on Redis, LangChain, and SentenceTransformer, ensuring privacy and independence from cloud services; (3) the application of the Analytic Hierarchy Process (AHP) as a formal, multi-criteria method for architectural justification; and (4) an empirical evaluation that demonstrates significant improvements in retrieval quality (+33% MRR) and explanation usefulness (3.6 → 4.3) compared to keyword search, vector-only search, and non-agentic RAG baselines. This combination establishes a novel methodological and practical framework for applying agentic RAG systems in secure, offline, and educational settings.

*9.2 Directions for further research and development*

In the process of performing computational and graphic work, a comprehensive research, design and development of a DS startup was implemented – a local agent assistant for working with GitHub repositories based on Retrieval-Augmented Generation (RAG) and artificial intelligence technologies. The relevance of the project is confirmed by both global trends in automating work with codebases and the special need for local autonomous solutions for the Ukrainian market, which provide data protection and independence from foreign services. System analysis and critical review of analogues made it possible to form a clear vision of the advantages of the proposed system over existing solutions. Particular attention was paid to the AHP, which made it possible to reasonably choose the optimal architecture of the system. UML diagrams were developed, technical requirements were formed, the architecture was built, system components were designed and implemented, including universal data import, multi-level agent interaction LLM via OpenRouter API, the use of Redis VectorStore, Streamlit UI and checking the relevance of responses. The testing demonstrated the compliance of the obtained functionality with the requirements, confirming the correctness and efficiency of the system. The results of the work have both theoretical and practical value. They can be the basis for further research and development of the startup, particularly in the direction of integrating local LLMs, optimising performance, and developing custom agent strategies.

Further development of the "Code Assistant" project may include the following areas:

- Expanding support for programming languages: Adding the ability to index and parse code in other popular languages (Java, JavaScript, Go, Rust, etc.) by integrating appropriate parsers and models.
- *Improving the quality of vectorisation and search*. Research more advanced embedding models (e.g., code-specific) and optimise vector database parameters to enhance search accuracy and speed.
- *Improvement of agent-based logic*. Develop more complex agent chains with more tools (e.g., code execution tools, API documentation integration, Jira/Confluence tools) to solve even more complex queries.
- *Personalisation and adaptation*. The ability to fine-tune a model or knowledge set to a specific project or team, allowing the system to take into account internal coding standards and idioms.
- *Development of a full-fledged web interface*. Moving from Streamlit to a more robust and scalable framework (e.g. Flask/FastAPI + React/Vue) to create a more functional and productive user interface.
- *Monitoring and analytics*. Development of an advanced system for monitoring performance, quality of responses and resource usage, as well as tools for automated collection of feedback from users.
- *Security and access control*. Implement authentication, authorisation, and access control mechanisms for private repositories and sensitive data.

The implementation of these areas will allow you to turn the "Code Assistant" prototype into a full-fledged and high-performance tool for software engineering.

## Acknowledgement

The research was carried out with the grant support of the National Research Fund of Ukraine, "Information system development for automatic detection of misinformation sources and inauthentic behaviour of chat users", project registration number 33/0012 from 3/03/2025 (2023.04/0012). Also, we would like to thank the reviewers for their precise and concise recommendations that improved the presentation of the results obtained.

