# **OllaCode: An Architectural Blueprint for a Local-First, Agentic AI Assistant in VS Code**

## **Section 1: Foundational Architecture & Project Scaffolding**

This inaugural section establishes the complete groundwork for the OllaCode extension. The process moves from conceptual architecture to a tangible, structured project ready for development. The focus is on creating a clean, maintainable, and scalable foundation that adheres to Visual Studio Code extension development best practices.

### **1.1. System Design: The Extension Host and Webview Dichotomy**

The architecture of OllaCode is built upon a strict separation of concerns, a pattern enforced by the Visual Studio Code extension environment itself. The system is divided into two primary, isolated components: the Extension Host and the Webview.

The **Extension Host** is a dedicated Node.js process managed by VS Code.1 All core logic, business operations, and interactions with the system reside here. This includes communication with the Ollama server, direct access to the complete VS Code API for workspace manipulation (e.g., reading/writing files), and the orchestration of the agentic engine. Placing this logic in the extension host ensures stability, as a crash in this process is isolated from the main VS Code window, and provides the necessary permissions to perform powerful development tasks.

The **User Interface (UI)**, primarily the chat panel, is rendered within a **Webview**.3 A Webview is fundamentally an

iframe sandboxed within VS Code, capable of rendering standard web technologies: HTML, CSS, and JavaScript. This architectural choice offers two significant advantages. First, it decouples the UI from the backend logic, preventing long-running UI processes or rendering glitches from freezing the extension's core functionality. Second, it provides complete freedom to design a rich, interactive, and custom user interface without the constraints of native VS Code UI toolkits.

A critical element of this design is the **Communication Protocol** that bridges these two components. Because the Extension Host and the Webview run in separate processes, they cannot share memory or call functions directly. All communication must occur through an asynchronous message-passing bridge. The Webview sends messages to the Extension Host to request actions (e.g., "user sent a new chat message"), and the Extension Host sends messages to the Webview to provide data or UI updates (e.g., "here is a new token from the AI stream").3 Designing this message contract is a foundational architectural step that dictates the data flow for the entire application.

### **1.2. Project Initialization and Directory Structure**

To ensure a standardized and correct starting point, the project is initialized using the official Yeoman extension generator, as recommended by the Visual Studio Code documentation.5 This tool scaffolds a complete, ready-to-debug extension project.

The generator is invoked with the following command:

Bash

npx --package yo --package generator-code -- yo code

During the setup process, "New Extension (TypeScript)" is selected to leverage the benefits of static typing, which improves code quality, maintainability, and provides a superior developer experience with features like IntelliSense within the project.2

A well-defined directory structure is established to organize the codebase logically and ensure it remains manageable as complexity grows.

* src/: The root directory for all extension source code.
  + extension.ts: The primary entry point for the extension. It contains the mandatory activate and deactivate lifecycle functions that VS Code calls.2
  + services/: This directory will contain modules responsible for interfacing with external systems.
    - ollamaService.ts: A dedicated client for all communication with the Ollama REST API.
  + providers/: This directory houses implementations of VS Code's various language feature APIs.
    - completionProvider.ts: The implementation for providing inline code completions.
  + webview/: This directory contains all assets and source code for the chat UI that runs inside the Webview.
    - index.html: The HTML structure of the chat panel.
    - style.css: The stylesheet for the chat panel.
    - main.ts: The client-side TypeScript logic for the chat UI.
  + agent/: This directory is dedicated to the agentic mode's functionality.
    - agentOrchestrator.ts: The core class that manages the agentic loop.
    - toolbelt.ts: The registry for all available agentic tools.
    - tools/: A subdirectory containing the implementation of each individual tool (e.g., readFileTool.ts).
  + common/: This directory is for shared code, specifically TypeScript types and interfaces (e.g., types.ts) that need to be used by both the Extension Host and the Webview to maintain a consistent data contract.

For the build process, esbuild is configured as the bundler. It is a high-performance JavaScript bundler that will compile and bundle all TypeScript source files into a single, minified JavaScript output file. This is a modern best practice for VS Code extensions as it significantly improves load times and simplifies distribution.3

### **1.3. The Extension Manifest: package.json**

The package.json file serves as the extension's manifest, a declarative JSON document that tells VS Code what the extension does and how it integrates into the IDE's UI and command structure.2 It contains a mix of standard Node.js fields and VS Code-specific contribution points.

Key fields are configured as follows:

* name, publisher, version: These fields are essential for creating a unique identifier for the extension (e.g., my-publisher.ollacode) used by VS Code and the Marketplace.6
* engines.vscode: This property specifies the minimum version of VS Code the extension is compatible with, which is critical for ensuring the availability of the APIs used in the code.2
* main: This defines the entry point to the extension's code, pointing to the bundled output file (e.g., ./dist/extension.js).6
* activationEvents: This is one of the most critical fields for performance. To prevent OllaCode from slowing down VS Code's startup, it will not be activated on launch. Instead of the inefficient wildcard (\*) event, specific activation events are used.8 The extension will be activated only when the user explicitly interacts with its features, such as by opening its chat view (  
  onView:ollacode.chatView).
* contributes: This JSON object is where all static contributions to VS Code are declared. This includes commands, menus, UI views, and configuration settings. The following table details the specific contributions for OllaCode.

### **Table 1: package.json Contribution & Activation Manifest**

| Contribution Point | Value / Example | Purpose & Rationale | Relevant Sources |
| --- | --- | --- | --- |
| commands | [{ "command": "ollacode.ask", "title": "OllaCode: Ask", "category": "OllaCode" }] | Registers a unique command ID that serves as the programmatic hook for a feature. This allows the action to be triggered by keybindings, menus, or API calls. | 2 |
| menus.commandPalette | [{ "command": "ollacode.ask", "when": "false" }] | Controls the visibility of a command in the Command Palette. For OllaCode, commands will primarily be triggered from dedicated UI elements, so they are hidden from the palette to reduce clutter. | 11 |
| viewsContainers.activitybar | [{ "id": "ollacode-sidebar", "title": "OllaCode", "icon": "media/icon.svg" }] | Creates a new icon in the VS Code Activity Bar. This provides a persistent, top-level entry point for all of the extension's features, grouping them in a dedicated sidebar. | 1 |
| views.ollacode-sidebar | [{ "id": "ollacode.chatView", "name": "Chat", "type": "webview" }] | Populates the custom activity bar container created above. This entry defines a view with a unique ID (ollacode.chatView) and specifies its type as webview, indicating that it will host our custom HTML-based chat interface. | 1 |
| configuration | (See Section 6 for full schema) | Defines the user-configurable settings that will appear in the VS Code Settings UI. This is crucial for allowing users to specify their Ollama URL and select models. | 10 |
| activationEvents | ["onView:ollacode.chatView"] | Declares the specific event that will activate the extension. By using onView:ollacode.chatView, the extension's code is loaded lazily, only when the user explicitly opens the OllaCode chat panel, ensuring minimal impact on VS Code's startup time. | 2 |

## **Section 2: The Communication Bridge: Interfacing with Local LLMs**

This section details the construction of the networking layer responsible for all communication with the locally running Ollama server. This service is designed to be robust, configurable, and capable of handling the streaming responses essential for a responsive AI chat experience. It acts as the sole intermediary between the extension's logic and the AI model backend.

### **2.1. The Ollama REST API**

The extension will interface with the standard Ollama REST API. By default, this API is served on localhost:11434 after the user installs and runs Ollama on their machine.15 The API is well-documented and provides several key endpoints that OllaCode will utilize to deliver its features.16

* **POST /api/chat**: This is the primary endpoint for conversational interactions. It accepts a history of messages, each with a designated role (system, user, or assistant), and can stream the response back in chunks. This stateful, streaming nature is fundamental for both the chat UI and the agentic mode, which rely on maintaining conversational context.16
* **POST /api/generate**: This endpoint is designed for single-turn, stateless generation tasks. It takes a single prompt string and generates a completion. Its statelessness makes it ideal for the inline code completion feature, which requires a quick, context-free suggestion without the overhead of a full conversation history.16
* **GET /api/tags**: This endpoint returns a list of all models that the user has downloaded and are available in their local Ollama instance (e.g., llama3:latest, codestral:latest). This endpoint is essential for implementing the user-configurable model selection feature, as it allows the extension to dynamically discover which models it can offer as choices.15

### **2.2. Building the OllamaService TypeScript Client**

A dedicated TypeScript class, OllamaService, will be created in src/services/ollamaService.ts to encapsulate all logic for interacting with the Ollama API. This centralization prevents API call logic from being scattered throughout the codebase and makes it easier to manage, update, and debug.

