Gemini CLI Conceptual Architecture

Authors:

Riche Cai: 22twq4@queensu.ca

Logan Jarvis: 21lsj2@queensu.ca

Colin Dreany: 22lns3@queensu.ca

Kalan Morris-Poolman: 22lyvd@queensu.ca

February 13th, 2026

Queen's University - CISC 322

Instructor: Prof. Bram Adams

Table of Contents:

1. Abstract

This report analyzes the conceptual architecture of Gemini CLI, an open-source command-line system that integrates Google's Gemini large language models into developer workflows. It focuses on the system's high-level structure, describing key subsystems such as the Interactive CLI, Core, Tooling and Execution, State and Context Management, and External Model Services, and how they interact to manage input, context, and AI-driven actions.

By emphasizing modular design, layered dependencies, and controlled execution, the report highlights how Gemini CLI maintains extensibility, user oversight, and maintainability. Treating AI reasoning as an external service, the analysis focuses on system organization rather than implementation details, providing insight into how an AI-assisted CLI can be structured for iterative development while remaining adaptable and clear.

2. Introduction and Overview

2.1 Overview of Gemini CLI

The Gemini CLI is an open-source, command-line interface system that utilizes large language models (LLMs). Built around Google's Gemini family of models, the system enables Users to issue natural language commands that can be interpreted, reasoned, and translated into development actions within the User's workspace. Gemini CLI is primarily intended for developers and technical Users who want to augment their productivity using AI-driven assistance without leaving the terminal environment. Its architecture emphasizes safety, transparency, and extensibility, enabling new tools and integrations to be added over time without disrupting existing functionality.

2.2 Purpose of This Report

The purpose of this report is to analyze and document the conceptual architecture of Gemini CLI. The report aims to identify the system's major components, describe their responsibilities, and explain how they interact to support the overall functionality of the software. Additionally, this report serves as an academic exercise for CISC 322, applying architectural concepts to a real-world, open-source system.

2.3 Scope and Limitations

The scope of this report is limited to the high-level architectural design of Gemini CLI. It covers the software's main subsystems, their dependencies, and the architectural patterns used to coordinate User input, AI querying, and tool execution. The report does not provide a full implementation of walkthrough, detailed source code analysis, or performance benchmarking. Low-level algorithms, model internals, or aspects of the Gemini models themselves are considered black boxes and will not be investigated. Furthermore, the architecture described in this report reflects the system as it exists at the time of analysis. As Gemini CLI is an actively evolving open-source project, some components or interfaces may change in future versions.

3. Conceptual Architecture

3.1 Overall Structure of the System

Gemini CLI is an open-source system that enables User's to directly send natural language queries to the Gemini large language models of Google via a command-line interface in their terminal. Conceptually, the system has been structured into a modular, layered architecture with some elements of an implicit invocation architecture and a client-server architecture. Key components are mediated by a central orchestration component that enables the numerous architectural dependencies.

Like any architecture, there are also performance-critical points in the architecture of Gemini CLI. First, and most importantly, due to Gemini CLI's extreme emphasis on LLM's, the model API component is the primary bottleneck. There can be a significant amount of network latency and server-side processing time buildup; nearly any usage of Gemini CLI involves making an API call. Additionally, the Tooling & Execution Layer has numerous, intensive, I/O operations like shell execution and file system traversal. This brings in a second possible hindrance to the architecture of Gemini CLI as a User's System side bottleneck. These key issues have inspired the efficient and modular approach that the Gemini CLI uses.

In the following sections, we will break down and explain the interpreted conceptual architecture of Gemini CLI.