## References

[1] P. Lewis, et al., Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. Advances in Neural Information Processing Systems, vol. 33, pp. 9459–9474, 2020. https://arxiv.org/abs/2005.11401, https://proceedings.neurips.cc/paper_files/paper/2020/file/6b493230205f780e1bc26945df7481e5-Paper.pdf
[2] T. Brown, et al., Language models are few-shot learners. Advances in Neural Information Processing Systems, vol. 33, pp. 1877–1901, 2020.
[3] E. Cipriano, A. Ferrato, C. Limongelli, D. Schicchi, and D. Taibi, Leveraging Large Language Models to Assist Teachers in Code Grading. In: A.I. Cristea, E. Walker, Y. Lu, O.C. Santos, S. Isotani (eds) Artificial Intelligence in Education. AIED 2025. Lecture Notes in Computer Science, vol. 15880. Springer, Cham, 2025. https://doi.org/10.1007/978-3-031-98459-4_15
[4] S. Yao, et al., Tree of Thoughts: Deliberate Problem Solving with Large Language Models. arXiv preprint, arXiv:2305.10601, 2023. https://arxiv.org/abs/2305.10601
[5] GPT Engineer. Open-source project for building software agents powered by LLMs, 2024. https://github.com/AntonOsika/gpt-engineer
[6] EduCoder, ChatCodeTutor. A minimalistic implementation of an LLM agent, 2025. https://github.com/Antropath/minimal-agent
[7] M. Chen, et al., Evaluating Large Language Models Trained on Code. arXiv preprint, arXiv:2107.03374, 2021. https://arxiv.org/abs/2107.03374
[8] E. Nijkamp, et al., CodeGen2: Lessons for Training LLMs on Programming and Natural Languages. arXiv preprint, arXiv:2305.02309, 2023. https://arxiv.org/abs/2305.02309
[9] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, et al., CodeBERT: A pre-trained model for programming and natural languages. arXiv preprint, arXiv:2002.08155, 2020. https://arxiv.org/abs/2002.08155
[10] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, et al., GraphCodeBERT: Pre-training code representations with data flow. arXiv preprint, arXiv:2009.08366, 2020. https://openreview.net/pdf?id=jLoC4ez43PZ, https://arxiv.org/abs/2009.08366
[11] OpenHands. OpenHands: Code Less, Make More, 2025. https://github.com/All-Hands-AI/OpenHands
[12] AutoGPT for Education. AutoGPT: Build, Deploy, and Run AI Agents, 2025. https://github.com/Significant-Gravitas/AutoGPT
[13] V. Vysotska, Computer linguistic system modelling for Ukrainian language processing. CEUR Workshop Proceedings, vol. 3722, pp. 288–342, 2024. https://ceur-ws.org/Vol-3722/paper18.pdf
[14] V. Vysotska, Computer linguistic system architecture for Ukrainian language content processing based on machine learning. CEUR Workshop Proceedings, vol. 3723, pp. 133–181, 2024. https://ceur-ws.org/Vol-3723/paper9.pdf
[15] V. Vysotska, Modern State and Prospects of Information Technologies Development for Natural Language Content Processing. CEUR Workshop Proceedings, vol. 3668, pp. 198–234, 2024. https://ceur-ws.org/Vol-3668/paper15.pdf
[16] V. Vysotska, Computer Linguistic Systems Design and Development Features for Ukrainian Language Content Processing. CEUR Workshop Proceedings, vol. 3668, pp. 229–271, 2024. https://ceur-ws.org/Vol-3688/paper18.pdf
[17] V. Vysotska, O. Markiv, S. Vladov, V. Sokurenko, L. Chyrun, and Y. Lytvynenko, Embedding Model for Low-Resource Carpathian Ruthenian Language as Western Ukrainian Dialect Based on NLP and Machine Learning. Proc. 2024 IEEE 17th Int. Conf. on Advanced Trends in Radioelectronics, Telecommunications and Computer Engineering (TCSET), pp. 1–6, IEEE, 2024. https://ieeexplore.ieee.org/abstract/document/10755854
[18] R. Fedchuk, and V. Vysotska, Mathematical model of a decision support system for identification and correction of errors in Ukrainian texts based on machine learning. CEUR Workshop Proceedings, vol. 4005, pp. 29–50, 2025. https://ceur-ws.org/Vol-4005/paper3.pdf
[19] GitHub Copilot. https://github.com/features/copilot.
[20] Cody by Sourcegraph. https://sourcegraph.com/cody.
[21] Tabnine: AI code assistant. https://www.tabnine.com/.
[22] LangChain Documentation. https://docs.langchain.com/.
[23] Redis Vector Similarity Search. https://redis.io/docs/interact/search-and-query/query/vectors/.
[24] Streamlit Documentation. https://docs.streamlit.io/.
[25] H. Husain, H.H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, CodeSearchNet challenge: Evaluating the state of semantic code search. arXiv preprint, arXiv:1909.09436, 2019. https://arxiv.org/abs/1909.09436.
[26] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, UniXcoder: Unified cross-modal pre-training for code representation. arXiv preprint, arXiv:2203.03850, 2022. https://arxiv.org/abs/2203.03850.
[27] L. Di Grazia, and M. Pradel, Code search: A survey of techniques for finding code. ACM Computing Surveys, vol. 55, no. 11, pp. 1–31, 2023. https://dl.acm.org/doi/10.1145/3565971.
[28] S. Setty, H. Thakkar, A. Lee, E. Chung, and N. Vidra, Improving retrieval for RAG based question answering models on financial documents. arXiv preprint, arXiv:2404.07221, 2024. https://arxiv.org/html/2404.07221v2.
[29] T. Liu, C. Xu, and J. McAuley, RepoBench: Benchmarking repository-level code auto-completion systems. arXiv preprint, arXiv:2306.03091, 2023. https://openreview.net/forum?id=pPjZIOuQuF.
[30] Leolty, RepoBench: Benchmarking Repository-Level Code Auto-Completion Systems - ICLR 2024. https://github.com/Leolty/repobench.

