CISC 322/326
A1: Conceptual Architecture Report
February 13, 2026

Conceptual Architecture of Gemini CLI

Group 1: Scott Rosenberg Fan Club

Authors

Jordan Bouckley - 22htx@queensu.ca
Andreea Cobzaru - 22hfx1@queensu.ca
Zachary Gazaille - 22jr22@queensu.ca
Callum McIntyre - 22wpd2@queensu.ca
Isaac Ouellette - 22bq31@queensu.ca
Nicholas Saloufakos - 22sst1@queensu.ca

Table of Contents

1 Abstract

Gemini CLI is a command line interface AI development tool that integrates the Google Gemini AI model. With Gemini CLI, developers are now able to directly utilize the power of AI by installing this lightweight tool that operates within their terminal and assists their development needs. A major issue surrounding the modern age of AI is that automation and system interaction is lacking often behind a separate web based interface, which heavily reduces the developers flexibility and scriptability. Gemini CLI clearly addresses these challenges by providing an AI assistant that lives inside of your terminal, which enables direct access to files, commands and workflows which were previously unreachable by the current crop of AI models.

The conceptual architecture is broken into 3 main components which are the CLI layer, Core layer and the Gemini API layer. The CLI layer handles interaction with the user, such as receiving prompts and displaying messages back to the user. This is the primary interface between the developer and the AI system, and effectively provides information, clarity and responsiveness in a sleek terminal environment. The Core layer is responsible for constructing effective, context aware prompts, tracking and executing tools and sending API requests to the Gemini API. It works as the primary logic and control center of Gemini CLI where it manages the workflow and coordinates how to incorporate the Gemini models to their fullest capability. The Gemini API layer is a remote server that will be prompted by the Gemini API client that resides in the core. The Gemini API layer will compute all of the actual AI processes and modelling and return a solution to the developers needs. These layers create a modular design that allows Gemini CLI to evolve independently across all of its components.

2 Introduction and Overview

This document provides a comprehensive architectural analysis of the Gemini CLI, a command-line interface tool designed to interact with Google's Gemini models. The purpose of this report is to deconstruct the system's design decisions, identifying the architectural styles, structural patterns and dynamic behaviours that enables it to function securely and efficiently in a local development environment. This report analyzes official documentation, showcasing a technical understanding of how Gemini CLI manages user input, processes natural language, and interacts with the Operating System and external APIs.

The scope of this architectural description includes the software components of the Gemini CLI packages and core and their interactions with the immediate external environment. The internal layering of the application, the "Human in the Loop" security mechanism, and the asynchronous data streaming from the Gemini API are included in the architecture.

Section 1 describes a system overview of Gemini CLI. It functions as an agent that resides in the user's terminal. Unlike web-based AI agents, this software is designed to complete integration tasks, read local files, execute shell commands, and learn from message history. The system utilizes a layered architecture for local component separation and a client-server architecture for Gemini intelligence offloading. This structure ensures that the execution of the Gemini CLI tools are under local control during the LLM's heavy computational logic. Section 2 outlines the system's design including architecture with an analysis of key components like the InputHandler, Prompt Constructor, and ToolManager.Section 3 defines the External Interfaces that form the boundaries of the system. It details the specific. It details the specific protocols used to communicate with the user, the Gemini API, and the local Operating System. Section 4 illustrates the system's dynamic behaviour through key use cases. It provides sequence diagrams that contrast a standard request-response interaction against a complex tool execution workflow.

Finally, this report presents the most significant conclusions derived from the architectural study. The analysis determines that Gemini CLI utilizes hybrid architecture combining a Layered local structure with a Client-Server model to balance the computational power of remote LLMs with the security requirements of a local development environment.

3 Architecture

3.1 Overall Structure

The Gemini CLI is a highly modular system with three main components that each have their own roles, but integrate with each other to provide data, permissions, and execution of instructions. These layers, from most to least abstract, are:

Layer 1: packages/cli

- **Goals and Requirements:** This layer is the user-facing presentation layer of the system. Its goal is to handle and process command-line input, manage the user-experience through theme and UI, and render the final output/display for the user. The CLI layer also manages history and handles configuration settings.

- **Interfaces and Interactions:** The CLI package interacts directly with the core package. It passes the user's inputs and commands as requests to packages/core and also receives processed data from the core package to format for the terminal. The CLI might also receive a request from the core to use a tool, and the user must accept/reject before the answer is sent back to the core. This layer never interacts directly with the Gemini API layer, and does not even know that it exists.

- **Evolvability and Testability:** By separating the system's UI from the backend logic, you can make changes to the UI or styling without breaking the AI logic. If you want to use a different terminal framework, you can do so without touching the core's code. This also allows the capability of having different frontends while retaining the same backend/core. For testing, you can test that things are being displayed properly with mock data without having to run the AI to check a visual change.

Layer 2: packages/core (The "Client")

- **Goals and Requirements:** The core layer acts as the local backend for Gemini CLI. Some of its requirements include prompt construction, conversation/session state management and the registration/execution of tools. In the Client-Server architecture style, the core also acts as the API client, managing the state before communicating with the Gemini API.

- **Interfaces and Interactions:** This layer sits between the CLI and the Gemini API. One critical interaction that occurs internally in this layer is with the Tools sub-module (packages/core/src/tools). The core interacts with the other layers by receiving the user's input from the CLI and constructing a prompt which includes conversation history and available tool definitions. It then sends the prompt to Gemini API and awaits a response. The response is either a direct response or a tool request. If it is a tool request, the core layer prepares to execute it and passes details of the tool to the CLI and awaits approval. Once approval is confirmed, the core executes its action with the necessary tool and sends the result to the Gemini API, awaiting a response. After the core receives a final response, it passes it to the CLI.