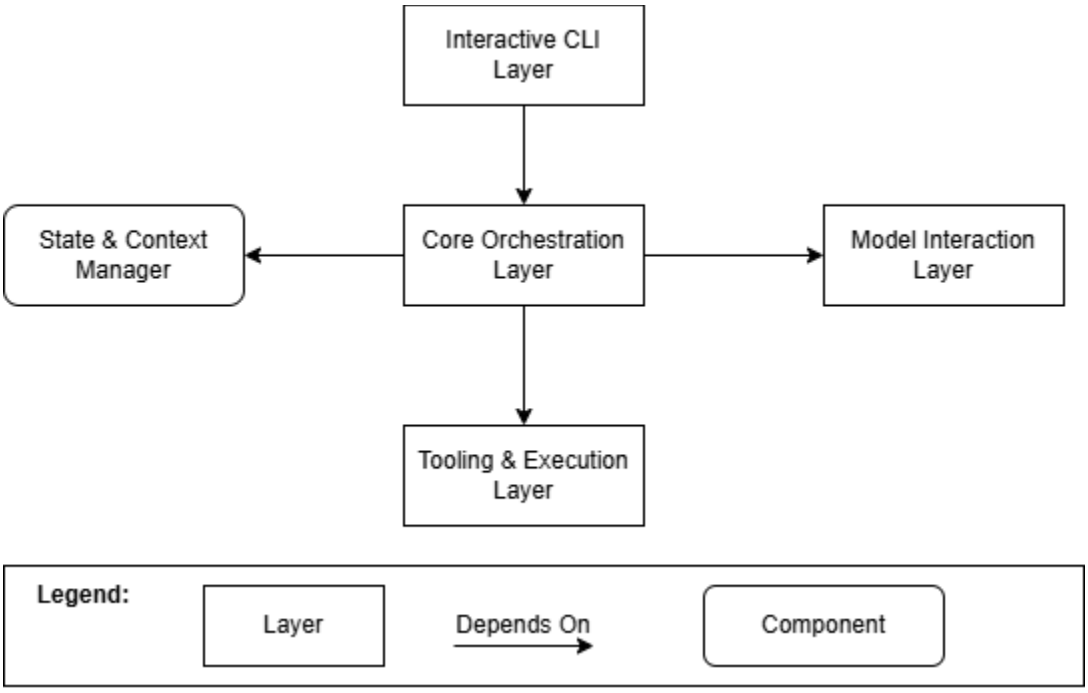


Figure 1: Box and Line Diagram of the Gemini CLI, composed of five layers.

3.2 Key Subsystems and Their Responsibilities

Interactive CLI Layer (External Interface):
The Interactive CLI Layer contains the user interface of Gemini CLI and receives natural language input and commands by the Users and displays the proposed changes, outcomes, and system messages in a readable terminal format. Architecturally the CLI Layer is deliberately not associated with reasoning and execution logic and both sends all requests to the Core Orchestration Layer and displays all responses.

Core Package Layer:
The Core Package Layer is the mind and foundation of Gemini CLI, it manages control flow among subsystems and the lifecycle of User requests. It manages all operations of the system, like interpreting the intent of the User, creating context-based prompts, triggering the Gemini model, communicating model responses with the user via the CLI layer, executing the tools, and maintaining the state and session data. Since it introduces centralization, this layer is performance-sensitive and efficiency is critical for the system to run smoothly, beyond it's major bottlenecks.

Tooling & Execution Layer (External Interface):
The Tooling Layer offers the power of the system to interact with the User's system, by performing actions like access to filesystem, shell code execution, web searching, and third-party tools. The actual tools of this layer are uncovered via standard interfaces, so that the Core Orchestration Layer can be independent of their actual implementations.

State and Context Management Component:
The State and Context Management component is in charge of conversational memory and operational context. This state of being serves as a conceptual blackboard, which allows processes, orchestration and tools coordination. The architecture implements this manager into the core, making this a key component.

Gemini API (External Interface):
The External Model Services layer consists of communication between the Gemini CLI core and external Gemini model services. This layer is limited to interactions with the Core Orchestration layer; it passes adjusted User prompts to the Gemini models and passes the returned information to the core. The simplicity and isolation of this layer is important to maintain secure control on how the LLM interacts with the User's system.

3.3 Dependency Relationships Between Subsystems

Gemini CLI follows a primarily layered dependency structure, where higher-level subsystems depend on the subsystems below them to provide the services and data required for correct operation. This dependency structure enables a clear separation of concerns and improves modularity, aligning with classic layered architectural principles described in software architecture literature (Garlan and Shaw). By doing this, each core subsystem can perform its respective tasks with minimal interference from other subsystems.

