Roo Code LLM Docs

Table of contents

1 Welcome to the Roo Code Docs

This is the homepage for the Roo Code documentation.

This explains the codebase

2 API Architecture

The `api` directory defines the API for interacting with different language models (LLMs) and providers.

2.1 Key Components

- `index.ts`: This file likely defines the main API entry point and provides functions for accessing the different LLM providers.
- `providers/`: This directory contains the implementations for the different LLM providers, such as OpenAI, Anthropic, and others.
- `providers/base-provider.ts`: This file likely defines the base class or interface for all LLM providers.
- `providers/openai.ts`: This file likely implements the OpenAI provider, which uses the OpenAI API to interact with the OpenAI models.
- `transform/`: This directory likely contains the code for transforming the input and output of the LLM providers.

2.2 Relationships

The API components likely interact with each other to provide a consistent interface for accessing the different LLMs. The `index.ts` file might use the `providers/` to access the different LLMs, and the `transform/` to transform the input and output of the LLMs.

2.3 External APIs

This area interacts with external APIs such as OpenAI, Anthropic, etc. Each provider implements the specific API calls and data transformations required for each LLM service.

3 Core Architecture

The `core` directory contains the core logic of the Roo Code extension.

3.1 Key Components

- `Cline.ts`: This file likely contains the main class for the extension, responsible for managing the overall state and behavior of the extension.
- `CodeActionProvider.ts`: This file likely implements the code action provider, which provides suggestions and actions to the user based on the current code context.
- `contextProxy.ts`: This file likely manages the context proxy, which provides access to VS Code's API and other extension-related information.
- `EditorUtils.ts`: This file likely provides utility functions for interacting with the VS Code editor.
- `mode-validator.ts`: This file likely validates the current mode of the extension.
- `webview/`: This directory likely contains the code for managing the webview, which is used to display the user interface.

3.2 Relationships

The core components likely interact with each other to provide the main functionality of the extension. For example, the `CodeActionProvider` might use the `EditorUtils` to get information about the current code context, and the `Cline` class might use the `webview/` to display information to the user.

4 Cost Calculation

The src/utils/cost.ts file defines functions for calculating the cost of using the LLMs.

4.1 Functions

The following functions are defined:

- `calculateApiCostInternal(modelInfo, inputTokens, outputTokens, cacheCreationInputTokens, cacheReadInputTokens)`: Calculates the cost based on the model info, input tokens, output tokens, cache creation input tokens, and cache read input tokens.
- `calculateApiCostAnthropic(modelInfo, inputTokens, outputTokens, cacheCreationInputTokens cacheReadInputTokens)`: Calculates the cost for Anthropic compliant usage.
- `calculateApiCostOpenAI(modelInfo, inputTokens, outputTokens, cacheCreationInputTokens, cacheReadInputTokens)`: Calculates the cost for OpenAI compliant usage.
- `parseApiPrice(price)`: Parses the API price.

4.2 Cost Calculation

The cost is calculated based on the following factors:

- Model info
- Input tokens
- Output tokens
- Cache creation input tokens
- Cache read input tokens

5 How to Make Roo Code Run as a Standalone Cross-Platform App

This document provides instructions on how to make the Roo Code codebase run as a standalone cross-platform app.

5.1 Technologies

- **Electron:** A framework for building cross-platform desktop applications with JavaScript, HTML, and CSS.
- **Node.js:** A JavaScript runtime environment that executes JavaScript code outside of a web browser.
- **React:** A JavaScript library for building user interfaces.
- **TypeScript:** A superset of JavaScript that adds static typing.

5.2 Steps

1. **Create a new Electron project.** Use the Electron Forge or Electron Builder to create a new Electron project.
2. **Copy the Roo Code codebase to the Electron project.** Copy the `src` directory and the `webview-ui` directory to the Electron project.
3. **Create a new `main.js` file.** This file will be the entry point for the Electron application.
4. **Implement the Electron main process.** The Electron main process is responsible for creating the browser window and managing the application lifecycle.
5. **Implement the Electron renderer process.** The Electron renderer process is responsible for rendering the user interface. You can reuse the existing React components in the `webview-ui` directory.
6. **Adapt the VS Code API calls to the Electron API.** The VS Code API calls need to be replaced with the corresponding Electron API calls. This will require significant effort, as the two APIs are very different.
7. **Build and test the application.** Use the Electron Forge or Electron Builder to build and test the application.

5.3 Reusing Code

The following code can be reused from the Roo Code codebase:

- **LLM Interaction Code:** The code in the `src/api` directory can be reused to interact with the LLMs.
- **MCP Integration Code:** The code in the `src/services/mcp` directory can be reused to integrate with the Model Context Protocol (MCP).
- **Core Utilities:** The code in the `src/utils` directory can be reused for various utility functions.
- **React Components:** The React components in the `webview-ui` directory can be reused to implement the user interface.

