Conceptual Architecture - Gemini CLI

Group Name: The Rat Pack

Aatman Gupta 22ag32@queensu.ca

Caleb Gao 21cg49@queensu.ca

Cory Boys (presenter) 21alb25@queensu.ca

Finnegan MacAdams (presenter) 23bmd@queensu.ca

Jude Campanelli-Henderson (leader) 22nbx2@queensu.ca

Rémy Brown-Vandervecht 22WLHR@queensu.ca

CISC 322

Professor Bram Adams

Abstract

This report examines the architectural foundations of Gemini CLI. Gemini CLI is an AI agent that integrates Google's Gemini into a user's terminal. To provide an architectural overview, we have decomposed Gemini CLI into its components and subsystems. The CLI package component, which handles front-end processes when executed in the terminal; the core engine component, which manages back-end packages in response to user input; and the tools component, which manages other tools available to the user within the system. The report examines two primary architectural styles: client-server and feedback control systems. The client-server model depicts the two identities and their relationship, explaining how input is computed via Gemini. The feedback control system describes the front-end and back-end interaction: it takes user input, packages it into a computation performed by the Gemini API, and returns the result to the display. The report also states two user use cases. The first describes the process of a developer using code context to diagnose and fix failing tests, which show Gemini CLI's debugging and approval capabilities. The second use case describes the process for generating a program from online specifications. This demonstrates the system's ability to retrieve web content and to apply file-system tools to protect data.

1. Introduction and Overview

The goal of this report is to document the conceptual architecture of Gemini CLI. Gemini CLI (command line interface) is an open-source AI agent that integrates Google's Gemini large language model into a user's terminal. Released in June 2025, Gemini CLI was designed to be a terminal-first tool for developers, providing users with powerful models and Gemini 2.5 Pro in their command line. It was also designed to be fully open and extensible, enabling users to customize their terminal and fully leverage the agent. Gemini CLI has undergone several major upgrades since its release, which have enhanced security features, enabled users to connect Gemini to their cloud data and integrate with Google Workspace, supported IDE plugins, and allowed users to install or create extensions within Gemini CLI.

We discuss the two main packages of Gemini CLIs: the CLI package and the Core package. The CLI is a software mechanism for interacting with a computer via a text-based user interface. This means it is responsible for the user-facing portion of the Gemini CLI, including handling user input and computed output. The core package contains essential classes and interfaces and serves as the back end, interacting with the Gemini API to execute and return user requests.

We also study the two primary architectural styles used in the Gemini CLI: client-server and feedback control systems. Client-server architecture is used when multiple clients request services from a central server, such as the Gemini API. Client-server architecture benefits from modularity. This is because dividing the system into the distinct models allows for easier development, testing, and maintenance. Gemini CLI is also a direct example of how modularity supports reusability. This is because the Gemini API is being used with a CLI package to allow the Gemini AI to be used within Gemini CLI.

2. Derivation Process

The derivation process began with each group member reviewing the provided documentation on the Gemini CLI architecture. We then conducted our own research to gain further background knowledge of software architecture in general and of the Gemini CLI, drawing on lecture slides and class resources, Gemini-released documentation, technical blogs and articles, and YouTube videos. We would then use our tutorial, TA meeting times, and extra work sessions to discuss our findings and how they connected to our understanding of the architecture. Based on our findings, we were able to connect Gemini CLI's architecture to the client-server model and the feedback control system. To construct box-and-arrow diagrams representing the described styles, our members learned to use Lucidchart and Krita. Creating the diagrams helped clarify and deepen our understanding of the processes among the packages. Using these charts and additional online research on the Gemini CLI's functionality, we developed a case study of its user-facing front end, focusing on its command-line interface and its interaction with the Gemini API. Our first use case focused on the iterative CLI-to-Core-to-Gemini-API workflow for running tests, gathering code context, proposing patches, and re-running tests, with explicit user approval required before any shell command is executed or any filesystem change is applied. Our second use case focused on a 'build-from-spec' workflow: the developer provides a URL, the CLI forwards it to Core for structured prompt construction, Core uses the web-fetch tool to retrieve requirements, and then (after explicit user approval) invokes filesystem tools to create/write the new program file before returning a final summary to the CLI.

3. Architecture Components & Subsystems

3.1 System Overview

Gemini CLI operates in an elevated command-line environment, offering customizable terminal themes and configurations via a React-based Terminal UI. It distinctly separates its front-end and back-end components (CLI and Core) into separate packages within the same repositories. This improved extensibility, enabling different frontend components to share the same backend. These packages are supported by a registry of tools that the system can use when operating in the command line.
Below is a diagram representing the interactions between the high-level components. As shown, when interacting with the system, a user types in the terminal; the CLI Package manages the input and sends it to the Core package. This package handles all AI orchestration and tool management, supporting extensibility by integrating these components into the system and providing numerous options for user configuration. When a solution is reached, the Core Engine will send it back to the CLI package, which will format and present it to the user.

