CISC 322 Assignment 1

A Report on the Conceptual Architectural Design of the Gemini CLI

February 13, 2026

Authors:

Jacob Badali
20jasb1@queensu.ca

Zhu Yuan
18zy11@queensu.ca

Chloe Atherton
20cga3@queensu.ca

Will Pritchard
21whp2@queensu.ca

Julian Tiqui
21jtt4@queensu.ca

Connor Rewa
20car8@queensu.ca

Abstract

Gemini CLI is a developer-focused command-line tool that integrates Google's Gemini models with controlled access to local tools such as the file system, shell, web resources, and external Model Context Protocol (MCP) servers. Conceptually, the system follows a three-tier layered architecture that separates presentation, orchestration, and execution concerns, enabling both interactive and headless usage while maintaining clear subsystem boundaries. The design emphasizes extensibility through a registry-based tool system, portability via Node.js abstractions, and centralized safety enforcement for system-modifying actions.

This report presents a conceptual analysis of Gemini CLI's architecture, its major subsystems, and their interactions. It examines how the CLI, Core, and Tools subsystems collaborate to process user requests, invoke Gemini models, and execute tool-based actions efficiently and safely. The report evaluates the system's strengths in modularity, responsiveness, and extensibility, while also considering challenges related to asynchronous coordination and consistent policy enforcement.

Finally, through representative use cases, this report illustrates how Gemini CLI's layered design supports real-world developer workflows. Overall, the analysis highlights how the architecture enables maintainability, testability, and independent subsystem evolution while balancing power and safety.

Contents

1 Introduction and Overview

Gemini CLI is a command-line system that combines large language model (LLM) capabilities with developer tooling to enable AI-assisted programming, debugging, and automation directly from the terminal. Unlike traditional chat-based AI interfaces, Gemini CLI can reason about a user's environment and interact with it through structured tools, including file access, shell execution, code search, web requests, and external integrations via the Model Context Protocol (MCP).

Conceptually, Gemini CLI follows a three-tier layered architecture. The CLI Subsystem forms the presentation layer, handling user input and rendering terminal output using React and Ink. The Core Subsystem serves as the orchestration layer, managing communication with the Gemini API, maintaining conversational state, coordinating tool execution, and enforcing safety policies. The Tools Subsystem constitutes the execution layer, providing concrete capabilities for interacting with the local environment. Dependencies flow unidirectionally from the CLI to the Core to the Tools Subsystem, enabling independent evolution of each layer.

A key architectural strength of Gemini CLI is its extensibility. Tools are dynamically registered with the Core, and MCP servers allow third-party capabilities to be integrated without modifying core system logic. Concurrency is managed through asynchronous, event-driven patterns: the CLI remains responsive while the Core orchestrates parallel API calls and tool executions, and the Tools Subsystem isolates shell commands in child processes.

Safety is centrally enforced within the Core through a policy-based model that distinguishes between read-only and system-modifying actions, requiring explicit user approval for potentially destructive operations. This ensures consistent behavior across both interactive and headless modes.

This report analyzes Gemini CLI's conceptual architecture, describing its layered structure, major components, and their interactions. The Architecture section details each subsystem and key design principles. The External Interfaces section outlines how Gemini CLI communicates with users and external systems. The Use Cases section demonstrates how the architecture supports practical developer workflows through sequence diagrams. The report concludes with key findings and lessons learned, highlighting the strengths and limitations of Gemini CLI's design.

2 Architecture

This section provides an overview of the conceptual architecture and major components of the Gemini CLI. After discussing general architectural details, specific components and their interactions are explored.

2.1 Derivation Methodology

The architectural design was derived through a structured process beginning with a domain analysis and culminating in a component-based structure. The process commenced with a thorough review of project documentation, including the Gemini API guidelines and the provided reference architecture on onQ, to establish foundational requirements. Following this, we formulated key use cases, such as debugging and refactoring, and modeled them using sequence diagrams to identify necessary system interactions. Finally, these dynamic interaction models were synthesized into the static box-and-arrow diagram presented in Figure 1, mapping identified responsibilities to concrete subsystems.

2.2 Alternative Architectures

Before finalizing the layered architecture, alternative styles were considered. A Monolithic Architecture was evaluated for its simplicity in deployment but rejected because it couples presentation and logic, limiting the ability to support both interactive and headless modes effectively. Conversely, a Microservices Architecture was considered for its scalability but deemed overly complex for a local CLI tool, as it introduces unnecessary network overhead and deployment challenges. The chosen Layered Architecture strikes the optimal balance, offering the separation of concerns needed for modularity without the infrastructure overhead of distributed services.

2.3 Overall Architecture Principles

The Gemini CLI system employs a three-tier layered organization that separates presentation concerns, business logic, and execution capabilities into distinct architectural boundaries. This structural approach enhances modularity and maintainability by establishing clear responsibilities for each tier while supporting the system's requirements for extensibility, cross-platform compatibility, and responsive user interaction. Although implemented as a monorepo containing packages/cli and packages/core, the logical architecture defines three conceptual layers whose boundaries do not directly correspond to physical package organization. The layers and external systems can be seen in Figure 1, below.