5.4 Adapting VS Code API Calls

The following VS Code API calls need to be replaced with the corresponding Electron API calls:

- `vscode.commands.registerCommand`: Use `electron.ipcMain` to register a new command.
- `vscode.window.createWebviewPanel`: Use `electron.BrowserWindow` to create a new browser window.
- `vscode.ExtensionContext`: You will need to manage the application state manually, as there is no direct equivalent to `ExtensionContext` in Electron.

5.5 Building and Testing the Application

Use the Electron Forge or Electron Builder to build and test the application.

Note: Due to the significant differences between the VS Code and Electron architectures, it may not be possible to reuse a large portion of the codebase. However, the core logic, utilities, and React components can be reused to reduce the amount of code that needs to be rewritten.

6 Extension Activation

The src/extension.ts file is the main entry point for the Roo Code extension. The activate function is called when the extension is activated.

6.1 Activation Process

1. **Load Environment Variables:** The extension loads environment variables from the .env file using the dotenvx library.
2. **Initialize Output Channel:** The extension creates an output channel named "Roo-Code" for logging messages.
3. **Initialize Telemetry Service:** The extension initializes the telemetry service for collecting usage data.
4. **Initialize Terminal Registry:** The extension initializes the terminal registry for managing terminal-related commands and actions.
5. **Register Webview View Provider:** The extension registers a webview view provider for the sidebar, which is used to display the user interface.
6. **Register Commands:** The extension registers various commands, such as those for interacting with the LLMs and managing the extension's settings.
7. **Register Text Document Content Provider:** The extension registers a text document content provider for displaying diff views.
8. **Register URI Handler:** The extension registers a URI handler for handling custom URIs.
9. **Register Code Actions Provider:** The extension registers a code actions provider for providing suggestions and actions to the user based on the current code context.
10. **Create Roo Code API:** The extension creates the Roo Code API, which is used by other extensions to interact with the Roo Code extension.

6.2 Deactivation Process

The deactivate function is called when the extension is deactivated. It performs the following tasks:

1. **Log Deactivation Message:** The extension logs a message to the output channel indicating that the extension has been deactivated.
2. **Clean Up MCP Server Manager:** The extension cleans up the MCP server manager, which is responsible for managing the MCP servers.
3. **Shutdown Telemetry Service:** The extension shuts down the telemetry service.
4. **Clean Up Terminal Handlers:** The extension cleans up the terminal handlers.

7 File System Utilities

The `src/utils/fs.ts` file defines utility functions for working with the file system.

7.1 Functions

The following functions are defined:

- `createDirectoriesForFile(filePath)`: Creates all non-existing subdirectories for a given file path.
- `fileExistsAtPath(filePath)`: Checks if a path exists.

8 Git Utilities

The `src/utils/git.ts` file defines utility functions for working with Git.

8.1 Functions

The following functions are defined:

- `searchCommits(query, cwd)`: Searches for commits matching a query in the specified working directory.
- `getCommitInfo(hash, cwd)`: Retrieves information about a specific commit in the specified working directory.
- `getWorkingState(cwd)`: Retrieves the status of the working directory.

9 How-To Guide

This document provides guidance on how to perform common tasks in the Roo Code codebase.

For a general overview of how VS Code extensions work, see the VS Code Extensions document.

- How to Create an IntelliJ Plugin
- How to Make Roo Code Run as a Standalone Cross-Platform App

9.1 Changing System Prompts

To change the way system prompts work, you need to modify the `src/shared/modes.ts` file. This file defines the different modes of the extension, and each mode has a `roleDefinition` and `customInstructions` property that define the system prompt for that mode.

1. **Edit the `src/shared/modes.ts` file.**
2. **Find the mode configuration you want to modify.** The `modes` array contains the configuration for each mode.
3. **Update the `roleDefinition` and/or `customInstructions` property.** The `roleDefinition` property defines the role of the mode, and the `customInstructions` property provides custom instructions for the mode.

9.2 Adding a New Chat UI Element

To add a new chat UI element, you need to modify the `webview-ui/src/components/chat/ChatView.tsx` file. This file defines the chat interface.

1. **Edit the `webview-ui/src/components/chat/ChatView.tsx` file.**
2. **Add the corresponding React component to the `ChatView` component.** You can add the component to the existing layout or create a new layout for the component.
3. **Update the state management logic to handle the new UI element.** You may need to add new state variables to the `ChatView` component and update the event handlers to handle the new UI element.

