WWW.IJRPR.COM

International Journal of Research Publication and Reviews

Journal Homepage: www.ijrpr.com ISSN 2582-7421

Agentic Retrieval-Augmented Generation for Intelligent GitHub Repository Analysis and Automated Code Assistance

Prof. S. Chakole, H. Peddulwar, K. Wanjari, Y. Sharma, and A. Kulkarni

Department of Robotics & Artificial Intelligence, Priyadarshini College of Engineering, Nagpur, India

ABSTRACT—

A GitHub repository appears to be a simple collection of files, but anyone who has tried reading other people's code knows well that most information is hidden in the code itself, not documented. Research among professional developers has shown that reading and understanding others' code takes between 58% and 70% of a developer's working time, more than writing it, and a newcomer always pays this price when exploring a new repository for the first time. Keyword searches offer little benefit over guessing, and a general-purpose large language model (LLM) is likely to answer incorrectly if it has not been trained on the code in that repository. In this project, we have designed and implemented an Agentic Retrieval-Augmented Generation (Agentic RAG) system tailored to address the specific needs of exploring GitHub repositories. Our system clones the target repository, splits source code and documents into chunks with their file paths, function names, and line numbers, and embeds each chunk using a code-specific embedding model into a vector database. Instead of retrieving and answering based on a fixed number of chunks, an agentic AI planner decides what to retrieve and potentially performs multiple retrieval steps. We describe the system architecture and the planning and retrieval process and provide an estimate of how much time an Agentic RAG system can save a developer by reducing the cost of repository exploration.

Keywords*—Agentic AI, Retrieval-Augmented Generation, large language models, GitHub repository analysis, vector database, semantic code search, automated code assistance*

I. INTRODUCTION

GitHub is exploding in popularity, and the numbers are astounding. GitHub's own estimates from their 2025 Octoverse report suggest that there are now 630 million repositories on the platform, of which over 121 million have been created in 2025 alone, which amounts to roughly 230 repositories appearing every minute [13]. Every one of these repositories, sooner or later, ends up in the hands of someone who did not write it: a new contributor, an open-source maintainer, a student learning from the code, or the original author returning to the code after some time away.

That is not a trivial cost. Multiple studies of working developers have found that most of a developer's time does not actually go into writing code, it goes into reading and understanding code that already exists. Xia et al. tracked 78 professional developers across seven real projects for more than 3,100 working hours and found comprehension activity took up about 58% of that time [11]. An earlier study by Minelli et al., which logged mouse and keyboard activity for 18 developers, put the number even higher, closer to 70% [12]. Whichever figure you trust, the pattern holds: developers spend more time figuring out code than writing it, and a repository nobody on the team has seen before is the worst version of that problem.

The tools most people reach for do not really solve this. Searching a repository by keyword only finds files containing that literal word or phrase, which is a poor match for a question phrased in natural language. Static analysis can tell you what calls what, but it cannot tell you why any of it exists. Large language models are genuinely good at explaining code, but only code they have effectively seen before. Point an LLM at a repository it was not trained on, which for anything private or recently updated is basically guaranteed, and it will either admit it does not know, or, more often, confidently make something up that sounds plausible but does not match the actual code.

Retrieval-Augmented Generation is meant to fix exactly this. Instead of relying only on what the model already knows, the system retrieves the actual relevant material from the repository and feeds it to the model before it answers, which grounds the response in something real. A single retrieval step is not always enough, though. A question about how a feature works might touch three or four files that do not share much vocabulary with each other, and a fixed top-k search can easily miss half of them. That is the gap an agentic layer is meant to close: instead of retrieving once and hoping for the best, an agent can look at what it found, decide if that is actually enough, and go search again if it is not.

This paper describes an Agentic RAG system built with GitHub repositories specifically in mind. We kept coming back to four goals while designing it. First, the system should ingest and index any GitHub repository without someone manually annotating it beforehand. Second, retrieved chunks need to carry enough structural context, file path, function name, class name, line numbers, to actually make sense on their own. Third, retrieval should be a multi-step process the agent controls rather than a single fixed lookup. Fourth, every answer the system gives should be traceable back to something that is actually in the repository, not invented by the model.