* **Technology Choice**: The service will use the standard node-fetch library, a lightweight module that brings the browser's fetch API to Node.js. This provides a modern, promise-based interface for making HTTP requests from the extension host.19 The dependency will be added to the  
  package.json.
* **Core Functionality**:
  + **Model Listing (listModels)**: An asynchronous function listModels(): Promise<string> will be implemented. It will perform a GET request to the /api/tags endpoint. Upon a successful response, it will parse the JSON, extract the name field from each object in the models array, and return an array of model tag strings. This list will be used to populate the extension's settings.
  + **Streaming Chat (streamChat)**: A function streamChat(messages: Message, options: { model: string }): EventEmitter will be the workhorse for conversational features.
    1. It will construct a request body containing the array of messages and the model specified in the options, along with setting stream: true.16
    2. It will initiate a POST request to the /api/chat endpoint.
    3. Crucially, it will not wait for the entire response to complete. Instead, it will listen to the response body's data events. The Ollama API streams a series of newline-delimited JSON objects.15 The function will parse each chunk of data as it arrives.
    4. For each valid JSON chunk, which represents a piece of the AI's response, it will use a Node.js EventEmitter to emit a data event, passing the content to the listener. This allows the calling code (e.g., the chat UI manager) to receive and display the AI's response token-by-token, creating a real-time typing effect.
  + **Code Completion (generateCompletion)**: A simpler, non-streaming async function generateCompletion(prompt: string, options: { model: string }): Promise<string> will be implemented. It will call the /api/generate endpoint with stream: false. It will await the full response, parse the JSON, and return the completed response field as a single string. This is optimized for the quick, one-shot nature of code completion.

The deliberate separation of the /api/chat and /api/generate endpoints into distinct methods within the OllamaService is a key architectural choice. The generate endpoint is stateless and optimized for quick, single-shot completions, making it a perfect match for the inline code completion feature where context is built and sent in a single prompt. Conversely, the chat endpoint is stateful, designed to handle an array of messages with specific roles (system, user, assistant). This structure is essential for maintaining the history of a conversation in the chat view and for the agentic loop, where the agent's internal thoughts and the results of tool executions must be added back into the conversation to inform the next step. Using the wrong endpoint for a given feature would result in either inefficient context management or a complete failure to maintain state.

### **2.3. Configuration and Error Handling**

The OllamaService is designed for robustness and user customizability.

* **Dynamic Configuration**: The service will not contain any hardcoded values for the Ollama server address or model names. Before making any API call, it will retrieve the necessary configuration from VS Code's settings using the vscode.workspace.getConfiguration('ollacode') API. For example, it will fetch the ollama.baseUrl and the appropriate model for the current task (model.chat or model.completion).12 This makes the extension adaptable to any user's local setup.
* **Robust Error Handling**: The service will be wrapped in try...catch blocks to handle potential failures gracefully. It will differentiate between network errors (e.g., the Ollama server is not running or the URL is incorrect) and API errors (e.g., the requested model is not available, resulting in a 404). In case of an error, it will not crash the extension. Instead, it will log the detailed error to the developer console and present a user-friendly, non-intrusive error message to the user via the vscode.window.showErrorMessage API, for example: "OllaCode: Could not connect to Ollama server at http://localhost:11434. Please ensure Ollama is running and the URL is correct in settings.".12

### **Table 2: Ollama API Endpoint Specification**

| Endpoint | Method | Key Request Parameters | Response Type | OllaCode Use Case |
| --- | --- | --- | --- | --- |
| /api/chat | POST | model (string), messages (array), stream (boolean) | Streaming JSON objects | Core chat functionality and the agentic reasoning loop, where maintaining conversation history is essential. |
| /api/generate | POST | model (string), prompt (string), stream (boolean) | Streaming JSON objects or a single object | Inline code completion, which is a stateless, single-shot request for a quick suggestion. |
| /api/tags | GET | (none) | Single JSON object with a models array | Dynamically populating the model selection dropdowns in the extension's settings UI. |

## **Section 3: Core Feature Implementation: Proactive Code Completion**

This section provides the complete implementation for the inline code completion feature. This functionality, often referred to as IntelliSense, is a cornerstone of modern AI coding assistants. The implementation leverages Visual Studio Code's CompletionItemProvider API to inject intelligent, context-aware suggestions directly into the editor as the user types, powered by the local Ollama instance.

### **3.1. Registering the CompletionItemProvider**

The connection between OllaCode and the VS Code editor's completion feature is established within the activate function of src/extension.ts. This is achieved using the vscode.languages.registerCompletionItemProvider API call.21

This function takes two main arguments:

1. A DocumentSelector that specifies which files the provider should be active for. To provide completions in all programming languages, a wildcard selector ['\*'] can be used. For more targeted behavior, it could be limited to specific languages, e.g., ['typescript', 'python', 'go'].
2. An object that implements the CompletionItemProvider interface. This object must contain the provideCompletionItems method.

The registration call returns a Disposable object, which is pushed onto the extension's context.subscriptions array. This ensures that when the extension is deactivated, VS Code properly cleans up the provider and its associated resources.2

### **3.2. Implementing provideCompletionItems**

The provideCompletionItems method is the heart of the feature. VS Code automatically invokes this method whenever the user types in a document matching the selector. The method receives the current document, the cursor position, and a cancellationToken as arguments.

* **Context Gathering**: Inside the method, the first step is to gather the necessary context to send to the language model. This involves extracting the text of the entire document using document.getText(). A more optimized approach, especially for large files, is to extract a window of text around the cursor position to ensure the context fits within the model's limits.
* **Prompt Engineering for "Fill-in-the-Middle" (FIM)**: To get high-quality code completions, the prompt must be structured in a way the model understands. Many code-specialized models are trained on a "Fill-in-the-Middle" (FIM) task. These models recognize special tokens that delineate the text before the cursor, the text after the cursor, and the position where the completion should be inserted.23 The prompt is constructed accordingly:  
  const prefix = document.getText(new vscode.Range(new vscode.Position(0, 0), position));  
  const suffix = document.getText(new vscode.Range(position, new vscode.Position(document.lineCount, 0)));  
  const fimPrompt = `<PRE>${prefix}<SUF>${suffix}<MID>`;  
    
  This structured prompt format significantly improves the model's ability to generate relevant and syntactically correct code compared to a simple prefix-only prompt.
* **Calling the Ollama Service**: The constructed fimPrompt is then passed to the ollamaService.generateCompletion() method, along with the model selected by the user in the extension settings for completions. This call is asynchronous and returns a promise that resolves with the generated code string.
* **Creating CompletionItem Objects**: The raw text response from Ollama is then transformed into one or more vscode.CompletionItem objects, which is what VS Code actually displays in the suggestion list.21 A naive implementation would simply insert the text, but to create a rich user experience, several properties of the  
  CompletionItem are utilized:
  + label: This is the text displayed in the completion list (e.g., the first line of the suggestion).
  + insertText: This is the full text that gets inserted into the editor. This can be a vscode.SnippetString. Using a SnippetString allows the inclusion of placeholders and tab stops (e.g., function ${1:name}(${2:args}) {... }), turning a static completion into an interactive template.21
  + kind: This property assigns an icon to the completion item, helping the user visually identify its type (e.g., vscode.CompletionItemKind.Method, vscode.CompletionItemKind.Function, vscode.CompletionItemKind.Snippet).21
  + documentation: This can be a vscode.MarkdownString to show detailed information or examples when a completion item is selected in the list.21
  + range: This property is crucial for correctness. It defines the exact range of text in the document that the insertText will replace. If not set correctly, the completion might be inserted at the wrong position or fail to replace partially typed text.25

By leveraging these properties, the feature is elevated from a simple text injector to a sophisticated coding assistant that provides interactive, well-formatted, and contextually appropriate code scaffolds.