With the User and frontend at the top of this hierarchy, a smooth flow of information is possible for each individual User. These layered dependencies align well with how Gemini CLI is used.

This structure reflects the directed dependency constraints typical of layered systems, where higher layers rely on services of lower layers while preserving abstraction boundaries (Garlan and Shaw). A User typically interacts with the system through a typed command in a CLI, which is first handled by the CLI package. The CLI package processes this input and forwards it to the core package. From there, the core package acts as the coordinator of control and data flow. It transforms the given input, uses the transformed input to call the Gemini API and, when necessary, the tool system to construct an appropriate response using the Gemini large language model along with any required tools. All these interactions with the Core package, causes it to require help from the State and Context Management subsystem, where it helps keep memory and constant support. After all that, the final response is then returned and displayed to the User.

In summary:

User → depends on → CLI Package

CLI Package → depends on → Core Package

Core Package → depends on → Gemini API, Tool System, State & Context Management

This interaction model aligns with the behavior documented in the official Gemini CLI repository and documentation (Google-Gemini; "Gemini CLI Documentation").

3.4 Architectural Style(s) and Design Patterns

The conceptual architecture of Gemini CLI can be considered a hybrid of several different architecture styles, as categorized in foundational software architecture literature (Garlan and Shaw). It mainly revolves around a layered approach, with the complimentary styles being the implicit invocation and client-server architectural styles. These styles combined allow for an efficient execution of user requests and the intended modular framework of Gemini CLI.

Layered

Gemini CLI is best understood as a layered system. Its subsystems are organized into distinct layers, each with clear responsibilities. Higher layers depend on the services provided by lower layers, allowing most interactions to flow downward. The layered-style architecture causes fewer dependencies and reduces flexibility for optimization; this separation makes it easier to evolve the employed Gemini model or tooling infrastructure without interfering with User-facing commands, and it improves maintainability by keeping frontend and backend concerns separate.

Implicit Invocation

Implicit Invocation is extremely relevant to one aspect of Gemini CLI, the tool system. Implicit invocation, also referred to as event-based architectural style, allows components to react to events without explicit knowledge of the invoker, increasing extensibility while reducing direct coupling (Garlan and Shaw). Although a lot of aspects can be managed by the core package and Gemini API, it is necessary to call upon a tool system at times to acquire necessary tools. The best approach to this is by using an implicit invocation structure. Actions like model responses, tool execution requests, and tool completion events implicitly trigger processing steps without requiring components to explicitly invoke each other. There are many tools in the tool system

that may be needed, so it can use broadcasting and allow tools to select when they are called upon to complete the required task for each use, allowing for an efficient and accurate use of the tools at the system's disposal. Although there may be some complications introduced like reduced debugging ability, overall, it lines up perfectly with how Gemini CLI wants to utilize its tool system, allowing for the upmost optimized tool executions.

Client–Server

Because Gemini CLI depends on the Gemini API, it is necessary to use a client-server approach. This interaction reflects the client-server architectural model, in which distributed components communicate through request-response protocols across network boundaries (Garlan and Shaw). When working with large language learning models, it is important to have a lightweight client and powerful remote server, despite introducing the need for a dependency on global networks. It runs separately on each User's system, making them the client. This handles prompt input and User interaction. Where on the other hand, the Gemini model backend is the server. As the server, it handles executing largescale model inference, performing reasoning, planning, and tool selection; along with managing the model state, safety layers, and scaling. All together returning structured responses or tool outputs to the client. The connector between the client and server is the network through API calls to Gemini API from the core package, which receives the information from the CLI frontend package. Using this efficient approach, the powerful server behind Gemini powers each client-based prompt highly effectively.

