Roadmap for Building a Local Agentic Coding Dashboard

This guide provides a step-by-step strategic roadmap for architecting and implementing a sophisticated, autonomous coding agent that operates entirely on your local hardware. The design is heavily influenced by the principles observed in advanced systems like GitHub Copilot's agent mode, focusing on a modular, tool-augmented, and context-aware architecture.

1. Required Frameworks and Libraries

Your foundation will consist of an agent orchestration framework, a local model server, a vector database, and a dashboard interface.

- **Agent Orchestration:**
  - **LangChain (with LangGraph):** Highly recommended. LangChain provides the core building blocks for agents, tools, and memory. LangGraph is essential for creating the cyclical, stateful workflows (like plan -> act -> observe -> reflect) that define modern agentic systems, moving beyond simple linear chains.
  - **CrewAI:** A solid alternative that offers a higher-level, role-based abstraction for creating collaborating agents (e.g., a "Senior Developer" agent and a "QA Engineer" agent). It simplifies multi-agent orchestration.
- **Local Model Serving:**
  - **Ollama:** The simplest and most effective way to download, manage, and serve a wide variety of open-source LLMs on your local machine via a consistent API.
- **Vector Database (for RAG):**
  - **ChromaDB:** An open-source, developer-friendly vector database that's easy to set up and run locally. Perfect for storing codebase embeddings.
  - **FAISS (from Meta):** A library for efficient similarity search. More of a library than a full database, but highly performant for developers who want more control.
- **Dashboard & Interface:**
  - **Streamlit or Gradio:** The fastest way to build an interactive Python-based web UI for your dashboard to send tasks and visualize the agent's thinking process.
- **Sandboxing:**
  - **Docker SDK for Python:** Essential for safely executing agent-generated code in isolated environments.

2. Core Architecture and Protocols

Your architecture should be a modular system centered around an orchestrator that manages communication using a standardized protocol.

- **Core Architectural Pattern:** The system should follow an **Orchestrator-Agent-Tool** model.
  1. **Orchestrator (Main Loop):** The central engine that manages the agent's cognitive cycle (Reason -> Act -> Observe). This is where you'll implement your primary LangGraph or CrewAI workflow.
  2. **Specialized Agents/Roles:** A single LLM can be prompted to adopt different roles. Your orchestrator should call the LLM with system prompts for distinct tasks:
     - **Planner Agent:** Decomposes the user's high-level goal into a sequence of concrete steps.
     - **Contextualizer Agent:** Uses Retrieval-Augmented Generation (RAG) to fetch relevant code snippets and documentation from memory.
     - **Coder Agent:** Writes or modifies code based on the plan and context.
     - **Debugger/Tester Agent:** Executes code or tests, observes the output (errors, test failures), and feeds the results back into the loop for self-correction.
- **MCP-Style Context Protocol:** You don't need to implement the full MCP standard, but you must adopt its core principle: **a standardized communication schema**.
  - **Define a JSON Schema:** Create a clear, consistent JSON format for all messages passed between components. This is the "USB port" for your system.
  - **Key Message Types:**
    - tool_call: A message from the agent to the orchestrator requesting the execution of a tool (e.g., {"tool_name": "run_in_terminal", "arguments": {"command": "pytest"}}).
    - tool_observation: A message from a tool back to the agent with the result (e.g., {"tool_name": "run_in_terminal", "output": "2 tests passed, 1 failed..."}).
    - memory_request: A message to the memory module to retrieve context.
    - state_update: A message that updates the agent's internal "scratchpad" or plan.
  - This protocol decouples the agent's reasoning logic from the tool implementations, making the system extensible.

3. Memory and Storage Design

Memory grounds your agent in the context of your codebase, preventing it from hallucinating and enabling it to follow project-specific patterns.

- **Long-Term Memory (LTM) - The Knowledge Base:**
  - **Database:** Use a local vector database like **ChromaDB**.
  - **Structure:** The LTM will store *embeddings* of your codebase and documentation. You will ingest your entire project directory, splitting files into smaller, semantically meaningful chunks (e.g., functions or classes).
  - **Retrieval Operation (RAG):** When the agent needs context, its query is converted into an embedding. The vector database performs a similarity search to find the most relevant chunks of code/docs, which are then "augmented" into the LLM's prompt.
- **Short-Term Memory (STM) - The Scratchpad:**
  - **Structure:** This holds the context for the *current task*. It includes the initial goal, the generated plan, the history of tool calls and observations, and any self-correction notes.
  - **Implementation:** In LangGraph, this is managed by the central state object. For simpler setups, a Python dictionary or class that persists through the agent's execution loop is sufficient.

