CodeBuddy: Open Source Code Onboarder

Executive Summary

CodeBuddy, an intelligent, locally hosted onboarding assistant for open-source software (OSS) developers. The system leverages a dual-path architecture combining Retrieval-Augmented Generation (RAG) for static code analysis and Agentic Tool Calling for live repository data retrieval. By utilizing local Large Language Models (LLMs) via Ollama, CodeBuddy ensures zero-cost scaling and absolute data privacy.

---

1. Introduction

1.1 The Problem: Open-Source Onboarding Friction

Open-source software thrives on community contributions. However, new contributors face a steep learning curve when deciphering massive, undocumented, or complex codebases. Traditional search tools (like `grep` or IDE text searches) match keywords but fail to explain semantic logic, architecture, or data flow. This friction leads to developer burnout and abandoned pull requests.

1.2 The Solution

CodeBuddy is a conversational AI assistant that directly ingests a GitHub repository and answers developer questions in real-time. It translates complex code into plain English, explains system architecture, and interacts with the GitHub API to check live issues—all without the developer needing to leave their local environment.

---

2. High-Level Architecture Overview

The CodeBuddy system is built on a "split-brain" routing architecture to handle the distinct needs of static code comprehension and dynamic repository status updates.

The architecture is divided into three primary functional blocks:

1. **The Orchestration Layer:** Handles user input, classifies intent, and routes the query.
2. **The RAG Pipeline (Static Knowledge):** Clones the codebase, chunks it, embeds it into a local vector database, and generates contextual explanations.
3. **The Agentic Pipeline (Dynamic Knowledge):** Bypasses the database to query live REST APIs for real-time data.

3. System Design & Decision Matrix

3.1 Orchestration: Conversation & Intent Routing

Before processing a query, the system must understand *what* the user is asking. Sending every query to a vector database is inefficient and often inaccurate if the user is asking for real-time data.

- **Implementation:** I implemented an LLM-based Intent Router utilizing PromptTemplate and JSON enforcement. The user's query is classified into specific intents: PROJECT_SETUP, CODE_EXPLANATION, LIVE_ISSUES, or AMBIGUOUS.
- **Decision:** Use a lightweight local LLM prompt configured with temperature=0.2 and strict JSON output schema.
- **Justification:** A low temperature ensures the model acts deterministically. Enforcing JSON output allows our Python backend to programmatically parse the routing decision without regex parsing errors. The inclusion of the AMBIGUOUS intent acts as a human-in-the-loop guardrail, preventing hallucinations by asking the user for clarification rather than guessing.

3.2 Knowledge Acquisition: Dynamic Data Ingestion

To be useful, the assistant must ingest code dynamically from the web.

- **Implementation:** I utilized LangChain's GitLoader to clone public repositories directly into a temporary local directory.
- **Decision:** Programmatic git clone via LangChain over manual file uploads or static local directories.
- **Justification:** This allows CodeBuddy to be highly dynamic. A developer can point it at any public repository, and the system prepares itself automatically.I implemented an OS-level cleanup script (shutil.rmtree) to ensure previous repository data is wiped before a new ingestion, preventing database contamination.

3.3 Processing: Code-Aware Chunking & Vector Storage

LLMs have strict context window limits.I cannot feed an entire repository into a single prompt.

- **Implementation:** The raw code is split using LangChain's RecursiveCharacterTextSplitter.from_language(Language.PYTHON). These chunks are converted to high-dimensional vectors using local OllamaEmbeddings and stored in a local Chroma database.

- **Decision 1 (Chunking):** Use an AST-aware (Abstract Syntax Tree) / Language-specific text splitter rather than a standard character splitter.
    - *Justification:* Standard splitters break text blindly based on character count. A language-aware splitter understands code syntax, ensuring that a Python `def` or `class` block is kept intact within a single chunk, preserving the logical context for the LLM.
- **Decision 2 (Embeddings & Database):** Local Llama 3.1 embeddings and ChromaDB.
    - *Justification:* ChromaDB is an open-source, lightweight, embedded vector database that requires no external server setup. Generating embeddings locally ensures that proprietary or sensitive repository code is never sent to a third-party server (like OpenAI), ensuring absolute privacy.

3.4 Retrieval-Augmented Generation (RAG) Engine

Once the data is stored, the system needs to retrieve it accurately to answer code-specific queries.

- **Implementation:**I utilized LangChain's `create_retrieval_chain` to link a ChromaDB Retriever (fetching the top `k=4` most relevant chunks) with an LLM Document Chain (`create_stuff_documents_chain`).
- **Decision:** Instructing the LLM via System Prompts to strictly rely on retrieved context.
- **Justification:** Open-source models can confidently hallucinate code. By strictly defining the persona ("You are CodeBuddy...") and the constraints ("If the context does not contain the answer, say you do not know"),I drastically reduce hallucinations. The system also outputs the source file names metadata alongside the answer, establishing trust and traceability for the developer.

3.5 Agentic Extension: Function Calling (Tool Use)

Static RAG cannot answer questions like "Are there any open PRs?" because the vector database is only a snapshot of the code at the time of ingestion.

- **Implementation:**I designed a Python function wrapped in a LangChain `@tool` decorator that pings the GitHub REST API for open issues and PRs.I upgraded the routing LLM to an Agent (`create_tool_calling_agent`) capable of autonomous decision-making.
- **Decision:** Implement an Agentic Tool Call path specifically for the `LIVE_ISSUES` intent.
- **Justification:** This upgrades the system from a passive search engine to an active agent. By parsing the GitHub API natively, the LLM can synthesize live JSON data (which is often messy) into a clean, human-readable summary of repository health and active work.

---

4. Conclusion and Future Roadmap

CodeBuddy successfully demonstrates how local Large Language Models can be orchestrated into complex, split-brain architectures to solve real-world developer friction. By combining semantic vector search with real-time agentic tool calling, it provides a comprehensive onboarding experience.

Future Enhancements (Phase 4):

1. **Multi-Language Support:** Expand the GitLoader file filters and Text Splitters to dynamically support JavaScript/TypeScript, Rust, and Go within the same repository.
2. **Cross-Repository Context:** Upgrade the Chroma vector store to accept multiple namespaces, allowing developers to query how different microservices within an organization interact.
3. **Automated PR Review Agent:** Extend the tool-calling agent to fetch the diff of a specific Pull Request and critique it based on the coding standards found in the RAG vector database.

Code in: https://github.com/jagdsh/gen_ai_code_onboarder.git