II. RELATED WORK

Fan et al.'s survey on Retrieval-Augmented Large Language Models is a reasonable starting point for understanding where RAG systems in general are headed. It lays out the standard retriever-index-generator setup that most systems in this space, ours included, are built around, and goes over the open problems around retrieval quality and latency that keep showing up in later papers [1]. Singh et al. take this further with a survey focused specifically on Agentic RAG, describing how an autonomous planning layer on top of retrieval lets a system decide for itself when it has gathered enough context and coordinate more than one retrieval step [2], which is close to the architecture we ended up building. Neha and Bhati run a more direct comparison between Traditional RAG and Agentic RAG, and their finding that the agentic approach costs some extra latency and complexity but handles multi-hop questions noticeably better lines up with why we went with the agentic route instead of the simpler one.

The whole point of grounding an LLM in retrieved content is to cut down on hallucination, and Li et al.'s survey walks through why retrieval helps here along with how researchers have tried to measure it [4]. One idea that shaped our own retrieval logic fairly directly is Rashtchian and Juan's work on "sufficient context," a way of checking whether what has been retrieved is actually enough for the model to answer correctly rather than just assuming a fixed number of results will do [5]. Speed is a separate concern from accuracy, and Speculative RAG addresses it by having a smaller model draft an answer quickly while a larger model checks it over, trading some accuracy risk for speed [6]. We did not build this into the current version of the system, but it is something we would like to try later.

Retrieved context is not always reliable either, especially across a repository where a README might describe something differently than the code that actually implements it. The (D)RAGged Into a Conflict study builds a taxonomy for these kinds of contradictions and finds that telling the model what type of conflict it is dealing with improves its answer [7]. Wang et al.'s Astute RAG pushes this further, proposing a way to reconcile what the model already "believes" internally with what retrieval actually turns up, which helps specifically when retrieval comes back incomplete or misleading [8].

Two papers come closer to what this project is actually about, which is using RAG on code rather than general text. Bhaumik et al. compare Graph RAG against standard vector-similarity RAG for migrating enterprise code, and find that keeping the structure and dependencies of the code intact, instead of treating every chunk as an isolated piece of text, cuts down on API-level hallucination during translation [9]. That result is a large part of why our own chunking keeps file paths, function names, and line numbers attached to every chunk rather than simply splitting text by length. Rashtchian and Juan's later work on Gemini's Agentic RAG platform shows an agent pulling context iteratively across several enterprise data sources instead of one fixed index [10], which is close to what our agent does across a single repository's code, documentation, and configuration files.

III. PROBLEM ANALYSIS AND MOTIVATION

Once you actually watch how people try to understand a repository they did not write, the same handful of problems come up again and again, regardless of the size or language of the project.

- Keyword search only matches the words you typed, not what you meant, so it misses code that is relevant but uses different terminology.
- Static analysis shows structure, what calls what, but has nothing to say about intent or purpose.
- A general-purpose LLM with no access to the actual repository will guess, and it guesses confidently even when it is wrong.
- An entire repository rarely fits in a single prompt, so some form of selective retrieval is unavoidable.
- A single, fixed retrieval step tends to miss questions that span more than one file or require following a reference across the codebase.
- None of this scales well: the bigger and older a repository gets, the worse the onboarding problem becomes, and open-source projects with high contributor turnover feel this constantly.

None of this is hypothetical. It plays out the same way almost every time — someone new opens a repository, searches for a term that is not the one the original author actually used, gives up, and asks a maintainer directly. That maintainer then spends their own time answering a question they have probably answered before, for someone else, more than once. Multiply this across every new contributor a project ever gets, and the comprehension cost from Section I does not stay flat, it adds up.