- **Evolvability and Testability:** The modular design of the Tool system sub-module of this layer ensures that new capabilities and tools can be added without having to change the API client or the CLI code. Keeping this modular also allows for easier testing. Each tool can be tested in a mock environment to ensure that it works correctly before plugging it into the main system.

Layer 3: Gemini API (The "Server")

- **Goals and Requirements:** The Gemini API is an external, remote sub-system that provides LLM capabilities. Within the Client-Server architecture style, this acts as the server. This layer's main job is to fulfill requests for complex natural language processing and to either return a generated response or tool execution instructions.
- **Interfaces and Interactions:** This layer interacts directly with the core layer. It receives the full context from the core including the user's messages, the chat history and the list of available tools. Then it processes this information, produces a response and returns it to the core layer. This layer has no idea that the CLI layer exists, it just processes data from the core and returns responses to it. For testing,
- **Evolvability and Testability:** Because this is an external service, it has great evolvability. When Google drops a new version of Gemini, the API version can simply be updated in the core packages. The rest of the application can stay the exact same, yet have better capability because of its updated Gemini API. For testing you can simply use API mocks to simulate API responses and ensure that the core and CLI layers handle this data gracefully and correctly.

The evolvability and testability of this system is inherent in its structure. By isolating the interface from the logic and the logic from the computations, developers can mock any single component to verify the others.

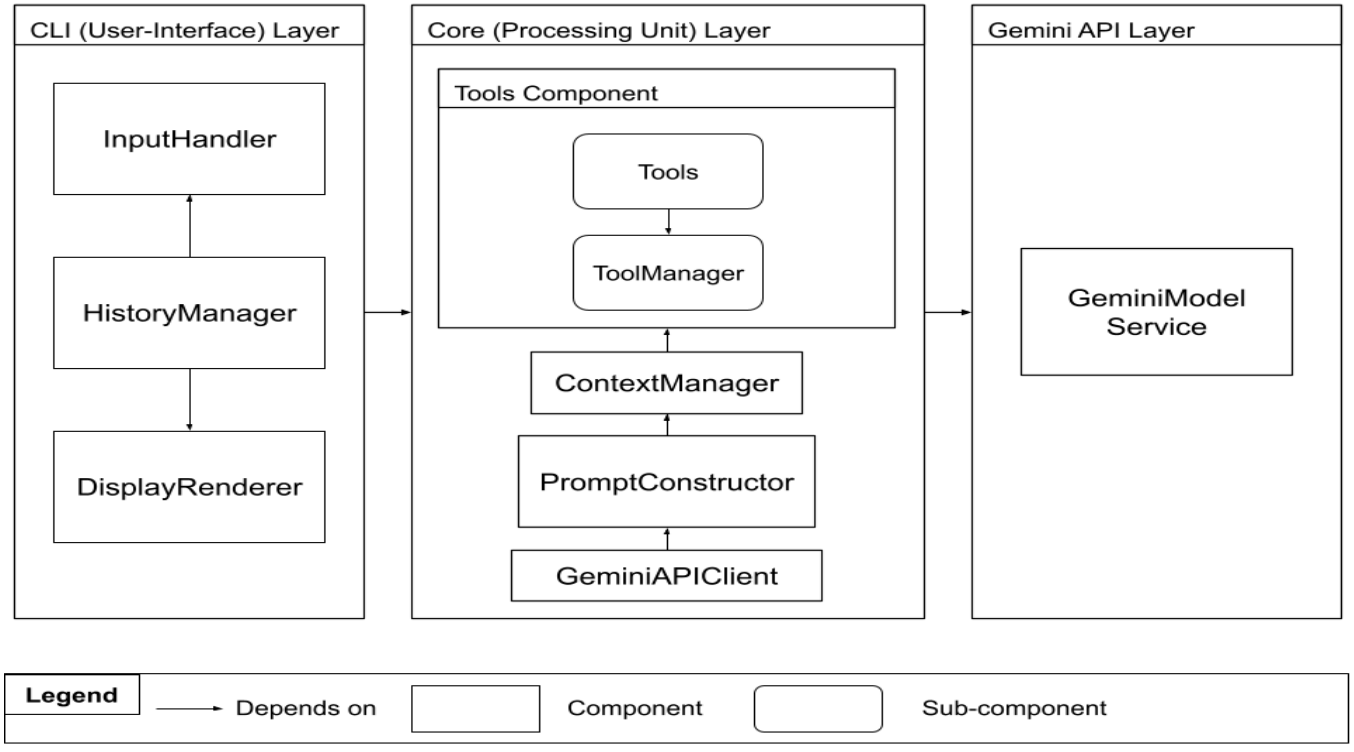


Figure 1: Shows Gemini CLI following a layered architecture within the local application and a client-server model between the core and Gemini API.

3.2 Architectural Styles

Gemini CLI can be characterized through two possible architectural lenses:

Style A: Layered Architecture

The three main components are organized hierarchially, where each layer provides a specific class of service to the layer above it, and acts as a client to the layer below. Information flows sequentially between adjacent layers through restricted interfaces, ensuring that upper layers have no visibility into the implementation details of the deeper layers.

Each component also has its own isolated role, only interacting with other layers to delegate tasks outside its own responsibility. For example, the core package may construct prompts, but never performs actual language processing or computation. Conversely, the API layer may request a tool execution, but relies on the core to actually perform tasks on the user's environment.

