CMPE 322: Assignment 1

Google Gemini CLI Conceptual Structure
CMPE 322: Software Architecture
Queen's University

Presented to: Dr. Bram Adams

Keven Li, 21zl47@queensu.ca, 20322014
Dylan Spilberg, 20dms3@queensu.ca, 20266708
Omar Afify, 20oamz@queensu.ca, 20287159
Muhammad Ibrahim, 20mi17@queens.ca, 20273315
Rowan Mohammed 20233462
Zonghan Li 20321365
AI agent: Google Gemini 3 Pro

Submitted on Feb 13, 2026

Statement of Originality
Following professional engineering practice, we bear the burden of proof for original work. We have read the Policy on Academic Integrity posted on the Faculty of Engineering and Applied Science website (http://engineering.queensu.ca/policy/Honesty.html) and confirm that this work is in accordance with the Policy.

Abstract

Gemini CLI is an open-source AI-driven command-line assistant that brings the capabilities of Google's Gemini models directly into a developer's terminal. This report presents a high-level conceptual architecture of the Gemini CLI, focusing on its purpose, structure, and key design goals. We begin by outlining the CLI's scope and target audience, primarily software developers and engineers who want to leverage AI in a terminal-first workflow.

The Gemini CLI enables users to perform complex development tasks through natural language, such as reading and modifying code, running build or test commands, and fetching web information, all while preserving project context. To achieve this, the system is architected as a modular monorepo comprising two primary packages (a front-end CLI interface and a back-end core engine) plus an extensible tool subsystem. We describe how these components interact to handle user requests, orchestrate AI model calls, and safely execute actions in the local environment.

The report emphasizes Gemini CLI's design principles, clear separation of concerns, extensibility through a rich tool ecosystem, and a focus on user control and experience, which together enable powerful yet trustworthy AI-assisted development workflows. All findings are derived from official documentation and developer commentary, aiming to help newcomers quickly understand the Gemini CLI's conceptual structure and goals.

Introduction and Overview

The purpose of the Gemini CLI is to provide developers with a terminal-first AI assistant that can integrate into their software development workflow. Unlike traditional code editors or cloud-based AI tools, Gemini CLI runs directly in the terminal, allowing users to query and manipulate codebases, generate content, and automate tasks using natural language commands. This report focuses on the conceptual architecture of Gemini CLI, the high-level structure of its system components and their interactions, rather than low-level implementation or source code details. By examining design documentation and commentary from the Gemini CLI project, we recover the abstract architecture that underpins its functionality. The scope of this report includes the major subsystems, their responsibilities, and the design rationale behind this structure, as relevant to a newcomer trying to understand how the system is organized.

Gemini CLI is primarily intended for software professionals who frequently work in the command-line environment, such as developers, DevOps engineers, and SREs (Site Reliability Engineers). These users benefit from an AI agent that can understand natural language requests and perform development tasks on their behalf, from running tests to refactoring code. For example, a developer can ask, "Explain this error" or "Find and fix all TODOs in the project", and Gemini CLI will intelligently handle the request. Because it is open-source and extensible, Gemini CLI also appeals to technically advanced users who may contribute plugins or adapt it to custom workflows. In the context of this report, the target audience is anyone new to the Gemini CLI project (e.g. new team members or external collaborators) who needs a high-level understanding of what the system does and how it is structured. We assume the reader has basic familiarity with command-line tools and AI assistants, but not necessarily any prior knowledge of Gemini CLI itself.

The conceptual architecture presented here was derived by studying Gemini CLI's official documentation, developer discussions, and an architectural overview published by a contributor. We analyzed how the system is described in terms of components and behavior, without referring to the actual source code. Specifically, we reviewed the high-level *Architecture* guide in the documentation, which outlines the system's core components and interaction flow. We also incorporated insights from a Medium article, *"Unpacking the Gemini CLI: A High-Level Architectural Overview,"* which provides commentary on design decisions and the internal workings of the CLI. By correlating these sources, we identified the main architectural elements of Gemini CLI and their relationships. This process ensures that our conceptual model reflects the intended design as described by the creators, rather than an implementation-specific view. The resulting architecture can be visualized as a set of interacting subsystems (CLI front-end, core back-end, and tool modules) cooperating to handle user requests. In the next sections, we provide an overview of the system's functionality and break down each major subsystem's role in achieving Gemini CLI's goals.

The Gemini CLI architecture is built upon a sophisticated, decoupled framework that establishes a clear boundary between user-facing interactions and the underlying computational logic. This design follows a classic division between a frontend interface and a backend processing engine, which is further enhanced by a modular, plugin-based tool system. By separating these concerns, the system ensures that the complexities of AI reasoning and execution remain isolated from the presentation layer, allowing for a more stable and maintainable codebase. This high-level separation is the foundation upon which the system's scalability and technical integrity are built.

A central element of this architectural strategy is the adoption of a monorepo structure, where both the CLI and the core engine are housed and maintained within a single repository. This deliberate choice allows the development team to version and evolves both components in tandem, ensuring strict type compatibility and synchronized releases. In practice, this means that features requiring updates to both the user interface and the core logic can be implemented through atomic commits, effectively preventing integration drift. Furthermore, the monorepo simplifies dependency management and testing by providing a unified build process and shared interface definitions, which significantly improves developer velocity and system reliability.