9.3 Adding Additional Settings Pages

To add additional settings pages, you need to modify the `webview-ui/src/components/settings/SettingsVie` file. This file defines the settings interface.

1. **Edit the `webview-ui/src/components/settings/SettingsView.tsx` file.**
2. **Add a new tab to the settings interface.** You can use the existing tab component or create a new tab component.
3. **Create a new React component for the settings page.** This component will display the settings for the new page.
4. **Update the state management logic to handle the new settings.** You may need to add new state variables to the `SettingsView` component and update the event handlers to handle the new settings.

9.4 Adding a New UI Panel

To add a new UI Panel, you need to modify the `src/extension.ts` file. This file is the main entry point for the extension.

1. **Edit the `src/extension.ts` file.**
2. **Register a new webview view provider in the `activate` function.** You can use the `vscode.window.registerWebviewViewProvider` function to register a new webview view provider.
3. **Create a new React component for the UI panel.** This component will display the content of the UI panel.
4. **Update the state management logic to handle the new UI panel.** You may need to add new state variables to the extension and update the event handlers to handle the new UI panel.

9.5 Adding Extensions to ROO

To add extensions to ROO, you need to modify the `package.json` file. This file contains the extension's metadata, dependencies, and build scripts.

1. **Edit the `package.json` file.**
2. **Add the extension as a dependency in the `package.json` file.** You can use the `npm install` command to add the extension as a dependency.
3. **Update the build scripts to include the extension in the build process.** You may need to modify the `esbuild.js` file to include the extension in the build process.

9.6 How to Run Tests

To run tests, you can use the following command:

```
npm run test
```

This command will run all the tests in the project. The test files are located in the `src/__tests__` and `webview-ui/src/__tests__` directories.

9.7 How to Modify the MCP Tool Calls

To modify the MCP tool calls, you need to modify the `src/services/mcp/McpHub.ts` file. This file contains the code for calling the MCP tools.

1. **Edit the `src/services/mcp/McpHub.ts` file.**
2. **Find the `callTool` function.** This function is responsible for calling the MCP tools.
3. **Modify the `callTool` function to change the way the MCP tools are called.** You can modify the arguments that are passed to the tools or the way the results are handled.

9.8 How to Launch Child Tasks

To launch child tasks, you can use the `newTask` tool. This tool allows you to create a new task with a specified mode and initial message.

1. **Use the `newTask` tool to create a new task.** You need to specify the mode and initial message for the new task.
2. **The new task will be launched in a new Cline instance.**

9.9 How to Chain Agents Together

To chain agents together, you can use the `newTask` tool to launch a new task with a specific mode and initial message. The new task can then use the `newTask` tool to launch another task, and so on. This allows you to create a chain of agents that can work together to accomplish a complex task.

1. **Use the `newTask` tool to launch the first agent in the chain.** You need to specify the mode and initial message for the first agent.

2. **In the first agent, use the `newTask` tool to launch the second agent in the chain.** You need to specify the mode and initial message for the second agent.
3. **Repeat step 2 for each agent in the chain.**
4. **The agents will be launched in a sequence, with each agent passing its results to the next agent in the chain.**

9.10 How to Have a Chain with a Coordinator Agent

To have a chain with a coordinator agent, you can use the `newTask` tool to launch a coordinator agent. The coordinator agent can then use the `newTask` tool to launch the other agents in the chain. The coordinator agent can be responsible for coordinating the work of the other agents and for combining their results.

1. **Use the `newTask` tool to launch the coordinator agent.** You need to specify the mode and initial message for the coordinator agent. The initial message should describe the overall task and the roles of the other agents.
2. **In the coordinator agent, use the `newTask` tool to launch the other agents in the chain.** You need to specify the mode and initial message for each agent. The initial message should describe the specific task that each agent is responsible for.
3. **The coordinator agent can then use the results from the other agents to accomplish the overall task.** The coordinator agent can use the `read_file` tool to read the results from the other agents and the `apply_diff` tool to combine the results.

9.11 Does this codebase support sub sub tasks?

While there's no explicit code preventing the creation of sub-sub-tasks (a task launched from a child task), full support isn't guaranteed. The system is designed to launch child tasks, but the implications of deeply nested task chains haven't been fully explored. Use with caution.

10 Integrations Architecture

The `integrations` directory integrates the Roo Code extension with various VS Code features.

10.1 Key Components

- `diagnostics/`: This directory likely contains the code for integrating with VS Code's diagnostics system, which is used to display errors and warnings in the editor.
- `editor/`: This directory likely contains the code for integrating with VS Code's editor features, such as code actions and completions.
- `terminal/`: This directory likely contains the code for integrating with VS Code's terminal, allowing the extension to interact with the terminal.
- `theme/`: This directory likely contains the code for customizing the theme of the extension.
- `workspace/`: This directory likely contains the code for integrating with VS Code's workspace features, such as file watching and project management.