3.5 Alternate Architectural Style(s)

Repository

At first glance, a repository architecture style was strongly considered due to the use of a mono-repository for the codebase of Gemini CLI. However, the is misleading. A monorepo is a version control/organization decision, separate from the ideas of styles present in the Garlan & Shaw ideas. In contrast, a true repository architectural style involves a central shared data store coordinating subsystem interaction (Garlan and Shaw). To expand on that, multiple subsystems coordinate by reading/writing to a shared and central database, which becomes the main mechanism behind the architecture. This simply is not present in Gemini CLI; if it were, one would expect to see all the components interact through a central database instead of direct calls and the request/response control Gemini CLI currently displays. Although having a repository-based styled architecture would simplify interactions and minimize dependencies, it would pose way bigger problems in the form of evolution, consistency, and coordination; especially if concurrency increased.

Pipe-and-Filter

A second alternative architectural style we considered is pipe-and-filter. The pipe-and-filter style structures systems as a series of independent transformations connected by data streams (Garlan and Shaw). Gemini CLI's request handling looks deceivingly like a series of filters and pipes, due to the linear data flow. A data system like pipe-and-filter would provide increased testability and replaceability between the subsystems. However, we chose not to adopt the pipe-and-filter style in our conceptual architecture since Gemini CLI's workflow is not necessarily always a single linear pass of control and data flow. As shown in our use cases, there are circumstances such as the need for the state and context component to constantly send updates to the core, or

the conditional branching that may exist from the tool system. A pipe-and-filter style architecture would add many boundaries and difficulties for these kinds of data and control flow.

3.6 Supporting Future Change and Evolution

The overall architecture of Gemini CLI is designed for supporting incremental evolution due to its clear subsystem boundaries and layered dependencies. The encapsulation of the Gemini API with a dedicated subsystem allows the complex and constantly changing model versions or backend interfaces to be altered at any time with minimal effects on other layers such as the CLI or Tool System layers. Whereas the separation between User interface and the core orchestration allows for possible alternate interfaces like a Graphical User Interface (GUI) to be introduced without needing to redesign most core reasoning components. Additionally, the use of a distinct Tool System allows for any new tools to be added without system interference. However, because orchestration is centralized within the Core Package Layer, maintaining modular boundaries within this layer is critical to prevent architectural complexity from accumulating over time. Despite that, Gemini CLI's overall structure supports significant possibilities for growth and evolution in the future, as is necessary with an AI-focused application.

4. External Interfaces

4.1 User Interface

Gemini CLI utilizes a command-line User interface that operates within the User's terminal environment using standard input and output streams. Users issue commands and prompts through textual input and receive response messages that are rendered as output. The interface also supports asynchronous User control, allowing for intervention mid-query. This enables responsive User interaction without adding more complicated external User-facing interfaces.

4.2 Gemini API Interface

Gemini CLI operates as an executor and medium for terminal interaction with different Gemini LLMs; as such, the system utilizes an external Gemini API component. This interface is responsible for transmitting altered User prompts, contextual data, and configuration parameters to the remote LLM, and receiving responses in return. The communication flow follows a request and response pattern, where the core requests that the LLM work over the provided information and then interprets the response. By delegating the LLM interaction to the API component, the CLI maintains a lightweight and modular framework.

5.3 Gemini CLI Tools Interface

The Gemini CLI's heavy focus on terminal execution is handled by the external tools interface. This interface enables Gemini CLI and the local system resources to interact with. Through this interface, the Gemini CLI can read, write, and modify files, execute User-approved scripts, and incorporate local context into prompts to send to Gemin. This interface acts as the boundary between proposed AI driven actions, and their execution.