Conceptually, the system is organized into three primary functional pillars: the CLI interface, the core engine, and an extensible set of tools. The CLI interface acts as the frontend package, managing all direct terminal interactions and rendering output for the user. The core engine serves as the backend, orchestrating communication with Gemini AI models and managing the business logic required to fulfill user requests. Supporting these two is a suite of tools that extend the AI's capabilities, providing it with the means to interact with the host's file system, access web resources, and perform other auxiliary operations. This modularity allows the system to bridge the gap between abstract reasoning and practical execution.

The operational lifecycle of the Gemini CLI follows a logical flow where user input is captured by the interface and passed to the core engine for processing. The engine then determines if specific tools are required to complete the task; if so, it invokes them and integrates the results

before communicating back with the AI model. Finally, the processed data flows back to the CLI for display to the end user. This circular data flow, supported by a rigorous separation of concerns, ensures that the system is not only easier to debug but also agile enough to adapt to the rapid advancements in generative AI technology.

System Functionality and Goals

Gemini CLI is an AI-powered assistant for software development that runs directly in the command line. It supports natural-language workflows like inspecting and editing code, running builds and tests, searching documentation or the web, and maintaining project context across a session. It uses Google's Gemini large language models to interpret user intent and decide what to do next. Unlike a traditional chat interface, Gemini CLI can act on the local environment by creating or modifying files and executing shell commands, with explicit user approval required for system-changing actions. By connecting developer intent to low-level developer tools, it offloads repetitive or time-consuming work and helps developers stay focused on higher-level problem solving.

The architecture is guided by four goals: responsiveness, safety, extensibility, and user experience. Responsiveness comes from separating the interactive CLI from heavier AI and file-processing work, allowing output to stream while longer tasks run in the background. Safety is enforced through confirmation gates before any operation that writes files or executes commands. Extensibility is enabled through a modular design that supports adding new tools and third-party extensions without reworking the system. Finally, a React-based terminal UI provides a modern interactive interface with structured, readable output. To meet these goals, Gemini CLI is organized into three major subsystems: a CLI front-end, a core back-end, and an integrated tool system managed by the core. Each component has a clear responsibility:

The CLI Package

The CLI package is the user-facing terminal interface. It captures user requests, parses input, renders formatted responses, maintains session history, and applies user preferences like theme and verbosity. Built with Ink (a React-based CLI framework), it supports a modern interactive experience while keeping business logic minimal. The CLI primarily forwards requests to the core and displays the results, which also makes it feasible to reuse the same backend for other interfaces such as GUIs or IDE integrations.

The Core Package

The core package (@google/gemini-cli-core) is the orchestration engine that translates user requests into model prompts and tool executions. It builds enriched prompts for the Gemini API using relevant context like code snippets, conversation state, and the set of available tools. It then interprets the model output, decides whether a tool must be invoked, and coordinates execution. The core manages conversation state, authentication, caching, and security boundaries, and it enforces confirmation before any system-modifying operation. Keeping this logic centralized allows consistent behavior across frontends while maintaining safe execution policies.

The Tool System

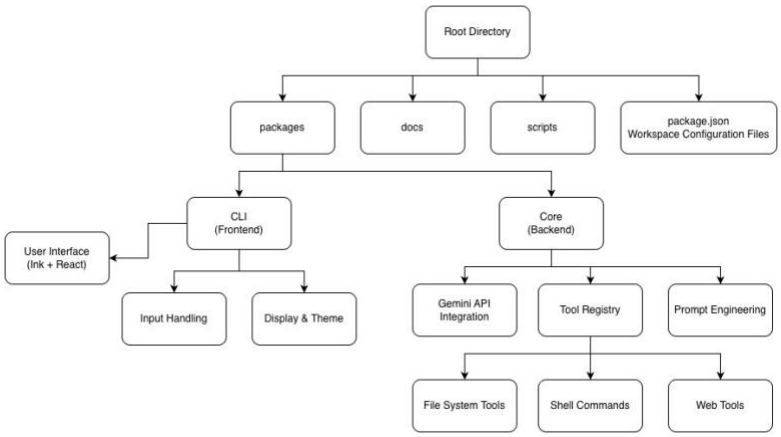
The tool system defines how the AI interacts with the environment in a controlled way. Tools are modular plugins that provide capabilities such as file reads and writes, shell command execution, web search and URL fetching, and limited memory functions. These tools live within the core and are invoked only when requested by the AI model. The core acts as a gatekeeper: read-only tools can run automatically, while side-effect tools require explicit approval. This structure keeps the assistant powerful while ensuring users retain control over changes to their machine or codebase.

A typical Gemini CLI session follows a consistent flow across these subsystems. The CLI captures a request and passes it to the core, which adds relevant project context and sends the prompt to the Gemini API. The model may answer directly or request a tool (for example, searching the repository). The core runs the tool, incorporates the results back into the model context, and produces a final response. If the requested action could modify the system (such as editing files), the core pauses for confirmation before proceeding. The CLI then renders the final output for the user. This clean separation improves maintainability, supports parallel development across components, and allows the system to evolve as models and tools change.

