Codebase Guy: Efficient AI-Guided Development Across Multiple Repositories within the Same Codebase

Victor Bona
Independent
Blumenau, Brazil
victor.bona@hotmail.com

***Abstract*—Navigating and understanding a large codebase spread across multiple repositories is a challenging and time-consuming task for software developers. Modern tools aim to help with this problem, but often only supports closed scopes within one repository or directory. *Codebase Guy* is a system that uses artificial intelligence to assist developers with code comprehension, search, and generation tasks in such extensive codebases. The platform integrates local large language models (LLMs) (e.g., via the Ollama engine) with a highly efficient retrieval-augmented generation (RAG) pipeline and other AI-driven features to provide interactive natural language assistance. The Codebase Guy architecture includes a chat-based user interface, APIs for tool integration, a semantic code embedding service, and an agent that indexes and monitors multiple repositories. The system supports conversational queries about the code, chain-of-thought reasoning with tool use (such as web research when needed), and even assists in ticket resolution by locating relevant code for bug fixes or feature requests. We detail the technical design of Codebase Guy, including its hybrid semantic-lexical search strategy, context reduction through summarization and prompt optimization, and techniques for deploying these capabilities locally under constrained computational resources. Our implementation in TypeScript demonstrates how developers can seamlessly chat and code with LLM across different project scopes. We focus on efficiency optimizations that balance accuracy and performance, enabling Codebase Guy to deliver relevant answers with low latency and manageable resource usage. The paper concludes with real-world use cases and a discussion of future enhancements, such as an autonomous coding agent and an orchestrator for coordinating multiple AI agents.**

I. Introduction

Modern software projects often consist of millions of lines of code distributed across numerous files, modules, and even multiple repositories. Developers spend a significant portion of their time reading and understanding existing code rather than writing new code. This challenge is exacerbated in large **brownfield** projects where documentation may be sparse or outdated, and knowledge is spread among different teams or code repositories. When trying to locate functionality or debug an issue, an engineer might need to search through several services or libraries that collectively form the codebase. Traditional code search tools (e.g., grep or simplistic text search) can quickly find exact matches but lack semantic understanding, often leading to irrelevant results or missed context. Maintaining a mental map of how different repositories interact (for example, how an API call in one service corresponds to code in another) is difficult and error-prone.

Large Language Models (LLMs) have shown promise in aiding code comprehension and generation by providing natural language explanations and even writing code given a prompt. In principle, an LLM could answer questions about the codebase or generate new code by reading the relevant files. However, applying LLMs directly to a massive multi-repository codebase faces two key hurdles: **context window limitations** and **compute constraints**. State-of-the-art LLMs can only accept a few thousand tokens of input context, which is far smaller than the entire codebase. Simply concatenating all relevant files or even large file fragments into a single prompt would exceed these limits and be computationally prohibitive. Moreover, running powerful LLMs in a cloud-based environment raises privacy and cost concerns—organizations often prefer local deployment to keep proprietary code and data secure. Local deployment, however, means we must work with limited hardware (such as a developer's laptop or a small server), where memory and processing power are constrained. This scenario demands efficient techniques to make AI assistance feasible without sacrificing too much accuracy.

Codebase Guy addresses these challenges by combining retrieval-based techniques with local LLM inference to provide an AI assistant for large codebases. Instead of treating the LLM as an *silver bullet* that must ingest entire files, Codebase Guy uses a *retrieval-augmented generation (RAG)* approach tailored for code. The system preprocesses and indexes the code across all repositories to enable fast lookup of relevant snippets or summaries. When a developer asks a question or requests help (for instance, "How is user authentication implemented across our services?" or "Find the source of this error message"), the platform searches its index to fetch the most pertinent pieces of code from the various repositories. Only these relevant code fragments (and/or their summaries) are provided to the LLM, along with the developer's query, in a constructed prompt. By narrowing the LLM's context to just a tiny fraction of the codebase (often a few functions or a single module), Codebase Guy dramatically reduces the amount of information the model must process at once. This reduction in prompt size not only fits within the model's context window but also speeds up inference and decreases memory usage.

In addition to code retrieval, Codebase Guy incorporates other AI-driven methodologies to enhance assistance. It can perform **chain-of-thought reasoning**, wherein the system may break down complex queries into multiple steps, possibly consulting external knowledge sources (e.g., performing a web search for an error code or a library usage example) before formulating a final answer. The platform is also designed to integrate with issue trackers or project management tools: given a bug report or ticket description, Codebase Guy can identify relevant code areas and suggest potential fixes, effectively assisting in **ticket resolution**. By integrating these components, Codebase Guy acts as an intelligent development companion that not only finds and explains code from multiple repositories but also learns to navigate the broader context around the code (such as documentation or known issues).

This paper presents a comprehensive overview of Codebase Guy's technical architecture and methodologies. We describe in detail how the system is built to maximize efficiency, allowing it to run locally. Key contributions include: (1) a hybrid semantic-lexical code search mechanism that achieves high recall with minimal overhead, (2) a hierarchical summarization and context compression technique that preserves essential information while staying within token limits, (3) a practical implementation of a chat-based coding assistant that can operate across multiple repositories seamlessly, and (4) an outline of future extensions like an autonomous coding agent and a multi-agent orchestrator. The remainder of this paper is organized as follows: Section II describes the overall system architecture and components. Section III outlines the technical implementation details, including how the LLM, embeddings, and search are integrated. Section IV discusses the methodologies and techniques employed to achieve efficient retrieval and prompt optimization. Section V illustrates use cases demonstrating the system in action. Section VI explores planned future enhancements, and Section VII concludes the paper with final thoughts on the system's impact and efficiency.