This modularity ensures evolvability. Since each layer is a collection of procedures with a stable interface, a developer could swap the packages/cli for a graphical interface (GUI) without needing to rewrite code in other components. This principle applies across all three components: the core logic can be changed, or the Gemini API can be swapped, all while leaving the other components completely untouched.

Style B: Client-Server Architecture

The architecture of Gemini CLI follows a request-response pattern where the system is partitioned into two distinct roles. The CLI and the core together make the “client” and the Gemini API is the “server”. In this relationship, the local application is responsible for logic such as capturing input, managing local files and rendering the UI, while the remote Gemini API acts as the brain. This ensures that any computational load is offloaded to Google’s infrastructure, allowing the client to remain lightweight on the user’s machine.

This interaction is defined by a request-response protocol over a network. When a user makes a prompt, the local client must bundle the prompt with necessary context and send it as a formal request to the Gemini API. The server then processes the request and sends back a structured response. From the user’s perspective, this process is synchronous as the client remains in a waiting state until the server responds with the necessary data or instructions.

Since the core logic of the language model resides on the server, this allows the service provider to make improvements to the models accuracy, safety and capabilities without requiring the user to update their local Gemini CLI. This separation also allows the Gemini API to maintain a state of versions and features, which ensures that any client using any interface receives constant up-to-date service.

3.3 Component Descriptions

InputHandler (CLI Layer)

The main functionality of the InputHandler is to receive and handle the input message sent by the user in the terminal.

DisplayRenderer (CLI Layer)

The DisplayRenderer is responsible for outputting messages to the user in the CLI.

HistoryManager (CLI Layer)

The HistoryManager component saves the conversation history to the user's computer whenever the user submits input or receives output. Therefore the HistoryManager is dependent on the InputHandler and DisplayRenderer. It can also be used to retrieve history when constructing a prompt.

GeminiAPIClient (Core Layer)

The GeminiAPIClient handles all communication between the Core layer and the Gemini API layer. It sends a prompt created by the PromptConstructor and returns model responses to be interpreted. Therefore, GeminiAPIClient is dependent on the PromptConstructor.

PromptConstructor (Core Layer)

The PromptConstructor is responsible for creating the final, structured prompt. It combines the user's raw input with the conversation history and the available tools ensuring that GeminiAPIClient receives a fully context-aware request. Because PromptConstructor requires this context, it is dependent on the ContextManager.

ContextManager (Core Layer)

The ContextManager gathers all the necessary context for the prompt. This includes the current conversation history and the available tools. Therefore, the ContextManager is dependent on the ToolManager.

ToolManager (Core Layer)

The ToolManager is responsible for managing the tools. It knows what tools are available, it is responsible for requesting access to use a tool and it is responsible for executing the required tools.

Tools (Core Layer)

The Tools component includes the actual tools of Gemini CLI, such as File System Tools for reading and navigating local directories, Shell Commands for executing terminal instructions and Web Tools for retrieving external sources. The tools can only be executed by the ToolManager, so Tools is dependent on ToolManager.

GeminiModelService (Gemini API Layer)

GeminiModelService is the external Gemini API and is responsible for LLM reasoning and output.

3.4 Performance Critical Parts

Gemini CLI has three primary areas where performance is critical to maintaining a responsive user experience: network latency, tool execution blocking and context window overhead.

Network Latency

The most significant bottleneck is the delay between Gemini CLI and the Gemini API server. Because Gemini CLI follows a client-server model, every user query must traverse over the network, be processed and then be returned. To prevent the user interface from appearing frozen during this delay, the

architecture implements asynchronous streaming. By using the streaming capabilities of the Gemini API, the CLI is able to incrementally output text as it is generated, which reduces the user's perceived latency.

Tool Execution Blocking

Within the Tools sub-module, certain operations such as recursive file system searches or complex shell commands are I/O bound and can be slow. In a standard synchronous flow, these would block the entire system and prevent the CLI from rendering updates. However, the system combats this by using a Non-blocking Event Driven Pattern. Tools are executed as asynchronous child processes, which ensures that the core remains responsive to CLI signals even while heavy computation is occurring in the background.

Context Window Overhead

As a conversation continues, the amount of history sent from the client to the server increases linearly. This context bloat can impact performance as the API takes longer to ingest the larger payloads. The architecture addresses this through Selective Context Pruning. This means that instead of sending a raw dump of all history data, the core package creates a summarization.

3.5 Control Flow and Concurrency

The global control flow of Gemini CLI is governed by an Asynchronous Event-Driven model. Rather than linear execution where each part of the system freezes while waiting for other parts, the architecture uses the Node.js event loop to manage non-blocking operations. Control is passed between components using requests, promises and event emissions, which ensures the user interface remains responsive even during heavy computation.

Global Delegation and the ReAct Loop

Control typically originates in the CLI when a user submits a prompt. The CLI delegates logic to the core package, which then initiates a ReAct (Reason + Act) loop, where control is passed to the Gemini API via an HTTPS request. One of the critical architectural features is the interrupted control flow required for tool execution. When the API returns instructions to use a tool, the core regains control to perform the task. If the tool requires user permission, the core pauses execution and requests confirmation from the CLI. This ensures that the core handles logic and the CLI remains ultimate authority.

Units of Concurrency

1. Asynchronous promises: Most interactions between the core and Gemini API are handled through promises. This ensures the system remains live to handle user interrupts while waiting for network responses, and preventing the UI from freezing up during high-latency calls.
2. Streaming Buffers: The system uses a producer-consumer model for text generation. As Gemini API produces chunks of data, the core sends them to the CLI for immediate rendering, which allows the user to read responses in real-time before the full payload arrives.
3. Child Processes: For shell-based tools or heavy computations, the system creates independent child processes. This isolates the long-running local tasks from the main application, ensuring that a slow command does not lock the terminal interface.