System Goals in Practice

In practice, this design achieves its goals by balancing capability with control in a way that feels practical for day-to-day development. The split between the CLI and the core keeps the terminal experience responsive even when the backend is doing heavier work, like scanning a repository, assembling context, or waiting on model/tool results. It also makes the system more adaptable: the same core logic can be reused across different frontends (for example, a GUI or IDE integration) without rewriting how requests are interpreted, how tools are called, or how safety policies are enforced. The tool system is what makes Gemini CLI genuinely action-oriented, but it stays safe because the core acts as a gatekeeper. Read-only operations can run automatically to gather context quickly, while anything that could change the codebase or system is routed through explicit user approval. That permission step is key for trust: users can see what the assistant is about to do, decide whether it’s appropriate, and avoid accidental edits or commands. Overall, the layered architecture delivers a strong baseline set of capabilities while keeping the design clean enough to extend over time, whether that means adding new tools, supporting new workflows, or incorporating improved model behavior without forcing a redesign of the system.

How Subsystems Interact

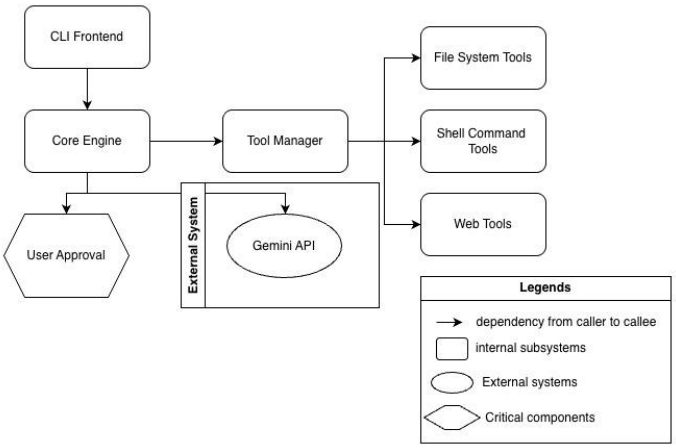


The interactions between subsystems within Google Gemini strictly follows the delegate style, where each request passes through component by component. Every request to the AI model starts with user typing the input prompt, and the prompt is handled by the CLI frontend layer. This component is responsible for interacting with the user and presenting the result to the user. The request will then be sent the core of the entire system, located at packages/core. It is also described as “The AI Orchestration Engine” by the article titled “Unpack the Gemini CLI: A High-Level Architectural Overview” written by Jim Alateras. The core engine constructs the appropriate prompts for Gemini API to process the request, consist of the input prompt and some context information associated with the request. The set is then compiled and sent to the external Gemini API to obtain a response from the AI model. When tools are required to execute certain task(s), the core engine will also invoke the tool manager to access system functions. To avoid any accidental destructive operations, any modification on files or shell commands execution will be gated, requiring explicit approval from the user. In the end, the result is passed back to the CLI frontend subsystem, and the formatted result is displayed to the user.