II. System Architecture

At a high level, Codebase Guy consists of several interconnected components that work together to turn user queries into helpful answers or code changes. The architecture (illustrated conceptually in Fig. **??** if we had one) is composed of the following major parts: a user interface for conversation, a backend AI engine that includes the local LLM and reasoning logic, a retrieval index of the codebase, and supporting services like embedding computation and agent coordination. In this section, we describe each component and how they interact.

A. User Interface and Interaction

The user interface (UI) is designed to allow developers to interact with Codebase Guy in a conversational manner. This could be implemented as a chat window in an IDE (e.g., a VS Code extension), a web-based chat application, or even a messaging platform integration (like Slack or Teams bot). Through the UI, the developer can pose questions in natural language, such as *"Explain how module X parses configuration files"* or *"Which functions would be affected if we change the user authentication logic?"*. The UI sends these queries to the backend system via a defined API. In turn, the assistant's responses (explanations, code snippets, or step-by-step guidance) are displayed back in the chat interface. The UI may also provide controls for the user to refine the query (for example, selecting which repository or component to focus on), and to view any retrieved code context that the AI is using to formulate its answer. This conversational setup aims to make interacting with the codebase as intuitive as talking to a knowledgeable teammate.

The backend exposes APIs that the UI or other tools can use. Typically, a primary API endpoint accepts a user query (and possibly additional context like conversation history or a specified project scope) and returns an AI-generated answer. Other API endpoints might allow more direct interactions, such as asking for "search results only" (bypassing the LLM, useful if the developer just wants to see relevant code snippets) or providing feedback on an answer. The API layer ensures that Codebase Guy can be integrated not only via the provided UI but also into other developer workflows or tools. For instance, a continuous integration system could query Codebase Guy for hints when a build fails, or a documentation generator could ask it for summaries of code.

B. Local LLM Engine via Ollama

At the core of the assistant is a local large language model that generates the natural language responses and code suggestions. Codebase Guy uses **local LLM deployment**, meaning the model runs on the developer's own hardware or a dedicated server under the organization's control. We integrate the LLM using the Ollama tool, which streamlines running open-source LLMs on local machines. Ollama allows models like LLaMA 2, CodeLlama, or other fine-tuned variants to be loaded and queried via a simple interface. In our implementation, the backend communicates with the Ollama engine (which hosts the model) through an API call: the prompt (containing the user's question and retrieved context) is sent, and the generated completion (the answer) is returned.

Using a local LLM offers privacy (no code or queries are sent to external services) and can be cost-efficient after the initial setup. However, it requires optimization to perform well under limited resources. We typically run a model in the 7B to 13B parameter range, which can fit and infer on a high-end laptop or small server when quantized (e.g., 4-bit or 8-bit weights). For example, a 7B model quantized to 4-bit can often produce responses in a few seconds on a modern CPU, which is acceptable for interactive use. Larger models (30B or more) can improve answer accuracy and depth but may introduce latency on local hardware, so Codebase Guy allows model selection based on available compute. The architecture is modular: if a GPU is available or if the user prefers a different backend (like running the model in a container or via another library), the system can adapt as long as it exposes a similar "generate text from prompt" interface.

To maintain responsiveness, the LLM engine is typically kept loaded in memory, ready to serve requests, rather than loading/unloading models for each query. We also employ a **system prompt** or persona for the model (e.g., ”You are Codebase Guy, an expert AI assistant for our codebase...”) to consistently guide its style and restrict it to use only provided information when answering code-specific questions. By doing so, we minimize hallucinations and ensure that answers remain grounded in the actual code context retrieved.

C. Embedding and Indexing Service

A crucial component of Codebase Guy is the code retrieval system, which allows the assistant to *find relevant pieces of code across multiple repositories* efficiently. This is handled by an embedding and indexing service. During an offline (or initial) phase, Codebase Guy scans through the codebase — which may consist of multiple repositories or projects — and processes all the source files to create an index that can be queried later.

The indexing pipeline works as follows: an **agent** traverses each repository's file structure, reading source files (for example, only files with certain extensions like `.ts`, `.js`, `.py`, etc., depending on the project). Each file is broken down into logical *chunks* suitable for retrieval. Rather than splitting arbitrarily, the agent uses code structure to inform chunk boundaries. For instance, it may split code by function or class definitions, or logical blocks, so that each chunk represents a coherent piece of functionality. This avoids situations where a relevant piece of code is cut off mid-function, which would make it harder for the AI to interpret.

For each chunk of code, the agent generates a semantic embedding using an embedding model. The embedding is a high-dimensional vector representation that captures the meaning of the code snippet (for example, it will place similar code or related topics near each other in vector space). We use a code-aware model for this purpose (such as CodeBERT or a smaller SentenceTransformer fine-tuned on code and text) to get better embeddings for code than a generic text model would. The embeddings are computed either by an integrated library or by calling out to a dedicated embedding service. In our setup, this embedding computation can also run locally; for efficiency one might use a smaller model for embeddings or quantize the embedding model as well. All computed vectors are stored in a vector index (using an approximate nearest neighbor structure like HNSW or IVF from libraries such as FAISS or similar), enabling fast similarity search by embedding.

