CISC 322
Assignment 1

Conceptual Architecture Analysis of Gemini CLI

February 13, 2026

The Unemployed
Gabrielle Garey (22dny@queensu.ca)
Calvin Wong (23dt5@queensu.ca)
Kesiya Jacob (22ccf5@queensu.ca)
Sofiia Bushareb (22wx56@queensu.ca)
Ryan Walsh (22bfr3@queensu.ca)
Grisha Tyukin (22ld34@queensu.ca)

Table of Contents

Table of Contents 2
1. Abstract 3
2. Introduction and Overview 3
3. Derivation Process 4
4. Overall Architecture 4
    4.1. Division of Responsibilities 6
    4.2. Data Flow 6
    4.3. Subsystems 7
        4.3.1. API Client 7
        4.3.2. Prompt Construction 7
        4.3.2. Tool Registration & Execution Logic 7
        4.3.4. State Management 8
        4.3.5. Server Configurations 8
        4.3.6. Input Processing 8
        4.3.7. Display Rendering 8
        4.3.8. CLI Configuration Settings 9
        4.3.9. System Tools 9
    4.4. Concurrency 9
5. Use Cases 9
    5.1. Use Case 1: Tool Request 9
    5.2. Use Case 2: Direct Response 10
6. Data Dictionary 11
7. Naming Conventions 12
8. Lessons Learned 12
9. Conclusion 13
10. AI Collaboration Report 13
11. References 15

1. Abstract

Our team examines the conceptual architecture of Gemini CLI, Google's open-source, terminal-based AI assistant designed to support developers by accessing their local codebase directly. Through our analysis, we determined that Gemini CLI primarily follows a client-server architectural style, separating the user-facing responsibilities from backend processing and remote AI services. The client side consists of the CLI package, which handles user input, presents final output, and manages the overall user experience. On the server side, the core package functions as the backend of the system, where it manages communication between the API client and the Gemini API, prompt construction and management, tool registration and execution logic, state management for conversation sessions, and server-side configuration. This separation of responsibilities supports a modular design, making the system easier to update and maintain. In particular, this design supports extensibility through the Model Context Protocol, allowing new tools to be added without changing the core request-response flow. However, it also introduces trade-offs, such as network dependency, network latency, and coordination overhead between components.

2. Introduction and Overview

Gemini CLI is Google's open-source AI assistant that runs directly in your local terminal, understands your codebase, and helps you fix bugs using natural-language prompts. It is Google's response to Anthropic's Claude Code (Dutt, 2026). Gemini CLI connects directly to your local environment, and while preserving the context of your project, can run shell commands, conduct web searches, and read and edit your files (Gemini CLI Documentation, 2026).

We were given the responsibility of researching and reporting on Gemini CLI's conceptual architecture regarding topics such as system evolution, architecture styles, concurrency, and control and data flow among parts. As a team, we concluded that the main architecture style of this system follows a client-server style because it separates user-facing responsibilities from backend processing.

Our derivation process and how we arrived at these theories are discussed in the first section of our report. We then explore the architecture, examining its general design, subsystems, data flow and control, concurrency, and other aspects. Additionally, we go over distinct use cases for Gemini CLI and provide a sequence diagram for each. Lastly, we discuss the report's conclusions and the lessons we learned. To complete our evaluation, there is also a data dictionary, naming conventions, and an AI collaboration report.

3. Derivation Process

To understand the conceptual architecture of Gemini CLI, our team first divided the work so that different group members could investigate individual aspects of the system simultaneously, including the overall architecture, component responsibilities, and data and control flow. Within these smaller groups, we analyzed the official Gemini CLI documentation and relevant technical articles, while also referencing the course material to ground our findings. This allowed us to develop a shared understanding of how Gemini CLI operates and how its major components interact. Once we had a thorough understanding of the system, we identified Gemini CLI's key features, including prompt handling and session management, and used those to build our use case designs and architectural analysis. We came up with two use cases that use some of Gemini CLI's core features: tool request and direct response. We also concluded that the separation between the CLI and core package aligns with a client-server architecture style.