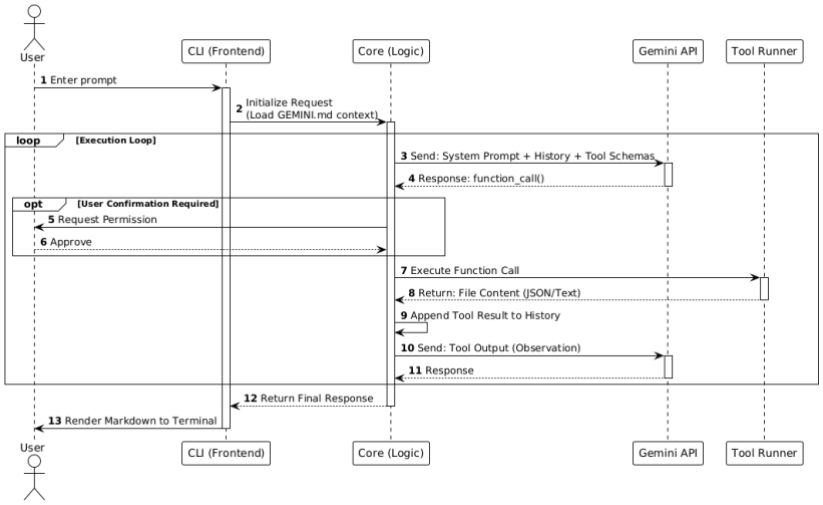
Dependency Direction

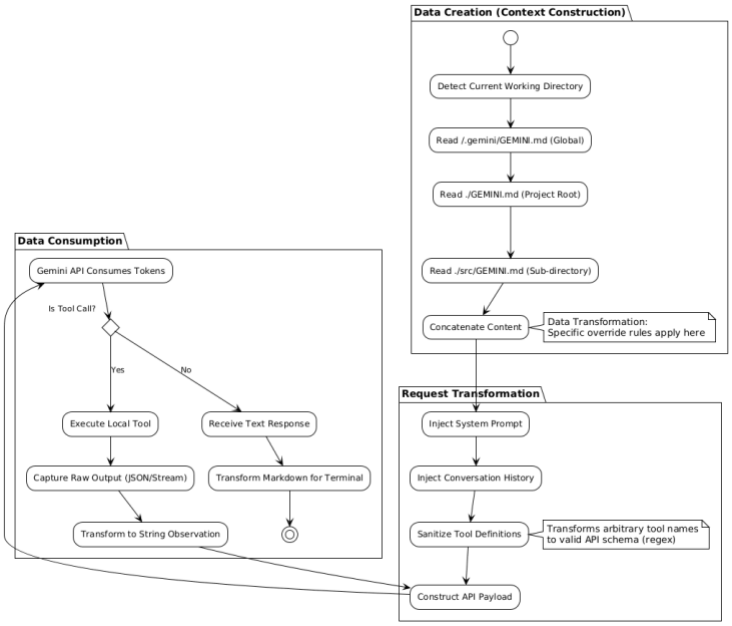
The dependency of each component in the system is intentional designed as unidirectional; in other words, the flow of information is in one direction only. The CLI frontend, which is responsible for interacting with users and presenting formatted response to users, heavily depends on the core orchestration engine for reasoning and decision-making processes. But the coordination of tasks that require system-level tools is out of reach for the CLI package, as it doesn’t contact with Tool Manager directly. The core engine relies on the Tool Manager to call tools and libraries to complete system-level operations, and it depends on the external Gemini API for inferences that will be done by the AI model. In the subsystem, the tool implementations are abstracted behind Tool Manager and not allowing direct access of low system-level tools for execution from high abstraction layers. This dependency design decision forces the separation of concerns and focus to different components based on their specialties. It also reduces coupling system-wise. This also enables reusing some part of the system in different combinations.

Box-and-arrow Concept Architecture Diagram (with Legend)



This directed box-and-arrow diagram shows the dependency relationships between each major component, or subsystem, in overall system. The diagram is drawn in a way where the solid line arrow is pointing from the caller to the callee, which shows the direction of information flow. And each kind of shapes in the diagram represents a type of subsystem of Google Gemini: the round rectangle is each individual subsystem, the ellipse represents external system, and the hexagon represents the critical component of the system. Every request to the system starts with the frontend of the CLI package, the prompt is packaged and sent to the core engine for processing and coordination. If one or more tools are needed to perform certain tasks, the Tool Manager will be called to invoke system tool functions to complete the executions. The prompt and context information will also be passed from core engine to the external Gemini API subsystem. To ensure maximum safety, any modifications on files and tool access will be gated.





Concurrency Model

Architecturally, the Gemini CLI operates on a Single-Threaded Event Loop model (typical of Node.js applications). While the logical orchestration is sequential where a user request triggers a deterministic chain of Context Loading, API Inference, and Tool Execution, the underlying execution is non-blocking. This allows the CLI frontend (built on Ink/React) to remain responsive, rendering UI updates (spinners, streaming text) even while the Core engine awaits high-latency network responses from the Gemini API.

The system uses a Streaming Pipe-and-Filter pattern for model interactions. Rather than blocking for a complete HTTP response, the Core opens a persistent stream with the API, pushing tokens to the CLI's rendering buffer as they arrive. This creates a "Time-to-First-Token" latency that is near-instantaneous from the user's perspective, masking the computational overhead of the LLM.
However, this asynchronous behaviour is deliberately interrupted by the Safety Gate architectural pattern. When the Core detects a side-effect tool call (e.g., fs.write), it enforces a Synchronous Barrier, pausing the execution pipeline until a “Permission Granted” signal is received from the user. This ensures that while data processing is concurrent, state-changing operations remain deterministic and human-supervised.

System Evolution

How the System Changes Over Time

The architecture has evolved from a stateless "Chatbot" model to a context-aware "Agentic" system. Early iterations relied on naive context loading. This has evolved into a Hierarchical Context Architecture, where the system now recursively merges GEMINI.md files from the global, project, and directory levels. This evolution addressed the "Context Dilution" problem

inherent in large codebases. The security model evolved from simple binary prompts ("Allow Y/N?") to a Tiered Policy Engine. This allows the system to support complex, rule-based automation (e.g., "Always allow read operations, but gate write operations"), reflecting a shift towards enterprise-grade usability.

Extensibility and Constraints

The system's extensibility is architecturally decoupled through the Model Context Protocol (MCP). By treating tools as external "Servers" rather than internal library calls, the Core can dynamically discover and register new capabilities without recompilation. This style allows the ecosystem to grow independently of the core CLI release cycle.

To preserve system integrity, the architecture enforces a strict Unidirectional Dependency Rule. The CLI Frontend is strictly a presentation layer where it cannot bypass the Core to invoke tools directly. Furthermore, the Sandbox Constraint dictates that while the agent runs on the host, it cannot execute side effects outside of the Safety Gate, ensuring safety.