4. Model Selection and Specification

Running locally requires a careful balance between model capability and hardware constraints.

- **Recommended Local LLMs:**
  - **Code Llama (7B or 13B):** Specifically fine-tuned for code generation and understanding. An excellent starting point.
  - **Mistral (7B) / Mixtral (8x7B):** Highly capable general-purpose models that perform very well on coding tasks.
  - **Phi-3 (Mini or Small):** Surprisingly powerful small models from Microsoft that are excellent for resource-constrained systems.
  - **Key Consideration:** Always use **instruction-tuned** and **quantized** (e.g., GGUF, AWQ) versions of these models to reduce VRAM and RAM usage.
- **Minimum Hardware Requirements:**
  - **CPU:** Modern 8-core+ processor.
  - **GPU:** An NVIDIA GPU with **at least 8 GB of VRAM** is strongly recommended for reasonable performance. 12-16 GB is ideal for running larger, more capable models.
  - **RAM:** 32 GB. (16 GB is a bare minimum but you may face swapping issues).
  - **Disk:** A fast SSD with at least 50 GB of free space for models and databases.

5. Tool Integrations

Tools are what allow the agent to interact with the world and turn plans into actions. Safety is paramount.

- **Code Execution Environment:**
    - **Sandboxing is Mandatory:** Never execute LLM-generated code directly on your host machine.
    - **Implementation:** Create a run_code tool that uses the **Docker SDK** to spin up a temporary, isolated container. It should copy the necessary files, run the command, capture the stdout and stderr, and then tear down the container.
- **Version Control Interface:**
    - Implement tools that are thin wrappers around **Git commands**:
        - read_file(path)
        - write_file(path, content)
        - list_files()
        - apply_git_diff(patch_string)
- **Testing and Debugging Utilities:**
    - Create tools to run your project's test suite (e.g., a run_tests tool that executes pytest or npm test in the Docker sandbox).
    - The tool must be able to parse the test output to determine success or failure and return the results as a structured observation.

6. Implementation Steps: A Strategic Roadmap

Follow these steps in sequence to build your dashboard from the ground up.

1. **Step 1: Set Up Your Development Environment**
    - Install Python, Docker Desktop, and your IDE (e.g., VS Code). Create a virtual environment for the project.
2. **Step 2: Install and Configure Frameworks**
    - Install **Ollama** and pull a coding model (e.g., ollama pull codellama).
    - pip install langchain langgraph crewai chromadb-client streamlit docker openai.
    - Start the ChromaDB server (or configure it to run in-memory).
3. **Step 3: Define Your MCP Context Protocol Schema**
    - In a Python file, define Pydantic models or simple dictionaries for your tool_call and tool_observation message structures. This formalizes your internal communication.
4. **Step 4: Build the Memory-Storage Layer (RAG Pipeline)**
    - Write a script (ingest.py) that can:
        - Point to a local code repository.
        - Load the files and split them into chunks (using LangChain's

RecursiveCharacterTextSplitter).
        - Generate embeddings for each chunk using a local embedding model via LangChain.
        - Store these embeddings in your ChromaDB instance.
    - Test the retriever by writing a simple query function.
5. **Step 5: Integrate Your Local LLM**
    - Use LangChain's Ollama integration to connect to your running model.
    - Create a simple agent (e.g., create_tool_calling_agent in LangChain) and test that it can make a basic call to the LLM and receive a response.
6. **Step 6: Wire Up Tool Callbacks**
    - Implement the Python functions for each of your core tools (read_file, write_file, run_code_in_docker, run_tests).
    - Define these as Tools within your agent framework (LangChain or CrewAI) so the LLM knows about their existence, descriptions, and arguments.
7. **Step 7: Build and Test the End-to-End Agentic Workflow**
    - Using **LangGraph**, design your agent's state graph. Nodes will represent actions like PLAN, RETRIEVE_CONTEXT, EXECUTE_TOOL, and GENERATE_RESPONSE. Edges will define the conditional logic (e.g., on error, go to DEBUG node; on success, proceed to next step).
    - Give the agent a simple, end-to-end task (e.g., "Add a docstring to the calculate function in main.py").
    - Trace the execution meticulously. Log every step: the initial plan, the RAG results, the tool calls, and the final output. This is the most critical debugging phase.
8. **Step 8: Build the Dashboard and Optimize**
    - Create a simple **Streamlit** application that provides a text input for your high-level goal.
    - The app should call your agent orchestrator and display the agent's thought process, current plan, and final proposed code changes in real-time.
    - Investigate model quantization (e.g., using 4-bit GGUF files) to significantly reduce resource consumption and improve the dashboard's responsiveness.