4. Overall Architecture

The studied system, Gemini CLI, is built using a client-server architecture that allows a client to access the Google AI model through the command line. The system demonstrates modular organization with three distinct packages, each working to distribute data and process that data across a range of components. This architecture exhibits the characteristic dependability on a stable connector, that being, the HTTP protocol to connect a client in the terminal to the Gemini AI model. On the client side lies the CLI package, containing the user-facing portion of Gemini CLI. It handles initial user input, presents final output, and manages the overall user experience. On the server side, the core packages act as a backend for Gemini CLI. These packages receive requests sent from packages/CLI, orchestrate interactions with the Gemini API, and manage the execution of available tools.

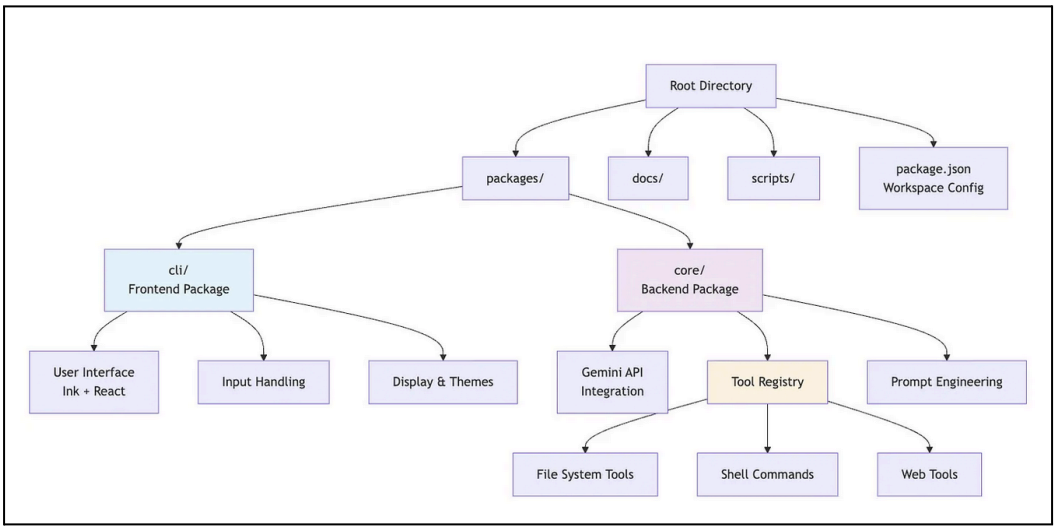

Figure 1: Directory Map of Gemini CLI (*Medium*)

The application supports two main ways of running on the client's local machine: through the interactive UI and in non-interactive CLI mode. The interactive UI facilitates back-and-forth communication between the user and the LLM, while the non-interactive mode is useful for automation and scripting. Both of these are managed by packages/cli, whose main job is managing the user experience. Built on Ink, a React renderer for command-line apps, it transforms the static terminal into a more dynamic and interactive environment. This package handles capturing user input, displaying the AI model's response with proper formatting, managing conversation history so that a user can reference previous exchanges, and applying various colour themes and display settings. packages/cli communicates with packages/core to send users' input to be processed and sent to the LLM.

While the CLI package focuses on user interaction, the core package operates behind the scenes as the system's backend. This package constructs appropriate prompts for the Gemini API, potentially including conversation history and available tool definitions, and sends these prompts to the API for processing. The core package also manages a suite of tools, individual modules that extend the capabilities of the Gemini model by integrating it with other environments through operations like file system manipulation, shell command execution, and web fetching. When the Gemini API requests a tool execution, the core package prepares and manages the execution process and implements a security model where operations that can modify the file system or execute shell commands require explicit approval from the user before proceeding.

The client-server architecture provides several benefits for the system's evolution, allowing new features to be added, bugs to be fixed, and capabilities to scale without disrupting existing functionality. As mentioned previously, the system is split into packages, including "packages/cli" and "packages/core". This separation of concerns enables individual packages to be modified, tested, or replaced independently, allowing teams to work in parallel and reducing the risk that changes in one area will break others. As the system grows, Gemini CLI scales by extending existing modules within the core package rather than modifying its fundamental architecture. New capabilities are typically added through the tool system in the core package, where tools act as modular extensions that allow the model to interact with the local environment, such as the file system or shell commands. New tools can be added or existing ones improved without altering the fundamental request–response flow between the CLI, core, and Gemini API.