Division of Responsibility

Google Gemini system is clearly designed around two main packages, packages/cli and packages/core, each with their own unique responsibilities. The CLI package handles both user interaction and formatted presentation to users. It handles the user aspect of each request, such as input prompt handling, formatted output display, as well as some internal roles, such as conversation history management, theme and UI rendering (with Ink library), and present the alert dialog to obtain user approval for execution. It doesn't touch anything related to reasoning, the actual task execution processes and API calls. In other words, it purely focuses on UI and presentation.

The core package is another main component of Google Gemini system, and it is in charge of orchestrating the rest of the system, including but not limited to AI inference, reasoning and decision-making processes, and manages tools. Just like described in the article titled by Unpacking the Gemini CLI: A High-Level Architecture Overview, the core package can be seen as the "behind the scenes intelligence coordinates". Its responsibility includes but not limited to managing and coordinating AI processes and tools, also ensuring the safety of each execution.

There are also some surrounding packages that support the execution of each request, for example, the tool subsystem. It consists of a set of tools for file system, shell command execution, web search and discovery, as well as memory access and management. Its sole responsibilities are defining tool sets and registering them with the system; later at runtime, implement system capacities with those tools registered beforehand, and the confirmation message will be displayed by the CLI frontend component. The remaining part is the external system, that interact with the Gemini system, which includes Gemini API, the operating system, and some services that are available online.

Parallel Development Implication

The mono-repository structure means all related package in the Google Gemini system are managed and packaged in a single repository. This design decision allows for tighter coordination between sub-components. In this style, when adding new features, developers can just commit once, with all the modifications, even some cross package ones. It is less likely to produce broken links between components caused by the update. For interfaces and dependencies, since they are shared across packages, they are less error prone, and this style ensures all dependencies are on the same version.

The architectural design choice of using mono-repository style but decompose into several subsystems allows partial parallel development, while maintaining coordinated integration points. It also allows individual evolution of a single component without affect other ones. Developers can work the frontend part, without touching a single line of code for the backend core engine code. Similarly, improvements made to the core will be inside the core package, not impacting the CLI package.

The tool manager extends the parallelism by acting as a plugin-style architecture in the entire program. New tools or functions can be added without needing to modify the higher-level functions. However, some specific functions do require coordinated updates to both parts of the system. Features that include user visible behaviours will be an example of this category. But this architecture style reduces the risk to a minimal level, as a single commit can publish all modifications to the code, even those cross-package ones, and with shared interfaces and dependencies, the risk is minimal.

Overall, this architectural style allows each component to act independently but also control them with coordination points. This not only reduces the overhead when integrate them together but also maximizes the extensibility.

Identified Styles, Benefits and Tradeoffs

In this complex CLI system, there are multiple architecture styles that co-exist with each other. For example, the most obvious one is the layered architecture. Where the information is passed from the user, as a prompt through the CLI's the presentation layer to the core engine. The response of the AI model is then passed to tools layer and executed on the external system layers. The advantages are clear separation of components, which means each one can be swapped or changed, but it also introduces some slight communication overhead and requires stricter interface definitions for communication.

Another type of architectural system suitable for this system is client and server, where the CLI and the core package combine as the client, and Gemini API, web services, and the operating system's shell commands turn into the server, as it is where the tasks are executed. For this style, it allows for better and cleaner external integration, with clear modular boundaries. But the disadvantages are also obvious: the performance depends heavily on the network speed, and API exchanges might break.

Clearly, it can also be framed as a mono-repository architecture, where all components, and packages are packaged in a single repository. This allows for coordinated release of the entire system, and the system shares interface and dependencies. It also reduces the complexity of testing. But the downside is that the size of the repository will be extremely large and requires discipline to manage all the dependencies and files.

Alternative 1: Sandboxed Execution (Claude Code Approach)
We considered a "Hermetic Isolation" architecture similar to Anthropic's Claude Code, which executes all agent commands inside a containerized environment (Docker/microVM) to maximize safety.

We rejected this in favor of a Host-Based Execution model. While sandboxing offers higher safety by default, it introduces significant friction for local development (latency in file syncing, inability to access local toolchains/databases). Gemini CLI prioritizes latency and integration, allowing the agent to use the developer's existing environment (git, linters, compilers) directly. The risk is mitigated via the "Safety Gate" pattern instead of architectural isolation.

Alternative 2: Cloud-Native SaaS Architecture

A thin-client CLI that sends all context to a remote cloud environment for processing and execution.

Rejected due to Privacy and Data Sovereignty. A local-first architecture ensures that sensitive source code and environment variables remain on the user's machine. Only the necessary inference tokens are transmitted, whereas a cloud-native approach would require uploading the entire codebase state, raising significant security concerns for enterprise users.