Method of Passing Control

Control is passed between layers through restricted interfaces. The CLI interacts with the core through a set of asynchronous methods, while the core interacts with the tools through a separate interface. This

ensures that each layer only handles data it is responsible for. For example, the CLI could never execute a tool by itself.

3.5 Critical Abstractions and Patterns

The intelligence of Gemini CLI relies on the ReAct pattern. Instead of a simple one-way conversation, the core acts as a middleman that lets the AI pause to use a tool, observe the result and continue thinking. This allows the system to solve multi-step problems by treating local tool outputs as new context for the AI.

To ensure safety, the system incorporates a Human-in-the-Loop pattern. This serves as a security feature that suspends the control flow and waits for an explicit confirmation from the user before the AI can perform any potentially destructive local commands.

The system also uses a Singleton pattern to centralize the API client and conversation history. By keeping a single source for the session state, the system ensures that the AI’s memory remains synchronized across multiple sessions.

3.6 Alternative Architectural Styles

Pipe and Filter Architecture

In a pipe and filter architecture, the system would be viewed as a series of independent transformations, each performed by a different component. The user's input would represent a data "stream" associated with this architecture, passing through the core component for prompt engineering, then moving to the API for tool execution and response creation.

This style improves concurrency, an area that the layered and client-server styles are weaker in. Through the pipe and filter lens, "filters" can run in parallel; specifically, the system could start streaming the beginning of a response from the API while another section of the response is still being processed or a tool is concurrently executing.

However, this style worsens the system in terms of interactivity. By the very nature of a command line interface, the user is meant to interact with the system in a back-and-forth manner, but pipe-and-filter treats the system as transformational and mostly linear. In addition, the constant parsing and unparsing required to view data as a stream is less efficient than the direct procedure calls used in the current layered style.

Interpreter Architecture

The Gemini CLI could be viewed through the lens of the interpreter style. In this model, the core acts as an execution engine that processes the user's prompt, while the Gemini API serves as the instruction set used to determine which actions to take.

This style is relevant because the CLI essentially interprets human-text commands and translates them into executable actions, sometimes even specific Bash commands through the tooling sub-module of the core layer. This follows the classic interpreter process of translating a high-level language into executable actions. Additionally, this structure shows how the interpreter-based CLI abstracts underlying LLM logic, allowing the user to interact with their system without needing to understand the complex prompt engineering or API calls happening behind the scenes.

The primary disadvantage of this architecture is the added layer of latency. Unlike a traditional local interpreter (like Python or Bash), every user input must be transmitted, tokenized, and processed by a remote system before an action can be performed. This makes perceived execution significantly slower than direct-access systems. Furthermore, because the "instruction set" (the LLM) is non-deterministic, there is a risk of the model generating unwanted actions. This requires the core layer to implement strict constraints and validation that a traditional, rule-based interpreter would not need.

4 Derivation Process

The derivation of the Gemini CLI architecture followed a structured three step process that focused on achieving a full understanding of the core functionality and use cases of the software.

The research began with reading the available documentation provided on OnQ to establish a foundational understanding of the capabilities and core components. From this step it became clear that this was a complex system with many abilities that, in order to remain organized, is divided into two main packages, the frontend CLI and backend Core. Exploring deeper into the documentation yielded information about the specific responsibilities and sub modules the CLI and Core contained, such as Input Handling for CLI and the Gemini API integration and Tool modules for Core. The documentation also offered a broader understanding into Google's key design principles for the architecture, the most important of which being modularity and extensibility. This was critical to interpreting the systems overall architecture, as it became clear that only certain architectural styles such as Layered and Client Server would be valid choices given these design goals.

Additionally, several third party sources such as articles discussing Gemini CLI and Youtube videos showcasing a demo of its functionality were also reviewed to provide further context that highlighted the many potential unique use cases. From this understanding of the potential applications the most important and frequent use cases were analyzed to determine the specific step by step interactions between components during run time. Specifically, the use cases of a user entering a prompt and a user entering a prompt that required tool execution were deemed the most noteworthy and as such were then documented with sequence diagrams. The process of creating these diagrams in itself served to deepen the group's understanding of the system's structure as it required further research to determine which components, based on the documentation, should serve as the initial set to begin creating the diagrams with.

Lastly, using the knowledge amassed during the first two steps, a box-and-arrow diagram was created using the components found in the sequence diagrams. The goal of the box-and-arrow diagram was to highlight the static dependencies between components, in particular to depict how information flows between the layers and how the client server interaction works between the Core package and the Gemini API. By focusing on overall structure this diagram provides a holistic overview and summary of the derived architecture and how the layered and client server styles combine to create Gemini CLI.

5 External Interfaces

User Interface (CLI)

The user interface is a text-based command line interface, which is the external point of interaction allowing the human user to operate the Gemini CLI.

- **Input (User → System):** Handled by InputHandler component
    - e.g. Raw text strings (prompts), Control commands (exit, clear), Confirmation signals (Y/N for permissions)

- **Output (System → User):** Handled by DisplayRenderer component.
    - e.g. Formatted Markdown responses from API, Visual indicators, Error Messages

Network Interface (HTTPS)

The network interface connects the local core layer/client to the external API layer/server. This operates over the internet using the HTTPS protocol.

- **Request sent to Server:**
    - e.g. JSON payloads (containing user prompt and context window), Tool definitions
- **Response received from server:**
    - e.g. Streamed text responses, Tool requests, Completion signals

File System Interface

This is the interface between the HistoryManager and the user's local file system, used for data persistence across sessions.