The Gemini CLI is designed to be extensible through the Model Context Protocol (MCP), which provides a standardized interface for integrating additional capabilities. MCP allows external MCP servers to expose tools and resources to the CLI without requiring changes to its core codebase. An MCP server is an independent application that acts as a bridge between the Gemini model and local or external resources, such as files, scripts, APIs, or databases. Through MCP, the CLI can dynamically discover available tools and resources and execute them when requested by the model. Together, these architectural choices allow the Gemini CLI to evolve in a controlled and maintainable way.

Despite its advantages, the client-server architecture does introduce several technical challenges for software architects. For example, the performance of the overall system is dependent on the performance of the network. If the Gemini API is down, there is no way to communicate with the

AI model, and the entire system is essentially down. Network latency is also a concern, as the system relies on HTTP communication with its API servers. Additionally, as is usually the case with client-server applications, it can be hard to find what services are available. Gemini CLI handles this by placing every operation under two packages, but that places a large workflow overhead on the system because every user interaction must flow through both the CLI and core packages, with each transition adding processing and coordination costs.

The non-interactive CLI mode also uses the pipe and filter architectural style. It allows Gemini CLI to integrate into shell scripts by accepting input through standard input (stdin) and producing output (stdout). Users can pipe data from other commands into Gemini CLI for processing, such as cat file.txt | gemini "summarize this". They can also chain Gemini's output to other tools (e.g. Gemini "generate data" | grep keyword | awk '{print $1}'). The pipe and filter pattern aligns well with the system's overall modular design. Each component, whether Gemini CLI or another Unix tool, processes data and passes it along without needing to know about the internal implementation of other components in the chain. However, this pattern does introduce the constraint that data must flow in one direction through the pipeline, which contrasts with the interactive CLI mode's bidirectional conversation model.

4.1. Division of Responsibilities

Responsibilities for further developing Gemini CLI are divided via the github and organized in the roadmap published on the public repository. The repository is linked on the official website and is open source to everyone. The docs are also linked and available to consult. New preview releases are published each week at UTC 2359 on Tuesdays, and the repo encourages users to help test and install the software. The roadmap, which is a compiled list of what is currently being undertaken, is updated every quarter based on who is assigned what task on GitHub, and acts as a summary for features in development. The repo also allows users to host discussions and Q & A sessions to talk about new feature implementations or troubleshoot issues.