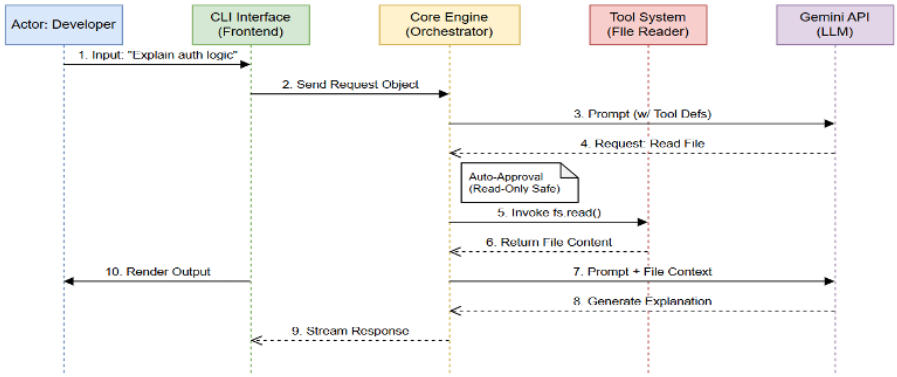
Use Case 1: Contextual Inquiry and Read-Only Analysis

This use case exemplifies the "read-only context query" pattern within the Gemini CLI architecture. The scenario centers on a software developer who seeks to comprehend a specific codebase segment without altering the source code. The system initiates this process when the developer submits a natural language query through the terminal interface. This example effectively demonstrates how the architecture autonomously resolves dependencies and acquires the necessary context.

The workflow illustrates the seamless collaboration established between the frontend components and the core engine. The command-line interface receives the input and subsequently packages the raw text with the session configuration for the backend. The core engine functions as a central coordinator and constructs a prompt that contains both the user query and the definitions of available tools. The Large Language Model (LLM) analyzes the intent upon receiving this prompt via the external Gemini API and resolves the question using the provided context.

A key architectural feature demonstrated here is the Policy-Based Automation. The Core Engine intercepts the model's request to invoke the file-system-read tool. Because the system policy classifies file reading as a side-effect-free (safe) operation, the Core grants automatic permission. It invokes the Tool System to fetch the file contents, injects this data back into the

conversation context, and re-queries the Gemini API. Finally, the model synthesizes the code explanation, which is streamed back through the Core to the CLI Interface, rendering a formatted Markdown response to the user. This entire loop occurs without manual intervention, prioritizing speed and responsiveness.

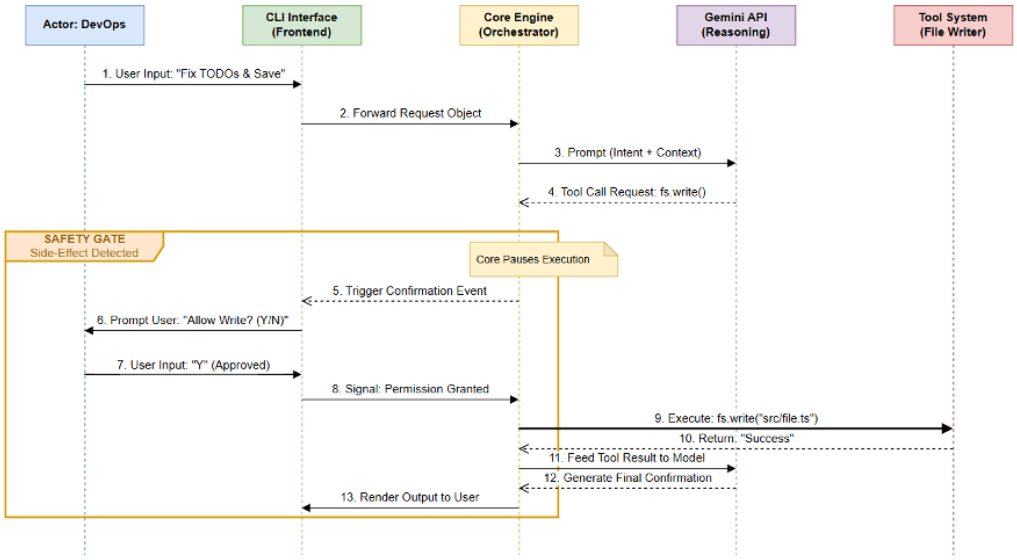


Use Case 2: Safety-Gated Code Modification

This use case serves to demonstrate the system's rigorous implementation of the Safety Gate and Side-Effect Control architectural patterns. In this scenario, the Actor (a DevOps role) initiates a command that implies a tangible change to the project state: "Fix TODOs & Save." This trigger activates a workflow distinct from passive inquiries, specifically engaging the system's protection mechanisms against unauthorized file system mutations.

The flow commences with the CLI Interface forwarding the request object to the Core Engine (Orchestrator). The Core consults the Gemini API, which analyzes the intent and context, subsequently issuing a fs.write() tool call request. However, upon receiving this request, the Core Engine detects a "Side-Effect." Instead of immediately fulfilling the tool call, the architecture triggers the Safety Gate. A "Core Pauses Execution" state is enforced, illustrating a synchronous blocking mechanism designed to prevent autonomous system overreach.

At this juncture, the system demands a Human-in-the-Loopvalidation. The Core triggers a confirmation event, causing the CLI to prompt the user specifically: "Allow Write? (Y/N)." The execution thread remains suspended until the user provides an affirmative "Y" input. Only once this Permission Grantedsignal is received does the Core Engine unlock the gate and permit the Tool System to execute the actual fs.write("src/file.ts") operation. The workflow concludes as the tool's success result is fed back to the model, which generates the final confirmation output for the user.