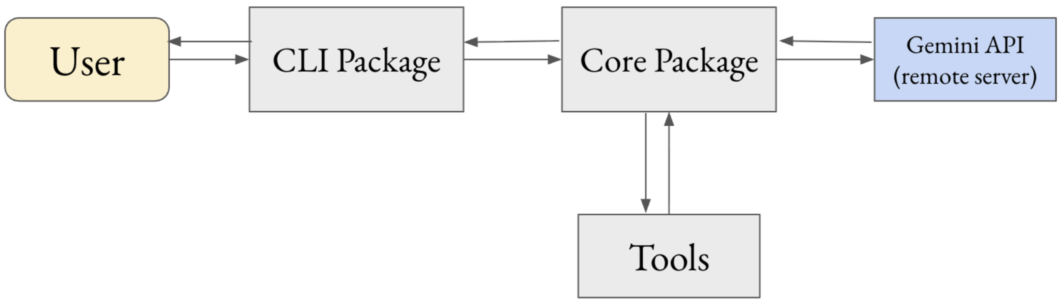


Figure 3.1 high level conceptual architecture box and arrow diagram

3.2 CLI Package Component

This component is responsible for the system's front-end processes, handling all interactions between the user and the system. It uses a terminal-based React component and builds on the Ink library for the terminal user interface. It handles user input by parsing commands, maintains control of the prompt loop for sessions, and tracks the project and user history. The system also offers many advanced configurations at the on-system, global, and project-specific levels, supports both customizable and built-in themes, and manages connection security for the model. It then formats responses from the Core Engine into comprehensible, readable output for presentation to users.

3.2.1 User Interaction System handles all user interaction with the CLI. It manages the presentation of the interactive terminal interface to users. It also manages user input by parsing commands and flags and maintaining the prompt loop and session flow. It then forwards user requests to the Core Engine and formats responses and status messages for clear presentation to users.
3.2.2 Configuration System is responsible for the more personalized aspects of the system. It offers multiple configuration levels, including system-wide default settings, user-specific global settings, and lower-level settings that provide advanced configuration for specific projects and environments. It also handles authentication, manages API keys, and handles account setup for secure connections to the model service. It is overall responsible for the CLI configuration and authorization before input is sent to the Core Engine.

3.3 Core Engine Component

This component is the foundation of Gemini CLI, providing backend logic for interactions between the Gemini AI, tools, and the frontend CLI component. It takes the instructions from the frontend and constructs them into strong, effective prompts to be sent to the AI model, including relevant tool information and conversation history. It then receives the AI response, sends instructions to the relevant components, and outputs the final result to the CLI.

3.3.1 Prompt Management takes the input passed along by the CLI package and formats it into an effective prompt to be given to Gemini AI. It sends prompts that combine the user input with conversation history and any relevant files. The system ensures consistency and proper formatting across model interactions to enhance the quality of the responses it receives.
3.3.4 Agent Control Logic manages the AI's workflow in the Core Package, ensuring that the process for component interaction is well structured. It determines whether to invoke tools or return answers, and orchestrates system interactions until the request is complete. It also includes several execution controls, such as confirmation prompts and safety checks, to ensure that responses comply with the security policy and that actions occur in the correct order and only when authorized.

3.4 Tools Component

This component operates within the Core Engine, managing the system's many tools. It provides access to built-in commands and supports third-party tools. Managing the logic and invocation validation of the tools in the registry, it ensures proper integration with the Gemini model's command execution.

3.4.2 Tool Invocation is responsible for actually executing the tools that are requested by the model when the Core Engine is running. It validates the requested tool and its arguments, runs the tool, and then stores the results for transmission to the Core Engine. It also applies permission checks and confirmation rules when needed to ensure safety and correct execution flow.
3.4.3 Tool Registry maintains the list of tools that are available to the system during task execution. It stores each tool's definition, capabilities, and other logic to provide the core engine with context for available actions. It is also responsible for discovering and registering tools that are not already in the system

Since there are such a large number of subsystems in Gemini CLI, many were excluded from this review. The items relevant to the descriptions of architectural styles and use cases were included, along with others deemed most relevant to the overall structure.

4. Architecture Styles

4.1: Architectural Style #1: Client-Server

Definition and Applicability:

The primary architectural style used in Gemini CLI is the Client-Server pattern. In this context, the system is strictly divided into two independent entities connected via a network. The Client is the local Gemini CLI application running on the user's machine, while the Server is Google's Gemini API infrastructure. The CLI acts as a "thick client", as it performs significant local processing (things such as gathering content from files, history management, and input formatting) before making a request to the server.

Component Interaction:

The interaction is request-driven. The CLI (Client) initiates the communication by constructing a structured prompt that contains the user's query and relevant file context. It transmits the payload via HTTPS (using either REST or gRPC) to the Gemini endpoint (Server). This Server, which hosts the LLM, then performs the expensive computations and streams the response back to the client. The Client can then render this response into the terminal as markdown.