4.2. Data Flow

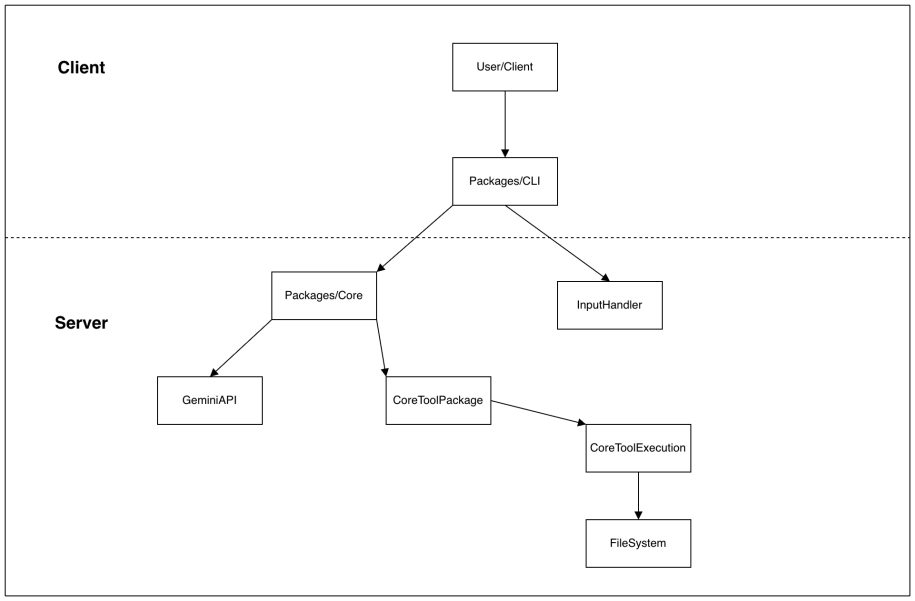

Figure 2: Box-and-Line Diagram of Gemini CLI's Client-Server Architectural Style

4.3. Subsystems

As an interactive command-line application built on Node.js, Gemini CLI organizes its functionality across two primary packages that work together to deliver AI-powered development assistance from the Google AI model. The CLI package is responsible for user interaction and presentation, utilizing Ink, a react renderer for command line applications, to create the terminal interface. The core package handles backend operations, including API communication and tool execution. With these packages, Gemini CLI's components can be divided into user interface elements managing input and display, backend orchestration managing API interactions and prompt construction, extensible tool systems that allow the AI model to interact with different local environments, and configuration management that oversee authentication, user settings, and extensions.

4.3.1. API Client

The API client component resides within the core package and is responsible for communicating with the Google Gemini API. It sends a constructed request, consisting of the prompt, conversation context, and tool schemas to the Google Gemini API and receives either a direct model response or a tool invocation request. It interacts with multiple other components, receiving constructed prompts from the prompt construction module, transmits tool schemas from the tool registry, and processes responses. When the API returns a tool execution request, the API client coordinates with the tool execution logic to process the request and send results back to the mode for further processing. This multidirectional communication is what allows Gemini CLI to perform its complex, multistep operations for extended user sessions.

4.3.2. Prompt Construction

The Prompt construction module lies within the core package and is responsible for assembling a comprehensive prompt that gets sent to the Gemini API. The prompt combines the user's input with the conversation history and possible tools available to produce a structured prompt. It works with other components, retrieving information from the state manager to maintain context, accesses tool schemas from the tool registry, and takes instructional context from GEMINI.md files. To manage the context window, when a conversation approaches the token limit for the configured model, the module will compress the conversation history before sending it to the API client.

4.3.3. Tool Registration & Execution Logic

The tool registration and execution logic lies within the core package and holds resources that extend the capabilities of Gemini CLI. Tool registration defines which tools are available to the system and how they can be called. The registry connects to the Model Context Protocol servers to list and register tools as different instances. The execution logic provides the runtime machinery that invokes a tool when the Gemini model requests it. These tools extend the model's

capabilities by enabling interaction with the local environment, such as the file system, shell commands, or web fetching. This module interacts with the prompt construction module to provide tool specifications, coordinates with the API client to receive tool requests and return results, and communicates with the CLI package to get user approval for operations and display execution feedback.

4.3.4. State Management

The state management module resides in the core package. It maintains conversation and session context across multiple turns, allowing later requests to reference earlier messages, tool results, and ongoing tasks. This ensures interactions remain coherent rather than being treated as independent prompts and ensures that the user can interrupt their work and resume where they left off. Sessions are hashed into a subfolder of the module, allowing the user to switch directories to a different project, and load all of the history of that project, all while the previous directory information is saved. This module interacts with other components to provide conversation history to the prompt construction module, coordinates with the CLI package to enable session browsing through commands like /resume, and works with the configuration settings to allow the user to work on multiple continuous streams.

4.3.5. Server Configurations

The server-side configuration module in the core package provides adjustable settings that control backend behaviour. These settings determine endpoints, behaviours, and security policies. Configuration is a hierarchical ordering of precedence, where default values are hardcoded, a system defaults file for default settings, user settings for personal preference, workspace settings for project-specific configurations, and command line arguments that can override configurations for specific settings. This module manages connections to the MCP servers for using custom tools, handles environment variables, and allows administrators to specify system-level overrides.

4.3.6. Input Processing

The input processing module lies within the CLI package and manages parsing user input, including commands, prompts, and flags. It then converts it into a structured request. This request can then be passed to the Core along with any relevant local CLI settings. This module coordinates with the display rendering module to show command results and API responses to the user, and sends natural language prompts and file contents to the core package for it to be structured and sent to the Gemini API.