IV. PROPOSED SYSTEM ARCHITECTURE

Our first instinct was to keep things simple: a single retrieval step feeding directly into an LLM prompt, the standard RAG setup, which would have been quicker to build. We changed course after actually thinking through what a typical repository question looks like in practice. Explaining how a feature is wired across multiple files, tracing a class hierarchy, or debugging something that only makes sense once you have read two or three related functions is not something a single fixed retrieval step answers well. Letting the agent look at what it retrieved, judge whether that is enough, and go back for more if it is not costs a little extra time per query, but it is worth it for the kind of multi-hop questions people actually ask about a codebase.

A. Components

- Ingestion layer: clones the target repository from a URL or accepts a local upload, and filters out build artifacts, dependency and cache folders, and binary or media files before anything downstream has to deal with them.
- Parsing and chunking layer: breaks source files, documentation, and configuration into logically meaningful chunks while preserving file paths, function names, class names, and line numbers.
- Embedding layer: converts each chunk into a dense vector representation using a code-aware transformer model.
- Vector store layer: indexes embeddings and their metadata in Qdrant for fast approximate nearest-neighbor search.
- Agentic retrieval layer: plans, executes, and refines semantic search over the vector store depending on what the query actually needs.
- Generation layer: synthesizes a grounded, repository-specific answer from the retrieved context using a large language model.
- Interface layer: a chat-style web interface where a developer asks questions and gets back answers, explanations, and suggestions.

V. SYSTEM DESIGN AND COMPONENTS

A. Repository Ingestion and Filtering

The system accepts either a Git URL or a local repository path. On ingestion it excludes version-control internals, dependency directories such as node_modules or vendor, build and cache output, and non-text or media files by extension and size, so indexing effort only goes toward files a developer would actually read.

B. Content Representation and Chunking

Rather than splitting files into fixed-length token windows, chunk boundaries follow logical code units, functions, classes, and documentation sections, using language-aware parsing. Every chunk carries metadata: its file path, enclosing function or class name, start and end line numbers, and detected language, so a retrieved chunk still makes sense even outside the context of the file it came from.

C. Embedding Generation

Each chunk is converted into a dense vector using the jina-embeddings-v2-base-code model, a transformer trained on code across multiple programming languages and further tuned on coding question-answer and docstring pairs [14]. We picked a code-aware embedding model over a general-purpose text embedder because it captures syntactic and semantic similarity between code fragments better, which matters for retrieval quality on natural-language questions about code.

D. Vector Database Construction

Embeddings and their metadata are stored in Qdrant, an open-source vector database that indexes vectors for approximate nearest-neighbor search and also supports payload-based filtering alongside similarity search. This lets a query be scoped by file type, directory, or language on top of semantic closeness, which starts to matter once a repository has thousands of chunks in it.

E. Agentic Retrieval Strategy

An AI agent sits between the user's query and the vector database, instead of a fixed retrieval function. It interprets what the query is actually asking for, explanation, debugging, architecture summary, or implementation guidance, plans an initial retrieval strategy, and runs one or more semantic searches. After each search the agent checks whether the retrieved context is sufficient to answer the query, using the same idea as the "sufficient context" approach [5]. If it is not, the agent refines the search by rephrasing the query, widening its scope, or following a reference such as a class it depends on, before consolidating and de-duplicating the final context.

F. Context-Aware Response Generation

The consolidated context and the original query are passed to a large language model, which is prompted to answer only from the supplied context and to point back to the specific files and lines it drew on. Depending on the query, the output might be a code explanation, an architecture summary, a debugging suggestion, or a recommended change, but all of it is grounded in retrieved repository content rather than the model's own training data.

VI. METHODOLOGY AND APPROACH

We built this in phases so each layer could actually be tested as we went, rather than everything coming together, or falling apart, right at the end. With four of us on the project, splitting work by layer also kept us from stepping on each other's code. Two of us stayed on the ingestion, parsing, and embedding pipeline, and the other two worked on the agentic retrieval logic and the response and interface layers.

- Phase 1 (Weeks 1–2) — Requirements analysis and literature review: figuring out the query types the system needs to support and picking the embedding model, vector database, and LLM.