In addition to the vector index, we also maintain a lightweight **keyword index** (lexical index). This could be a simple inverted index mapping words/tokens to the locations (file and chunk) where they appear. We index important identifiers, function names, error messages, and other keywords from the code. This dual indexing strategy (semantic and lexical) is used to implement hybrid search (described later in Section IV). The index stores not only the references to the code but can also store a pre-computed short summary of each chunk. For example, alongside each function's code, we may store a one-line description of what that function does. These summaries can be generated by the agent either by parsing documentation and signatures or by using the LLM itself in a preprocessing step to summarize the code. Summaries of larger units (like entire files or modules) can also be stored to help answer high-level questions.

Because the codebase may evolve over time, the indexing service is designed to handle updates. The agent can watch for repository changes (via hooks or periodic scanning) and update the index incrementally. New or modified files get re-embedded and old embeddings are removed if code is deleted. This ensures that Codebase Guy's knowledge stays in sync with the actual code.

D. Agent Orchestration and Tools

While the core loop of Codebase Guy is straightforward (query $\rightarrow$ retrieve context $\rightarrow$ generate answer), the system is built to allow more complex reasoning patterns when needed. This is achieved through an *agent orchestration* layer that can coordinate multiple steps and tools. In the current implementation, this is relatively simple: the assistant primarily uses the code search index as its tool for answering queries. However, the architecture anticipates adding more capabilities.

For example, one tool integrated is a **web research module**. If the user's query cannot be answered solely from the code (for instance, ”What does error code XYZ typically mean?” or ”Is there a known library function to do X?”), the assistant can optionally perform a web search or consult online documentation. In practice, this might be implemented by detecting that the query is out-of-scope of the codebase and then calling an external search API. The results (e.g., relevant documentation pages or Stack Overflow answers) can then be summarized and fed into the conversation. This chain-of-thought process, where the AI breaks the problem into sub-questions (first find relevant info, then answer using it), is akin to how a developer would approach an unfamiliar error: search the web, read up, then apply the knowledge to the code at hand.

The orchestration layer is also what manages context between turns in a multi-turn conversation. If a developer asks a follow-up question like ”Okay, now show me how that function is used in repository B,” Codebase Guy uses the conversation history to understand the context (which function was being discussed) and performs a new retrieval in the context of repository B. Some state (like the focus of discussion) is kept to avoid starting from scratch on every question. The orchestrator ensures that each query to the LLM is accompanied by the most relevant information, which might include previous Q&A summary or the code snippet currently under discussion.

Overall, this architecture is modular. The UI, embedding service, LLM (via Ollama), and orchestrator communicate through clear interfaces (often HTTP or IPC calls in our implementation). This modularity means each component can be scaled or improved independently: e.g., one could swap out the vector search engine for a different one, or upgrade the LLM to a new model, without overhauling the entire system.

III. Technical Implementation

In this section, we delve into how we implemented the above architecture, focusing on the interplay between components and the practical considerations. The current prototype of Codebase Guy is built primarily in **TypeScript**, running on Node.js for the server components, with a front-end in React for the chat UI (alternatively, the core can be accessed via a CLI for terminal-based interaction). We chose TypeScript to easily integrate with developer tools and because the target codebases (on which we tested) included a lot of TypeScript/JavaScript, which allowed for reuse of parsing utilities.

A. Initializing and Indexing the Codebase

On startup or upon a user command, Codebase Guy performs an indexing routine if one has not been done or if the repositories have changed significantly. Pseudocode for the indexing process is shown in Listing 1. We used Node.js file system APIs to recursively traverse directories. For each file, we determine if it should be indexed (via extension whitelists and by skipping large auto-generated files or dependencies). We then read the file content and pass it to a chunking function.

Listing 1. Pseudo-code for codebase indexing

```
async function indexRepository(repoPath: string) {
  for (const filePath of walkFiles(repoPath)) {
    if (!isCodeFile(filePath)) continue;
    const text =
        await fs.promises.readFile(filePath, "utf-8");
    const chunks =
        chunkCodeByStructure(text, filePath);
    for (const chunk of chunks) {
      const vec = embed(chunk.text);
      vectorIndex.add(vec, {
        file: filePath, loc: chunk.range
      });
      keywordIndex.addKeywords(
        extractKeywords(chunk.text),
        chunk.id
      );
      const summary = summarizeCode(chunk.text);
      if (summary) {
        metadataIndex.storeSummary(chunk.id, summary);
      }
    }
  }
}
```

As shown, ‘chunkCodeByStructure‘ is responsible for splitting files. In our implementation, this function uses simple heuristics: for languages like JavaScript/TypeScript, it looks for function or class boundaries (using regex or a lightweight parser) to cut the file. Each chunk is tagged with its location (‘filePath‘ and a line range or byte range) so we can later retrieve or display the snippet from the original source. We limit chunk size (e.g., to a few hundred tokens) to ensure the embeddings capture focused topics and to avoid extremely large snippets that wouldn't fit in the LLM context anyway.

The ‘embed‘ function calls a pre-trained embedding model. We integrated a small Transformer model (through a library) to get a 768-dimensional vector for each chunk. This model runs locally; it can be somewhat slow if the codebase is huge, so in practice we would run this indexing once and save the index to disk (using a binary file for vector index and a JSON or database for metadata). We also considered using a service like OpenAI's embedding API for faster vector generation, but opted for local to preserve privacy. The ‘vectorIndex.add‘ stores the vector along with metadata (file and location, and maybe an ID). We used a library for approximate nearest neighbor search so that even if tens of thousands of vectors are added, the query remains fast (sub-linear time). The ‘keywordIndex.addKeywords‘ might simply add entries to an in-memory map or use a full-text search library; for instance, one could use Elasticsearch or Lunr.js to enable efficient text queries. We extract keywords by removing stopwords and taking identifiers and string literals from the chunk.