4.3.7. Display Rendering

The Display rendering module resides within the CLI package and manages the visual presentation of all information in the terminal interface using React and Ink for UI rendering. As text content arrives from the API, large AI responses are split to improve rendering performance. This module receives formatted responses from the core package for presentation, coordinates

with the input processing module to show user commands and prompts, and applies theme settings from the configuration settings to give the user a smoother experience.

4.3.8. CLI Configuration Settings

CLI configuration settings load and apply options that affect the CLI's behaviour and appearance. Relevant configuration may also be passed through to the Core as part of requests, keeping configuration concerns separate from backend handling.

4.3.9. System Tools

This subsystem can be broken up into three different subgroups. The file system tool lets the system read and write files and list folders (for example, readFile, writeFile, and listDir). Access is limited to specific directories. The web tool component enables the system to make outbound HTTP or HTTPS requests, such as fetching a URL, calling an external API, or downloading a web page. The shell tool component allows the execution of operating system commands, such as git status or ls and returns standard output and error streams.

4.4. Concurrency

Gemini CLI uses an asynchronous, non-blocking concurrency model to inherit its Node.js foundation. Because the system follows a client-server architecture where the server must coordinate multiple long-running tasks, such as HTTP requests to the Gemini API, file system I/O, and external shell integration, it relies on the Node.js event loop to handle these operations without freezing the UI (Gemini CLI Documentation, 2026). The display rendering module is also worth noting here, because it handles streaming text content from the API. To maintain performance, the module splits large AI responses and renders them incrementally using React and Ink. Furthermore, the tool execution logic uses a coordinated concurrency flow. The system must pause the primary conversation stream to wait for user approval and tool output, essentially maintaining multiple states of execution simultaneously. For example, the system can be waiting for the API, waiting for user input, and executing logical tasks all at the same time.

5. Use Cases

5.1. Use Case 1: Tool Request

This use case assumes the developer has Gemini CLI installed and authenticated with their Google account. The user types a prompt in the terminal requesting file creation (e.g., gemini "fix the bug in parser.py"). The CLI package receives the user input and forwards it to the Core package, which constructs an API prompt including conversation history and available tools. The Core sends this request to the Gemini API, which processes the prompt and determines that the write_file tool is needed to fulfill the request. The API returns a tool use request to the Core package. The Core sends an approval request to the CLI, which displays it to the user. When the

user approves, the CLI forwards the approval to the Core, which executes the write_file tool to modify the file. The Core sends the execution results back to the Gemini API, which generates a confirmation message. The API returns this confirmation to the Core, which forwards it to the CLI package. The CLI displays the success message in the terminal, confirming the file was modified.