Motivation and Benefits:

- **Scalability:** This style is essential because running an LLM of Gemini's size locally is impossible for most users' hardware. Offloading the inference to Google's servers allows the tool to provide Gemini's flagship intelligence on any machine.
- **Centralized Evolution:** Google can update the model (the Server in this scenario) behind the scenes without requiring users to install updates or new CLI binaries. The Client automatically benefits from all the backend improvements.

Challenges and Repercussions:

- **Latency:** Every interaction incurs a network round-trip penalty. Unlike a tool that runs locally (eg. grep), the responsiveness is bound by the user's internet speed and the API processing time.
- **Availability:** The tool requires an internet connection. If the server is down or the user is offline, Gemini CLI cannot function.
- **Data Privacy:** This style requires sending local code and data to a remote server, which entails potential privacy risks that the user has no choice but to accept. Data leaks can occur at any time, and this approach may put users' data at risk.

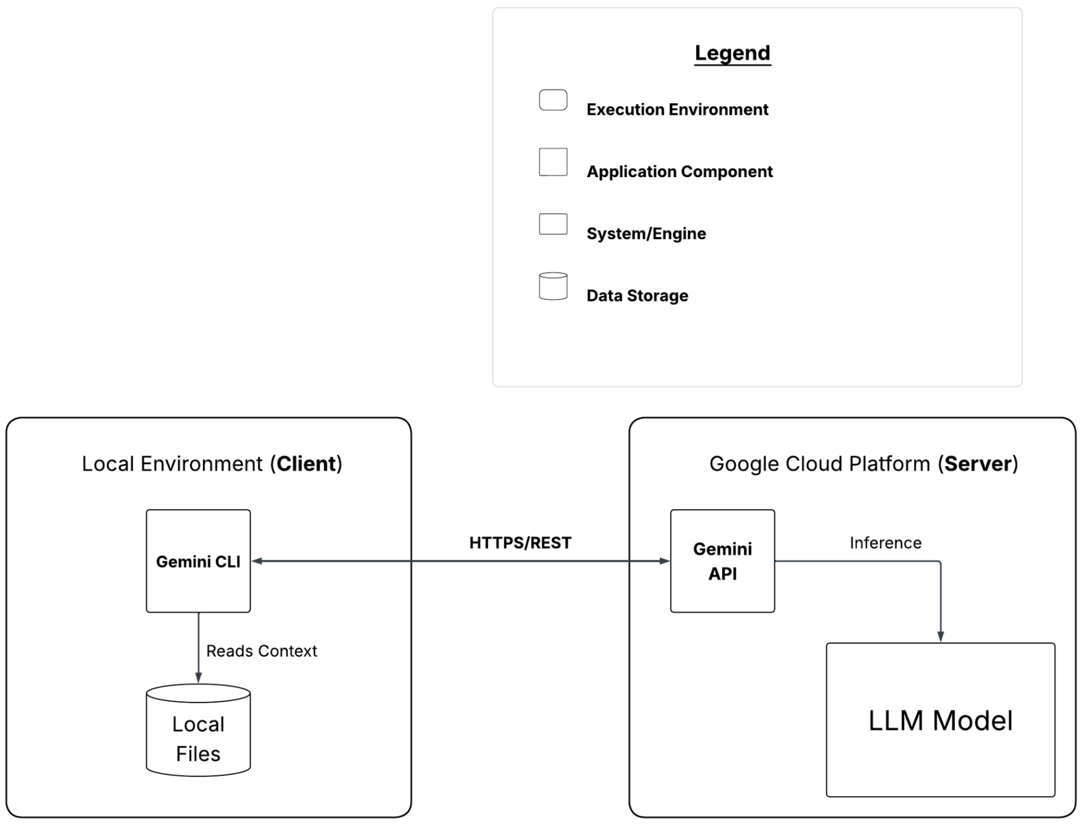


Figure 4.1 diagram representing the client-server style

4.2 Architectural Style #2: Feedback control system

Definition and Applicability:

The secondary architectural style employed in Gemini CLI is the Feedback-Control System (specifically, an Agentic Control Loop). Unlike a linear "input-process-output" program, the system operates in a dynamic loop in which the software continuously observes the environment, takes action, and adjusts based on the results. This style is implemented through the interaction between the Core Package (the Controller) and the CLI Package (the Environment). This is implemented using a ReAct (Reason and Act) pattern, enabling the Gemini CLI to autonomously solve multi-step problems.

Component Interaction:

The interaction operates as a continuous loop starting with the Controller (the Core Package), which acts as the system's "brain". It processes the user's input to establish a target state, and then constructs a prompt for the Gemini API. If the API determines that an action is required, the Core package activates Actuators (Tool Execution) to perform tasks such as modifying files or executing commands; for critical actions, this step includes a manual confirmation check. The results of these actions, either successful outputs or error messages, are captured as Feedback and fed back into the Controller's context window. This enables the system to compare the current state with the target goal and dynamically determine the next required action. The cycle is repeated until the user's request is fully resolved, and the final result is returned to the CLI package for display to the user.