The optional ‘summarizeCode‘ step uses a lightweight summarization strategy: if the chunk is above a certain length or if it's a recognized logical unit (like a long function), we call a summarizer (which could be a smaller LLM or even rule-based). In our tests, we used the same LLM via Ollama in a quicker ”summarization mode” (by prompting it to summarize the code in one sentence) for a subset of important functions. These summaries are stored in a metadata index keyed by chunk or file. This adds some upfront cost, but it pays off when answering high-level questions, as we can supply summaries instead of raw code to the main LLM, saving tokens.

B. Hybrid Retrieval and Query Processing

When the user asks a question, Codebase Guy must retrieve the relevant context. The query processing module first interprets the query. If the query references a specific repository or component (e.g., ”in the payments service, how do we validate user input?”), the system will focus the search on that repository's index. Otherwise, it searches across all indexed repositories to not miss relevant results.

We employ a **hybrid search strategy** combining semantic and lexical search:

- *Semantic Search*: We compute an embedding for the user's query (using the same embedding model as during indexing). Then we perform a vector similarity search in the vector index to find code chunks whose embeddings are closest to the query vector (indicating those code snippets are likely related in theme or content to what the query is asking).
- *Lexical Search*: In parallel, we take important terms from the query (for example, if the query contains function names, error messages, or specific keywords) and query the keyword index. This yields code locations that exactly contain those terms. We often use a ranked retrieval like BM25 to get the top textual matches, which is good for catching cases where a query term directly appears in code (like a specific function name or log message).

The results of these two searches are then combined. We might take the top-k from each or do a score fusion (both the semantic similarity score and lexical score can be normalized and compared). The combination ensures high recall: semantic search might return a relevant function that uses different variable names than the query, while lexical search will catch an exact config key or error code mentioned. By blending them, Codebase Guy avoids the pitfalls of relying solely on one method.

After retrieval, we have a set of candidate code snippets (and possibly summaries). Typically, we select the top few (say 3-6) pieces to include as context. If any retrieved snippet is very large, we might truncate it or replace parts of it with a summary to fit the length. The system also filters out obviously irrelevant or redundant results (for example, multiple snippets from the same file that overlap heavily—we would merge them or choose the most relevant segment).

If the user's query is a follow-up in a conversation, we also consider conversation context. For instance, if previously the user was asking about a particular class and now just asks ”where is it used?”, we carry over the subject (the class name or ID) and incorporate that into the new query logic.

C. Prompt Construction and LLM Response

Once relevant context is collected, the next step is to build the LLM prompt. Prompt construction is critical for guiding the model to produce a useful and correct answer. We follow a structured prompt format. In simplified form, it looks as follows.

System Message: You are Codebase Guy, an AI assistant that helps with the codebase. You have access to the following code snippets and related information. Use them to answer the question. If you do not have enough information, state so or invoke the appropriate tools.

User Question: `<the user's query>`

Relevant Code:

- **Snippet 1:** (with file path and an optional summary)
- **Snippet 2:** (with file path and code content)
- . . .

Assistant Task: Provide a helpful answer or code solution using the given context.

In the actual implementation with Ollama, system and user messages are combined since we are using a single prompt for a single-turn answer. We interleave the retrieved code into the prompt, often prefacing each snippet with a delimiter and source info, so the model knows it's reference material. For example, we might include a line like ”‘// From file: auth/utils.ts‘” before a code snippet. This helps the model attribute the snippet to the codebase and use it for grounding the answer.

We also apply some prompt optimization techniques: we strip out extraneous or irrelevant lines from code (like license headers, large comment blocks that aren't directly useful, or import statements if the question is about logic). We ensure each snippet is clearly separated to avoid the model merging them confusingly. If a summary is available and the raw code is long, we may use ”‘¡Summary of file X: ...¿‘” instead of the code itself. The idea is to pack as much relevant info as possible within the token limit.

The LLM then generates an answer. Thanks to the prompt, this answer will typically cite or make use of the code provided. For instance, it might say ”The function ‘validateInput‘ in the payments service (see snippet from ‘payments/util.ts‘) checks the format of the input and throws an error if invalid. It uses a regex defined in ‘constants.ts‘ to do the validation...”. Because the model has the code context, it can be specific and correct. The chain-of-thought reasoning mostly happens implicitly inside the model: it has the information it needs, and it composes an explanation or solution step by step. For transparency, we could prompt the model to output its reasoning (this is sometimes done in research using techniques like ReAct, where the model first outputs reasoning and tool usage before final answer), but in our user-facing mode we suppress the raw reasoning and just give the final answer. Internally, however, we might log the reasoning if using an agent approach for debugging or improvement.

To illustrate how the system ties everything together, consider a quick example. Suppose a user asks: *”Why does the application crash when I input an empty username?”*. The query processor identifies keywords like ”crash” and ”empty username”. The lexical search might find an error message ”username cannot be empty” in the authentication module. The semantic search might retrieve a function ‘validateUser‘ that deals with user input. The system then provides the code from ‘validateUser‘ and maybe the error handling code around it to the model, along with the question. The LLM sees from the code that an exception is thrown when username is empty and responds with an explanation: ”The application crashes because the ‘validateUser‘ function throws an exception when the username is empty. In the code snippet provided, if the ‘username‘ string length is 0, it calls ‘throw new Error(”username cannot be empty”)‘. This error isn't caught, so it causes a crash. The fix would be to catch this error or handle the empty input before calling ‘validateUser‘.”

The answer is derived directly from real code, which increases trust in the result. The developer can double-check the cited code if needed.