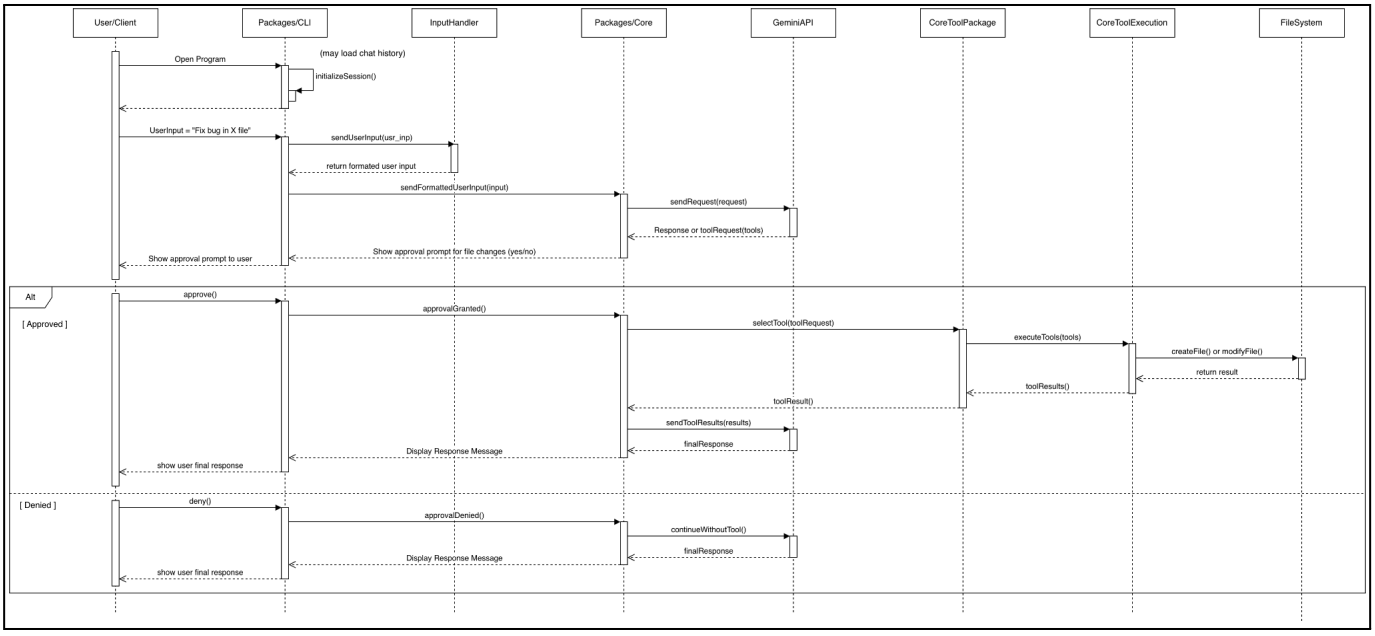

Figure 3: Tool Request Sequence Diagram

5.2. Use Case 2: Direct Response

The user types a prompt in the terminal requesting an explanation (e.g., gemini "explain how list comprehensions work in Python"). The CLI package receives the user input and forwards it to the Core package, which constructs an API request including the prompt and conversation history. The Core sends this request to the Gemini API, which processes the prompt and generates an explanation as a direct text response. The API returns the text response to the Core package, which forwards it to the CLI package. The CLI then formats and displays the explanation in the terminal for the user to read.

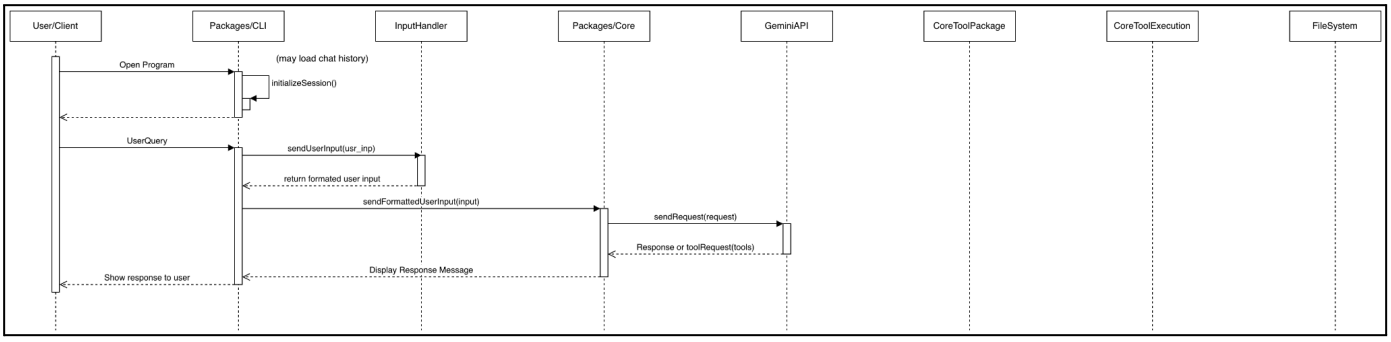

Figure 4: Direct Response Sequence Diagram

6. Data Dictionary

Application Programming Interface: Application programming interface, a set of functions and procedures allowing the creation of applications that access the features or data of an operating system, application, or other service.

GUI: Graphical User Interface, a system that allows users to interact with digital devices through visual elements like icons, windows, menus, and pointers, rather than by typing text-based commands.

Client: A program, device, or app that sends requests to the server and waits for a response

Command Line Interface: It is a type of method to interact with a computer by typing text commands, usually within a terminal or console window

Command: A single executable input that performs an action in the CLI (ex. /chat)

Roadmap: A project board linked on GitHub that visualizes the weekly division of tasks as a byproduct of the developer meetings from Google. Tasks are organized into categories such as backlog items, new features, and feature requests.