10.2 Relationships

The integration components likely interact with each other and with the core components to provide a seamless experience for the user. For example, the `diagnostics/` might use the `core/` to analyze the code and display errors and warnings in the editor.

11 How to Create an IntelliJ Plugin from the Roo Code Codebase

This document provides detailed instructions on how to create an IntelliJ plugin from the Roo Code codebase.

11.1 Prerequisites

- IntelliJ IDEA IDE
- Basic knowledge of IntelliJ plugin development

11.2 Steps

1. **Create a new IntelliJ plugin project.** Use the IntelliJ IDEA IDE to create a new plugin project. Select "New Project" from the File menu, then select "IntelliJ Platform Plugin" from the project types.
2. **Identify reusable code.** The core logic of the Roo Code extension, such as the code for interacting with the LLMs, the MCP integration, and the core utilities, can be reused in the IntelliJ plugin. This code is located in the `src` directory.
3. **Adapt the VS Code API calls to the IntelliJ API.** The VS Code API calls need to be replaced with the corresponding IntelliJ API calls. This will require significant effort, as the two APIs are very different. You will need to consult the IntelliJ Platform SDK documentation to find the appropriate API calls.
4. **Implement the UI using the IntelliJ UI framework.** The webview UI needs to be reimplemented using the IntelliJ UI framework. This will also require significant effort, as the two UI frameworks are very different. You will need to use Swing or JavaFX to create the UI.
5. **Create a `plugin.xml` file.** This file describes the plugin to the IntelliJ IDEA IDE. The `plugin.xml` file should be located in the `src/main/resources/META-INF` directory.
6. **Build and test the plugin.** Use the IntelliJ IDEA IDE to build and test the plugin. Select "Build" from the Build menu, then select "Build Project". To test the plugin, select "Run" from the Run menu, then select "Run".

11.3 Reusing Code

The following code can be reused from the Roo Code codebase:

- **LLM Interaction Code:** The code in the `src/api` directory can be reused to interact with the LLMs.
- **MCP Integration Code:** The code in the `src/services/mcp` directory can be reused to integrate with the Model Context Protocol (MCP).
- **Core Utilities:** The code in the `src/utils` directory can be reused for various utility functions.

11.4 Adapting VS Code API Calls

The following VS Code API calls need to be replaced with the corresponding IntelliJ API calls:

- `vscode.commands.registerCommand`: Use `com.intellij.openapi.actionSystem.AnAction` to register a new action.
- `vscode.window.createWebviewPanel`: Use `com.intellij.openapi.ui.SimpleToolWindowPanel` to create a new tool window.
- `vscode.ExtensionContext`: Use `com.intellij.openapi.components.ApplicationComponent` or `com.intellij.openapi.components.ProjectComponent` to access the plugin's context.

11.5 Implementing the UI

The webview UI needs to be reimplemented using the IntelliJ UI framework. You can use Swing or JavaFX to create the UI.

11.6 plugin.xml

The `plugin.xml` file describes the plugin to the IntelliJ IDEA IDE. Here is an example `plugin.xml` file:

```xml
<idea-plugin>
    <id>com.example.roocode.intellij</id>
    <name>Roo Code IntelliJ Plugin</name>
    <version>1.0</version>
    <vendor email="support@example.com" url="http://www.example.com">Example</vendor>
    <description><![CDATA[
    This plugin provides AI-powered code assistance for IntelliJ IDEA.
    ]]></description>

    <depends>com.intellij.modules.platform</depends>

    <extensions defaultExtensionNs="com.intellij">
        <!-- Add your extensions here -->
    </extensions>

    <actions>
        <!-- Add your actions here -->
    </actions>
</idea-plugin>
```

11.7 Running Roo Code in a Container/Sandbox

To enhance security and isolation, you might consider running the core Roo Code logic (LLM interactions, MCP) within a container or sandbox environment inside the IntelliJ plugin. Here are a few options:

- **Docker Container:** Package the core logic into a Docker container and use the IntelliJ plugin to communicate with the container via a REST API. This provides strong isolation and allows you to manage dependencies separately.
- **JVM Sandbox:** Use the Java Security Manager or a similar sandboxing mechanism to restrict the access of the Roo Code logic to system resources. This is a lighter-weight option than Docker, but it may not provide as strong isolation.
- **GraalVM Native Image:** Compile the core Roo Code logic to a native image using GraalVM. This can improve performance and reduce the memory footprint of the plugin.