D. Performance Optimizations

Given that Codebase Guy is intended to run with limited computing resources, we implemented several optimizations to keep it fast and lightweight:

- Token Minimization: By aggressively trimming the prompt to only include necessary information, we reduce the number of tokens the LLM must process. This speeds up response generation and lowers memory use. For example, if a function has 100 lines but only 10 are relevant to the query, we try to include just those 10 (or a summary of the rest).
- Caching: We cache results at multiple levels. Embeddings of code chunks are stored from the indexing phase, so we never recompute those. We also cache query embeddings—if the same or similar question is asked again, we can reuse the embedding vector (and even the retrieved snippets if the code hasn't changed). Additionally, if

the LLM is asked something very similar to a previous question, we might reuse part of the answer or at least the context, avoiding duplicate computation. An LRU cache of recent queries helps when a user iteratively refines a question.

- Incremental Index Updates: As mentioned, the indexing agent updates only affected parts of the index when code changes. This avoids reprocessing the entire codebase for small modifications. It also means the system can be kept running and up-to-date continuously (for example, integrating with Git hooks to re-index on new commits).
- Parallel and Async Operations: The retrieval of code (both vector search and keyword search) is done asynchronously in parallel. We also parallelize embedding computations during indexing across files. The TypeScript async model helps overlap I/O (like reading files from disk) with CPU work (like computing embeddings).
- Model Quantization and Loading: Using Ollama, we run models that are quantized (4-bit weight quantization for larger models), which drastically reduces memory footprint and can even speed up inference due to better cache usage. We found that a 7B parameter model quantized to 4-bit can answer typical questions in around 2-3 seconds on an 8-core CPU machine, which is adequate for a conversational assistant. Meanwhile, a 13B 4-bit model might take around 5-7 seconds for the same, and larger models would be slower. By default, we opt for the 7B model for speed, but allow switching to a 13B model if the user requires more detailed answers and has the time.
- Balancing Precision and Recall: Our retrieval tries to fetch relevant context with slightly more emphasis on recall (to not miss something important). This sometimes means the model gets a bit more information than necessary. To ensure efficiency, we cap the number of snippets. The prompt is tuned so that the model isn't overwhelmed or sidetracked by tangential code. In practice, we found that providing 3 focused snippets yields better and faster answers than giving 10 loosely related ones. This balance keeps the answers accurate (since they have the key info) without unnecessary slow-down.

Through these optimizations, Codebase Guy maintains a relatively small resource footprint. The vector index of a medium-sized repository (say 50k lines of code resulting in a few thousand chunks) might be on the order of tens of megabytes, which is easily handled in memory. The LLM model, once loaded, is the main memory consumer (several GB), but remains constant regardless of query. The search operations are all very fast (millisecond range for index lookup). Thus, the primary latency in responding to a query is the LLM inference time, which we've optimized by keeping the context lean. The result is an AI assistant that feels responsive and can run on commodity hardware.

IV. Methodologies and Techniques

Codebase Guy combines several state-of-the-art methodologies in AI and information retrieval to achieve its capabilities. Here we highlight the key techniques and how we adapted them for our multi-repository code assistant scenario.

A. Retrieval-Augmented Generation (RAG)

Retrieval-Augmented Generation is a paradigm where an AI model is supplemented with an external knowledge base that it can query. Instead of relying solely on the knowledge encoded in its weights, the model "augments" its input with relevant data fetched from a repository or database. In Codebase Guy, the code index serves as this external knowledge. Before the LLM attempts to answer a question, the system performs retrieval to gather relevant code context. This methodology allows the use of smaller local models effectively, because the model doesn't need to have seen every possible API or code pattern during training—it can be given the exact code from the project at inference time.

A notable aspect of our RAG implementation is its use in a constrained environment. We borrowed strategies from RAG in NLP (like open-domain question answering systems) but tailored them to code. This includes the hybrid search (semantic + lexical) to deal with code-specific querying (since code has structure and identifiers), and the hierarchical context approach where we might use summaries for broad questions and actual code for detailed ones. By structuring the prompt with retrieved snippets, we essentially turn the free-form generative model into a grounded, project-specific assistant.

B. Hybrid Semantic-Lexical Retrieval

As discussed earlier, combining semantic and lexical search is crucial for code. Pure semantic search (embedding based) might miss an exact match on an error string, and pure lexical search might miss conceptually similar code that uses different naming. Our hybrid approach draws on research in information retrieval that shows combining different signals yields better results. In practice, one can implement this by either merging result lists or by cascaded filtering. We chose to first union the top candidates from both methods and then rank that union by a heuristic score. The heuristic could be a weighted sum of normalized semantic similarity and normalized BM25 score, or even a small learned model if we had training data. For now, a simple combination worked well: semantic similarity primarily ensures conceptual relevance, while lexical ensures precision on terms.

One advantage we observed with hybrid retrieval is improved **recall** (we rarely completely miss the relevant code in the top results) at the cost of some additional overhead (running two searches). However, since our indexes are local and the codebase sizes we target are moderate (on the order of thousands to tens of thousands of chunks), the extra overhead is negligible (both searches complete in a blink, maybe a few tens of milliseconds).

C. Hierarchical Summarization and Context Reduction

To manage very large contexts, Codebase Guy uses hierarchical summarization. This technique involves creating summaries at multiple levels of granularity. For example, during indexing, each function might get a one-line summary (its purpose), each file might get a paragraph summary (describing the main classes or functions in it), and an entire repository or module might have a summary describing its overall role. When a user question is broad (e.g., ”How does the caching system work?”), it might retrieve the module-level summary for ”caching” rather than details of specific functions. Those summaries are fed to the LLM, which then can ask follow-up or the user can drill down. Conversely, if the question is very specific (e.g., ”What is the return value of function X in scenario Y?”), the system will retrieve the exact function code or even the specific branch in the code if possible.