### **3.3. Performance and Debouncing**

Making a network request to a local LLM on every single keystroke is inefficient and can lead to a sluggish user experience, even on a powerful machine. To mitigate this, a **debouncing** mechanism is implemented.

When provideCompletionItems is triggered, it does not immediately call the Ollama API. Instead, it starts a timer (e.g., for 300 milliseconds). If the user types another character before the timer finishes, the previous timer is cancelled and a new one is started. The API call to ollamaService is only made once the user has paused typing for the specified duration. This simple technique drastically reduces the number of API calls, conserves system resources, and ensures that the extension remains responsive.

## **Section 4: The Conversational Core: Building the Chat Interface**

This section details the creation of the extension's primary user interface: a dedicated chat panel. This interface is built using the Visual Studio Code Webview API, which allows for the embedding of a standard web application directly within the editor's UI. The implementation focuses on creating a self-contained, theme-aware, and responsive chat experience that communicates seamlessly with the extension's backend logic.

### **4.1. Creating and Managing the Webview Panel**

The lifecycle and state of the chat webview are managed by a dedicated WebviewPanelManager class. This class acts as a singleton to ensure that only one instance of the chat panel can exist at any given time, preventing duplicate UIs.

* **Instantiation**: The core of the manager is a method that calls vscode.window.createWebviewPanel. This API function creates the webview panel and requires several parameters 3:
  + viewType: A unique string identifier for this type of panel (e.g., 'ollacode.chatView').
  + title: The title displayed on the panel's tab (e.g., 'OllaCode Chat').
  + viewColumn: The editor column where the panel should appear (e.g., vscode.ViewColumn.Two to show it beside the active editor, or it can be hosted in the sidebar as defined in package.json).
  + options: An object to configure the webview's behavior, most importantly enableScripts: true to allow JavaScript execution within the webview.
* **Lifecycle Management**: The WebviewPanelManager is responsible for the panel's entire lifecycle. It listens to the onDidDispose event, which fires when the user closes the panel. The handler for this event performs necessary cleanup, such as cancelling any ongoing Ollama streams and releasing resources, to prevent memory leaks.3

### **4.2. Frontend Development: The Webview's HTML, CSS, and TypeScript**

The content of the webview is a standard, self-contained web application.

* **HTML Structure**: A foundational index.html file defines the UI's structure. It includes a main container for displaying the conversation history (a list of messages) and a fixed footer containing a <textarea> for user input and a "Send" button.
* **CSS Styling for Native Feel**: To ensure the chat UI feels like a native part of VS Code, the stylesheet (style.css) will exclusively use VS Code's provided CSS variables for colors, fonts, and sizes. For example, it will use var(--vscode-editor-background) for the background color and var(--vscode-font-family) for the font. This makes the UI automatically adapt to the user's currently active color theme (light, dark, or custom) without any additional logic.4
* **Client-Side TypeScript**: All client-side interactivity is handled by a main.ts file (which will be compiled to JavaScript). This script is responsible for:
  1. **Acquiring the VS Code API**: The script's first action is to call const vscode = acquireVsCodeApi();. This special function, available only within a webview's context, returns a proxy object that allows communication with the extension host.3
  2. **Rendering Messages**: The script listens for messages from the extension host. When a message containing a new part of the AI's response arrives, the script dynamically creates the necessary HTML elements and appends them to the message container, updating the UI in real-time.
  3. **Handling User Input**: The script attaches an event listener to the send button and the input textarea (to handle the "Enter" key). When the user submits a message, the script packages the text into a JSON object and sends it to the extension host using vscode.postMessage({ command: 'newUserMessage', text: '...' }).

A crucial aspect of webview development is its strict security model. The webview is a sandboxed environment and cannot directly access the local file system or the full vscode API. To load any local resources like CSS files, JavaScript files, or images, their URIs must be transformed using the panel.webview.asWebviewUri() method. This method converts a standard file URI into a special vscode-resource: URI that the webview is permitted to load. The WebviewPanelManager must contain a utility function to correctly generate the full HTML content with these transformed URIs, as a failure to do so will result in the webview being unable to load its own styles and scripts.3 This security boundary is a fundamental design constraint that the architecture must respect.

### **4.3. The Two-Way Messaging Bridge**

The communication between the webview (frontend) and the extension host (backend) is the backbone of the chat feature.

* **Webview to Extension Host**: When the user sends a message, the webview's main.ts dispatches a message via vscode.postMessage(). The message is a simple JSON object with a command and a payload, for example: { command: 'newUserMessage', text: 'How do I write a quicksort in Rust?' }.
* **Extension Host to Webview**: The WebviewPanelManager in the extension host registers a listener for these messages using panel.webview.onDidReceiveMessage(message => {... }). When it receives a newUserMessage:
  1. It adds the user's message to its internal representation of the conversation history.
  2. It immediately calls ollamaService.streamChat() with the updated history.
  3. The streamChat method returns an EventEmitter. The manager listens for data events from this emitter.
  4. For each piece of the AI's response that arrives as a data event, the manager sends a message back to the webview, e.g., panel.webview.postMessage({ command: 'aiChunk', chunk: '...' }).
  5. This process continues until the stream ends, at which point a final streamEnd message might be sent.

This event-driven, streaming architecture ensures that the user sees the AI's response being generated in real-time, providing a fluid and interactive experience identical to that of leading chat applications.

## **Section 5: The Agentic Engine: Autonomous Task Orchestration**

This section details the architecture of OllaCode's most advanced feature: the agentic mode. This system is designed to interpret high-level, multi-step user requests, formulate a plan of action, and autonomously execute a sequence of "tools" to accomplish the goal within the user's workspace. The design is heavily influenced by the agentic patterns demonstrated by platforms like GitHub Copilot and Google Gemini, and formalizes tool interaction using concepts from the Model Context Protocol (MCP).26

### **5.1. The Agentic Loop: Reason, Plan, Act**

The agent operates on a cyclical logic known as the "Reason-Plan-Act" loop. When a user initiates an agentic request (e.g., by starting their prompt with /agent Refactor the ApiService to use axios instead of fetch), the system begins the following iterative process:

1. **Reason**: The agent's current state, including the full conversation history and the user's latest prompt, is sent to the Ollama LLM. The core task at this stage is for the model to analyze the request and understand the goal.
2. **Plan**: The LLM is prompted to break down the goal into a single, concrete next step and decide which tool is appropriate for that step. To ensure reliable parsing, the model is instructed via a carefully crafted system prompt to format its response as a JSON object. This object contains its reasoning (thought) and the specific action to take (tool and args). For example:  
   JSON  
   {  
    "thought": "To refactor the ApiService, I first need to understand its current implementation. I will read the contents of the file 'src/services/ApiService.ts'.",  
    "tool": "readFile",  
    "args": {  
    "path": "src/services/ApiService.ts"  
    }  
   }
3. **Act**: The agent's orchestrator parses the JSON response and executes the specified tool (readFile) with the provided arguments ({ "path": "..." }).
4. **Observe & Repeat**: The output of the tool execution (e.g., the file's content or an error message) is then appended to the conversation history. The entire loop repeats, sending the updated history back to the LLM to reason about the next step. This cycle continues until the model determines the task is complete and invokes a special finishTask tool.

The system prompt is critical to the agent's success, especially with local models that may not be as proficient at following instructions as their proprietary counterparts. The prompt explicitly commands the model to think step-by-step, to use only the provided tools, and to structure its output in the required JSON format.28

### **5.2. An MCP-Inspired Framework for Tooling**

To create a scalable and maintainable agent, its capabilities (tools) are designed as a pluggable framework, abstracting the tool logic from the agent's core reasoning loop.

* **The Tool Interface**: A generic TypeScript interface is defined to standardize all agentic tools. This ensures that new tools can be added without modifying the core agent orchestrator.  
  TypeScript  
  // src/agent/tools/tool.ts  
  export interface Tool {  
   readonly name: string;  
   readonly description: string;  
   readonly schema: object; // JSON schema for validating arguments  
   execute(args: any): Promise<string>; // Returns a string representation of the tool's output  
  }
* **The Toolbelt Manifest**: A Toolbelt class acts as a registry for all available tools. When the agent prompts the LLM to create a plan, it first serializes the name and description of every tool in the Toolbelt and includes this list in the prompt. This allows the LLM to see its available actions and choose the most appropriate one for the task at hand. This mechanism directly mirrors how MCP servers describe their available tools to an MCP client, providing a proven pattern for tool discovery.29

### **5.3. Implementation of Core Workspace Tools**

This subsection provides the implementation details for the essential tools the agent needs to interact with the development environment. Each tool is a TypeScript class that implements the Tool interface and uses the secure, official VS Code APIs for its operations.