Note: Due to the significant differences between the VS Code and IntelliJ plugin architectures, it may not be possible to reuse a large portion of the codebase. However, the core logic and utilities can be reused to reduce the amount of code that needs to be rewritten.

12 Language Models (LLMs)

The Roo Code extension supports various language models (LLMs) through the API providers in the `src/api/providers` directory.

12.1 Supported LLMs

The following LLMs are currently supported:

- **OpenAI:** The OpenAI provider uses the OpenAI API to interact with the OpenAI models, such as GPT-3 and GPT-4.
- **Anthropic:** The Anthropic provider uses the Anthropic API to interact with the Anthropic models, such as Claude.
- **Bedrock:** The Bedrock provider uses the AWS Bedrock API to interact with various LLMs available on AWS Bedrock.
- **Gemini:** The Gemini provider uses the Google Gemini API to interact with the Gemini models.
- **Mistral:** The Mistral provider uses the Mistral API to interact with the Mistral models.
- **Ollama:** The Ollama provider uses the Ollama API to interact with the Ollama models.
- **LM Studio:** The LM Studio provider uses the LM Studio API to interact with the LM Studio models.
- **Vertex:** The Vertex provider uses the Google Vertex AI API to interact with the Vertex AI models.
- **Deepseek:** The Deepseek provider uses the Deepseek API to interact with the Deepseek models.
- **OpenRouter:** The OpenRouter provider uses the OpenRouter API to interact with various LLMs available on OpenRouter.
- **VSCode-LM:** The VSCode-LM provider uses the VSCode Language Model API to interact with local language models.
- **Unbound:** The Unbound provider uses the Unbound API to interact with various LLMs available on Unbound.
- **Human Relay:** The Human Relay provider allows the user to manually provide input for the LLMs.

12.2 API Providers

Each LLM has its own API provider in the `src/api/providers` directory. The API providers implement the specific API calls and data transformations required for each LLM service. The `base-provider.ts` file defines the base class or interface for all LLM providers.

13 MCP Integration

The `src/services/mcp` directory contains the code for integrating with the Model Context Protocol (MCP), which is used to communicate with external tools and services.

13.1 Key Components

- `McpServerManager.ts`: This file manages the MCP server instances. It ensures that only one set of MCP servers runs across all webviews.
- `McpHub.ts`: This file is the main class for managing the MCP connections. It is responsible for connecting to the MCP servers, fetching the tools and resources, and handling the communication between the extension and the MCP servers.

13.2 MCP Server Management

The `McpServerManager` class is a singleton that manages the MCP server instances. It provides the following functionalities:

- `getInstance()`: Returns the singleton instance of the `McpHub` class.
- `cleanup()`: Cleans up the singleton instance and all its resources.

13.3 MCP Connection Management

The `McpHub` class manages the MCP connections. It provides the following functionalities:

- `getServers()`: Returns the list of enabled MCP servers.
- `getAllServers()`: Returns the list of all MCP servers, regardless of their state.
- `connectToServer()`: Connects to an MCP server.
- `deleteConnection()`: Deletes an MCP connection.
- `updateServerConnections()`: Updates the MCP server connections based on the settings file.
- `readResource()`: Reads a resource from an MCP server.
- `callTool()`: Calls a tool on an MCP server.

- `toggleToolAlwaysAllow()`: Toggles the always allow setting for a tool on an MCP server.

13.4 MCP Settings

The MCP settings are stored in a JSON file named `mcp_settings.json`. The file contains a list of MCP servers, each with its own configuration. The configuration includes the command to run the server, the arguments to pass to the server, and the environment variables to set for the server.

13.5 File Watching

The `McpHub` class uses `chokidar` to watch for changes in the MCP server files. When a change is detected, the `McpHub` class restarts the connection to the server.

14 Extension Messages

The `src/shared/ExtensionMessage.ts` file defines the messages that are sent between the extension and the webview. The `ExtensionMessage` interface has a `type` property that indicates the type of message.

14.1 Message Types

The following message types are defined:

- `action`: Indicates an action that should be performed in the webview.
- `state`: Indicates the state of the extension.
- `selectedImages`: Indicates the selected images.
- `ollamaModels`: Indicates the available Ollama models.
- `lmStudioModels`: Indicates the available LM Studio models.
- `theme`: Indicates the current theme.
- `workspaceUpdated`: Indicates that the workspace has been updated.
- `invoke`: Indicates that a function should be invoked in the webview.
- `partialMessage`: Indicates a partial message.
- `openRouterModels`: Indicates the available OpenRouter models.
- `glamaModels`: Indicates the available Glama models.
- `unboundModels`: Indicates the available Unbound models.
- `requestyModels`: Indicates the available Requesty models.
- `openAiModels`: Indicates the available OpenAI models.
- `mcpServers`: Indicates the available MCP servers.
- `enhancedPrompt`: Indicates an enhanced prompt.
- `commitSearchResults`: Indicates the commit search results.
- `listApiConfig`: Indicates the list of API configurations.
- `vsCodeLmModels`: Indicates the available VSCode Language Model models.
- `vsCodeLmApiAvailable`: Indicates whether the VSCode Language Model API is available.
- `requestVsCodeLmModels`: Indicates a request for VSCode Language Model models.
- `updatePrompt`: Indicates an update to a prompt.
- `systemPrompt`: Indicates the system prompt.
- `autoApprovalEnabled`: Indicates whether auto approval is enabled.
- `updateCustomMode`: Indicates an update to a custom mode.

- `deleteCustomMode`: Indicates a deletion of a custom mode.
- `currentCheckpointUpdated`: Indicates an update to the current checkpoint.
- `showHumanRelayDialog`: Indicates that a human relay dialog should be shown.
- `humanRelayResponse`: Indicates a response from the human relay dialog.
- `humanRelayCancel`: Indicates that the human relay dialog has been cancelled.
- `browserToolEnabled`: Indicates whether the browser tool is enabled.
- `browserConnectionResult`: Indicates the result of a browser connection.
- `remoteBrowserEnabled`: Indicates whether the remote browser is enabled.

15 Modes

The `src/shared/modes.ts` file defines the different modes of the extension. Each mode has a `slug`, `name`, `roleDefinition`, `customInstructions`, and `groups`. The `groups` property specifies the tool groups that are allowed for the mode.

15.1 Modes

The following modes are defined:

- `code`: This mode is for general code editing and development.
- `architect`: This mode is for planning and designing the architecture of the codebase.
- `ask`: This mode is for asking questions about the codebase and getting information about software development.
- `debug`: This mode is for debugging the codebase.

15.2 Mode Configuration

Each mode has the following configuration properties:

- `slug`: A unique identifier for the mode.
- `name`: The display name for the mode.
- `roleDefinition`: A description of the role of the mode.
- `customInstructions`: Custom instructions for the mode.
- `groups`: The tool groups that are allowed for the mode.

15.3 Tool Groups

The `groups` property specifies the tool groups that are allowed for the mode. The tool groups are defined in the `src/shared/tool-groups.ts` file.

16 Project Overview

This project appears to be a VS Code extension named “Roo Code” that provides AI-powered code assistance.

16.1 Key Areas

- **Core:** Contains the core logic of the extension, including the `Cline` class, code action provider, and webview management.
- **API:** Defines the API for interacting with different language models (LLMs) and providers.
- **Webview UI:** Implements the user interface using React, TypeScript, and Tailwind CSS.
- **Integrations:** Integrates the extension with various VS Code features, such as diagnostics, editor actions, and terminal actions.
- **Services:** Provides various services, such as browser integration, MCP integration, and telemetry.

16.2 Top-Level Files

- `src/extension.ts`: The main entry point for the extension.
- `webview-ui/src/App.tsx`: The main component for the webview UI.
- `package.json`: Contains the extension’s metadata, dependencies, and build scripts.

16.3 External Dependencies

- **Model Context Protocol (MCP):** Used for integrating with external tools and services.
- **Language Models (LLMs):** Supports various LLMs through the API providers.

16.4 Program Flow

16.4.1 Plugin Initialization

```
sequenceDiagram
    participant VSCode as VS Code
    participant Extension as src/extension.ts
    participant ClineProvider as ClineProvider [[../src/core/webview/ClineProvider.ts]]
    participant McpServerManager as McpServerManager [[../src/services/mcp/McpServerManager.t
    participant TelemetryService as TelemetryService [[../src/services/telemetry/TelemetrySe
    participant TerminalRegistry as TerminalRegistry [[../src/integrations/terminal/TerminalR

    VSCode->>Extension: activate(context)
    Extension->>TelemetryService: initialize()
    Extension->>TerminalRegistry: initialize()
    Extension->>ClineProvider: new ClineProvider(context, outputChannel)
    Extension->>VSCode: registerWebviewViewProvider(ClineProvider.sideBarId, sidebarProvider)
    Extension->>VSCode: registerCommands(context, outputChannel, provider)
    Extension->>VSCode: registerTextDocumentContentProvider(DIFF_VIEW_URI_SCHEME, diffConten
    Extension->>VSCode: registerUriHandler(handleUri)
    Extension->>VSCode: registerCodeActionsProvider(pattern, new CodeActionProvider())
    Extension->>McpServerManager: cleanup(context)
    Extension->>TelemetryService: shutdown()
    Extension->>TerminalRegistry: cleanup()
```