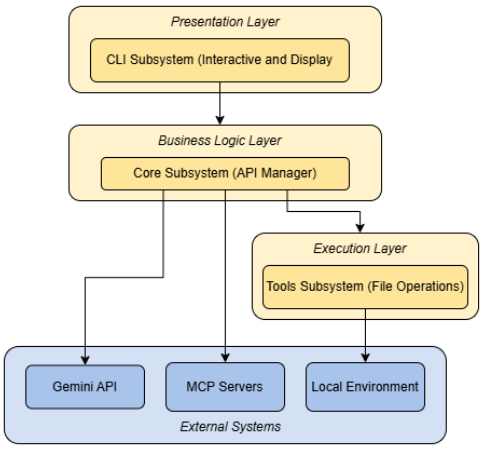

Figure 1: High-level conceptual diagram showing three layer design.

2.3.1 Layered Organization

The architectural foundation consists of the Execution Layer, implemented through the Tools Subsystem that manages all local environment interactions [6]. This layer abstracts file operations, shell command execution, web requests, and code search behind a uniform interface, isolating platform-specific logic to achieve portability across Windows, macOS, and Linux. Child processes handle shell execution and Model Context Protocol (MCP) server communication, preventing blocking operations from degrading overall system responsiveness.

The Business Logic Layer occupies the middle tier, orchestrating interactions between the presentation interface and execution capabilities [6]. The Core Subsystem within this layer coordinates API communication with Gemini models, constructs prompts enriched with conversation context, maintains a dynamic tool registry for capability discovery, manages application state, and integrates external MCP servers. Asynchronous operation patterns enable parallel tool execution while the registry-based design supports runtime extensibility without core logic modifications.

The Presentation Layer provides user-facing interfaces through the CLI Subsystem, which renders terminal output using React and Ink components [3, 2]. This layer handles both interactive mode for real-time user engagement and headless mode for automation scenarios requiring structured JSON output. The single-threaded event loop maintains UI responsiveness while awaiting asynchronous operations from lower layers.

Dependencies flow unidirectionally downward: the CLI depends on the Core, which depends on the Tools layer, enabling independent evolution of each tier. It is critical to distinguish dependency structure from communication flow—while dependencies are strictly unidirectional, communication is inherently bidirectional. Requests propagate downward through the layers while responses return upward through well-defined interfaces that mirror typical frontend-backend relationships.

2.3.2 Design Principles and Quality Attributes

The three-tier organization embodies several architectural principles that address both functional and non-functional requirements. Separation of concerns isolates presentation logic, orchestration responsibilities, and execution mechanisms into distinct layers, simplifying independent development, testing, and modification of each subsystem.

The system achieves extensibility and portability through registry-based tool management, MCP integration for third-party extensions, and Node.js abstractions that encapsulate platform-specific logic. These architectural choices directly drive high-tier quality attributes: performance is optimized via asynchronous execution, token caching, and native binaries like ripgrep; testability and maintainability are anchored by a modular design and a test suite exceeding 113,000 lines; and security is strictly enforced at the orchestration layer through policy-based restrictions and mandatory user confirmation for all destructive operations.

2.3.3 Architectural Styles

The Gemini CLI uses a heterogeneous architecture that integrates multiple Garlan and Shaw styles. Structurally, the system follows a Layered style, where there is a separation between the user interface, the core orchestration engine, and the tools subsystem [5]. This separation isolates presentation concerns from business logic. This structural separation also enables a Client-Server interaction style. The CLI functions as a frontend client that requests orchestration services, while the Core acts as the backend server managing state and execution [5]. The system also utilizes an implicit invocation architecture style through a Message Bus where components don't call each other directly; instead, they publish typed messages and subscribe to messages they care about. This can be seen in Figure 2, below.

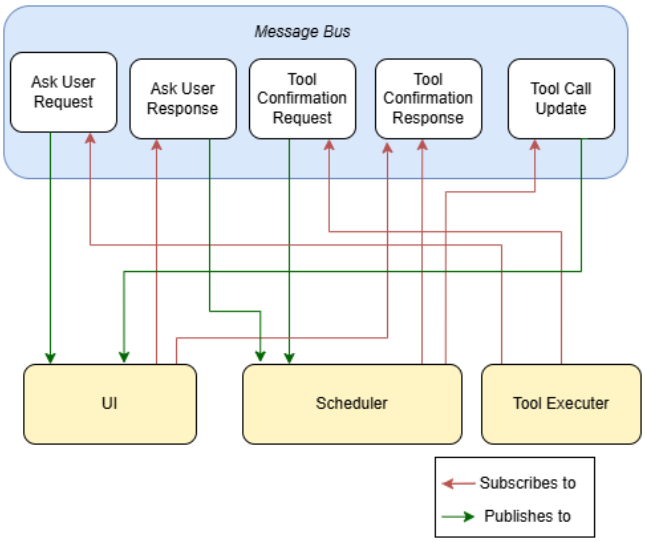

Figure 2: Publisher Subscriber visual representation

The UI listens for Tool Call Update and Ask User Request, the Scheduler listens for Tool Confirmation Request and Ask User Response, and the Tool Executor publishes requests [6]. While these interaction patterns are essential to system behavior, the layered organization remains the primary architectural style because it defines the fundamental structural decomposition and dependency constraints across the entire system. The client-server and implicit invocation styles operate within the layered boundaries, facilitating communication between layers rather than replacing the layered structure itself. This can be seen in Figure 3, below.