External Interfaces

Beyond its own CLI, core, and tool subsystems, Gemini CLI communicates with several external entities. These interfaces make it clear how the system communicates with users, services, and the local environment while also defining its operating bounds.

User Interface

The primary external interface of Gemini CLI is the terminal-based user interface. This interface enables bidirectional communication between the user and the system.

Interaction Model

Through the command line, users offer prompts in natural language. After capturing input and parsing configuration options and flags, the CLI sends the request to the core engine. For readability, responses are broadcast back to the terminal in structured Markdown format. AI generated responses can be rendered incrementally before the entire computation is finished because of this streaming feature.

Session Management

Throughout a session, the CLI keeps track of the previous conversations, maintaining context across several prompts. Additionally, user settings like theme configuration, formatting style, and verbosity are applied.

Confirmation Gates

The CLI displays a clear confirmation prompt when a requested action has the potential to alter files or run shell commands. Only with the user's consent does execution begin. The user is guaranteed to maintain control over all side-effect activities thanks to this interface.

Boundary Role

The CLI only functions as a presentation layer from an architectural standpoint. It sends requests and renders outputs, but it doesn't do logic or tool execution. This maintains the separation of computational logic from user engagement.

External Services

To process user requests, Gemini CLI relies on different external systems, with the Gemini LLM service being the primary external dependency. The core engine constructs structured API payloads containing the system's prompt, chat history, tool schemas, and contextual project data. The Gemini API receives these payloads across the network, processes them, and returns either a direct text response or a structured function-call request indicating that a specific tool should be executed. Due to the remote nature of the Gemini model, this interaction requires secured network connectivity and introduces factors such as network latency, token limits, and API throughput constraints.

For interactions with the local execution environment, Gemini CLI utilizes a tool subsystem architected around the Model Context Protocol (MCP). This protocol serves as a universal standard that abstracts the interface between the AI model and external capabilities. Through MCP, the system can seamlessly alter files, execute shell commands, or read project directories to build contextual awareness, treating each capability as a standardized "MCP Server." To guarantee safe execution, all side-effect operations (such as writing files or running commands) are gated behind explicit user confirmation. Results from these operations are serialized according to the MCP specification and sent back to the core for additional analysis.

Web-based retrieval services can also be integrated to enhance local context. The system may employ specialized MCP-compliant tools to fetch external data or documentation from the web. These interactions broaden the scope of the model's reasoning capabilities while ensuring that all external data flows through the same predetermined tool contracts and safety measures defined by the protocol.

Information Exchange

Natural language prompts entered through the terminal require the main input from the user to the system. Additional inputs include: Configuration settings, CLI flags, and the acceptance or rejection of suggested side-effect operations.
Responses are sent in a structured Markdown format from the system to the user. These outputs could be in the form of error messages, confirmation prompts, execution results, code recommendations, or explanations. As tokens are received from the model, streaming allows the incremental showcase of incomplete responses.
Structured JSON payloads are used for communication between the Gemini API and the core engine. These consist of tool descriptions, contextual information, conversation history, and tokenized prompts. The API requires that a tool should be run by returning either a structured function-call requests or a streaming text.

A specified invocation contract rules the interface between the core and the tool subsystem: the tool gives structured results like file contents, command output, or retrieved data, where as the core sends the tool name and inputs. Before being reincorporated into the model's reasoning loop, these outcomes are normalized and added to the conversation history. Contextual information is put together, system prompts are added, and tool schemas are cleaned up before being sent to the Gemini API. Outputs are converted into uniform observation forms after tool execution. The traceability, consistency, and secure incorporation of external data into the AI-driven process are guaranteed by this multi-layered transformation pipeline.

Lessons Learned

We learned that even with Gemini 1.5 Pro's 1-million-token window, "infinite context" is not a silver bullet. Naive context loading leads to latency degradation and "lost in the middle" phenomena. The architecture had to evolve to include a Hierarchical Context Loading mechanism (Global vs. Project vs. Directory in GEMINI.md) to prioritize relevant information over raw volume.

Another key lesson was that a strict "Safety Gate" (asking for confirmation on every write) leads to user fatigue, causing them to blindly approve actions. The architecture evolved to include a Policy Engine (Tiered Priority System) that allows users to define rules (e.g., "Always allow ls", "Never allow rm -rf"), shifting the safety model from purely interactive to policy-driven.

Conclusion

Gemini CLI's basic design has been presented in this study as a modular, layered, delegate-driven system made up of a standardized tool subsystem, a core orchestration engine, and a CLI front-end. In order to maintain strong separation of concerns and minimize coupling, the architecture is purposefully organized around unidirectional dependencies, where requests go from the display layer to the core and outward to external systems.

The Model Context Protocol (MCP), which formalizes the interface between the AI model and external capabilities, is a distinguishing architectural characteristic. Gemini CLI maintains standardization, traceability, and safe governance of environmental actions by abstracting file operations, shell commands, and web retrieval behind MCP-compliant tool contracts. This protocol-based approach preserves distinct operational boundaries while enhancing extensibility.