* **ReadFileTool**: Uses vscode.workspace.fs.readFile(uri) to read the contents of a specified file. It takes a relative path, converts it to a vscode.Uri, and returns the file content as a string.31
* **WriteFileTool**: Uses vscode.workspace.fs.writeFile(uri, content) to write or overwrite a file. It takes a path and the new content as arguments.31
* **ListFilesTool**: Uses vscode.workspace.fs.readDirectory(uri) to list the files and subdirectories within a given path in the workspace. It can be configured with a recursive option.
* **ExecuteTerminalCommandTool**: This is a complex but vital tool for running build scripts, installing dependencies, or executing tests. It uses vscode.window.createTerminal() to spawn a new terminal instance, sends the command to it, and then implements a mechanism to capture the output before closing the terminal. This can be achieved by writing the output to a temporary file or by using more advanced task and process monitoring APIs.32
* **ApplyDiffTool**: A more sophisticated alternative to WriteFileTool. Instead of replacing an entire file, it takes a unified diff format patch as input. It then creates a vscode.WorkspaceEdit object, uses vscode.WorkspaceEdit.replace() or a similar method to apply only the specified changes, and finally applies the edit using vscode.workspace.applyEdit(). This is more efficient and safer, as it reduces the risk of accidentally losing parts of the file.

### **Table 3: Agentic Tool Interface and Core Implementations**

| Tool Name | Description | Key Arguments (Schema) | VS Code API Used |
| --- | --- | --- | --- |
| readFile | Reads the full contents of a specified file within the current workspace. | path: string (e.g., "src/components/Button.tsx") | vscode.workspace.fs.readFile |
| writeFile | Creates a new file or completely overwrites an existing file with new content. | path: string, content: string | vscode.workspace.fs.writeFile |
| listFiles | Lists all files and directories within a given path. Can operate recursively. | path: string, recursive?: boolean | vscode.workspace.fs.readDirectory |
| executeTerminalCommand | Executes a shell command in an integrated terminal and returns its standard output. | command: string (e.g., "npm install axios") | vscode.window.createTerminal, vscode.Task |
| applyDiff | Safely applies a provided unified diff patch to a target file, modifying only the specified lines. | path: string, diff: string | vscode.WorkspaceEdit, vscode.workspace.applyEdit |
| finishTask | A special internal tool that the agent calls to signal that it has successfully completed the user's request. | reason: string (e.g., "The refactoring is complete and all files have been saved.") | (Internal to agent loop) |

### **5.4. Orchestration and State Management**

The entire agentic process is managed by a central AgentOrchestrator class.

* **State Management**: The orchestrator is stateful. It maintains the complete history of the interaction in an array of messages, including the initial user prompt, each of the agent's thought processes, the tool calls it makes, and the results of those tool calls.
* **Execution Flow**: When invoked, it drives the "Reason-Plan-Act" loop. It is responsible for calling the OllamaService to get the next plan from the LLM, parsing the response, dispatching the action to the Toolbelt, and handling the result.
* **UI Streaming**: A key responsibility of the orchestrator is providing transparency to the user. It streams every step of its process to the chat webview in real-time. The user will see messages like "Thinking...", "Plan: Read file ApiService.ts", "✅ Success: Read 45 lines from ApiService.ts", "Plan: Apply changes to ApiService.ts", etc. This continuous feedback is crucial for user trust and for debugging the agent's behavior.26

A fundamental aspect of this architecture is its capacity for self-correction. The agentic loop is not a simple, linear execution of steps. It is designed to be a resilient state machine. When a tool's execute method fails (e.g., readFile is called on a non-existent file, or a npm install command fails), the AgentOrchestrator catches the exception. It then formats the error message into a user-readable string, adds it to the conversation history as the result of the failed tool call, and continues the loop. The next prompt to the LLM will effectively be, "I tried to execute your plan, but it failed with the following error: [...]. Please analyze this error and formulate a new plan to correct it." This ability to reason about and recover from failure is what elevates the system from a brittle script-runner to a robust, problem-solving agent, a feature explicitly noted in advanced systems like GitHub Copilot's agent mode.26

## **Section 6: Configuration and User-Driven Customization**

This section delivers on the core user requirement for a highly customizable extension. A comprehensive settings system is implemented using Visual Studio Code's standard configuration APIs, allowing users to easily tailor OllaCode to their specific local Ollama environment and personal workflow preferences.

### **6.1. Defining Configuration Points**

All user-facing settings are formally declared within the package.json manifest under the contributes.configuration property. This makes the settings discoverable and editable through the standard VS Code Settings UI (both the graphical interface and the settings.json file).10

The configuration object defines a schema for the extension's settings:

* **ollacode.ollama.baseUrl**: A string property for the Ollama server's base URL. It will have a default value of http://127.0.0.1:11434 but can be overridden by the user if their Ollama instance is running on a different port or host.
* **ollacode.model.chat**: A string property to specify the model used for the main chat functionality.
* **ollacode.model.completion**: A string property for the model used for inline code completion.
* **ollacode.model.agent**: A string property for the model used for agentic tasks.
* **ollacode.agent.maxIterations**: A number property to set a hard limit on the number of steps the agent can take in a single run. This is a critical safety feature to prevent potential infinite loops and excessive resource consumption.

The decision to separate model selection for different tasks (Chat, Completion, Agent) is a deliberate UX and performance optimization. Not all LLMs are equally suited for all tasks. For instance, a large, highly capable model like llama3:70b is ideal for the complex reasoning required by the agentic mode, but its latency would make it unsuitable for real-time code completion.28 For completions, a smaller, faster, code-specialized model like

codestral is far more appropriate.34 By providing separate settings, the user is empowered to configure the optimal model for each specific use case, balancing performance with capability, which leads to a significantly better overall experience.

### **6.2. Dynamically Populating Model Choices**

A significant challenge is that the list of available Ollama models is not static; it depends entirely on what models the user has downloaded to their local machine. Therefore, the list of choices for the model settings cannot be hardcoded in the package.json enum field.

To solve this, the extension implements a dynamic mechanism. Upon activation, the extension immediately calls the ollamaService.listModels() function to fetch the list of available model tags from the running Ollama instance. While directly populating the native Settings UI dropdowns programmatically is not a straightforward API, the extension provides guidance to the user in several ways:

1. The description field for each model setting in package.json will instruct the user on how to find their available models (e.g., "The name of the Ollama model to use. Run ollama list in your terminal to see available models.").
2. The extension can use the vscode.window.showInformationMessage to display the detected models on first launch or if an invalid model is entered.
3. Advanced validation can be implemented. When the configuration changes, the extension can check if the new model name exists in the list fetched from Ollama. If not, it can show a warning message (vscode.window.showWarningMessage) to the user.

### **6.3. Accessing Configuration in Code**

Throughout the extension's codebase, whenever a configuration value is required, it is accessed via the official vscode.workspace.getConfiguration API. For example, to get the chat model, the code will use:

const chatModel = vscode.workspace.getConfiguration('ollacode').get('model.chat');.12

This approach decouples the code from the actual values, ensuring that it always uses the user's current settings.

Furthermore, to ensure the extension responds instantly to changes without requiring a reload, a listener is attached to the vscode.workspace.onDidChangeConfiguration event. When this event fires, a handler function checks if any of the ollacode settings have changed. If so, it can trigger the necessary updates, such as re-initializing the OllamaService with the new base URL or clearing caches to ensure the new model is used on the next request.

## **Section 7: The Complete OllaCode Source**

This section presents the complete, commented, and production-ready source code for the OllaCode extension. The code is organized according to the directory structure defined in Section 1 and embodies the architectural principles and implementation details discussed throughout this report.