This approach ensures that we give the LLM **just enough** detail. Summaries compress the information so that the gist is conveyed in fewer tokens. Of course, summaries risk omitting details, so our system chooses when to use them: if a piece of code is too long or not directly needed, a summary is used; if the precise code logic is needed (like for debugging or ensuring correctness), the raw snippet is used. The hierarchical nature means the system could first supply a high-level answer and then progressively reveal more detailed code if asked.

We found that generating these summaries can be done reasonably well with the LLM itself by prompting it in a different mode. To keep things efficient, one could also generate summaries on the fly for a retrieved snippet if needed, but since we can do it offline, we prefer to store them.

D. Prompt Engineering and Few-Shot Hints

Even with relevant code context, the way the question and context are presented to the model (the prompt) greatly influences the output. We treated prompt engineering as an iterative design process. Early versions of Codebase Guy's prompt simply appended code and the question, which led to mixed results—the model sometimes ignored code or went off-topic. We then introduced a clear system instruction as shown earlier, explicitly telling the model to use the code and to be accurate. We also experimented with giving a few-shot example in the prompt (for instance, showing a dummy Q&A with code). While this can improve reliability, it consumes tokens and in our local setting with smaller models, we opted not to include a full example exchange in every prompt. Instead, we might have the system remember some formatting from earlier interactions.

One technique we do use is **instruction tuning** via the system message: e.g., ”If the question is about code, include references to filenames or function names in your answer. If you need more information, you can respond with a clarification question.” This helps the conversation feel more natural and interactive. In practice, the user rarely sees the prompt we construct; they only see the final answer, but these behind-the-scenes prompt tweaks help the model deliver better results.

E. Local Deployment Under Constraints

All these methodologies are chosen with local deployment in mind. Running everything locally means we avoid network latency and can integrate deeply with local files, but it also means careful use of CPU/GPU. We leveraged efficient libraries (for example, using WebAssembly or native addons for heavy math in Node, or offloading some tasks to Python scripts if needed for performance). We also ensure that if the system is idle, the model can be unloaded or switched to a low-power mode (to free memory for the developer's other tasks).

A key technique for constrained environments is using smaller specialized models rather than one giant model for everything. Codebase Guy uses: - A moderate-sized LLM for understanding and generating code explanations. - A separate embedding model for search (which could be smaller and faster). - Optionally, a distinct summarization model or a faster mode of the main model for quick tasks. This specialization means each task is done by a model suited for it. The embedding model, for instance, might be a tiny 100 million parameter model which is very fast, whereas the LLM might be 7B which is slower but more capable for generation. This division helps maintain overall efficiency.

In summary, the methodologies in Codebase Guy reflect a blend of IR (Information Retrieval) techniques and LLM integration tricks, all geared towards making a useful tool for developers without requiring supercomputer-level hardware.

V. Use Cases

The true value of Codebase Guy is best illustrated through example scenarios. Here we present a few real-world use cases that demonstrate how developers can utilize the system to enhance their workflow across multiple repositories.

A. Cross-Repository Code Comprehension

Consider an organization that has a microservices architecture with different repositories for each service. A developer is tasked with understanding how the ”Notification Service” interacts with the ”User Service” when a new user registers. Traditionally, the developer would have to manually find where in the Notification Service code it calls out to the User Service API, then open the User Service repository and see how that API is implemented.

With Codebase Guy, the developer can simply ask in natural language: *”How does the Notification Service use the User Service when a user signs up?”*. The assistant will interpret this question, perform a search across both repositories, and retrieve relevant snippets. For example, it might find in the Notification Service code a function that handles a ”User-Created” event and calls an API client for the User Service. It will also retrieve the User Service endpoint definition for handling that event or request. The model then assembles an explanation such as: *”When a new user signs up, the Notification Service receives a ‘UserCreated‘ event. In the ‘notifications/events.js‘ file, the handler ‘onUserCreated‘ calls ‘UserAPI.getUserDetails(userId)‘ (see code snippet). This is a client call to the User Service. On the User Service side, the ‘getUserDetails‘ function in ‘user/routes.js‘ fetches the user info from the database and returns it. The Notification Service uses that data to compose a welcome email.”*

This answer saves the developer from searching through multiple repos and reading boilerplate. It directly gives the integrated view of the interaction between services, citing specific files and functions.

B. Debugging and Issue Resolution

Imagine a scenario where an application composed of several components is throwing an error, e.g., ”Payment gateway timeout error”. The error is logged in one microservice, but the cause might be in another, or it could be a configuration issue. A developer with this bug report can ask Codebase Guy: *”What could cause a ’Payment gateway timeout’ error in our system?”*.

The assistant will search all repositories for the error message ”Payment gateway timeout”. Suppose it finds that this exact phrase appears in the Payment Service repository in a file ‘payment/client.ts‘ where a specific exception is thrown, and also finds references in a configuration file where a timeout duration is set. It provides those to the LLM. The response could be: *”The error ’Payment gateway timeout’ is thrown in ‘payment/client.ts‘ (Payment Service) when the external payment API does not respond within the configured timeout (see snippet). The config file ‘payment/config.yaml‘ sets ‘gatewayTimeout: 5000‘ (5 seconds). If the payment provider is slow or the network is down, this error will occur. Also, the Order Service calls the Payment Service and does not catch this exception, so it propagates and might crash the order workflow.”*

It might then suggest a solution: *”To resolve this, you might increase the timeout in the config or add retry logic in the Payment Service. Also ensure the Order Service handles the exception gracefully.”*. This use case shows how Codebase Guy can connect the dots across code (finding where the error is thrown) and configurations, and provide insight quickly.