Core Package: Backend package that manages prompts and tool execution

Extensions: Modular add-ons to extend functionality

Headless Mode: Non-interactive mode suitable for scripts/automation

Large Language Model: A type of artificial intelligence that's trained on massive amounts of text so it can understand, generate, and work with human language

Model Context Protocol: A standardized way for a language model to receive, interpret, and use contextual information and additional data (ex. user info, settings, files, or system state) from a system or application

Model: A specific Gemini AI version (ex. gemini-2.5-flash)

NodeJS: An open-source, cross-platform JavaScript runtime environment that lets you run JavaScript outside of a web browser

GitHub: A proprietary developer platform that allows developers to create, store, manage, and share their code. It uses Git to provide distributed version control, and GitHub itself provides access control, bug tracking, software feature requests, and other project resources.

Prompt: The text or query sent to the Gemini model

Repository: A central place where project files live and are managed (ex. code, documents, configurations, and data)

Server: A computer that provides services, data, or resources for clients over a network

Session: A saved browser of interactions (ex. conversation history)

Shell Command: Commands prefixed by ! to execute system shell commands from inside Gemini CLI

7. Naming Conventions

API: Application Programming Interface

CLI: Command Line Interface

Env Var: Environment Variable

GEMINI.md: Project context file

geminiignore: Ignore list file

JSON: JavaScript Object Notation

MCP: Model Context Protocol

UI: User Interface

8. Lessons Learned

Throughout this project, we learned valuable and practical lessons about conceptual software architecture analysis. From our research into GeminiCLI's documentation, developing UML diagrams, and collaborating together as a group to derive and understand all the information we found, we gained a clearer understanding of how to justify and present conceptual system architectures.

Firstly, we learned the importance of identifying and understanding clear responsibility boundaries in system and component analysis. For GeminiCLI, the separation of the client-side CLI package from the core package (which handles prompt construction, tool logic, and coordination with the Gemini LLM) showed us how the client-server architecture is reflected in the interaction between components. This taught us that architectural styles can be deciphered by analyzing the interaction and responsibilities of components in a system, which greatly helped us justify our final architectural decision.

Secondly, we learned how a system can legitimately exhibit characteristics of different architectural styles depending on how it is used. With GeminiCLI, the non-interactive mode

aligns well with the pipe-and-filter style because of how it supports stdin/out pipelines and chaining with other tools. This taught us that architectural analysis should be considered from multiple perspectives (in this case, from an interactive and non-interactive perspective), since with different constraints, the ideal architectural style can change drastically.

Additionally, we learned that designing a system for easy integration is an architectural choice with real trade-offs. GeminiCLI's tool system and MCP-based extensions allow new tools and resources to be added without affecting the overall architecture of the system. However, we discovered that this can introduce extra coordination overhead between components and still depends on network communication with Gemini services. Since tools can interact with local environments, the architecture needs safeguards (such as asking for user permission to modify files). This taught us that designing a system with a focus on easy integration is not free. It requires serious consideration of how it will affect everything else in the system, and how we can balance the potential trade-offs.

Finally, we also learned how much proper documentation matters when performing conceptual architecture analysis. A large part of our ability to justify claims came from being able to point to and understand official documentation and third-party analyses. All the documentation we used was well-written and unambiguous, which was extremely helpful in performing the architectural analysis. This taught us that clear documentation isn't just for show, but is vital to making architectural decisions verifiable and defensible. This is not just for those writing architectural analyses, but for developers, users, and stakeholders alike who need to understand exactly how the system works.

9. Conclusion

In conclusion, our project provided a comprehensive analysis of GeminiCLI. We discussed the architectural design and style, its functionality, evolution, and control flow. After extensive research and analysis, we concluded that GeminiCLI uses a client-server architecture. We researched GeminiCLI's open-source repository, documentation, and third-party analyses of GeminiCLI, which led us to this conclusion. We discussed the benefits of the client-server architectural style for GeminiCLI, citing the benefits of the style, such as modifiability, scalability, and ease of maintenance. We also modelled the architecture of GeminiCLI and its use case switch a box-and-arrow diagram of the overall architecture, and two sequence diagrams demonstrating the two main use cases of GeminiCLI. Overall, our project highlights the strengths and capabilities of GeminiCLI, and the value provided by the client-server architecture it uses to the value provided to users, stakeholders, and developers alike.