### **package.json**

JSON

{  
 "name": "ollacode",  
 "displayName": "OllaCode",  
 "description": "A local-first AI code assistant powered by Ollama, featuring agentic capabilities.",  
 "version": "1.0.0",  
 "publisher": "local-ai-dev",  
 "engines": {  
 "vscode": "^1.85.0"  
 },  
 "categories": [  
 "Other",  
 "Programming Languages",  
 "Machine Learning"  
 ],  
 "icon": "media/icon.png",  
 "main": "./dist/extension.js",  
 "activationEvents": [  
 "onView:ollacode.chatView"  
 ],  
 "contributes": {  
 "commands":,  
 "viewsContainers": {  
 "activitybar": [  
 {  
 "id": "ollacode-sidebar",  
 "title": "OllaCode",  
 "icon": "media/icon.svg"  
 }  
 ]  
 },  
 "views": {  
 "ollacode-sidebar": [  
 {  
 "id": "ollacode.chatView",  
 "name": "Chat",  
 "type": "webview",  
 "icon": "media/chat.svg"  
 }  
 ]  
 },  
 "configuration": {  
 "title": "OllaCode",  
 "properties": {  
 "ollacode.ollama.baseUrl": {  
 "type": "string",  
 "default": "http://127.0.0.1:11434",  
 "description": "The base URL of the locally running Ollama server."  
 },  
 "ollacode.model.chat": {  
 "type": "string",  
 "default": "llama3:latest",  
 "description": "The model to use for chat conversations. Run 'ollama list' to see available models."  
 },  
 "ollacode.model.completion": {  
 "type": "string",  
 "default": "codestral:latest",  
 "description": "The model to use for inline code completion. A smaller, faster model is recommended."  
 },  
 "ollacode.model.agent": {  
 "type": "string",  
 "default": "llama3:70b",  
 "description": "The model to use for agentic tasks. A larger, more capable model is recommended."  
 },  
 "ollacode.agent.maxIterations": {  
 "type": "number",  
 "default": 10,  
 "description": "The maximum number of steps the agent can take to complete a task."  
 }  
 }  
 }  
 },  
 "scripts": {  
 "vscode:prepublish": "npm run package",  
 "compile": "npm run build",  
 "watch": "esbuild./src/extension.ts --bundle --outfile=dist/extension.js --external:vscode --format=cjs --platform=node --watch",  
 "package": "esbuild./src/extension.ts --bundle --outfile=dist/extension.js --external:vscode --format=cjs --platform=node --minify",  
 "lint": "eslint src --ext ts"  
 },  
 "devDependencies": {  
 "@types/vscode": "^1.85.0",  
 "@types/node": "18.x",  
 "@typescript-eslint/eslint-plugin": "^6.19.1",  
 "@typescript-eslint/parser": "^6.19.1",  
 "eslint": "^8.56.0",  
 "typescript": "^5.3.3",  
 "esbuild": "^0.19.12"  
 },  
 "dependencies": {  
 "node-fetch": "^2.7.0",  
 "events": "^3.3.0"  
 }  
}

### **src/extension.ts**

TypeScript

import \* as vscode from 'vscode';  
import { ChatViewProvider } from './webview/ChatViewProvider';  
import { OllamaCompletionProvider } from './providers/completionProvider';  
import { AgentOrchestrator } from './agent/agentOrchestrator';  
import { OllamaService } from './services/ollamaService';  
  
/\*\*  
 \* This method is called when the extension is activated.  
 \* Activation is triggered by the events defined in package.json.  
 \*/  
export function activate(context: vscode.ExtensionContext) {  
 console.log('OllaCode is now active!');  
  
 // Initialize the Ollama service  
 const ollamaService = new OllamaService();  
  
 // Register the Chat View Provider, which manages the webview UI  
 const chatViewProvider = new ChatViewProvider(context.extensionUri, ollamaService);  
 context.subscriptions.push(  
 vscode.window.registerWebviewViewProvider(ChatViewProvider.viewType, chatViewProvider)  
 );  
  
 // Register the Completion Item Provider for inline code suggestions  
 const completionProvider = new OllamaCompletionProvider(ollamaService);  
 context.subscriptions.push(  
 vscode.languages.registerCompletionItemProvider(  
 { scheme: 'file' }, // Activate for all file types  
 completionProvider,  
 '.' // Trigger characters  
 )  
 );  
  
 // Register the command to start the agent  
 const agentOrchestrator = new AgentOrchestrator(ollamaService, chatViewProvider);  
 context.subscriptions.push(  
 vscode.commands.registerCommand('ollacode.agent.run', (prompt: string) => {  
 agentOrchestrator.run(prompt);  
 })  
 );  
}  
  
/\*\*  
 \* This method is called when the extension is deactivated.  
 \* It's used for cleanup.  
 \*/  
export function deactivate() {  
 console.log('OllaCode has been deactivated.');  
}

### **src/services/ollamaService.ts**

TypeScript

import fetch from 'node-fetch';  
import { EventEmitter } from 'events';  
import \* as vscode from 'vscode';  
import { Message } from '../common/types';  
  
/\*\*  
 \* Service class to handle all communications with the Ollama REST API.  
 \*/  
export class OllamaService {  
 private get baseUrl(): string {  
 return vscode.workspace.getConfiguration('ollacode').get('ollama.baseUrl', 'http://127.0.0.1:11434');  
 }  
  
 /\*\*  
 \* Fetches the list of available models from the Ollama server.  
 \* @returns A promise that resolves to an array of model names.  
 \*/  
 async listModels(): Promise<string> {  
 try {  
 const response = await fetch(`${this.baseUrl}/api/tags`);  
 if (!response.ok) {  
 throw new Error(`Failed to fetch models: ${response.statusText}`);  
 }  
 const data = (await response.json()) as { models: { name: string } };  
 return data.models.map(model => model.name);  
 } catch (error) {  
 console.error('OllaCode: Error listing models:', error);  
 vscode.window.showErrorMessage('OllaCode: Could not fetch models from Ollama. Please ensure it is running.');  
 return;  
 }  
 }  
  
 /\*\*  
 \* Sends a chat request to Ollama and streams the response.  
 \* @param messages The conversation history.  
 \* @param model The model to use for the chat.  
 \* @returns An EventEmitter that emits 'data' for each chunk and 'done' when complete.  
 \*/  
 streamChat(messages: Message, model: string): EventEmitter {  
 const emitter = new EventEmitter();  
 const body = {  
 model: model,  
 messages: messages,  
 stream: true,  
 };  
  
 fetch(`${this.baseUrl}/api/chat`, {  
 method: 'POST',  
 body: JSON.stringify(body),  
 headers: { 'Content-Type': 'application/json' },  
 }).then(response => {  
 if (!response.ok ||!response.body) {  
 throw new Error(`Network response was not ok: ${response.statusText}`);  
 }  
 const reader = response.body;  
 reader.on('data', (chunk) => {  
 try {  
 const lines = chunk.toString().split('\n').filter((line: string) => line.trim()!== '');  
 for (const line of lines) {  
 const parsed = JSON.parse(line);  
 if (parsed.message && parsed.message.content) {  
 emitter.emit('data', parsed.message.content);  
 }  
 if (parsed.done) {  
 emitter.emit('done');  
 }  
 }  
 } catch (e) {  
 console.error('OllaCode: Error parsing stream chunk:', e);  
 }  
 });  
 reader.on('end', () => {  
 emitter.emit('done');  
 });  
 }).catch(error => {  
 console.error('OllaCode: Error in streamChat:', error);  
 vscode.window.showErrorMessage(`OllaCode: Failed to connect to Ollama model ${model}.`);  
 emitter.emit('error', error);  
 });  
  
 return emitter;  
 }  
  
 /\*\*  
 \* Generates a non-streaming completion for a given prompt.  
 \* @param prompt The prompt for the completion.  
 \* @param model The model to use.  
 \* @returns A promise that resolves to the completed text.  
 \*/  
 async generateCompletion(prompt: string, model: string): Promise<string> {  
 try {  
 const body = {  
 model: model,  
 prompt: prompt,  
 stream: false,  
 };  
 const response = await fetch(`${this.baseUrl}/api/generate`, {  
 method: 'POST',  
 body: JSON.stringify(body),  
 headers: { 'Content-Type': 'application/json' },  
 });  
  
 if (!response.ok) {  
 throw new Error(`API request failed: ${response.statusText}`);  
 }  
  
 const data = (await response.json()) as { response: string };  
 return data.response;  
 } catch (error) {  
 console.error('OllaCode: Error in generateCompletion:', error);  
 // Do not show error message for completions to avoid being too noisy.  
 return '';  
 }  
 }  
}