Motivations and Benefits:

- **Autonomous Problem Solving:** The Feedback-Control style enables Gemini CLI to handle complex tasks without constant user intervention. For example, the agent can write code, encounter an error, receive that error as feedback, and then automatically generate a fix in the next iteration.
- **Dynamic Adaptation:** The path to any given solution is not hard-coded. The Controller adapts its strategy in real time using trial and error, using the feedback it receives from the environment (for example, realizing a file is missing and deciding to create it)
- **Self-Correction:** This feedback loop provides a mechanism for error recovery. If the AI "hallucinates" a command that doesn't exist, the immediate "Command Not Found" feedback triggers, and the model then tries a valid alternative.

Challenges and Repercussions:

- **Infinite Loops:** One critical risk of control loops is the system becoming "stuck," repeating the same action indefinitely. This necessitates strict "circuit breaker" mechanisms (such as maximum iteration limits) to prevent runaway processes.
- **Cost and Latency Accumulation:** Because each step in the control loop triggers a full API request to the Server, complex tasks can become significantly expensive and slow. A 5-step loop effectively increases the architecture's latency by a factor of 5.
- **Non-Determinism:** Debugging a control loop can be quite difficult, because the "Controller" (the LLM) is probabilistic. The same input state may yield different outcomes across runs, making its behaviour difficult to test and predict consistently.

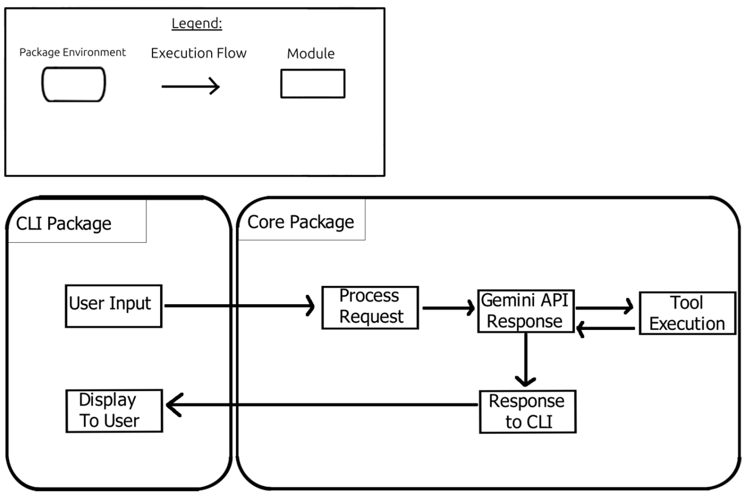


Figure 4.2 diagram of the feedback control style

4.3 Alternatives Considered

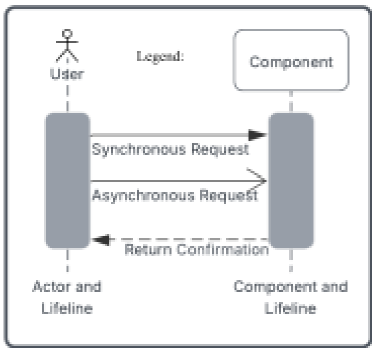
4.3.1 Layered Architecture: In a strict Layered style, we would organize Gemini CLI into tiers (for example, Terminal UI → CLI Command Parsing → Core orchestration → Tool adapters → Gemini API). This could improve the software's maintainability because each layer would have a clear role, and changes in the UI layer are less likely to cause ripple effects in the core orchestration or tool execution. That being said, it could potentially worsen development speed and complexity for an AI agent, because the feedback-control loop we describe often needs fast back-and-forth across the "layers", and strict layering can force an awkward cross-layer interface. Therefore, we treated layering as a useful guideline for code organization, but not as the primary system style.

4.3.2 Pipe-and-Filter Architecture: Using a Pipe-and-Filter style, the system could be modelled as a mostly linear pipeline resembling: input → prompt construction → model call → output formatting, where each step transforms the data and passes it forward. This could potentially improve the testability and transparency of the software, since each stage can be tested independently. With that in mind, it could still worsen support for intensive, multi-step agent behaviour, because our use cases rely on repeated tool use with approvals, error feedback, and iterative processes until a successful outcome is reached. Hence, we preferred the feedback-control interpretation of the "agentic loop" behaviour, even though parts of the system locally mimic pipeline stages.

5. Use Cases

The Gemini CLI is a command-line tool designed for developers. With its seamless integration of a user-friendly front-end terminal and a robust back-end orchestration engine, it simplifies the process of testing, understanding, and modifying code. The CLI package efficiently captures user input and presents results directly in the terminal, while the Core package manages prompt creation, interacts with the Gemini API, and orchestrates various tool actions, including running tests and applying patches. Importantly, it maintains user control by requiring explicit approval before executing any shell commands or making modifications.