- Phase 2 (Weeks 3–4) — Repository ingestion and parsing: building the cloning, filtering, and language-aware chunking pipeline.
- Phase 3 (Weeks 5–6) — Embedding and vector storage: wiring up jina-embeddings-v2-base-code and Qdrant, and checking retrieval quality on a few sample repositories.
- Phase 4 (Weeks 7–9) — Agentic retrieval layer: building the planner, the sufficiency check, and the iterative refinement logic.
- Phase 5 (Weeks 10–11) — LLM integration and response generation: prompt design and getting the answer to reliably cite file and line numbers.
- Phase 6 (Weeks 12–13) — Interface development: the chat-style web interface developers actually interact with.
- Phase 7 (Weeks 14–15) — Testing: unit tests for individual modules, integration tests across the pipeline, and evaluation on real repositories of different sizes.
- Phase 8 (Weeks 16–17) — Deployment and documentation: rollout and getting a few users to actually try it.

On the technology side, the backend is Python with FastAPI, GitPython handles repository cloning, and tree-sitter does the language-aware parsing for chunking. jina-embeddings-v2-base-code generates the embeddings, Qdrant handles vector storage and search, and a large language model accessed through an API performs response generation, coordinated through a custom agent loop that plans and refines retrieval. The frontend is a React-based chat interface, and the system is meant to run on a standard cloud platform so it can handle bigger repositories without needing to be rebuilt from scratch.

VII. IMPLEMENTATION STRATEGY

A. System Workflow

End to end, it works like this. A developer submits a GitHub URL or uploads a local repository. The system clones it, filters out anything irrelevant, and parses what is left into metadata-tagged chunks. Each chunk gets embedded and stored in Qdrant. The developer then asks a question in plain language through the chat interface. That query is embedded and handed to the agent, which plans and runs one or more retrieval steps, checking after each one whether it has enough to work with. The consolidated context goes to the LLM, which generates a grounded, repository-specific response, and that response, along with the files and lines it drew on, is what gets shown back to the developer.

B. Security Considerations

Since the system clones and indexes repositories that may well be private, we treated security as something to design around from the start rather than add on later.

- OAuth-scoped GitHub access tokens with least-privilege permissions for cloning private repositories.
- Ephemeral, sandboxed cloning environments isolated per repository.
- TLS/HTTPS encryption for all data in transit.
- Encryption at rest for stored embeddings and metadata tied to private repositories.
- Role-based access control so a user can only query repositories they are actually authorized to see.
- Audit logging of queries and retrieved content for traceability.

VIII. EXPECTED OUTCOMES AND A WORKED ESTIMATE

It helps to put a rough number on this instead of just asserting the system saves time. Picture a mid-sized repository with a five-person team that brings on two new contributors every quarter. If it genuinely takes about three working days, call it 24 hours, for someone to get comfortable enough with an unfamiliar codebase to start contributing, and comprehension eats up even just the lower end of what Xia et al. found, 58% of that time [11], that is close to 14 hours per person spent purely on figuring out where things are rather than writing anything. An agent that can directly answer "where is this implemented" or "why does this function exist," using the actual repository content, will not erase that number entirely, some hands-on exploration is always going to be necessary and honestly useful, but it targets the search-and-re-search part of that time, which the same studies put at the bulk of it [11], [12]. Across eight new contributors a year, even cutting that time down by a third adds up to real hours returned to the team. This is our own back-of-the-envelope estimate rather than something we measured, but it gives a sense of the scale involved.

Beyond time saved, grounding responses in retrieved repository content also addresses the hallucination problem that comes with using a plain LLM on an unfamiliar or private codebase [4]. The outcomes we expect are faster onboarding for new contributors, fewer repeated interruptions for maintainers fielding the same orientation questions, more reliable explanations of legacy or sparsely documented code, and a system that can support code review and debugging with answers that point back to specific files and lines instead of being generated from memory.

IX. CONCLUSION AND FUTURE WORK

The Agentic RAG system described in this paper is aimed at a problem that is easy to underestimate precisely because it sounds simple, understanding a codebase someone else wrote. In practice that simplicity falls apart quickly once a repository grows past a handful of files, and the cost of getting it wrong is not visible on any dashboard the way a missed deadline is. It shows up quietly, as hours spent reading code, which Section I already puts a number on. By combining structure-preserving chunking, code-aware embeddings, a vector database built for semantic search, and an agent that plans and refines its own retrieval, the system grounds its answers in what a repository actually contains rather than in an LLM's training data.