10. AI Collaboration Report

For A1, our AI teammate was Perplexity (February 2026 version). We selected this tool because the assignment required conceptual analysis supported by documentation, and Perplexity is particularly strong at retrieving source-linked technical information. Unlike a single-model

assistant, Perplexity operates as an AI agent that can automatically select and route queries to different models such as GPT, Claude, or Gemini, depending on the task. This flexibility made it well-suited for our workflow, since different tasks such as retrieval, summarization, and critique benefited from different model strengths.

Tasks Assigned to the AI Teammate

Locate and cite descriptions of Gemini CLI's conceptual architecture.
Rationale: This task involved searching large amounts of documentation and extracting relevant source-backed statements, which is well-suited to an AI optimized for fast retrieval.

Summarize lengthy technical documentation into concise architecture-relevant takeaways.
Rationale: The sources were dense and technical, so using AI helped quickly produce structured summaries that we could then verify against the originals.

Compare our proposed architecture with a model-generated decomposition.
Rationale: This task involved checking how our components matched the AI's version and spotting any missing parts or unclear responsibilities, which the AI could do quickly to support our review.

Interaction Workflow

Our group used a collaborative workflow: each member drafted prompts for their section and shared both the prompts and the AI outputs with the group. Before sending prompts, we first discussed what we wanted to learn from the AI and gathered local context such as our current diagrams, assumptions, and assignment requirements. After receiving responses, we reviewed them together, decided what was useful, and edited or discarded anything unclear or unsupported.

One of our more complex prompts asked the AI to act like a software architecture TA and compare our proposed architecture of Gemini CLI with its own breakdown. We provided context such as our component list and the subsystems we were analyzing (for example, packages/cli and packages/core), and requested a clear, structured format. Initially, responses were too general, so we refined the prompt by adding more details, asking the AI to be more critical, and requiring it to clearly separate facts from assumptions. These refinements improved the relevance and clarity of the output.

Validation Procedures

All AI outputs were treated as drafts. We verified factual claims by checking primary sources, cross-referencing summaries with original documentation, and edited all text for clarity and accuracy. Any claim that could not be validated was removed or clearly labelled as an assumption.

Estimated Contribution

We estimate that the AI contributed approximately 13% of the final deliverable:

5% summarizing documentation

5% critique and logic checking
3% research support

All architectural decisions, interpretations, and diagrams remained human-generated.

Reflection on Team Dynamics

The AI improved efficiency in drafting, research, and brainstorming, but it also introduced additional verification work. It helped identify missing assumptions and improved clarity through rewriting support. However, some responses sounded confident even when partially inferred, so our group adopted a strict rule that factual claims required source validation. Overall, we learned that effective AI collaboration requires clear prompts, structured outputs, and consistent verification.

11. References

AI TL;DR. (2025, December 4). *Gemini CLI - The command line agent from Google.* YouTube. https://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

cz. (2025, June 28). *Gemini CLI Project Architecture Analysis*. DEV. https://dev.to/czmilo/gemini-cli-project-architecture-analysis-3onn

Dutt, A. (2025, June 27). *Gemini CLI: A Guide With Practical Examples*. DataCamp. https://www.datacamp.com/tutorial/gemini-cli

Gemini CLI documentation. (2026). Gemini CLI. https://geminicli.com/docs/

Gemini CLI Project Architecture Analysis. (2026, February 6). AI Coding Tools Docs. https://aicodingtools.blog/en/gemini-cli/architecture-analysis

google-gemini. (2025). *Gemini CLI Architecture Overview*. GitHub.

https://github.com/google-gemini/gemini-cli/blob/main/docs/architecture.md

MCP servers with the Gemini CLI. Gemini CLI. (n.d.). *https://geminicli.com/docs/tools/mcp-server/*

myyorku2002. (2025, July 15). *Diving Deep into the Gemini CLI Architecture: A Source Code Analysis - BI Practice*. BI Practice. https://www.bipractice.ca/diving-deep-into-the-gemini-cli-architecture-a-source-code-analysis/