16.4.2 User Chat Loop

```
sequenceDiagram
    participant User as User
    participant VSCode as VS Code
    participant ClineProvider as ClineProvider [[../src/core/webview/ClineProvider.ts]]
    participant Cline as Cline [[../src/core/Cline.ts]]
    participant APIProvider as APIProvider [[../src/api/providers/base-provider.ts]]
    participant LLM as LLM
    participant ToolManager as ToolManager

    User->>VSCode: Enters message in Cline
    VSCode->>ClineProvider: postMessageToWebview({type: "userMessage", text: message})
    ClineProvider->>Cline: Handles user message
    Cline->>ToolManager: Determines if tool is needed
    alt Tool is needed
        ToolManager->>APIProvider: Sends request to tool
        APIProvider->>LLM: Sends request to LLM to use tool
        LLM-->>APIProvider: Returns tool result
        APIProvider->>Cline: Transforms tool result
        Cline->>ClineProvider: postMessageToWebview({type: "toolResult", text: toolResult})
        ClineProvider->>VSCode: Displays tool result in Cline
        VSCode->>User: Displays tool result
    else Tool is not needed
        Cline->>APIProvider: Sends message to API Provider
        APIProvider->>LLM: Sends request to LLM
        LLM-->>APIProvider: Returns response
        APIProvider->>Cline: Transforms response
        Cline->>ClineProvider: postMessageToWebview({type: "assistantMessage", text: respons
        ClineProvider->>VSCode: Displays message in Cline
        VSCode->>User: Displays message
    end
```

17 Codebase Analysis Plan

1. **Create `llm_docs` directory:** Create a directory to store the markdown documentation.
2. **High-Level Overview:** Create a markdown file (`llm_docs/overview.md`) with a high-level overview of the project, based on the file structure and names.
3. **Key Architectural Areas:** Identify and document the main architectural areas (e.g., API, core, webview-ui) in separate markdown files within the `llm_docs` directory.
4. **Important Files:** For each architectural area, identify and document the most important files and their roles.
5. **Extension Activation:** Analyze `src/extension.ts` to understand how the extension is activated and what components are initialized.
6. **MCP Integration:** Analyze the `src/services/mcp` directory to understand how Model Context Protocol (MCP) is integrated into the extension.
7. **Webview UI:** Analyze the `webview-ui` directory to understand the structure and components of the user interface.
8. **Categories Documentation:** Identify and document categories like 'external APIs' and 'LLMs' in a structured way within the `llm_docs` directory.
9. **Review and Refine:** Review the generated documentation and identify any missing information or areas that need further clarification.
10. **Final Touches:** Add any final touches to the documentation, such as diagrams or examples.

17.1 Categories to Document

- External APIs
- LLMs

18 Services Architecture

The `services` directory provides various services used by the Roo Code extension.

18.1 Key Components

- `browser/`: This directory likely contains the code for integrating with a browser, allowing the extension to display web pages or interact with web-based services.
- `checkpoints/`: This directory likely contains the code for managing checkpoints, which are used to save and restore the state of the extension.
- `glob/`: This directory likely contains the code for performing globbing operations, which are used to find files that match a certain pattern.
- `mcp/`: This directory contains the code for integrating with the Model Context Protocol (MCP), which is used to communicate with external tools and services.
- `ripgrep/`: This directory likely contains the code for using ripgrep, a fast and efficient search tool.
- `telemetry/`: This directory likely contains the code for collecting telemetry data, which is used to track the usage of the extension.
- `tree-sitter/`: This directory likely contains the code for using tree-sitter, a parser generator tool.

18.2 Relationships

The service components likely interact with each other and with the core components to provide various functionalities for the extension. For example, the `mcp/` might use the `browser/` to display information from external tools and services, and the `telemetry/` might be used to track the usage of the different services.

19 Shell Utilities

The `src/utils/shell.ts` file defines utility functions for getting the shell path.

19.1 Functions

The following functions are defined:

- `getShell()`: Attempts to retrieve the shell path from VS Code config, `userInfo()`, environment variable, and finally falls back to a default.

20 Sound Utilities

The `src/utils/sound.ts` file defines utility functions for playing sounds.

20.1 Functions

The following functions are defined:

- `playSound(filepath)`: Plays a sound file, but only if sound is enabled and the minimum play interval has elapsed.
- `setSoundEnabled(enabled)`: Sets whether sound is enabled.
- `setSoundVolume(volume)`: Sets the sound volume.
- `isWAV(filepath)`: Determines if a file is a WAV file.