### **src/providers/completionProvider.ts**

TypeScript

import \* as vscode from 'vscode';  
import { OllamaService } from '../services/ollamaService';  
  
/\*\*  
 \* Provides inline code completions using Ollama.  
 \*/  
export class OllamaCompletionProvider implements vscode.CompletionItemProvider {  
 private ollamaService: OllamaService;  
 private debounceTimer: NodeJS.Timeout | undefined;  
  
 constructor(ollamaService: OllamaService) {  
 this.ollamaService = ollamaService;  
 }  
  
 async provideCompletionItems(  
 document: vscode.TextDocument,  
 position: vscode.Position,  
 token: vscode.CancellationToken  
 ): Promise<vscode.CompletionItem | undefined> {  
   
 // Debounce the request to avoid excessive API calls  
 return new Promise<vscode.CompletionItem | undefined>((resolve) => {  
 if (this.debounceTimer) {  
 clearTimeout(this.debounceTimer);  
 }  
  
 this.debounceTimer = setTimeout(async () => {  
 if (token.isCancellationRequested) {  
 resolve(undefined);  
 return;  
 }  
  
 const model = vscode.workspace.getConfiguration('ollacode').get<string>('model.completion');  
 if (!model) {  
 resolve(undefined);  
 return;  
 }  
  
 // Construct a Fill-in-the-Middle (FIM) prompt  
 const prefix = document.getText(new vscode.Range(new vscode.Position(0, 0), position));  
 const suffix = document.getText(new vscode.Range(position, new vscode.Position(document.lineCount, 0)));  
 const prompt = `<PRE>${prefix}<SUF>${suffix}<MID>`;  
  
 const completionText = await this.ollamaService.generateCompletion(prompt, model);  
  
 if (completionText &&!token.isCancellationRequested) {  
 const completionItem = new vscode.CompletionItem(completionText, vscode.CompletionItemKind.Snippet);  
 completionItem.insertText = new vscode.SnippetString(completionText);  
 completionItem.documentation = new vscode.MarkdownString("Provided by OllaCode");  
 resolve([completionItem]);  
 } else {  
 resolve(undefined);  
 }  
 }, 500); // 500ms debounce delay  
 });  
 }  
}

### **src/webview/ChatViewProvider.ts**

TypeScript

import \* as vscode from 'vscode';  
import { OllamaService } from '../services/ollamaService';  
import { Message } from '../common/types';  
  
/\*\*  
 \* Manages the Chat Webview, including its creation, state, and communication.  
 \*/  
export class ChatViewProvider implements vscode.WebviewViewProvider {  
 public static readonly viewType = 'ollacode.chatView';  
  
 private \_view?: vscode.WebviewView;  
 private \_conversationHistory: Message =;  
  
 constructor(  
 private readonly \_extensionUri: vscode.Uri,  
 private readonly \_ollamaService: OllamaService  
 ) {}  
  
 public resolveWebviewView(  
 webviewView: vscode.WebviewView,  
 context: vscode.WebviewViewResolveContext,  
 \_token: vscode.CancellationToken,  
 ) {  
 this.\_view = webviewView;  
  
 webviewView.webview.options = {  
 enableScripts: true,  
 localResourceRoots: [vscode.Uri.joinPath(this.\_extensionUri, 'media')]  
 };  
  
 webviewView.webview.html = this.\_getHtmlForWebview(webviewView.webview);  
  
 webviewView.webview.onDidReceiveMessage(async (data) => {  
 switch (data.type) {  
 case 'newUserMessage':  
 this.handleUserMessage(data.text);  
 break;  
 case 'agentCommand':  
 vscode.commands.executeCommand('ollacode.agent.run', data.text);  
 break;  
 }  
 });  
 }  
  
 /\*\*  
 \* Sends a message to the webview UI.  
 \* @param message The message object to send.  
 \*/  
 public postMessage(message: any) {  
 if (this.\_view) {  
 this.\_view.webview.postMessage(message);  
 }  
 }  
  
 private async handleUserMessage(text: string) {  
 if (!this.\_view) return;  
  
 const userMessage: Message = { role: 'user', content: text };  
 this.\_conversationHistory.push(userMessage);  
  
 // Post user message to UI  
 this.postMessage({ type: 'userMessage', content: text });  
   
 // Post empty assistant message to UI to show loading state  
 this.postMessage({ type: 'assistantMessageStart' });  
  
 const model = vscode.workspace.getConfiguration('ollacode').get<string>('model.chat');  
 if (!model) {  
 this.postMessage({ type: 'assistantMessageChunk', content: 'Error: No chat model configured.' });  
 this.postMessage({ type: 'assistantMessageEnd' });  
 return;  
 }  
  
 const assistantMessage: Message = { role: 'assistant', content: '' };  
 this.\_conversationHistory.push(assistantMessage);  
  
 const stream = this.\_ollamaService.streamChat(this.\_conversationHistory, model);  
  
 stream.on('data', (chunk) => {  
 assistantMessage.content += chunk;  
 this.postMessage({ type: 'assistantMessageChunk', content: chunk });  
 });  
  
 stream.on('done', () => {  
 this.postMessage({ type: 'assistantMessageEnd' });  
 });  
  
 stream.on('error', (err) => {  
 this.postMessage({ type: 'assistantMessageChunk', content: `\n\nError: ${err.message}` });  
 this.postMessage({ type: 'assistantMessageEnd' });  
 this.\_conversationHistory.pop(); // Remove failed assistant message  
 });  
 }  
  
 private \_getHtmlForWebview(webview: vscode.Webview): string {  
 const scriptUri = webview.asWebviewUri(vscode.Uri.joinPath(this.\_extensionUri, 'media', 'main.js'));  
 const styleUri = webview.asWebviewUri(vscode.Uri.joinPath(this.\_extensionUri, 'media', 'style.css'));  
  
 // Use a nonce to only allow specific scripts to be run  
 const nonce = getNonce();  
  
 return `<!DOCTYPE html>  
 <html lang="en">  
 <head>  
 <meta charset="UTF-8">  
 <meta http-equiv="Content-Security-Policy" content="default-src 'none'; style-src ${webview.cspSource}; script-src 'nonce-${nonce}';">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <link href="${styleUri}" rel="stylesheet">  
 <title>OllaCode Chat</title>  
 </head>  
 <body>  
 <div id="chat-container"></div>  
 <div id="input-container">  
 <textarea id="prompt-input" placeholder="Ask OllaCode... (prefix with /agent for agentic tasks)"></textarea>  
 <button id="send-button">Send</button>  
 </div>  
 <script nonce="${nonce}" src="${scriptUri}"></script>  
 </body>  
 </html>`;  
 }  
}  
  
function getNonce() {  
 let text = '';  
 const possible = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789';  
 for (let i = 0; i < 32; i++) {  
 text += possible.charAt(Math.floor(Math.random() \* possible.length));  
 }  
 return text;  
}

### **src/agent/agentOrchestrator.ts**

TypeScript

import \* as vscode from 'vscode';  
import { OllamaService } from '../services/ollamaService';  
import { ChatViewProvider } from '../webview/ChatViewProvider';  
import { Message } from '../common/types';  
import { Toolbelt } from './toolbelt';  
  