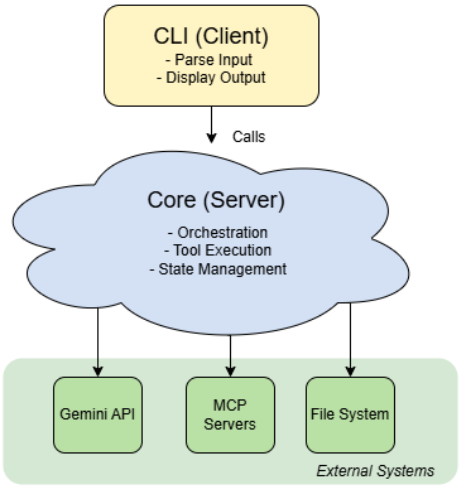

Figure 3: Conceptual Model of the Gemini CLI Subsystems

2.4 Analysis of Major Components

2.4.1 CLI Subsystem

The CLI Subsystem forms the presentation layer, acting as a thin client that mediates user interaction while delegating Logic to the Core [1]. It handles input processing and renders system responses using React and Ink, isolating presentation structure from orchestration logic [2, 3]. Supporting both interactive and headless modes, the CLI utilizes Node.js's event loop to maintain responsiveness, forwarding requests asynchronously to the Core while updating the UI incrementally to prevent blocking during long-duration operations [4]. By depending only on stable Core interfaces, the CLI achieves independent evolvability, allowing presentation changes without affecting backend logic.

2.4.2 Core Subsystem

The Core Subsystem serves as the central orchestrator, managing API communication, state, and tool execution [1]. It constructs prompts by enriching user input with conversation history and tool context, ensuring consistent interaction regardless of changes to underlying models or routing strategies. The Core maintains a dynamic tool registry that allows capabilities to be discovered and invoked by the Gemini API, while enforcing safety policies that distinguish between read-only and state-modifying actions [1]. By coordinating asynchronous operations and streaming responses, the Core ensures efficient parallel execution and manages session continuity across both interactive and headless modes [4]. This centralized orchestration decouples user interaction from execution mechanics, simplifying testing and maintenance.

2.4.3 Tools Subsystem

The Tools Subsystem provides concrete execution capabilities for file operations, shell commands, and web requests, isolating the AI model from direct environment access [6]. Executed exclusively under Core coordination, tools run in isolated child processes to prevent blocking the main event loop and to contain potential failures [4]. This design enforces safety policies by distinguishing between read-only inspection and modifying actions, requiring user confirmation for the latter [1]. Extensibility is supported through a registry-based pattern and MCP integration, allowing new tools to be added without modifying the core system structure. Performance is optimized via streaming I/O and native binaries for efficient code search.

2.4.4 Extended Components

Beyond the CLI-based workflow, Gemini CLI supports optional extended components including the Agent-to-Agent (A2A) server and IDE integrations. These act as alternative clients to the Core, reusing orchestration, safety, and tool execution mechanisms while maintaining consistent architectural patterns across implementations [1].

2.5 Interplay of Layers and Components

The interaction among subsystems follows a well-defined flow orchestrated through asynchronous message-passing [6]. A typical user session begins when the CLI captures input and transmits it to the Core, which enriches the request with conversation history and constructs prompts for the Gemini API. Based on the API response, the Core may invoke tools to perform operations like file access or shell execution. If the requested tool requires system modifications, the Core solicits explicit approval from the CLI before proceeding.

The layered organization (described in Section 2.3.1 Layered Organization and Section 2.4 Analysis of Major Components) ensures clear separation: the CLI Entry Layer handles user interface and interaction, the Core Engine orchestrates AI interaction and tool execution scheduling, and the Tool System performs file operations, system command execution, network requests, and extension tool support [6]. This separation maintains distinct responsibilities across the three architectural components.

Integration between layers employs asynchronous communication through async function calls, child processes, and HTTP connections [6]. Requests flow downward (CLI $\rightarrow$ Core $\rightarrow$ Tools/API), while responses propagate upward, with the Core mediating all interactions. This message-passing architecture ensures each module operates within its defined responsibilities, with components interacting through callbacks, promises, and event emitters rather than direct synchronous invocations. The design enables parallel tool execution where dependencies permit while maintaining responsiveness throughout the request lifecycle.

3 External Interface

The external environment that the Gemini CLI interacts with includes users, development environments, model services, tools, and collaborative software platforms. These interfaces define what information is exchanged at a given boundary, ensuring that the software agent can function as an intelligent development assistant within a specific environment. Since architectural analysis focuses on the information exchanged at the system boundary and its effects, rather than the specific interface implementation, the External Interfaces section should emphasize the content, direction, and function of information exchange between external entities and the system, rather than the interface presentation itself, thus more accurately reflecting the system's conceptual architectural structure.