Now imagine a more complex bug: a ticket says ”Users are not receiving password reset emails.” This could involve the Auth Service and Notification Service. The developer can ask: *”Why might password reset emails not be sending?”*. Codebase Guy might search for ”reset email” and find that the Notification Service expects a certain event or API call that the Auth Service isn’t making. For instance, maybe the Auth Service is supposed to emit a ”PasswordResetRequested” event but doesn’t. The assistant’s answer might highlight this by showing the code in Notification Service that listens for that event and noting that no corresponding emit is found in the Auth Service code. This effectively pinpoints a missing integration.

C. Exploratory Development and API Usage

Another use case is when a developer wants to use a function or API from another module but isn’t sure how. Suppose a developer in repository A wants to use a library from repository B. They might ask Codebase Guy: *”How do I use the PDF generation API from the Utilities repo?”*. The assistant will search the Utilities repository for anything related to PDF generation. It might find a ‘PdfGenerator‘ class with a function ‘generateReport(data, template)‘. It will also look for any usage examples, maybe tests or other services that call it. Upon gathering that, the assistant can reply with an explanation and example code: *”The Utilities repository provides a ‘PdfGenerator‘ class (in ‘utils/pdf.js‘). To use it, you need to instantiate it with a template. For example, as shown in ‘reporting-service/reportService.ts‘, you can do: ‘const pdfGen = new PdfGenerator(templatePath); const pdfBuffer = pdfGen.generateReport(data);‘. Make sure to handle the promise it returns. Also ensure the template file path is correct and accessible.”* and it might include a short code snippet showing this pattern.

This way, the developer learns by example, drawn from real usages possibly in other repos, how to use the functionality, without combing through readme files or asking someone else. Codebase Guy in this scenario acts like a smart internal documentation tool that not only points to the relevant function but also how it’s used in context.

D. Multi-Repository Refactoring Assistant

For a final use case, consider a refactoring task: the team wants to rename a microservice or change an API contract that affects multiple repositories. Doing this safely requires finding all the places to change. Codebase Guy can assist by being asked: *”I’m renaming the ’Billing’ service to ’Payments’. What do I need to update?”*. The assistant will search for ”Billing” across all repositories. It might find references in config files (like service registry or environment configs), in code (function names like ‘BillingServiceClient‘), and documentation strings. It can present a consolidated list or description: *”The term ’Billing’ appears in the following places: (1) ‘gateway/config.json‘ – as a service name for routing, (2) ‘billing-service/‘ repository (which is obviously the service name), (3) ‘order-service/billingClient.ts‘ – class name and references to the Billing service API, (4) ‘README.md‘ in the Ops repo – mentions Billing service. All these should be changed to ’Payments’. The API endpoints in ‘order-service/billingClient.ts‘ (e.g. ‘/api/billing/...‘) will need to be updated to ‘/api/payments/...‘ to match the new service name.”*.

The developer can use this as a checklist for their refactoring, ensuring nothing is missed. Future versions of Codebase Guy might even automatically generate a set of patches for these changes, but even identifying them is hugely helpful.

These use cases demonstrate that Codebase Guy can adapt to many scenarios: Q&A about code behavior, debugging, learning how to use internal APIs, and planning code modifications. By spanning multiple repositories, it provides a unified view that developers traditionally lack with siloed tools.

VI. Future Enhancements

While Codebase Guy already offers a robust set of features for AI-guided development, there are several planned enhancements that aim to push its capabilities even further. We outline some key future directions below.

A. Autonomous Coding Agent

Currently, Codebase Guy provides information, explanations, and code snippets on request. A natural next step is developing a **coding agent** that can take higher-level instructions and actually perform code modifications or generate new code modules on behalf of the developer. This goes beyond writing a snippet in the chat — it involves understanding user intent for a code change, editing multiple files, and potentially testing those changes.

For example, a developer could say, ”Add logging to all database calls in the User Service.” Rather than just explaining how to do that, an autonomous coding agent would attempt to implement it: scanning the User Service repository for database call sites, inserting appropriate logging statements, and then presenting a diff or patch to the developer for review. To accomplish this, the agent would need to combine the retrieval capabilities (to find where changes need to be made) with generation capabilities (to write the new code). It would also need to operate with caution, perhaps running tests or ensuring syntax correctness after changes.

We plan to integrate such an agent as a special mode of Codebase Guy. Under the hood, this might involve a series of LLM calls and tools orchestrated to plan the changes and apply them. Safety checks (like not introducing syntax errors or preserving logic) are crucial. Initially, the coding agent might focus on boilerplate tasks like adding logging, updating deprecated API usage, or creating template code for a new module, where the scope is well-defined.

B. Multi-Agent Orchestration

As we introduce more specialized agents (one for retrieving information, one for writing code, one for searching the web, etc.), an orchestrator becomes important. The **Orchestrator** in future versions of Codebase Guy will manage these multiple agents, deciding which tool or agent to invoke at each step to best serve the user’s request. This concept is inspired by recent advancements in agent frameworks where an LLM can reason about using different tools.