Additionally, the system developed a more sophisticated safety model by moving beyond basic confirmation gates. The implementation of a policy-driven execution paradigm permits tiered control over side-effect actions instead of depending exclusively on repeated user prompts. The design exhibits responsiveness in terms of performance, contextual accuracy, and safety enforcement when paired with hierarchical context loading mechanisms (e.g., structured GEMINI.md scopes at global, project, and directory levels).

Deliberate compromises are reflected in architectural choices like choosing a local-first execution architecture over a fully cloud-native or sandboxed alternative. Gemini CLI strikes a balance between automation and user sovereignty by giving priority to integration with the developer's current environment and reducing risk through gating patterns and policy enforcement. The monorepo structure further allows for synchronized evolution of tightly coupled subsystems, supporting proper development without sacrificing clarity.

Overall, Gemini CLI serves as an example of how contemporary AI-assisted systems must have layered design, protocol-based extensibility, policy-driven safety measures, and contextual intelligence to maintain their strength and reliability. Its architecture provides a workable plan for integrating big language models into developer processes while maintaining control, transparency, and long-term architectural stability.

AI Collaboration Report

Our group selected Google Gemini (Model: Gemini 3 Pro) as our primary virtual teammate for two strategic reasons:

- **Context Window Capacity:** The project required analyzing extensive PDF documentation (assignment details, rubrics) alongside large technical concepts. Gemini's large context window allowed us to upload the entire project specification and rubric simultaneously, ensuring the AI "understood" the constraints without hallucinating requirements.
- **Deep Research Capabilities:** Unlike standard LLMs which rely on static training data, Gemini's ability to perform autonomous, iterative web searches was useful for recovering the architecture of Gemini CLI which is a relatively new open-source project with sparse high-level documentation.

We treated the AI not as a writer, but as a Senior Architectural Consultant. The specific tasks delegated were:

- **Rubric-Based Compliance Audit:** We assigned the AI the role of a "strict marker." We uploaded our draft and the marking scheme, asking it to identify specific missing sections that would prevent us from scoring 100%. This was suitable for AI because LLMs excel at pattern matching against explicit constraints.
- **Gap Analysis & Deep Research:** We lacked detailed knowledge on specific architectural comparisons (e.g., Gemini CLI vs. Claude Code). We tasked the AI to perform "Deep Research" to find evidence of specific patterns in the GitHub repository and roadmap.
- **Section Synthesis & Technical Refinement:** Once the research was validated, we asked the AI to synthesize raw technical facts into cohesive architectural descriptions.

Our group utilized a Role-Based Prompting Strategy combined with Meta-Prompting. We did not simply ask "Write this section." Instead, we engineered a workflow:

- **Persona Definition:** We explicitly instructed the AI to "Act as a strict architectural critic. Do not rewrite yet; first, tell us what is missing based on the 'Great' column of the rubric." This prevented the AI from generating generic fluff and forced it to focus on the marking criteria.

- **Meta-Prompting for Research:** When we needed deep technical details, we used the AI to write a prompt for itself. We asked: "Acting as a prompt engineer, write a prompt I can use to deep research Gemini CLI." Here is an example of the 'Deep Research' Prompt:
    - "Conduct deep research using the official GitHub repository... specifically looking for evidence of the Model Context Protocol (MCP). Compare Gemini CLI's architecture to Claude Code... Why did they choose a local CLI-first approach over a cloud-only implementation?"
- **Iterative Refinement:** When the AI produced a draft section, we noticed it missed the specific protocol name. We refined the prompt: "Rewrite this to explicitly include the Model Context Protocol (MCP) as the architectural standard."

To mitigate the risk of "hallucination," we implemented a strict validation protocol:

1. **Source Traceability:** We explicitly requested that the AI identify where it found information. When the AI claimed the system used a "Monorepo Architecture," we verified this by checking the packages/core folder structure in the actual GitHub repository to confirm that all the parts belonged under a single parent directory.
2. **Comparative Fact-Checking:** When the AI provided the comparison with Claude Code (Sandboxed vs. Local Execution), we cross-referenced this with Anthropic's official documentation to confirm that Claude Code indeed uses a containerized environment, validating the AI's "Alternative Architecture" claims.
3. **Rubric Alignment:** We manually reviewed the AI's suggestions for the "Lessons Learned" section against the assignment rubric to ensure they met the criteria for "insightful reflection" rather than just surface-level observations.

We estimate the AI contributed approximately 35% to the final deliverable. Of this 35%, around 25% was for drafting section content that was later refined and validated as detailed above. The other around 10% was for providing structural guidance where the AI identified missing components and work remaining to align with the "great" column of the rubric.

Integrating the AI teammate shifted our workflow from "Drafting" to "Curating." The AI was a massive force multiplier for research. Manually comparing Gemini CLI's architecture to Claude Code would have taken hours of reading documentation. The AI's "Deep Research" tool accomplished this in minutes, providing a structured report we could immediately critique and integrate. The main challenge was context drift. We had to frequently remind the AI of the specific constraints (e.g., "This is A1, not A2 so keep it conceptual"). We learned that effective collaboration requires treating the AI not as a search engine, but as a junior engineer who is brilliant but prone to losing focus; precise, constraint-heavy prompts were the key to success.