3.1 User Command Interface

In Gemini CLI, the Command Line Interface (CLI) refers to the interaction between the user and the system. Here, the user simply enters text/code, presses Enter, and the system analyzes the input, executes it, and returns the result. This is a text-based command-line interface, and feedback can be text, execution results, error results, etc. Therefore, users can request more information based on the feedback, modify their requests accordingly, and continue interacting with the system.

3.2 Local Development Environment Interface

Between the Gemini CLI and the user's local computer development environment, there exists an information exchange interface called the Local Development Environment Interface (LDI). The user command interface serves as the interaction channel between the user and the system, while LDI acts as the data channel between the system and the local project environment. Through this channel, the system retrieves relevant data from the local development environment, enabling it to analyze and execute commands based on context. Output information flows from the system to the local environment; after analyzing and executing commands, the

system writes and updates the information to the local environment, thus achieving bidirectional information exchange.

In short, LDI enables the system to access and manipulate project data to fulfill user requests, rather than simply providing suggestive output.

3.3 Cloud Model Service Interface

The Cloud Model Service Interface (CMI) refers to the information exchange interface between the Gemini CLI and the remote model inference service. Essentially, it separates the CLI from the remote model inference capability, allowing the system to operate lightweightly while utilizing cloud server computing power. In this interface, the CLI provides input information to the cloud model service, including but not limited to user-inputted commands and context information, enabling the model to produce corresponding outputs based on the current commands. On the output side, the cloud model service outputs the results based on user commands to the CLI. Because of the CMI, the CLI itself does not need to run on a large model and can complete tasks by calling remote agents over the network, thus separating the local interaction layer from the remote intelligent inference layer, ensuring advanced inference capabilities while maintaining a lightweight design.

3.4 MCP Interface

The MCP, which stands for Model Context Protocol, is a bridge that connects our system to outside tools. This is where the interaction occurs, where the system is talking to other tools with the rules implemented by the MCP. It "leaves the system," which means it defines which tool should be used, which parameters should be applied for the task, and which context is currently being applied. Once the tool finishes, the return from the tool is then sent back and processed in a format that allows for the creation of the final answer for the prompt that was originally given.

If viewed from an architectural perspective, the MCP interface is the unified way in which the model can call the tools. This is important because the model is able to execute the inferences that have been made in a more tangible way, giving the system the ability to tap into a number of external services without necessarily sacrificing the basic role that the model plays in terms of inference.

3.5 Collaboration Interface

The Collaboration Interface essentially acts as a bridge that establishes the Gemini CLI with the team's collaborative development environment so that the team can share information with one another. Here, the system is able to access any data pertinent to the development of the project within the collaborative environment. It essentially develops an awareness of what the team's collaboration is looking like so it can make suggestions or carry out any relevant activities.

On the other hand, it can push information back to the collaboration platform, which includes, among other things, proposed code changes, ideas for code modification, and code review details, thus enabling it to contribute to code maintenance and improvement through the collaborative process.

From an architectural standpoint, the Collaboration Interface integrates the system into the collaboration environment of the team. This way, the system can access common information for projects and implement useful services in collaboration. This would make it possible to expand the isolated application for lone-wolf development into a robust collaboration development assistant for team-collaborative development.

3.6 Authentication Interface

The Authentication Interface is like a gate to a system; it's like a way of determining who is worthy of access to the external services or system. Simply put, the purpose of the Authentication Interface is to ensure that anyone is notallowed to get access to the core services of the system.

Typically, it involves sending out an identification request in the form of an external entity—an access token, an API key, and sometimes login credentials—and receiving the answer, which is acceptance or refusal, depending on the outcome of the identification process.

From the architectural perspective, the Authentication Interface acts as a bridge between the internal system and external services. On one side, it facilitates safe access to cloud-based services, tool interfaces, and other collaborative tools. On another side, it represents access control policies to secure the system from unauthorized entry and data misuses.

4 Use Cases

This section demonstrates the practical application of the Gemini CLI architecture through developer scenarios. By mapping specific user workflows to the system's core components described in Section 2 Architecture, it is illustrated how the CLI orchestrates reasoning, tool execution, and safety protocols to solve complex problems.

4.1 Use Case 1: Debug a Failing Command

Description:
A developer encounters a runtime error when executing a command in their project and asks Gemini CLI to diagnose and fix the issue. Gemini reproduces the failure, inspects relevant files and error output, applies a targeted fix, and validates the solution by re-running the command. As a specific example, the developer enters the prompt, *"This script crashes with a null reference error, debug and fix it."* This use case shows how Gemini CLI combines reasoning with tool execution and sandboxed validation to safely debug code.