- **Input (Read from disk):**
    - e.g. Configuration files storing user preferences, previous session logs
- **Output (Write to disk)**
    - e.g. JSON or text files containing chat logs

System/Shell Interface

This is the interface between the ToolManager component in the core and the user's operating system, used to execute shell commands.

- **Command Execution**
    - e.g. Bash/shell commands like ls and git generated by tool logic
- **Execution Result**
    - e.g. standard output and standard error data returned from OS to the Core

6 Use Cases

6.1 Use Case 1

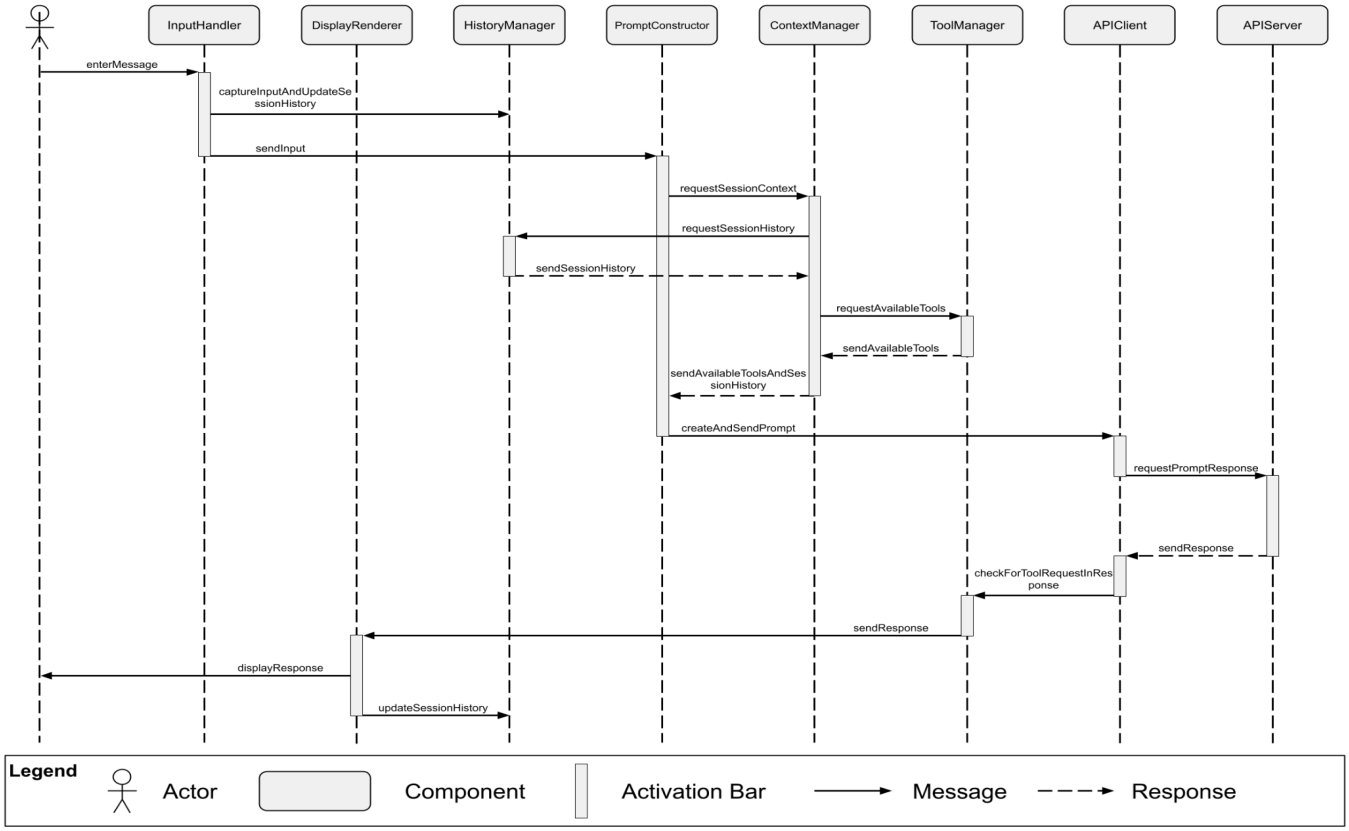
Standard User Prompt (No Tool Execution)

Phase one is the input reception sequence. When a user enters a message into the CLI, the input is sent into the InputHandler. The InputHandler will then simultaneously send off this input to HistoryManager as well as the Core, into the PromptConstructor.

Phase two is the core phase, which involves most importantly, the tools. Firstly, the PromptConstructor receives the input, and requests session context from the ContextManager. The ContextManager is responsible for giving all of the prior information and knowledge to the PromptConstructor to be eventually sent off to the API side. The ContextManager first requests the session history from the HistoryManager, which is returned to the ContextManager. Once the session history is received, the ContextManager will query the ToolManager for all of the tools that can be used, which will then list all of the tools within its database back to the ContextManager. The ContextManager compiles the session history, as well as the available tools, and finally sends this information back to the PromptConstructor. The PromptConstructor, with this information, then creates a prompt, and sends it to the APIClient.

This brings us into the API phase of the sequence. The APIClient will take the contextualized prompt, and ask the APIServer for a prompt response, which gets created by the server and returned to the client side. The APIClient then sends the response to the ToolManager, which checks for any necessary tools requested by the APIServer, but ToolManager sees that no tools are needed.

ToolManager will therefore send the response towards the DisplayRenderer. The DisplayRenderer will take the response, and make it human readable, then finally will bring the response back to the User, and update the session history, thus completing the sequence.