The same pattern, structured ingestion, semantic retrieval, agentic reasoning, grounded generation, is not really limited to GitHub and could extend to other code hosting platforms or internal enterprise codebases. A few directions we think are worth pursuing next: graph-based retrieval that captures dependency structure directly instead of relying on vector similarity alone, following what the Graph RAG results for code suggest [9]; an IDE or VS Code extension so the assistant sits where developers already work instead of a separate browser tab; extending the agent into something that reviews pull requests and comments automatically on incoming changes; support for multi-repository and monorepo-aware retrieval for larger organizations; and eventually moving from explaining and suggesting code toward generating patches for well-scoped changes on its own.

ACKNOWLEDGMENT

We thank Prof. Sakshi Chakole, Assistant Professor, Department of Robotics & Artificial Intelligence, Priyadarshini College of Engineering, Nagpur, for her guidance and feedback throughout this project during the 2026–27 academic session.

REFERENCES

[1] W. Fan, Y. Ding, L. Ning, S. Wang, H. Li, D. Yin, T.-S. Chua, and Q. Li, "A Survey on RAG Meeting LLMs: Towards Retrieval-Augmented Large Language Models," 2024.

[2] A. Singh, A. Ehtesham, S. Kumar, T. T. Khoei, and A. V. Vasilakos, "Agentic Retrieval-Augmented Generation: A Survey on Agentic RAG," 2026.

[3] F. Neha and D. Bhati, "Traditional RAG vs. Agentic RAG: A Comparative Study of Retrieval-Augmented Systems," 2024.

[4] Y. Li, X. Fu, G. Verma, P. Buitelaar, and M. Lu, "Mitigating Hallucination in Large Language Models (LLMs): An Application-Oriented Survey on RAG, Reasoning, and Agentic Systems," 2025.

[5] C. Rashtchian and D.-C. Juan, "Deeper Insights into Retrieval Augmented Generation: The Role of Sufficient Context," 2025.

[6] Multiple authors, "Speculative RAG: Enhancing Retrieval Augmented Generation through Drafting," 2025.

[7] Multiple authors, "(D)RAGged Into a Conflict: Detecting and Addressing Conflicting Sources in Retrieval-Augmented LLMs," 2025.

[8] F. Wang, X. Wan, R. Sun, J. Chan, and S. Arik, "Astute RAG: Overcoming Imperfect Retrieval Augmentation and Knowledge Conflicts for Large Language Models," 2025.

[9] S. Bhaumik, N. Jaiswal, A. Shukla, D. Malhotra, A. Agrawal, S. Garg, and S. Puri, "Beyond Vector Similarity: Hierarchical Context-Aware Graph RAG vs Standard RAG in Enterprise Code Migration," 2026.

[10] C. Rashtchian and D.-C. Juan, "Unlocking Dependable Responses with Gemini Enterprise Agent Platform's Agentic RAG," 2026.

[11] X. Xia, L. Bao, D. Lo, X. Xing, A. E. Hassan, and S. Li, "Measuring Program Comprehension: A Large-Scale Field Study with Professionals," IEEE Trans. Softw. Eng., vol. 44, no. 10, pp. 951–976, 2018.

[12] R. Minelli, A. Mocci, and M. Lanza, "I Know What You Did Last Summer: An Investigation of How Developers Spend Their Time," in Proc. 23rd IEEE Int. Conf. Program Comprehension (ICPC), 2015, pp. 25–35.

[13] GitHub, "Octoverse 2025: The State of Open Source and AI," GitHub Blog, 2025. [Online]. Available: https://github.blog/news-insights/octoverse/

[14] M. Günther et al., "Jina Embeddings 2: 8192-Token General-Purpose Text Embeddings for Long Documents," arXiv:2310.19923, 2023.