5. Use Cases

5.1 Use Case 1 Overview: Applying a Specific Model to Gemini CLI

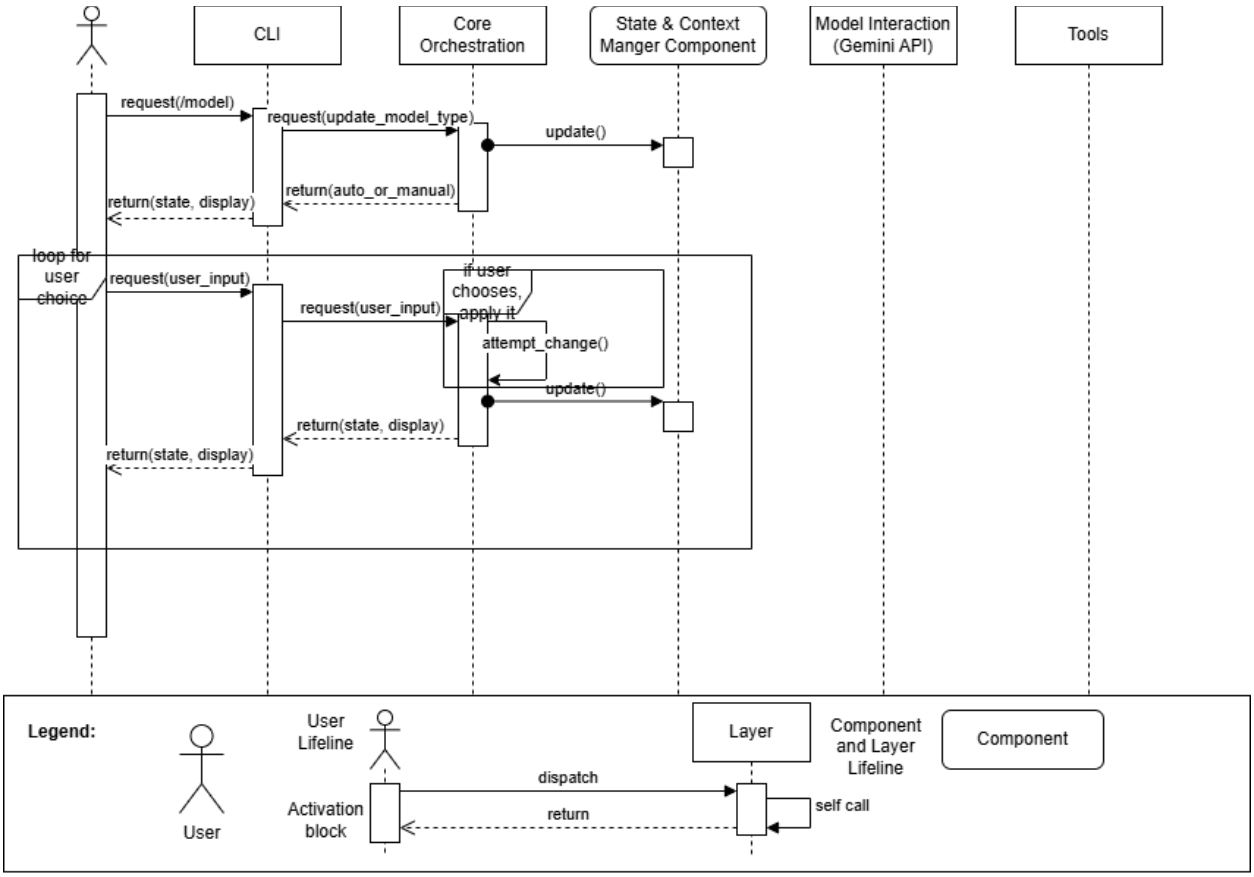
This use case models how a User applies a specific model configuration within Gemini CLI through the built-in settings interface. Instead of selecting a model at query time, the User modifies the session's default model by navigating a local configuration workflow. The purpose of this interaction is to demonstrate how the CLI interprets user preferences, presents available configuration modes, and updates the active session settings without invoking external services.

5.2 Process Explanation – Applying a Specific Model to Gemini CLI

At the beginning of the interaction, the User issues a command to modify the active model configuration. The CLI layer receives this request and invokes its Core Orchestration layer, which is responsible for retrieving and updating session configuration values. The Core provides the CLI with two configuration modes, manual selection or automatic selection. The system then enters an interactive selection flow in which the CLI and User exchange messages as the User navigates the model configuration options.

After the User selects manual configuration, the CLI forwards this choice to the Core. The Core responds by supplying a list of supported models that can be explicitly set. These options are displayed by the CLI, and the User selects the desired model. Once a valid selection is made, the Core updates the session configuration and the settings interface exits, returning the CLI to it's standard command prompt state. This interaction is handled entirely within the local Gemini CLI architecture and does not invoke external LLMs or Tools.