6.2 Use Case 2

User Prompt with Tool Execution

The second use case provides more context as to what happens when we do use tools, and how they affect the process. Tools essentially make up a large majority of what Gemini CLI can do. The beginning is largely the same; The InputHandler sends to the Core to collect all of the tools and context via the ContextManager, which sends into the API to generate a response, as well as the required tools depending on the response. From the API tool request, this is where we start to see some differences in the sequence.

Firstly, the ToolManager, after receiving the information of what tools are required, sends a permission request to the DisplayRenderer, since in most cases Gemini CLI cannot use any tools without explicit permission from the user. The DisplayRenderer takes this request and prompts the user to ask if they give permission to use the required tools.

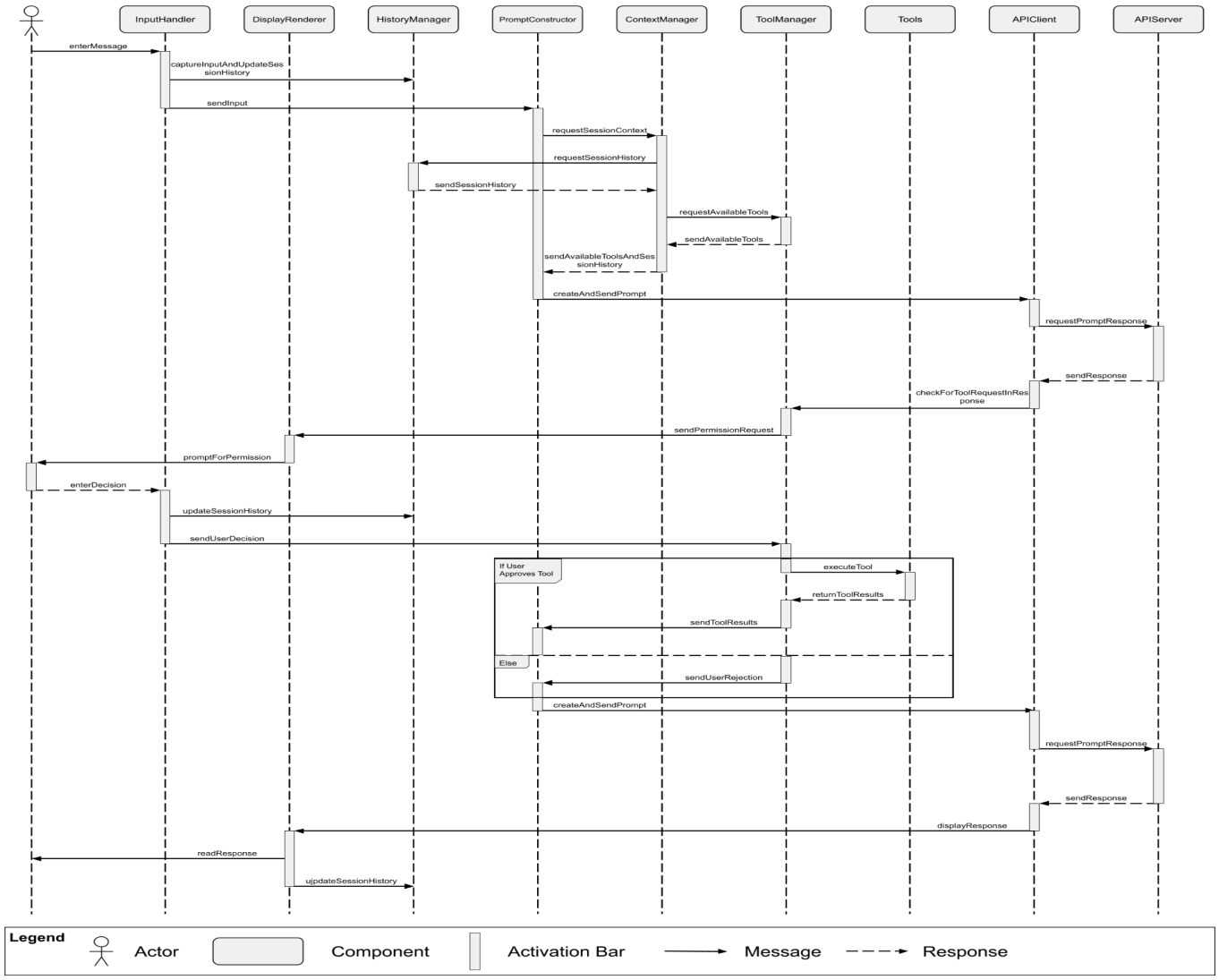
The user gives their response, a yes or no. Regardless of the response, the InputHandler once again will take the response, bring it to the HistoryManager for it to be added to the SessionData for

contextualization purposes. The InputHandler also sends the response towards the ToolManager for execution or to inform it that the user did not give permission.

The process then enters an If statement. If the user does provide permission, the ToolManager will access and execute the required Tool. The Tool will provide the response after the tool execution to the ToolManager. Once the manager has the results, it will give it to the PromptManager to eventually be sent over to the API side. If the user does not give permission, the ToolManager will directly send the rejection over to the PromptManager, without ever executing the necessary tool, to be sent over to the API side.

The PromptManager takes the prompt from the ToolManager, and creates and sends a prompt over to the APIClient. The rest is similar to the user prompt use case without the tool execution, since in this case the tool execution has been completed.

The API requests a response prompt to the server, which returns a response. The response gets sent to the DisplayManager, which processes the response and sends it to the user, and finally updates the session history to complete the sequence.



7 Data Dictionary

This glossary defines all key terms used in this architectural report.