[31] W. Cheng, Y. Wu, and W. Hu, Dataflow-guided retrieval augmentation for repository-level code completion. arXiv preprint, arXiv:2405.19782, 2024. https://aclanthology.org/2024.acl-long.431.pdf.
[32] R. Hu, C. Peng, J. Ren, B. Jiang, X. Meng, Q. Wu, et al., CodeRepoQA: A Large-scale Benchmark for Software Engineering Question Answering. arXiv preprint, arXiv:2412.14764, 2024. https://arxiv.org/pdf/2412.14764.
[33] D. Wu, W.U. Ahmad, D. Zhang, M.K. Ramanathan, and X. Ma, Repoformer: Selective retrieval for repository-level code completion. arXiv preprint, arXiv:2403.10059, 2024. https://arxiv.org/abs/2403.10059.
[34] G. Ggerganov, LLM inference in C/C++. https://github.com/ggml-org/llama.cpp.
[35] Which Quantization Method Is Best for You?: GGUF, GPTQ, or AWQ. https://www.e2enetworks.com/blog/which-quantization-method-is-best-for-you-gguf-gptq-or-awq.
[36] M. Grootendorst, Which Quantization Method is Right for You? (GPTQ vs. GGUF vs. AWQ). https://newsletter.maartengrootendorst.com/p/which-quantization-method-is-right.
[37] R. Devine, How to install and use Ollama to run AI LLMs locally on your Windows 11 PC. https://www.windowscentral.com/software-apps/how-to-install-and-use-ollama-to-run-ai-llms-on-your-windows-11-pc.
[38] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X.E. Tan, et al., Code Llama: Open foundation models for code. arXiv preprint, arXiv:2308.12950, 2023. https://arxiv.org/pdf/2308.12950.
[39] R. Devine, This Common Mistake in Ollama Could Be Killing Your AI Performance in Windows 11 — Here's How to Fix It. https://www.windowscentral.com/artificial-intelligence/mistake-with-ollama-on-windows-sucked-away-performance
[40] T.L. Saaty, Decision making with the analytic hierarchy process. International Journal of Services Sciences, vol. 1, no. 1, pp. 83–98, 2008. https://www.inderscienceonline.com/doi/abs/10.1504/IJSSCI.2008.017590
[41] I.B. Botchway, A.A. Emmanuel, N. Solomon, and A.B. Kayode, Evaluating software quality attributes using analytic hierarchy process (AHP). Int. J. of Advanced Computer Science and Applications, vol. 12, no. 3, 2021. https://thesai.org/Publications/ViewPaper?Volume=12&Issue=3&Code=IJACSA&SerialNo=21
[42] C.M. U-Dominic, J.C. Ujam, and N. Igbokwe, Applications of analytical hierarchy process (AHP) and knowledge management (KM) concepts in defect identification: a case of cable manufacturing. Asian Journal of Advanced Research and Reports, pp. 9–21, 2021. https://hal.science/hal-03384475v1/document#:~:text=The%20application%20of%20AHP%20begins,and%20to%20reduce%20process%20complexity
[43] UML Use Case Diagram. https://www.uml-diagrams.org/use-case-diagrams.html
[44] UML Class Diagram. https://www.uml-diagrams.org/class-diagrams.html.
[45] UML Sequence Diagram. https://www.uml-diagrams.org/sequence-diagrams.html.
[46] UML State Machine Diagram. https://www.uml-diagrams.org/state-machine-diagrams.html.
[47] UML Activity Diagram. https://www.uml-diagrams.org/activity-diagrams.html.
[48] UML Component Diagram. https://www.uml-diagrams.org/component-diagrams.html.
[49] N. Reimers, and I. Gurevych, Sentence-BERT: Sentence embeddings using Siamese BERT-networks. EMNLP 2019. arXiv:1908.10084.
[50] S. Wang, et al., MiniLM: Deep Self-Attention Distillation for Task-Agnostic Compression of Pre-Trained Transformers. NeurIPS 2020. arXiv:2002.10957.
[51] Y. Wang, W. Wang, S. Joty, and S.C. Hoi, CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. arXiv preprint, arXiv:2109.00859, 2021. https://arxiv.org/abs/2109.00859
[52] ISO/IEC 26514:2008. Systems and software engineering – Requirements for designers and developers of user documentation. International Organization for Standardization, 2008. https://www.iso.org/standard/43110.html.
[53] IEEE Std 1063-2001. Standard for Software User Documentation. Institute of Electrical and Electronics Engineers, 2001. https://standards.ieee.org/standard/1063-2001.html.
[54] S. Vladov, V. Jotsov, A. Sachenko, O. Prokudin, A. Ostapiuk, and V. Vysotska, Neural Network Method of Analysing Sensor Data to Prevent Illegal Cyberattacks. Sensors, vol. 25, no. 17, p. 5235, 2025. https://doi.org/10.3390/s25175235
[55] S. Vladov, L. Chyrun, E. Muzychuk, V. Vysotska, V. Lytvyn, T. Rekunenko, and A. Basko, Intelligent Method for Generating Criminal Community Influence Risk Parameters Using Neural Networks and Regional Economic Analysis. Algorithms, vol. 18, no. 8, p. 523, 2025. https://doi.org/10.3390/a18080523