The legend used in both sequence diagrams is represented by the following diagram:



5.1 Use Case 1:

This Use Case demonstrates a developer testing a project, diagnosing a failure using code context, applying a fix, and verifying the result through repeated test execution.

To start the process, we must be prompted by the developer (user) through the terminal. The CLI package receives the input and forwards it to the Core package, which prepares a structured prompt for the Gemini API. The structure prompt includes the developer's request and additional context (session state, available tool definitions, history) to provide the model with a better understanding of the system, thereby reducing guesswork.

Upon receiving the prompt, the Gemini API sends a request to use the shell-based tool. This must be authorized by the Developer. The Core package will receive the request from Gemini and route it to the CLI package. The CLI will prompt the Developer (approve/deny). If the developer approves, Core invokes the appropriate tool to run the test command. The Tools component returns the concrete test output (stdout/stderr and exit status), and Core forwards this evidence back to the Gemini API for diagnosis.

Once the failing test is identified, the Gemini API typically requests additional context by asking Core to read relevant files—such as the failing test file and the suspected source file(s). Core invokes read operations via Tools and sends the retrieved snippets back to the Gemini API. Using the test output and code context, the Gemini API proposes a patch and requests that it be applied. Since applying a patch modifies the filesystem, Core again triggers a user approval step through the CLI package. If approved, Tools applies the patch and returns the patch status to Core, which forwards it to the Gemini API.

At this point, the workflow enters an iterative verification cycle. The Gemini API requests that tests be rerun to validate the change. Core again requests approval for the shell command, Tools rerun the tests, and Core returns the new test output to the Gemini API. If tests still fail, the system repeats the same pattern—run tests, gather additional context if needed, keep applying patches until tests pass, or the developer decides to stop. The diagram clearly illustrates that debugging is gradual

When the test results indicate success, or if the developer decides to conclude, the Gemini API generates a comprehensive summary of the changes made and the rationale behind them. This summary is then sent back to the CLI package for display. The diagram also clearly illustrates the denial path: if the developer rejects approval at any point, the tool action will not be executed, thereby ensuring a secure interaction. Often, this process yields useful alternatives, such as step-by-step instructions or a suggested patch for manual application, thereby highlighting our dedication to providing a smooth and supportive experience.