Main Flow:

1. The CLI Subsystem parses the user's debugging request (e.g., "This script crashes with a null reference error, debug and fix it") and forwards it to the Core Subsystem.
2. The Core Subsystem loads session context, assembles relevant tool context (available tools + constraints), and constructs the initial prompt.
3. API Call #1 is sent from the Core Subsystem to the Gemini API. Gemini analyzes the request and determines the failure must be reproduced, returning a tool request to run the relevant command.
4. The Core Subsystem routes the execution request to the Tools Subsystem, which runs the command in the sandboxed environment and returns results (stdout, stderr, exit code).
5. The Core Subsystem updates the conversation history with the execution output and sends API Call #2 to the Gemini API so Gemini can analyze the failure and form a diagnosis.
6. Gemini requests additional evidence (e.g., source files/config) via read tools. The Core Subsystem invokes the Tools Subsystem (e.g., read_file) and receives file contents.
7. The Core Subsystem appends file contents to the conversation history and sends another Gemini API call (continuation of analysis) so Gemini can confirm root cause and propose a targeted fix.
8. Gemini requests a modifying action (e.g., edit_file) to apply the patch. The Core Subsystem enforces modification controls and, if user approval is required, sends a proposed diff/summary to the CLI Subsystem to present for confirmation.
9. After user approval, the CLI Subsystem returns the approval decision to the Core Subsystem, which executes the modification via the Tools Subsystem.
10. The Core Subsystem validates the fix by invoking the Tools Subsystem to re-run the original command and captures the new output/result.
11. The Core Subsystem sends API Call #3 to the Gemini API with the validation results, and Gemini returns a final explanation and summary.

12. The Core Subsystem returns the final response to the CLI Subsystem, which displays the "bug fixed + summary of changes" to the user.

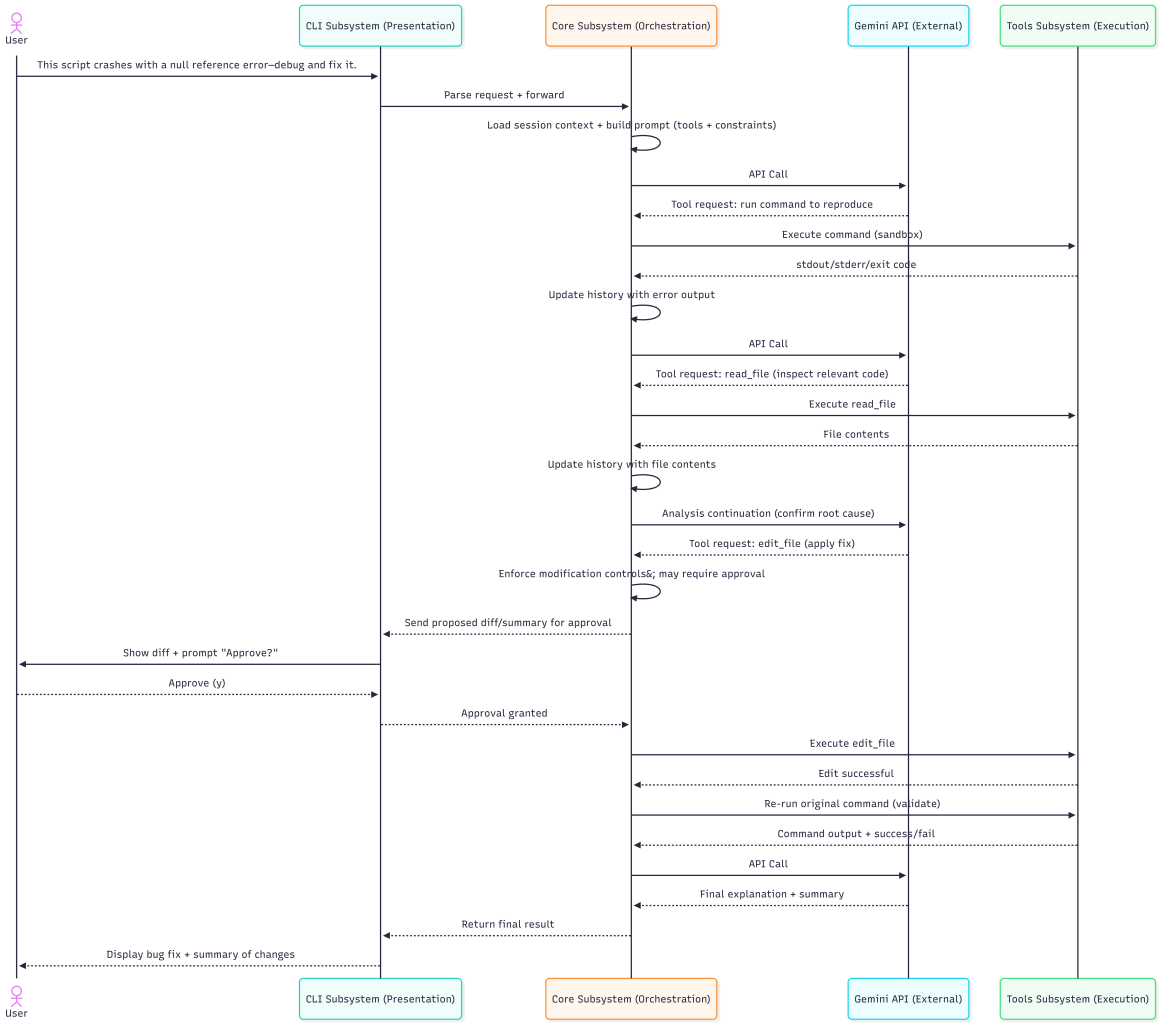

Figure 4: Sequence diagram for debugging a failing command

4.2 Use Case 2: Refactor Code

A developer requests a refactoring change on their code base through the CLI, Gemini implements this change directly within the codebase and returns a summary of the results. As a specific example, the developer enters the prompt, "*Refactor this function from promise-based .then() chains to modern async/await syntax*". This use case demonstrates how the core components orchestrate a multi-turn conversation with the Gemini API to accomplish this safely.