5.3 Sequence Diagram – Applying a Specific Model to Gemini CLI



Figrue 2: Use Case 2 Sequence Diagram: Adjusting the specific type of Gemini model used for the session.

5.4 Use Case 2 Overview: Fix Unit Testing

A developer has created a piece of software with certain functions critical to it's operation and a file for unit testing; the developer would like Gemini CLI to edit the project code so that it passes the unit tests. The goal is to adjust all of the code in the project so that the unit tests are passed.

5.5 Process Explanation – Fix Unit Testing

At the beginning of the interaction, the Developer prompts Gemini CLI to debug a project based on the results of a provided unit testing file. The CLI forwards the request to its Core layer, which initiates the process for querying Gemini. The Core updates the State Manager and invokes internal tooling to gather relevant system information and project context, like source files and tests to run. This contextual data is combined with the User's request to construct an engineered prompt.

The structured request is then sent to the Gemini model for analysis. Once a response is received, the core updates the State Manager and presents an initial proposed plan describing how the project can be modified to satisfy the unit tests to the user. The system then enters an iterative interaction cycle involving the User and the system. The model proposes specific code modifications or actions, and each proposed step requires User approval before execution (if it requires certain tools). When approval is granted, the CLI forwards the instruction to the Core,

which uses the tooling component to apply code changes and execute relevant operations. The system state is updated after each modification, and the updated context is sent back to the Gemini model for further analysis if necessary.

This cycle of proposal, approval, execution, and state update continues until the unit tests pass, or no further corrective actions are identified. Once the task is complete, the CLI presents a summary of the applied changes and returns to its standard command interface.