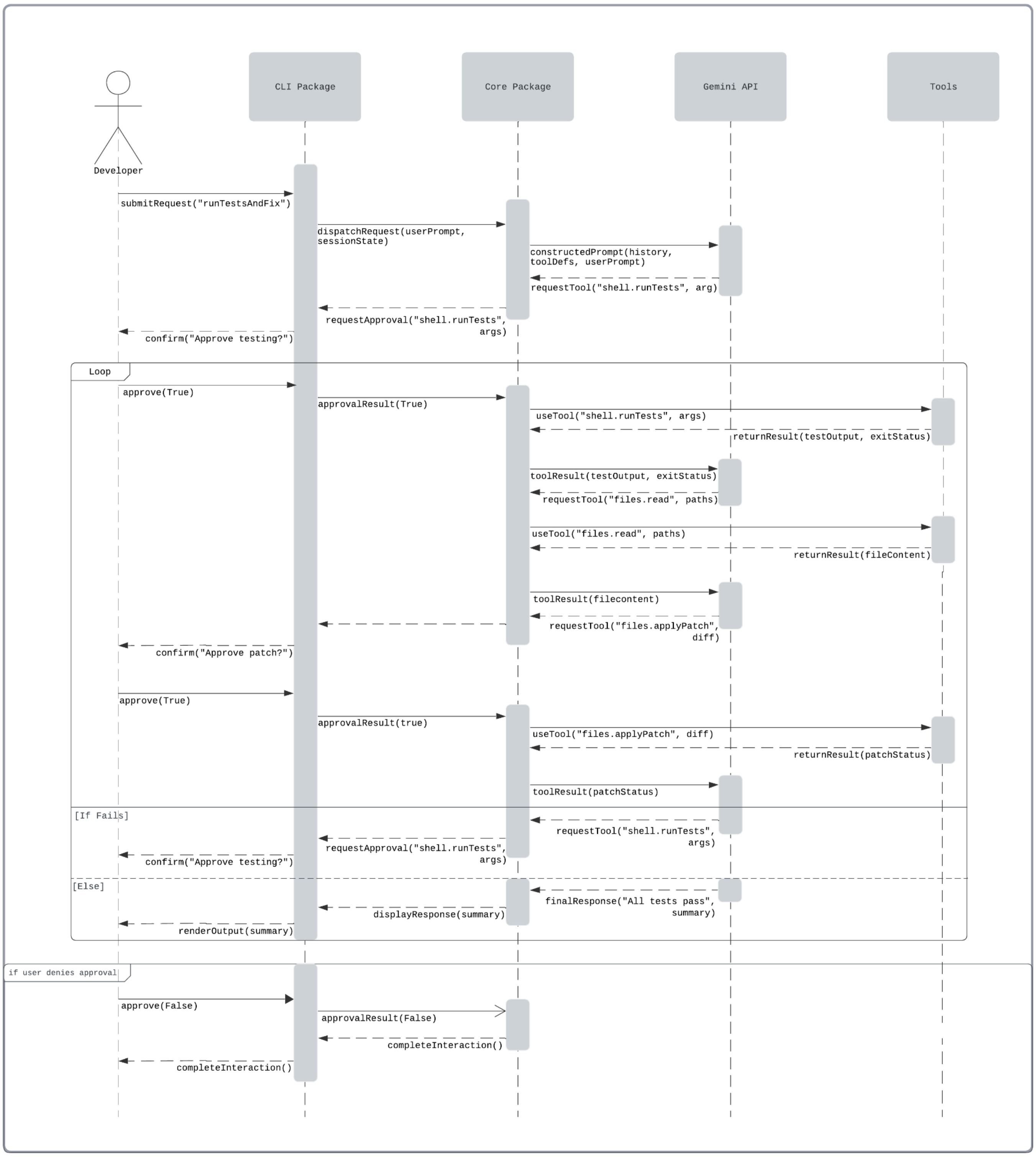

CLI Package
Core Package
Gemini API
Tools
Developer
submitRequest("runTestsAndFix")
dispatchRequest(userPrompt, sessionState)
constructedPrompt(history, toolDefs, userPrompt)
requestTool("shell.runTests", arg)
requestApproval("shell.runTests", args)
confirm("Approve testing?")
Loop
approve(True)
approvalResult(True)
useTool("shell.runTests", args)
returnResult(testOutput, exitStatus)
toolResult(testOutput, exitStatus)
requestTool("files.read", paths)
useTool("files.read", paths)
returnResult(fileContent)
toolResult(filecontent)
requestTool("files.applyPatch", diff)
confirm("Approve patch?")
approve(True)
approvalResult(true)
useTool("files.applyPatch", diff)
returnResult(patchStatus)
toolResult(patchStatus)
[If Fails]
requestTool("shell.runTests", args)
requestApproval("shell.runTests", args)
confirm("Approve testing?")
[Else]
finalResponse("All tests pass", summary)
displayResponse(summary)
renderOutput(summary)
if user denies approval
approve(False)
approvalResult(False)
completeInteraction()
completeInteraction()

Figure 5.1 sequence diagram of the first use case

5.2 Use Case 2: Create a program based on online specifications

The diagram below illustrates a use case in which a user asks the AI agent to create a file containing a program that meets the requirements or specifications listed on a website or online blog, with the URL provided.

To begin the process, the developer (user) inputs the request and URL into the CLI, which is handled by the user interaction system within the CLI package. This system forwards the request to the Prompt Management system in the Core package, which formats the request into a structured prompt for the Gemini AI, adding previous context, available tools, and message history. This expanded prompt, which provides the Gemini AI with a clearer understanding of the current development state for use, is then forwarded to the Gemini API, and a response is received.

Gemini's response requires the use of the Web Fetching Tool, which is invoked by the tools component and fetches the requirements from the provided URL. In this use case, we assume, for simplicity, that the request succeeds without interruption and that the URL contains the requested information; if the URL were malformed, the tool would return an error state, prompting the user for additional instructions or an updated URL. Similarly, if the Gemini API received insufficient instructions from the scanned data, it would return a request for more precise specifications. The tool returns the scanned data to the Core package, which forwards it again to the Gemini API.

Using the scanned web information and context provided by the Core package, the Gemini API proposes a solution that satisfies all requirements and requests to use the File System tools to create the file. Since this involves modifying files, it requires user (developer) approval, which the API requests from the Core package. The Core package reroutes this request to the CLI package, which then provides the user with information about what the AI intends to create or modify and prompts them to approve or deny. If the developer denies, the CLI package reports that state to the Core package, which then terminates the process without creating a file and returns a summary message to the user.

If the user approves the proposed file, the Core package invokes the File System tools to create the file and write the API's proposed solution to it. When that process is complete, the tools component returns the results to the Core package, from which the Gemini API generates its final summary, including the details surrounding the completed file, its capabilities, how it fulfills the requirements outlined for it, and other relevant information and advice regarding the use of the newly created program. The Core package then forwards this summary to the CLI package, which displays it to the user.