Main Flow:

1. The CLI Subsystem parses the user's refactoring request and forwards it to the Core Subsystem, which loads session context and constructs a prompt using available tool context (tool registry/capabilities).
2. API Call #1 is sent from the Core Subsystem to the Gemini API, providing the request plus the available tool options.

3. Gemini responds with a tool invocation request to inspect the code (e.g., read_file). The Core Subsystem routes this to the Tools Subsystem, which reads the requested file(s) and returns their contents.
4. The Core Subsystem updates the conversation history with the retrieved file contents and sends API Call #2 to the Gemini API, enabling Gemini to analyze the code in context and determine the needed refactor.
5. Gemini proposes specific changes and requests a modifying tool (e.g., edit_file) to apply the refactor.
6. The Core Subsystem enforces policy/controls for the modifying action and, if approval is required, sends the proposed diff/summary back to the CLI Subsystem to present to the user for confirmation.
7. After the user approves, the CLI Subsystem returns the approval decision to the Core Subsystem, which then executes edit_file via the Tools Subsystem and receives confirmation that the edit succeeded.
8. The Core Subsystem sends API Call #3 to the Gemini API with the edit outcome (and any relevant tool results), allowing Gemini to generate a final explanation and summary.
9. The Core Subsystem returns the final response to the CLI Subsystem, which displays the completion summary to the user.