Term	Definition
Child Process	An independent operating system process spawned by another process, used to execute shell commands or heavy computations in parallel, ensuring the main CLI application remains responsive.
Context Window	The maximum amount of text (tokens) that the Gemini API can process in a single request.
Event-Driven Model	A global control flow pattern used by the system, allowing the application to handle multiple operations without freezing the main execution thread.
Human-in-the-loop	A safety design pattern utilized in the ToolManager where the system pauses execution to require explicit human confirmation before performing sensitive actions (e.g., file deletion or shell execution).
Non-deterministic	A property of Large Language Models (LLMs) where the same input can produce different outputs.
Promises	A unit of concurrency used to handle asynchronous operations.
Prompt Engineering	The process performed by the PromptConstructor to format user input, history, and system instructions into a structured payload that optimizes the model's performance.
ReAct Pattern (Reason + Act)	A logic flow used by the Core layer where the LLM generates a thought, decides to use a specific tool, observes the output of that tool, and then continues its reasoning process.
Singleton Pattern	An architectural pattern ensuring a class has only one instance, ensuring a unified session state across the entire application lifecycle.
Streaming	A data transfer method used in the Network Interface. Instead of waiting for the full response to be generated, the API sends data in small chunks, allowing the DisplayRenderer to print text to the terminal in real-time to reduce perceived latency.
Token	Smaller units that raw text is broken down into that the LLM can process. It is the unit of "cost" and load for the API.
Tool	A specific function or script (e.g., read a file or run a command) that the AI model can request to execute. These bridge the gap between the text-based model and the operating system.

8 Naming Conventions

Classes/Components (PascalCase)

Major architectural components are named using PascalCase.

- Examples: InputHandler, GeminiAPIClient, HistoryManager

Packages and Directories/Layers (kebab-case)

Top-level packages and directory names that represent layers use lowercase letters. While the examples in this report are single words, standard convention dictates that multiple words in directory names are separated by hyphens (kebab-case).

- Examples: packages/cli, packages/core, src/tools

Functions in Use Case Diagrams (camelCase)

Functions, specifically those depicted in the Use Case and Sequence diagrams to represent actions or behaviors, are named using camelCase. These typically begin with a verb to indicate an operation.

- Examples: sendResponse, updateSessionHistory

Abbreviations

Abbreviation	Stands For
API	Application Programming Interface
CLI	Command Line Interface
JSON	JavaScript Object Notation
LLM	Large Language Model
UI	User Interface

9 Conclusions

The architectural analysis of the Gemini CLI demonstrates that the selected hybrid of Layered and Client-Server styles successfully meets the project’s critical requirements: modularity, responsiveness, and safety.

The architecture satisfies the requirement for responsiveness through its Event-Driven control flow. By handling processes asynchronously, the system prevents the UI from freezing during heavy computations by the API layer. The requirement for safety is met through the "Human-in-the-Loop" pattern implemented by the ToolManager, enforcing a strict permission boundary between the non-deterministic AI model and the user's operating system. Finally, the requirement for modularity is achieved by the inherent isolation of tool definitions from core logic from heavy computation from user interaction, allowing the system to support a wide range of capabilities that is easily scalable.

By strictly separating the CLI from the Core, the system achieves a high degree of evolvability. As detailed in the architectural definition, the user interface could theoretically be swapped for another graphical user interface without requiring any changes to the logic in the core. Furthermore, leaving the heavy computations to the Gemini API allows the client application to remain lightweight and capable of performing on the user's machine.

However, the analysis also highlighted some trade-offs with the hybrid Layered/Client-Server model. The reliance on a remote server introduces network delays, which necessitates an introduction of complexity through asynchronous streaming to maintain a smooth user experience.

10 Lessons Learnt

The Necessity of Strict Interfaces

While researching the different layers of the Gemini CLI, we realized that maintaining strict boundaries between the CLI, Core, and API layers was critical for efficient operation. We learned that this separation does not just enable scalability and adaptability, but it also actively prevents the blurring of responsibilities between layers, which would otherwise introduce unnecessary complexity and make the system fragile.

Asynchronous Requirements add Complexity

While the Event-Driven model is necessary for performance, it significantly increased the complexity of the control flow and sequence diagrams. In particular, the ReAct Loop was difficult to depict because the program state pauses and resumes multiple times (e.g. waiting for API, waiting for User Permission, waiting for Tool Execution). In the future, we would invest more time in creating detailed diagrams to map these asynchronous states in greater depth.

Safety Must Be a Priority

We initially viewed the "Human-in-the-Loop" pattern as a feature, but quickly realized it is a fundamental architectural requirement. Because LLMs are non-deterministic, allowing them to execute shell commands or perform other actions on a user's computer without a strict permission layer raises significant security risks. We learned that security mechanisms must be implemented into the ToolManager at the architectural level, rather than applied merely as a surface-level feature in the UI.

Visualization Driven Discovery

We found that we could not fully understand the individual components and interactions between them until we actually created the sequence diagrams. The process of visualizing the data flow revealed critical missing dependencies in our initial high-level model, forcing us to extend our architecture early on to include specific subcomponents like the ContextManager and HistoryManager that were previously overlooked.