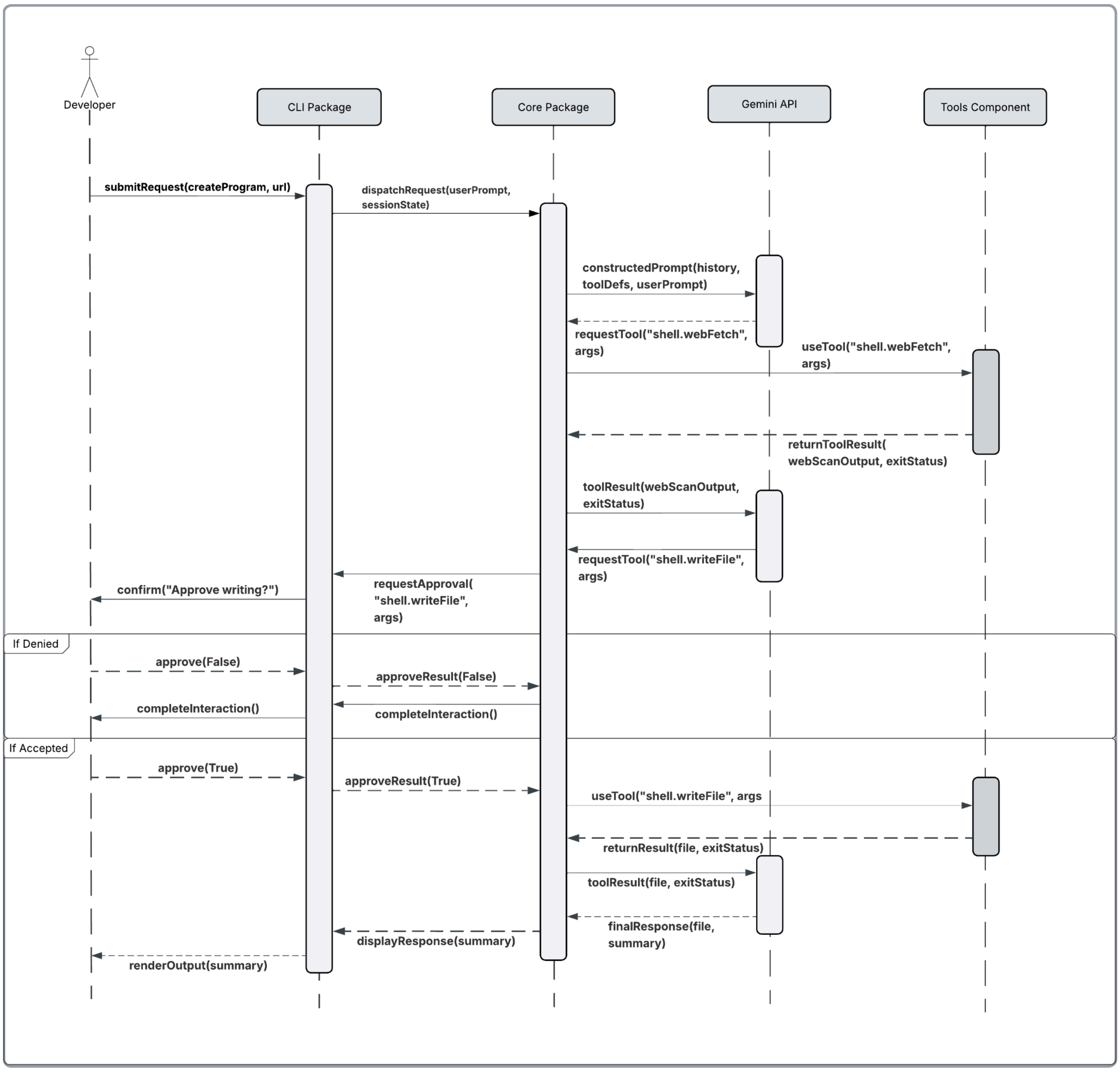

Developer
CLI Package
Core Package
Gemini API
Tools Component
submitRequest(createProgram, url)
dispatchRequest(userPrompt, sessionState)
constructedPrompt(history, toolDefs, userPrompt)
requestTool("shell.webFetch", args)
useTool("shell.webFetch", args)
returnToolResult(webScanOutput, exitStatus)
toolResult(webScanOutput, exitStatus)
requestTool("shell.writeFile", args)
requestApproval("shell.writeFile", args)
confirm("Approve writing?")
If Denied
approve(False)
approveResult(False)
completeInteraction()
completeInteraction()
If Accepted
approve(True)
approveResult(True)
useTool("shell.writeFile", args
returnResult(file, exitStatus)
toolResult(file, exitStatus)
finalResponse(file, summary)
displayResponse(summary)
renderOutput(summary)

Figure 5.2 sequence diagram of the second use case.

6. Conclusion

Gemini CLI provides access to the extensive Google Gemini open-source AI tool from the command line. It does this by working with modularity, primarily using the CLI package to manage front-end tasks, including user input, rendering, and terminal configuration. It operates in conjunction with the core package to handle back-end processes, receiving CLI requests, executing them via the Gemini API and the available tools, and leveraging the components those tools provide.