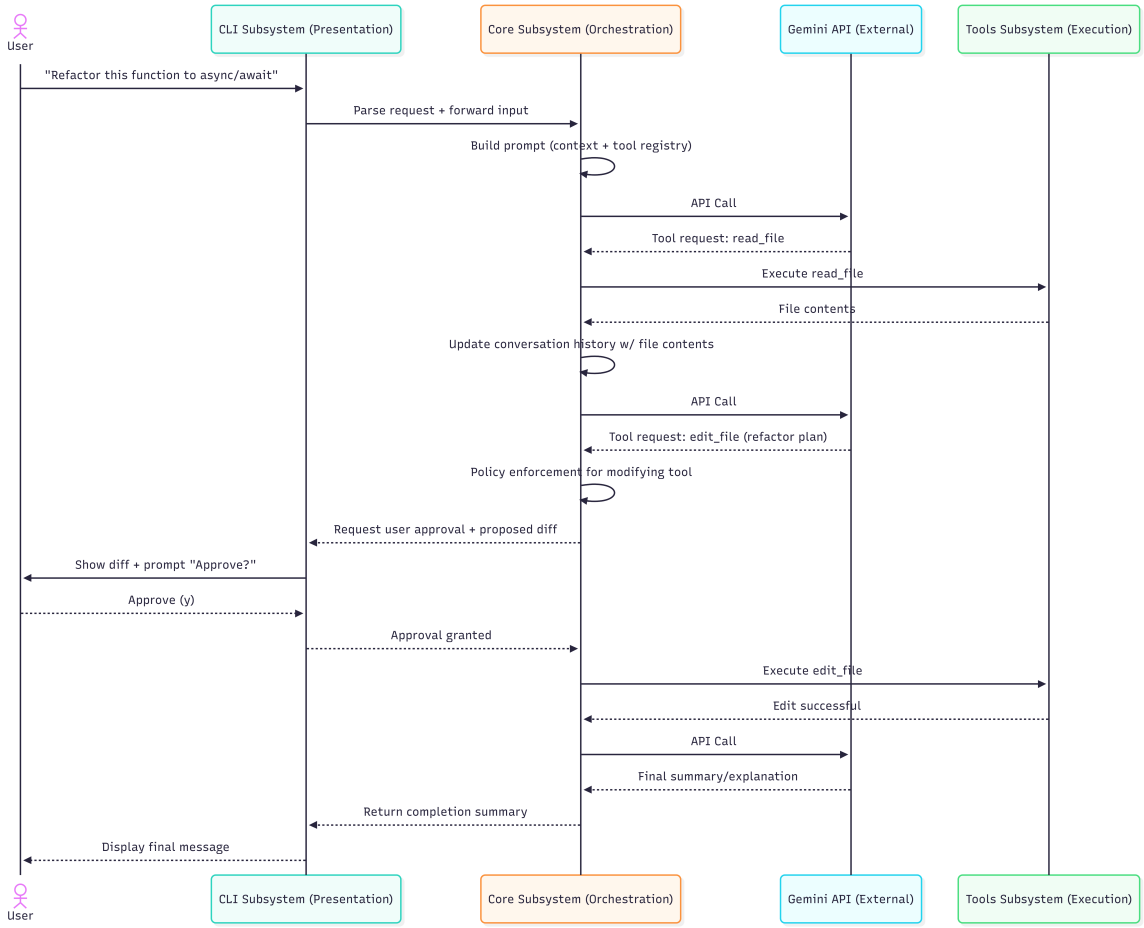

Figure 5: Sequence diagram for refactoring code

5 Conclusion

This report has presented a conceptual architectural analysis of Gemini CLI, examining how its layered design supports AI-assisted software development through controlled tool execution and seamless integration with Google's Gemini models. By structuring the system into three primary subsystems—the CLI (presentation), Core (orchestration), and Tools (execution)—Gemini CLI achieves a clear separation of concerns that enhances modularity, maintainability, and extensibility. The unidirectional dependency flow from the CLI to the Core to the Tools Subsystem enables each layer to evolve independently, reducing coupling while preserving coherent system behavior.

A key strength of Gemini CLI's architecture is its extensibility, achieved through a registry-based tool system and support for external Model Context Protocol (MCP) servers. This allows new capabilities to be integrated without requiring modifications to core orchestration logic or presentation components. Also, the system's asynchronous, event-driven design supports responsiveness and scalability, enabling parallel tool execution and streaming outputs while maintaining an interactive user experience.

Safety and control emerge as central architectural priorities. By locating policy enforcement within the Core subsystem and distinguishing between read-only and state-modifying operations, Gemini CLI ensures consistent and predictable behavior across both interactive and headless modes. This centralized approach mitigates risks associated with AI-driven tool execution while preserving developer flexibility.

However, the architecture also presents challenges. Coordinating multiple asynchronous processes—including API calls, tool execution, and MCP communication—introduces complexity that requires careful error handling and observability. Furthermore, as the ecosystem of tools and integrations grows, maintaining clear boundaries between layers and preventing architectural drift will be critical to sustaining long-term maintainability.

Overall, Gemini CLI demonstrates a robust and thoughtfully structured architecture that effectively balances power, safety, and usability. Its layered design provides a strong foundation for future enhancements, including deeper IDE integration, expanded tool ecosystems, and improved performance optimizations. The architectural principles analyzed in this report offer valuable insights into how AI-driven developer tools can be designed to be both flexible and reliable in real-world software engineering workflows.

6 Lessons Learned

Through analyzing the Gemini CLI's architecture, our team gained a deeper understanding of layered software design, separation of concerns, and the application of web technologies in terminal environments. One key takeaway was the effectiveness of the three-tier organization in isolating responsibilities. The abstraction provided by the Core Subsystem demonstrated how orchestration can be decoupled from user interface concerns, effectively allowing for dual mode operation without duplicating logic. While this design enhances testability and maintainability, it also highlighted the complexity involved in coordinating asynchronous event loops across distinct layers to ensure system responsiveness.

Another significant lesson was maintaining a balance between extensibility and security in applications. The integration of the MCP illustrated how a system can support third party extensions and runtime capability discovery without requiring modifications to the core codebase. However, this flexibility reinforced the architectural necessity of strict policy enforcement. We observed how the Policy Engine in the orchestration layer is critical when integrating non-deterministic models with local file system access. This emphasized the importance of implementing design patterns that include human decision to prevent unintended side effects in autonomous tools.

This project reinforced the need for clearly mapping asynchronous flows and component interactions. Given the Gemini CLI's reliance on Node.js single threaded event loops and streaming responses, understanding the exact sequence of data propagation required careful analysis. We found that mapping out the interaction between the CLI's React components and the Core's orchestration logic helped clarify how the system maintains UI responsiveness during long-running tool executions. Studying the Ink library provided insight

into how declarative UI paradigms, typically reserved for web development, can be effectively applied to terminal applications to ensure that presentation state remains distinct from business logic.

7 Data Dictionary

CLI Subsystem: The presentation layer of the Gemini CLI, responsible for capturing user input, rendering the terminal interface, and managing the event loop.

- **Core Subsystem:** The business logic and orchestration layer that coordinates API communication, state management, and tool execution requests between the CLI and the Tools Subsystem.
- **Tools Subsystem:** The execution layer of the architecture, containing the concrete implementations for file system operations, shell command execution, and network requests.
- **Ink:** A React-based library used within the CLI Subsystem to render component-based user interfaces in the terminal window.
- **React:** A declarative UI framework used to structure the CLI's presentation layer, enabling a component-based architecture similar to modern web development.
- **Headless Mode:** An operational state where interactive UI elements are suppressed to output structured JSON, facilitating automation and scripting workflows.
- **Policy Engine:** A safety mechanism within the Core Subsystem that enforces access controls and requires user confirmation for potentially destructive operations.
- **Tool Registry:** A dynamic directory within the Core Subsystem that manages the discovery and availability of tools for the AI model to invoke.
- **Orchestration:** The process managed by the Core Subsystem to coordinate the flow of data between the user, the Gemini API, and local execution tools.
- **Monorepo:** A version control strategy used to house both the packages/cli and packages/core within a single repository to streamline development.
- **Sandbox Runtime:** An isolated execution environment described in the Use Cases, used to safely run shell commands and tools during debugging sessions.