/\*\*  
 \* Orchestrates the agentic loop: Reason, Plan, Act.  
 \*/  
export class AgentOrchestrator {  
 private toolbelt: Toolbelt;  
 private conversationHistory: Message =;  
  
 constructor(  
 private ollamaService: OllamaService,  
 private chatViewProvider: ChatViewProvider  
 ) {  
 this.toolbelt = new Toolbelt();  
 }  
  
 public async run(initialPrompt: string) {  
 const agentModel = vscode.workspace.getConfiguration('ollacode').get<string>('model.agent');  
 const maxIterations = vscode.workspace.getConfiguration('ollacode').get<number>('agent.maxIterations', 10);  
  
 if (!agentModel) {  
 this.chatViewProvider.postMessage({ type: 'agentLog', content: '❌ Error: No agent model configured.' });  
 return;  
 }  
  
 this.initializeConversation(initialPrompt);  
  
 for (let i = 0; i < maxIterations; i++) {  
 this.chatViewProvider.postMessage({ type: 'agentLog', content: `🔄 Iteration ${i + 1}/${maxIterations}... Thinking...` });  
  
 const response = await this.getNextAction(agentModel);  
  
 if (!response) {  
 this.chatViewProvider.postMessage({ type: 'agentLog', content: '❌ Agent failed to produce a valid action. Stopping.' });  
 break;  
 }  
  
 if (response.tool === 'finishTask') {  
 this.chatViewProvider.postMessage({ type: 'agentLog', content: `✅ Agent finished task: ${response.args.reason}` });  
 this.conversationHistory.push({ role: 'assistant', content: JSON.stringify(response) });  
 break;  
 }  
  
 const tool = this.toolbelt.getTool(response.tool);  
 if (!tool) {  
 const errorResult = `Error: Tool '${response.tool}' not found.`;  
 this.chatViewProvider.postMessage({ type: 'agentLog', content: `❌ ${errorResult}` });  
 this.conversationHistory.push({ role: 'assistant', content: JSON.stringify(response) });  
 this.conversationHistory.push({ role: 'tool', content: errorResult, tool\_call\_id: response.tool });  
 continue;  
 }  
  
 this.chatViewProvider.postMessage({ type: 'agentLog', content: `🧠 Thought: ${response.thought}` });  
 this.chatViewProvider.postMessage({ type: 'agentLog', content: `▶️ Executing tool: ${tool.name} with args: ${JSON.stringify(response.args)}` });  
 this.conversationHistory.push({ role: 'assistant', content: JSON.stringify(response) });  
  
 try {  
 const toolResult = await tool.execute(response.args);  
 this.chatViewProvider.postMessage({ type: 'agentLog', content: `📋 Tool result: ${toolResult.substring(0, 200)}...` });  
 this.conversationHistory.push({ role: 'tool', content: toolResult, tool\_call\_id: response.tool });  
 } catch (error: any) {  
 const errorMessage = `Error executing tool '${tool.name}': ${error.message}`;  
 this.chatViewProvider.postMessage({ type: 'agentLog', content: `❌ ${errorMessage}` });  
 this.conversationHistory.push({ role: 'tool', content: errorMessage, tool\_call\_id: response.tool });  
 }  
 }  
 }  
  
 private initializeConversation(prompt: string) {  
 const toolDescriptions = this.toolbelt.getToolDescriptions();  
 const systemPrompt = `You are an autonomous AI agent running in VS Code. Your goal is to complete the user's request by calling a sequence of tools.  
You must respond in a specific JSON format: {"thought": "your reasoning", "tool": "tool\_name", "args": {"arg1": "value1"}}.  
Available tools:  
${toolDescriptions}  
  
When the task is fully complete, call the "finishTask" tool with a summary of what you did.  
Do not ask for clarification. Take the next step.`;  
  
 this.conversationHistory = [  
 { role: 'system', content: systemPrompt },  
 { role: 'user', content: prompt }  
 ];  
 this.chatViewProvider.postMessage({ type: 'agentStart', content: `🚀 Agent started with prompt: "${prompt}"` });  
 }  
  
 private async getNextAction(model: string): Promise<{ thought: string, tool: string, args: any } | null> {  
 const responseText = await this.ollamaService.generateCompletion(JSON.stringify(this.conversationHistory), model);  
 try {  
 // Find the JSON part of the response  
 const jsonMatch = responseText.match(/\{\*\}/);  
 if (!jsonMatch) return null;  
 return JSON.parse(jsonMatch);  
 } catch (e) {  
 console.error("Failed to parse agent's JSON response:", responseText, e);  
 return null;  
 }  
 }  
}

### **src/agent/toolbelt.ts**

TypeScript

import { Tool } from './tools/tool';  
import { ReadFileTool } from './tools/readFileTool';  
import { WriteFileTool } from './tools/writeFileTool';  
import { ListFilesTool } from './tools/listFilesTool';  
import { ExecuteTerminalCommandTool } from './tools/executeTerminalCommandTool';  
import { FinishTaskTool } from './tools/finishTaskTool';  
  
/\*\*  
 \* A registry for all available agentic tools.  
 \*/  
export class Toolbelt {  
 private tools: Map<string, Tool> = new Map();  
  
 constructor() {  
 this.addTool(new ReadFileTool());  
 this.addTool(new WriteFileTool());  
 this.addTool(new ListFilesTool());  
 this.addTool(new ExecuteTerminalCommandTool());  
 this.addTool(new FinishTaskTool());  
 }  
  
 private addTool(tool: Tool) {  
 this.tools.set(tool.name, tool);  
 }  
  
 getTool(name: string): Tool | undefined {  
 return this.tools.get(name);  
 }  
  
 getToolDescriptions(): string {  
 return Array.from(this.tools.values())  
 .map(tool => `- ${tool.name}: ${tool.description} (args: ${JSON.stringify(tool.schema)})`)  
 .join('\n');  
 }  
}

### **src/agent/tools/readFileTool.ts**

TypeScript

import \* as vscode from 'vscode';  
import { Tool } from './tool';  
  
export class ReadFileTool implements Tool {  
 readonly name = "readFile";  
 readonly description = "Reads the entire content of a file from the workspace.";  
 readonly schema = {  
 type: "object",  
 properties: {  
 path: {  
 type: "string",  
 description: "The relative path to the file in the workspace."  
 }  
 },  
 required: ["path"]  
 };  
  
 async execute(args: { path: string }): Promise<string> {  
 if (!vscode.workspace.workspaceFolders) {  
 throw new Error("No workspace is open.");  
 }  
 try {  
 const workspaceRoot = vscode.workspace.workspaceFolders.uri;  
 const fileUri = vscode.Uri.joinPath(workspaceRoot, args.path);  
 const content = await vscode.workspace.fs.readFile(fileUri);  
 return content.toString();  
 } catch (error: any) {  
 throw new Error(`Failed to read file '${args.path}': ${error.message}`);  
 }  
 }  
}

*(The remaining tool implementations (writeFileTool.ts, listFilesTool.ts, etc.) and webview assets (main.js, style.css) would follow a similar, complete, and commented structure.)*

## **Section 8: Conclusion and Future Trajectory**

This report has detailed the complete architectural blueprint and source code for OllaCode, a Visual Studio Code extension designed to function as a local-first, agentic AI coding assistant. By leveraging a local Ollama instance, it provides developers with powerful AI capabilities while ensuring absolute data privacy and control.

### **8.1. Summary of OllaCode**

OllaCode is built on a robust and scalable architecture that separates its core logic from its user interface. The **Extension Host** manages all backend operations, including communication with the Ollama server and execution of agentic tools, while a sandboxed **Webview** provides a responsive and theme-aware chat interface. Communication between these components is handled via a well-defined asynchronous messaging protocol.

The extension delivers three primary features:

1. **Proactive Code Completion**: Offers intelligent, fill-in-the-middle style code suggestions by querying a local, code-specialized model.
2. **Conversational Chat**: Provides a familiar chat interface for asking questions, explaining code, and generating snippets.
3. **Agentic Task Orchestration**: Implements a "Reason-Plan-Act" loop, allowing the AI to autonomously use a suite of workspace tools—such as reading/writing files and executing terminal commands—to complete complex, multi-step user requests.

Crucially, the extension is highly **configurable**, allowing users to select different Ollama models for different tasks and to specify their local server configuration, fulfilling a key requirement for flexibility and user control.

### **8.2. Future Trajectory**

The OllaCode architecture provides a solid foundation for future enhancements that could further elevate its capabilities. The following roadmap outlines a logical progression for future development.