21 Telemetry Settings

The `src/shared/TelemetrySetting.ts` file defines the `TelemetrySetting` type, which can be one of the following values: `unset`, `enabled`, or `disabled`.

21.1 Telemetry Settings

The following telemetry settings are defined:

- `unset`: The telemetry setting has not been set.
- `enabled`: Telemetry is enabled.
- `disabled`: Telemetry is disabled.

22 How VS Code Extensions Work

VS Code extensions extend the functionality of VS Code by using a combination of contribution points and VS Code APIs.

22.1 Key Concepts

- **Activation Events:** Events upon which your extension becomes active. See `src/extension.ts` for how Roo Code uses activation events.
- **Contribution Points:** Static declarations that you make in the `package.json` file to extend VS Code. See the `contributes` section in `package.json` for how Roo Code uses contribution points.
- **VS Code API:** A set of JavaScript APIs that you can invoke in your extension code. See `src/extension.ts` for how Roo Code uses the VS Code API.

22.2 Extension File Structure

```
.
  .vscode
     launch.json     // Config for launching and debugging the extension
     tasks.json      // Config for build task that compiles TypeScript
  .gitignore          // Ignore build output and node_modules
  README.md           // Readable description of your extension's functionality
  src
     extension.ts    // Extension source code
  package.json        // Extension manifest
  tsconfig.json       // TypeScript configuration
```

22.3 Extension Manifest

Each VS Code extension must have a `package.json` file. The `package.json` contains a mix of Node.js fields such as `scripts` and `devDependencies` and VS Code specific fields such as `publisher`, `activationEvents` and `contributes`. Here are some most important fields:

- `name` and `publisher`: VS Code uses `<publisher>.<name>` as a unique ID for the extension.
- `main`: The extension entry point.
- `activationEvents` and `contributes`: Activation Events and Contribution Points.
- `engines.vscode`: This specifies the minimum version of VS Code API that the extension depends on.

22.4 Extension Entry File

The extension entry file exports two functions, `activate` and `deactivate`. `activate` is executed when your registered Activation Event happens. See `src/extension.ts` for Roo Code's implementation. `deactivate` gives you a chance to clean up before your extension becomes deactivated.

23 Webview UI Architecture

The `webview-ui` directory implements the user interface for the Roo Code extension using React, TypeScript, and Tailwind CSS.

23.1 Key Components

- `src/App.tsx`: This file is the main component for the webview UI and likely renders the main layout and components.
- `src/index.tsx`: This file is the entry point for the webview UI and likely initializes the React application.
- `src/components/`: This directory contains the reusable components used in the webview UI.
- `src/context/`: This directory likely contains the React context providers, which are used to manage the state of the webview UI.
- `src/lib/`: This directory likely contains utility functions and helper classes used in the webview UI.
- `src/index.css`: This file contains the CSS styles for the webview UI, including VSCode CSS variables.

23.2 Technologies

- **React:** A JavaScript library for building user interfaces.
- **TypeScript:** A superset of JavaScript that adds static typing.
- **Tailwind CSS:** A utility-first CSS framework.
- **Vite:** A build tool that provides fast development and optimized production builds.

23.3 Relationships

The webview UI components likely interact with each other to provide the user interface for the extension. The `App.tsx` component might use the components in `src/components/` to render the different parts of the UI, and the `src/context/` might be used to manage the state of the UI.

24 Webview UI Structure

The `webview-ui/src/App.tsx` file is the main component for the webview UI. It defines the structure and layout of the UI.

24.1 Key Components

- `ChatView`: This component displays the chat interface, where the user can interact with the LLMs.
- `HistoryView`: This component displays the chat history.
- `SettingsView`: This component displays the settings for the extension.
- `McpView`: This component displays the MCP server management interface.
- `PromptsView`: This component displays the prompts management interface.
- `WelcomeView`: This component displays the welcome screen.
- `HumanRelayDialog`: This component displays a dialog for human relay, allowing the user to manually provide input for the LLMs.

24.2 State Management

The `App` component uses the `ExtensionStateContext` to manage the state of the UI. The `ExtensionStateContext` provides access to the following state variables:

- `didHydrateState`
- `showWelcome`
- `shouldShowAnnouncement`
- `telemetrySetting`
- `telemetryKey`
- `machineId`

24.3 Tab Navigation

The `App` component uses a tab-based navigation system to switch between the different views. The `tab` state variable determines which view is currently displayed. The `switchTab` function is used to switch between the different views.

24.4 Message Handling

The `App` component uses the `useEvent` hook to listen for messages from the extension. When a message is received, the `onMessage` function is called. The `onMessage` function processes the message and updates the state of the UI accordingly.