Gemini CLI operates in a client-server architecture, enhancing scalability by separating the CLI user from the Gemini back end. This design also simplifies deploying Gemini AI across multiple machines, since improvements on either side can be implemented independently. Additionally, Gemini CLI incorporates a feedback control system model in which user input, API responses, and tool use interact in iterative cycles to maintain control.

The analyzed use cases illustrate how Gemini CLI and its architecture can effectively support developers in real-world applications. They demonstrate how the CLI and core packages interact and how tools are implemented and used to ensure user safety. The Gemini CLI's architecture is shown to be carefully designed to enable seamless integration of the large language model into the user workflow.

7. Lessons Learned

Completing this report helped our group learn a great deal about the architecture of Gemini CLI. We learned the uses of a command-line interface and the benefits and drawbacks of its modular architecture.

In our research, learning to analyze documentation proved essential. We had to be able to parse through architectural information and learn how to apply it to the Gemini CLI. This would allow us to construct the conceptual architecture based on our understanding.

Overall, writing the report highlighted the importance of collaboration and the components that enable its success. We had to find effective ways to delegate tasks, manage our time, and collaborate in shared meeting spaces. We learned that working together and communicating our findings helped us grasp the concepts more effectively. Several group members also had to learn to use various drawing programs to create the diagrams in the report, which proved important for both our understanding of the architecture and for future reports, where diagrams will also be required.

8. AI (Gustapo the Rat):

For our AI Teammate, the group collectively decided to utilize Perplexity AI. This choice was driven by the platform's ability to create a shared space accessible to all team members, enabling the recording and easy retrieval of all chat threads. Additionally, the tool offers file uploads to a central repository and the ability to restrict search privileges so the AI's retrieval is limited to sources we explicitly approved for the Space, rather than arbitrary web browsing; in our case, this meant constraining it to course-provided materials and a small set of whitelisted external references we selected for relevance and consistency.

We uploaded all relevant course content and provided files, and we supplemented them with helpful resources and references—specifically: lecture slides, A1 reports from previous years, the Gemini CLI documents provided, and the following external resources: a Gemini CLI overview article on Medium, the official Gemini CLI changelogs, a Gemini CLI getting-started video, the official Gemini CLI architecture documentation, Google's

"Introducing Gemini CLI" blog post, and a conceptual-architecture reference page. This was done so the AI was constrained to the same information sources we were given as students (plus the explicitly listed additions), which is a clearer and testable constraint than claiming it "matched our style." (Gustapo was also expected to follow course norms: treat outputs as suggestions, and keep explanations consistent with the kind of justification-and-assumptions approach emphasized in class.)

Given our team's name, "Rat Pack," it was fitting to name our AI teammate "Gustapo the Rat" (he/they). Throughout this project, we used Gustapo sparingly to cultivate our own understanding; we primarily used it to clarify our final insights and validate our findings. As a limitation and ethics note, we treated all AI output as non-authoritative and verified it against our own diagrams and course materials, since LLM-based tools can produce confident but incorrect statements.

Regarding the Architectural Structure Diagrams, we asked Gustapo to review the Client-Server Architecture Diagram and provide feedback. While it identified some issues, it also made a few errors in component placement and interactions. Most notably, it briefly suggested (implicitly) that the Gemini LLM could be running on the client-side machine rather than being accessed as an external service via an API, which would change the deployment assumptions and responsibilities of the client. We used this mismatch as a checkpoint: we revalidated the intended deployment boundary (client vs. external LLM service) and revised our diagram explanations to ensure component responsibilities and interactions were consistent.

Gustapo was also used to help clarify the first Use Case. We prompted it to review the Sequence Diagram; it was strong at interpreting the flow and helped us produce a recap that improved our understanding of each step. However, it suggested moving the loop upward so that it begins before the Gemini API requests are executed when the platform is first run; based on our understanding of what the loop semantically represented in our sequence (i.e., repetition tied to later interaction rather than initial startup), we did not implement this change.

The most effective application of Gustapo was as a resource for dissecting our project and providing "autograding." To break down the assignment, we uploaded the detailed project marking scheme alongside comprehensive project details, and Gustapo parsed this into short, plain-language checkpoints (e.g., clarity/coverage/technical correctness) that were easier to follow during revision. In its role as a quality-control partner, it systematically cross-checked our draft against requirements (ranging from essential features to optional enhancements) and flagged potential omissions, thereby confirming that each rubric criterion was explicitly addressed and supported in the final submission.

9. References

1. https://medium.com/@anirudhsekar2008/gemini-cli-has-launched-and-it-changes-everything-for-developers-45bc770adfa3
2. https://geminicli.com/docs/changelogs/
3. https://www.youtube.com/watch?v=95--oibm6ac
4. https://geminicli.com/docs/architecture/
5. https://blog.google/innovation-and-ai/technology/developers-tools/introducing-gemini-cli-open-source-ai-agent/
6. https://sites.google.com/view/sixember/conceptual-architecture?authuser=0
7. https://www.geeksforgeeks.org/system-design/client-server-architecture-system-design/