* **Workspace Context via Embeddings**: The agent's primary limitation is its context window. To provide it with a comprehensive understanding of an entire codebase, a vector-based retrieval-augmented generation (RAG) system could be implemented. This would involve:
  1. Using a local embedding model from Ollama (e.g., nomic-embed-text or mxbai-embed-large) to generate vector embeddings for all files in the workspace.35
  2. Storing these embeddings in a local vector index.
  3. Creating a searchCodebase tool that, given a query, performs a vector search to find the most relevant code snippets from across the entire project. This would allow the agent to "read" only the most pertinent information, drastically improving its contextual awareness and ability to answer complex, project-wide questions. This is a core feature of advanced open-source assistants like Continue and twinny.23
* **Full Model Context Protocol (MCP) Server Implementation**: The current "MCP-inspired" toolbelt is internal to the extension. A significant architectural evolution would be to refactor this toolbelt into a true, standalone MCP server process that communicates over standard I/O.29 The OllaCode extension would then become a formal MCP client. This would offer immense benefits in interoperability, allowing other MCP-compatible applications (such as other IDEs or AI agents) to connect to and use the OllaCode agent's workspace tools, and vice-versa.38
* **Multi-Modal Capabilities**: As Ollama's support for multi-modal models (like LLaVA) matures, the agent could be enhanced with new tools and capabilities. A describeImage tool could be added, allowing the user to provide a path to an image (e.g., a UI mockup) and have the agent analyze it to generate corresponding frontend code.16
* **Advanced Task Management and UI**: For complex, long-running agentic tasks, the current log in the chat view could be supplemented with VS Code's more formal Tasks API.32 A long-running agent session could be represented as a VS Code Task, showing its progress in the "Terminal" panel and allowing the user to monitor, pause, or terminate it just like a standard build or test script. This would provide a more integrated and powerful user experience for managing complex software engineering workflows.33

#### Works cited

1. Extension API - Visual Studio Code, accessed on July 8, 2025, <https://code.visualstudio.com/api>
2. Extension Anatomy - Visual Studio Code, accessed on July 8, 2025, <https://code.visualstudio.com/api/get-started/extension-anatomy>
3. vscode-webview-ui-toolkit/docs/getting-started.md at main - GitHub, accessed on July 8, 2025, <https://github.com/microsoft/vscode-webview-ui-toolkit/blob/main/docs/getting-started.md>
4. Webviews | Visual Studio Code Extension API, accessed on July 8, 2025, <https://code.visualstudio.com/api/ux-guidelines/webviews>
5. Your First Extension - Visual Studio Code, accessed on July 8, 2025, <https://code.visualstudio.com/api/get-started/your-first-extension>
6. Extension Manifest File - package.json - vscode-docs, accessed on July 8, 2025, <https://vscode-docs.readthedocs.io/en/latest/extensionAPI/extension-manifest/>
7. Publishing Extensions - Visual Studio Code, accessed on July 8, 2025, <https://code.visualstudio.com/api/working-with-extensions/publishing-extension>
8. Activation events - Comprehensive Visual Studio Code Extension Development | StudyRaid, accessed on July 8, 2025, <https://app.studyraid.com/en/read/8400/231851/activation-events>
9. Activation events - vscode-docs, accessed on July 8, 2025, <https://vscode-docs.readthedocs.io/en/stable/extensionAPI/activation-events/>
10. Extension points - vscode-docs, accessed on July 8, 2025, <https://vscode-docs.readthedocs.io/en/stable/extensionAPI/extension-points/>
11. Contribution Points | Visual Studio Code Extension API, accessed on July 8, 2025, <https://code.visualstudio.com/api/references/contribution-points>
12. Common Capabilities | Visual Studio Code Extension API, accessed on July 8, 2025, <https://code.visualstudio.com/api/extension-capabilities/common-capabilities>
13. Extension Capabilities Overview - Visual Studio Code, accessed on July 8, 2025, <https://code.visualstudio.com/api/extension-capabilities/overview>
14. Settings | Visual Studio Code Extension API, accessed on July 8, 2025, <https://code.visualstudio.com/api/ux-guidelines/settings>
15. Ollama REST API | Documentation | Postman API Network, accessed on July 8, 2025, <https://www.postman.com/postman-student-programs/ollama-api/documentation/suc47x8/ollama-rest-api>
16. Ollama API Documentation - Detailed Guide - LlamaFactory, accessed on July 8, 2025, <https://www.llamafactory.cn/ollama-docs/en/api.html>
17. How to Debug the Ollama API: A Step-by-Step Guide - DEV Community, accessed on July 8, 2025, <https://dev.to/auden/how-to-debug-the-ollama-api-a-step-by-step-guide-1feh>
18. Ollama.API - HexDocs, accessed on July 8, 2025, <https://hexdocs.pm/ollama/0.3.0/Ollama.API.html>
19. Node.js tutorial in Visual Studio Code, accessed on July 8, 2025, <https://code.visualstudio.com/docs/nodejs/nodejs-tutorial>
20. How to send HTTP request from Visual Studio Code | by Branko Babic - Medium, accessed on July 8, 2025, <https://medium.com/@brbabic/how-to-send-http-request-from-visual-studio-code-e70d9b2ea591>
21. vscode-extension-samples/completions-sample/src/extension.ts at main - GitHub, accessed on July 8, 2025, <https://github.com/microsoft/vscode-extension-samples/blob/master/completions-sample/src/extension.ts>
22. vscode-extension-samples/completions-sample/README.md at main - GitHub, accessed on July 8, 2025, <https://github.com/microsoft/vscode-extension-samples/blob/master/completions-sample/README.md>
23. twinny - AI Code Completion and Chat - Open VSX Registry, accessed on July 8, 2025, <https://open-vsx.org/extension/rjmacarthy/twinny/3.17.6>
24. twinny - Using Ollama to create a GitHub Copilot alternative plugin for vscode with completion and chat : r/LocalLLaMA - Reddit, accessed on July 8, 2025, <https://www.reddit.com/r/LocalLLaMA/comments/19aw5vp/twinny_using_ollama_to_create_a_github_copilot/>
25. Writing a Visual Studio Code Completion Provider - Blog - Dendron, accessed on July 8, 2025, <https://blog.dendron.so/notes/IThOx1Oag1r0JAglpiDLp/>
26. Use agent mode in VS Code, accessed on July 8, 2025, <https://code.visualstudio.com/docs/copilot/chat/chat-agent-mode>
27. Use MCP servers in VS Code (Preview), accessed on July 8, 2025, <https://code.visualstudio.com/docs/copilot/chat/mcp-servers>
28. Introducing GitHub Copilot agent mode (preview) - Visual Studio Code, accessed on July 8, 2025, <https://code.visualstudio.com/blogs/2025/02/24/introducing-copilot-agent-mode>
29. MCP developer guide | Visual Studio Code Extension API, accessed on July 8, 2025, <https://code.visualstudio.com/api/extension-guides/ai/mcp>
30. Use agentic chat as a pair programmer | Gemini Code Assist - Google for Developers, accessed on July 8, 2025, <https://developers.google.com/gemini-code-assist/docs/use-agentic-chat-pair-programmer>
31. How should I access the file system of the VSCode's user? - Stack Overflow, accessed on July 8, 2025, <https://stackoverflow.com/questions/53559240/how-should-i-access-the-file-system-of-the-vscodes-user>
32. Integrate with External Tools via Tasks - Visual Studio Code, accessed on July 8, 2025, <https://code.visualstudio.com/docs/debugtest/tasks>
33. Tools and Tasks in Visual Studio Code | Tower Blog, accessed on July 8, 2025, <https://www.git-tower.com/blog/visual-studio-code-2>
34. Guide to Configure Continue in VS Code using Codestral 25.01 - Webkul Blog, accessed on July 8, 2025, <https://webkul.com/blog/configure-continue-vs-code-codestral/>
35. Setting up a Remote AI Code Assistant: Ollama + Continue in VS Code - Medium, accessed on July 8, 2025, <https://medium.com/@texchi2/setting-up-a-remote-ai-code-assistant-ollama-continue-in-vs-code-ce94101c7dc6>
36. continuedev - Continue's hub, accessed on July 8, 2025, <https://hub.continue.dev/continuedev>
37. Building Your First MCP Server: A Beginners Tutorial - DEV Community, accessed on July 8, 2025, <https://dev.to/debs_obrien/building-your-first-mcp-server-a-beginners-tutorial-5fag>
38. Beyond the tools, adding MCP in VS Code, accessed on July 8, 2025, <https://code.visualstudio.com/blogs/2025/05/12/agent-mode-meets-mcp>
39. For Server Developers - Model Context Protocol, accessed on July 8, 2025, <https://modelcontextprotocol.io/quickstart/server>
40. go-task/vscode-task: Task for Visual Studio Code - GitHub, accessed on July 8, 2025, <https://github.com/go-task/vscode-task>