In practice, the orchestrator could be another layer of the system that receives the user query and has a decision-making policy: e.g., ”If the query asks for code changes, engage the coding agent; if it asks a question, engage the Q&A agent; if external info is needed, use the web search agent.” It might also break a complex task into sub-tasks: for instance, for ”Implement feature X”, the orchestrator could break it into ”find similar implementations or references”, ”generate code for X”, ”review and test code”.

We envision the orchestrator using a form of chain-of-thought prompting internally. It may use a higher-level LLM prompt like: ”User asked for Y. Step 1: do I have the info in code or do I need to search web? Step 2: retrieve code A, B. Step 3: ask coding agent to modify code C.” Each step would be carried out and fed back into the loop. This design would allow Codebase Guy to handle more elaborate tasks that require planning or combining multiple information sources.

C. Enhanced Knowledge Integration

Currently, the system is primarily focused on source code. A future enhancement is to integrate other artifacts of the software development lifecycle to provide even richer assistance. This includes:

- **Documentation and Wikis**: Incorporating internal documentation pages or design docs into the retrieval index. This can help answer ”why” questions (e.g., ”Why was library X chosen for this task?”) if the info is documented somewhere.
- **Issue Trackers and Commit History**: By linking to JIRA or GitHub issues, Codebase Guy could provide context such as ”This function was changed in commit ABC to fix issue #1234, which was about improving performance.” This temporal and rationale information can be invaluable. If a user asks, ”Has the payment timeout value changed recently?”, the system could actually search commit messages or issue descriptions to answer that.
- **Runtime Logs (optional)**: In a debugging scenario, connecting to recent log databases could allow the assistant to see actual runtime error logs and correlate them with code. This would effectively bring an observability angle to the assistant.

Integrating these sources requires expanding the indexing and retrieval mechanisms (for example, storing vector embeddings of documentation paragraphs, or indexing issue titles and descriptions). The challenge is to keep the search precise—code queries should primarily return code, whereas design questions might return docs. The orchestrator can help determine the context.

D. Improved Conversational Memory and Learning

While Codebase Guy can handle a multi-turn conversation, the current approach is relatively stateless beyond using previous turn content. We plan to enhance this by giving the system a memory of past interactions and the ability to learn from feedback. For example, if a developer corrects the assistant or provides additional info (”No, that function was removed last month.”), the system could incorporate that knowledge (update the index or at least remember it for the session). Over time, this could lead to a personalized assistant that knows a developer’s preferences or common environment specifics.

Another aspect is keeping track of what parts of the code a developer has seen or discussed, to avoid redundant explanations. If earlier in the session the user already got an explanation of a certain module, the assistant can refer back to it instead of repeating it verbatim, which makes the conversation more efficient.

E. Performance and Scalability

Looking ahead, we intend to test Codebase Guy on much larger codebases (hundreds of thousands to millions of lines, and dozens of repositories). This will likely require more scalable infrastructure, such as sharding the vector index or using distributed search. We may employ more sophisticated indexing like AST-based chunking (so that references between chunks can be traced, enabling jumping directly to function definitions across the code graph). Adaptive retrieval techniques could be added, where the system learns which parts of the codebase are frequently relevant and which can be ignored to speed up search.

Finally, we are exploring ways to incorporate user feedback signals to continuously improve the relevance of retrieval. If the assistant frequently pulls a snippet that the user ignores or finds unhelpful, the system could learn to de-prioritize that in the future. Conversely, if certain files are often the answer, they could be cached or their embeddings refined.

In summary, the future roadmap of Codebase Guy is geared towards evolving from a powerful assistant to an even more autonomous collaborator in software development. By adding coding actions, better orchestration of specialized agents, deeper integration with project knowledge, and scaling up performance, we aim to cover more aspects of the development cycle and handle larger enterprise codebases while maintaining efficiency.

VII. Conclusion

We presented *Codebase Guy*, an AI-driven development assistant that efficiently supports understanding and working with large, multi-repository codebases. By leveraging local LLMs in conjunction with a retrieval-augmented generation approach, the system provides developers with a conversational interface to ask questions about their code, get explanations, and even receive help with coding tasks. The architectural design — combining a chat UI, an embedding-powered search index, and orchestrated agent capabilities — enables the assistant to deliver relevant information quickly, all while running on modest hardware in a privacy-preserving manner.

A core theme of Codebase Guy is **efficiency**. Through hybrid search techniques, context truncation, summarization, and careful prompt engineering, we show that it's possible to drastically reduce the token footprint and computation required for code intelligence. In practice, this means response times of only a few seconds using local models, which makes the tool practical for day-to-day use by developers. We demonstrated via use cases that the assistant can handle tasks like cross-repo navigation, debugging, and API usage queries that traditionally consume a lot of developer time. By automating the retrieval of relevant code and providing natural language insight, Codebase Guy helps developers focus on higher-level problem solving rather than wrestling with search tools or reading through large amounts of code.

Our implementation in TypeScript, together with tools like Ollama for LLM serving, validates that such a system can be built with readily available technologies. The approach is model-agnostic and can benefit from advances in both language models and search algorithms. As those improve, Codebase Guy can seamlessly adopt a more powerful model or a faster index to further enhance its capabilities.

In conclusion, Codebase Guy exemplifies a promising direction in software engineering: the fusion of AI with software development workflows. It provides a template for how AI assistants can be deeply integrated with code, not just as black-box code generators but as informed, context-aware partners in development. We believe that with continued refinement and the planned future features (like an autonomous coding agent and multi-agent orchestration), tools like Codebase Guy will become indispensable in managing the complexity of modern codebases, ultimately improving developer productivity and software quality.