11 References

AI TL;DR. (2025, December 5). *Gemini CLI - The command line agent from Google.* [Video]. YouTube. http://www.youtube.com/watch?v=QzJufbGhTeI

Alateras, J. (2025, July 8). *Unpacking the Gemini CLI: A high-level architectural overview*. Medium. https://medium.com/@jalateras/unpacking-the-gemini-cli-a-high-level-architectural-overview-99212f6780e7

Gemini CLI Architecture Overview. (n.d.). Gemini CLI. https://geminicli.com/docs/architecture/

Gemini CLI. (n.d.). Gemini CLI. https://geminicli.com/docs/cli/

Gemini CLI core. (n.d.). Gemini CLI. https://geminicli.com/docs/core/

Gemini CLI tools. (n.d.). Gemini CLI. https://geminicli.com/docs/tools/

Google. (n.d.). *CLI commands*. Gemini CLI. https://geminicli.com/docs/cli/commands/#built-in-commands

Hassan, A. E., & Adams, B. (n.d.). *Module 03: Architecture styles* [PowerPoint slides]. onQ. https://onq.queensu.ca/d2l/le/content/1132545/viewContent/6630152/View

12 AI Collaboration Report

12.1 Overview

Name: Google Gemini, **Version:** Gemini 3 Pro

Selection Process & Justification: Our team conducted a comparative analysis of leading LLMs, specifically OpenAI's GPT-4, Anthropic's Claude 4.5 and Google's Gemini 3 Pro. While all three possess strong capabilities, we decided that Gemini 3 Pro was the model we would use for three main reasons:

1. Since our project is the development of a Gemini CLI, using Gemini as our AI teammate provided a unique advantage. The model possesses the most up-to-date knowledge of its own API, documentation and parameters.
2. Google Gemini offers a free plan for students, which means that we had unlimited access to Gemini's Pro models. This allowed us to leverage the superior reasoning capabilities of the Pro model for complex architectural analysis.
3. We chose Gemini because of its large context window (1 million+ tokens). While other models can often forget earlier instructions when conversations become long, Gemini 3 Pro can maintain a perfect memory of our entire project history, including the full assignment rubric and all previous architectural decisions.

12.2 Tasks Assigned to the AI Teammate

- **Rubric Compliance:** The AI Teammate acted as a Rubric Compliance Officer. It reviewed our diagrams and excerpts against the criteria of the assignment. It identified specific missing elements in our assignment, such as the lack of a legend in our diagrams. This task was suitable for the AI because of its context window capability. It was able to hold the entire rubric in memory, allowing it to spot inconsistencies that a human reviewer might miss.
- **Architectural Analysis**: We tasked the AI Teammate with analyzing our system's structural design (specifically the separation between the CLI, Core logic, and API interactions). This helped us when determining which architectural styles our system used. This was an appropriate task for the AI Teammate because it possesses extensive knowledge of formal software engineering theory.
- **Technical Articulation:** We tasked the AI teammate with articulating the specific technical rationale for our key project decisions. We provided the facts and constraints and the AI formed these into formal narratives for the report. This was an appropriate task for the AI Teammate because the human team owned the decisions (the "what" and the "why") while the AI helped with articulation (the "how").

12.3 Interaction Protocol and Prompting Strategy

Methodology & Workflow: Our group used a collaborative "driver-navigator" approach. While the group collectively defined objectives and constraints, we had a designated "driver" that managed the actual AI interactions while the rest of us acted as "navigators". This helped us to maintain a consistent context window. Our workflow for interacting with the AI consisted of first defining the objective, then assembling the context, executing the prompt, and finally reviewing it as a group. We strictly treated the

AI as a drafting assistant, verifying all outputs against course materials.

Prompting Strategy & Iteration: Initial outputs from prompts were often too generic. To correct this, we refined prompts by adding specific constraints such as “limit words to 200” or “format as a table”. We also injected missing context rather than restarting the chat, slowly refining the responses until we arrived at an answer we liked.

Example Complex Prompt: “Act as a Senior Software Architect. Analyze the following system structure: We have a packages/cli folder for UI, packages/core for backend logic, and an external AI API. The CLI captures input but never communicates directly with the API, it must pass data through the Core. Based on this dependency restriction, classify the possible architectural styles for this system based on the previous context we have provided you. Produce the output as a table that lists reasons the system might follow an architectural style, and reasons it might not.”

12.4 Validation and Quality Control Procedures

Our validation process involved strict cross-referencing rather than passive review. We verified the AI’s architectural claims by cross-referencing them against the official Gemini CLI documentation. For example, when the AI identified a “Layered” style, we confirmed that this matched the system’s documented design guidelines to ensure accuracy. We ensured that every feature it described was actually present in the official docs.

12.5 Quantitative Contribution to Final Deliverable

We estimate that the AI contributed approximately 25% to the final deliverable. This figure represents the “intellectual labor” of refining and verifying our work, as well as aiding in our architectural analysis, rather than the generation of original content.

12.6 Reflection on Human-AI Team Dynamics

Integrating a virtual AI teammate into our group helped us to work efficiently, and make well thought-out decisions. Rather than eliminating collaboration, the AI teammate streamlined our workload, improving our pace and the structure of our workflow.

Our AI teammate generally saved time, mostly during the early stages of our report. It made an impact on our brainstorming, and decision-making, giving ideas for alternative architectural views, terminology and diagram structures. Although our time was saved through quick brainstorming of architectural comparisons, it also created new work. We often needed to refine prompts, verify its accuracy from explanations, and redesign our diagrams to better fit the Gemini CLI architecture and the rubric expectations. Overall, our AI teammate played an important role in the early stages of our report.

We encountered some challenges with aligning our AI teammate’s suggestions with our own understanding of software architecture. Specifically, the AI would sometimes hallucinate features that didn’t exist, which forced us to go back to our research to see why things weren’t lining up.

Throughout this process, our team learned that effective collaboration with an AI requires never taking the AI as a “source of truth”, but rather as a tool for critique and refinement. We learned that the most effective workflow is not to ask the AI to create from scratch, but rather providing it with context and asking it to audit or challenge our logic.