## Authors' Profiles

**Zhengbing Hu:** Prof., Deputy Director, International Center of Informatics and Computer Science, Faculty of Applied Mathematics, National Technical University of Ukraine "Kyiv Polytechnic Institute", Ukraine. Adjunct Professor, School of Computer Science, Hubei University of Technology, China. Visiting Prof., DSc Candidate in National Aviation University (Ukraine) from 2019. Primary research interests: Computer Science and Technology Applications, Artificial Intelligence, Network Security, Communications, Data Processing, Cloud Computing, Education Technology.

**Markiian-Mykhailo Paprotskyi** is a bright student pursuing a Bachelor's degree at the Department of Information Systems and Networks at Lviv Polytechnic National University. He is a beginning researcher with a passion for data science, in particular modern technologies such as machine learning and deep learning, and large language model(LLM).

**Victoria Vysotska** is a Professor at the Information Systems and Networks Department of Lviv Polytechnic National University. She defended her Doctoral degree in Technical Science, speciality in «structural, applied and mathematical linguistics», in 2023 on the topic "Analysis and synthesis of computational linguistic systems for processing Ukrainian textual content" Also, she received a PhD degree in Information Technologies from Lviv Polytechnic National University in 2014. She has currently published more than 600 publications. Her main research interests are focused on the identification of disinformation/fakes/propaganda, detection of sources of disinformation and inauthentic behaviour (bots) of coordinated groups based on artificial intelligence, NLP, computer linguistics, data science, system analysis, information technologies, machine learning, cyber security, modern education and distance learning system development.

**Lyubomyr Chyrun** is an Associate Professor at the Ivan Franko National University of Lviv. Since 2001, he worked at the Lviv Polytechnic University at the Institute of Computer Sciences and Information Technologies. as an associate professor of the Department of Information Systems and Networks. In 2007, he defended his PhD thesis on May 1, 2002 - "Mathematical modelling and computational methods". He co-authors the monograph "Continuous Fractions and Complex Numbers". He is the author of more than 120 scientific publications. His areas of scientific interest are pattern recognition, application of numerical methods in information technologies, object-oriented programming, web technologies, fake identification, natural language processing, computer linguistics, data science, system analysis, information technologies, and machine learning.

**Yuriy Ushenko:** Prof., Computer Science Department, Chernivtsi National University, Chernivtsi, Ukraine. Research Interests: Data Mining and Analysis, Computer Vision and Pattern Recognition, Optics & Photonics, Biophysics.

**Dmytro Uhryn** graduated from Yuriy Fedkovych Chernivtsi National University, Chernivtsi. He is a Doctor of Technical Sciences and associate professor at Yuriy Fedkovych Chernivtsi National University. He has currently published more than 170 publications. His research interests include data mining, information technology for decision support, swarm intelligence systems, and industry-specific geographic information systems.

**How to cite this paper:** Zhengbing Hu, Markiian-Mykhailo Paprotskyi, Victoria Vysotska, Lyubomyr Chyrun, Yuriy Ushenko, Dmytro Uhryn, "Local Agentic RAG-Based Information System Development for Intelligent Analysis of GitHub Code Repositories in Computer Science Education", International Journal of Modern Education and Computer Science(IJMECS), Vol.17, No.5, pp. 109-145, 2025. DOI:10.5815/ijmecs.2025.05.07