8 Naming Conventions

- **packages/**: The directory prefix used within the monorepo to physically separate the architectural layers, specifically distinguishing packages/cli (Presentation Layer) from packages/core (Business Logic Layer).
- **MCP (Model Context Protocol):** The standard acronym and protocol identifier used for integrating external tools and servers that extend the system's capabilities.
- **A2A (Agent-to-Agent):** An acronym used in the Extended Components section to denote server interfaces designed for automated, non-human client interactions.
- **CLI (Command Line Interface):** In this architecture, specifically refers to the user-facing client (packages/cli) rather than the entire system.
- **API (Application Programming Interface):** A set of rules and protocols for building and interacting with software applications.
- **JSON (JavaScript Object Notation):** A lightweight data-interchange format.
- **IDE (Integrated Development Environment):** Software for building applications that combines common developer tools into a single graphical user interface.

- **UI (User Interface):** The space where interactions between humans and machines occur.
- **LLM (Large Language Model):** A type of AI model capable of understanding and generating human language.

9 AI Collaboration Report

9.1 AI Member Profile and Selection Process

For this deliverable, our team worked with two AI tools: OpenAI GPT-4 class model (ChatGPT Plus) and Anthropic Claude (via Antigravity IDE integration). We selected these tools because this assignment required large-scale conceptual summarization, architectural abstraction, diagram generation (Mermaid), and structured technical writing. GPT-4 was used primarily for structured drafting, diagram syntax generation, and script refinement. Claude was used for repository summarization and architectural explanation within the IDE workflow.

9.2 Tasks Assigned to the AI Teammate

AI was used to summarize documentation and repository structure by condensing Gemini CLI docs and GitHub organization into conceptual explanations, leveraging language-model pattern recognition. It also drafted initial architectural prose, generating structured explanations of layered design, architectural styles, quality attributes, and subsystem responsibilities, which humans revised for accuracy and tone. Additionally, AI generated properly formatted Mermaid sequence diagram syntax from written use cases (Debugging and Refactoring), reducing syntax errors and formatting overhead. Finally, it helped shorten and simplify presentation scripts to meet timing constraints while consolidating key information.

9.3 Interaction Protocol and Prompting Strategy

Our group used a semi-centralized prompting workflow where multiple members interacted with AI tools, but outputs were shared and reviewed collectively before inclusion. Our prompting strategy evolved over time; early prompts were too broad, so we improved output quality by providing explicit constraints (e.g., "conceptual architecture only"), supplying documentation excerpts, requesting specific output formats, and iteratively refining drafts for clarity and tone. An example of a refined prompt is: "Write a debugging use case that matches this refactoring format using CLI, Core, Tools, Policy Engine, and Display." This iteration was essential to align AI output with assignment requirements and maintain consistent terminology.

9.4 Validation and Quality Control Procedure

We implemented strict quality control by cross-referencing all architectural claims against the official Gemini CLI documentation and GitHub repository. We verified the presence of packages/cli and packages/core in the repository before discussion, and confirmed descriptions of MCP, policy enforcement, and tool registration using primary sources. AI-generated architectural interpretations were compared with Dev.to and Medium articles cited in our references. We manually reviewed all sequence diagrams to ensure message flow alignment with our described architecture and edited all AI-generated text for clarity and consistency with our layered model.

9.5 Quantitative Contribution to Final Deliverable

We estimate that AI contributed approximately 30% of the final submitted deliverable, primarily through drafting support and structural refinement. This includes roughly 10% for architectural clarification (helping articulate the layered model and styles while we retained final decisions), 12–15% for prose drafting and refinement (improving explanations of components and use cases), 7–8% for Mermaid diagram syntax (formatting assistance), and 3–5% for script refinement (condensing content).

9.6 Reflection on Human–AI Team Dynamics

Integrating AI generally saved time, especially for drafting and formatting tasks like diagram syntax and script refinement, though it created additional work in prompt engineering and validation. AI supported brainstorming by offering alternative phrasings and structural suggestions, helping us explore organization options, but final decisions about subsystem boundaries and architectural style were always formed through team discussion. While it did not create major disagreements, alignment was required on how much we trusted AI-generated content versus rigorous verification. Overall, we learned that effective AI collaboration requires clear task delegation, precise prompting, and strict validation, and we plan to apply this structured, human-led approach in future projects.

References

[1] Google AI, “Gemini API Documentation,” https://ai.google.dev/gemini-api/docs/gemini-cli.

[2] Vadim Demedes, “Ink Documentation,” https://github.com/vadimdemedes/ink#why-ink.

[3] Meta, “Thinking in React,” https://react.dev/learn/thinking-in-react.

[4] OpenJS Foundation, “The Node.js Event Loop,” https://nodejs.org/en/learn/asynchronous-work/event-loop.

[5] John Alateras, “Unpacking the Gemini CLI: A High-Level Architectural Overview,” Medium, https://medium.com/@jalateras/unpacking-the-gemini-cli-a-high-level-architectural-overview-99212f6780e7.

[6] Cezar Milo, “Gemini CLI Project Architecture Analysis,” Dev.to, https://dev.to/czmilo/gemini-cli-project-architecture-analysis-3onn.