5.6 Sequence Diagram – Fix Unit Testing

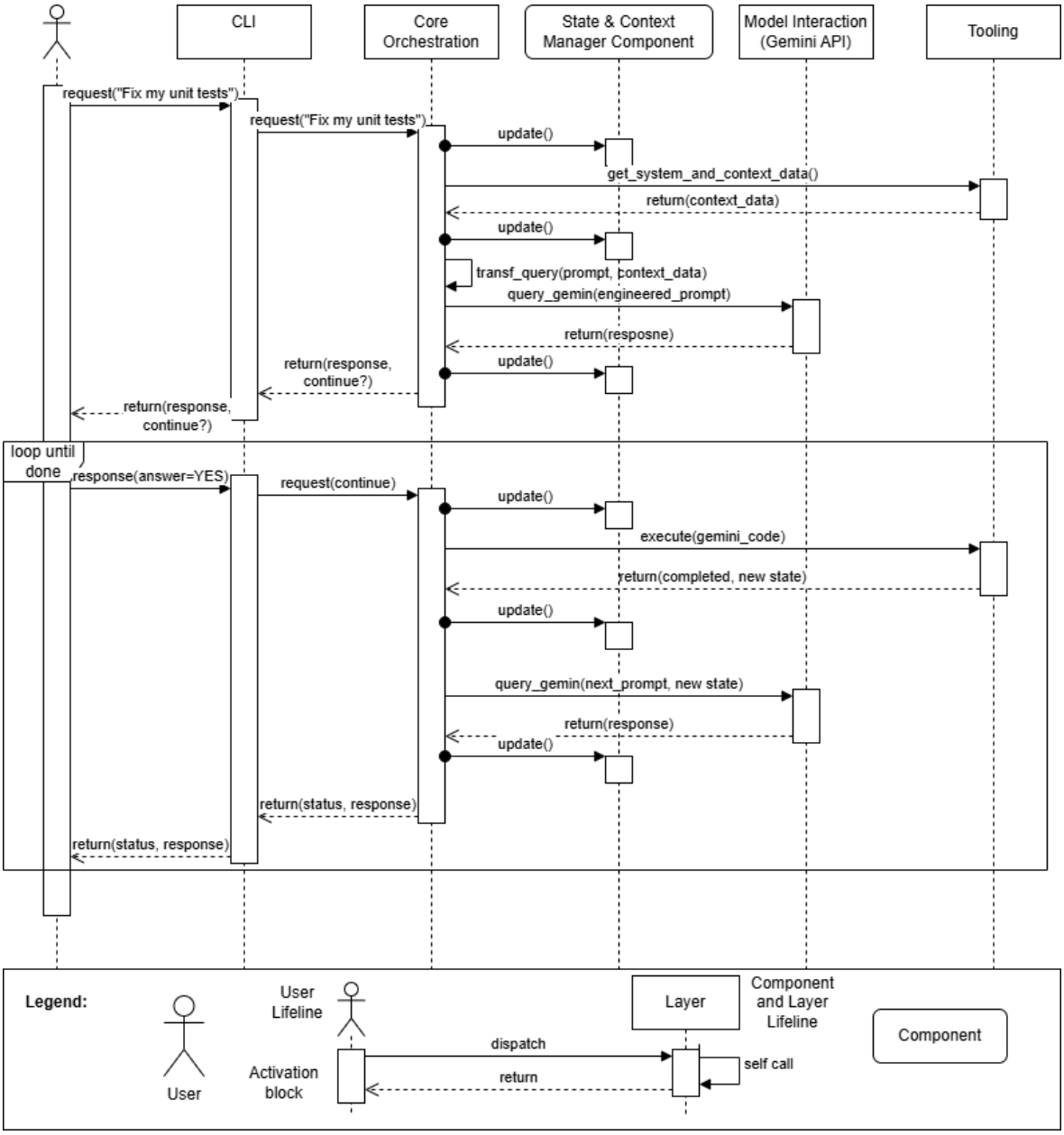


Figure 3: Use Case 2 Sequence Diagram: Adjusting project code so that all unit tests are passed.

6. Data Dictionary

Interactive CLI Layer:
The frontend subsystem that is responsible for capturing user input from the terminal and presenting responses and system messages to the user.

Core Orchestration Layer:
The central coordinating system that manages control flow, prompt construction, model interaction, safety checks, and tool invocation.

State and Context Management Component:
The subsystem responsible for storing conversation history and task context used to support multi-step reasoning and workflows.

External Model Services:
The layer that sends engineered prompts to the Gemini LLMs and receives responses.

Engineered Prompt:
A structured representation of the user's request combined with contextual information which is sent to the Gemini API.

Tool:
A discrete capability (e.g. file access, file writing, etc.) that can be used by the Tooling Layer to perform actions in the user's environment.

7. Naming Conventions

CLI (Command-Line Interface):
A text-based user interface that allows Users to interact with Gemini CLI by typing commands in a terminal environment.

LLM (Large Language Model):
A machine learning model trained on large datasets that performs reasoning, text generation, and analysis tasks used by Gemini.

API (Application Programming Interface):
A structured interface that enables Gemini CLI to send requests to external model services and receive responses programmatically.

8. Conclusions

In conclusion, Gemini CLI is a robust system that effectively integrates AI reasoning into developer workflows. It achieves this through a hybrid of layered and modular architectural styles, ensuring both flexibility and maintainability. This combination allows the system to remain well-structured and scalable.

Our analysis demonstrates how these components work together to manage user input, maintain context, coordinate tool execution, and interact with external AI services. By examining the relationships and dependencies between subsystems, this report provides a clear conceptual view

of Gemini CLI's architecture, highlighting how it supports controlled execution, extensibility, and iterative workflows while remaining adaptable for future development.

9. Lessons Learned

Working on the Gemini CLI project helped us understand how AI-assisted development tools are structured around controlled execution and structured context. A key lesson was that the model does not simply "fix code" independently - the CLI gathers project data, test results, and system state before constructing a request. This showed us that system effectiveness depends heavily on how context is collected, validated, and maintained. Studying this process changed how we interpreted component responsibilities in our sequence diagrams, particularly the roles of the Core, State Manager, and Tooling components.

We also learned that AI-driven workflows are inherently iterative and require careful modeling. Many interactions involved approval loops, repeated analysis, and state updates rather than a single request-response exchange. From a team perspective, the project emphasized planning and resilience. The work was designed for five to six members, but our group consisted of four, and near the final stage of this report one member became seriously ill and could no longer contribute. As a result, we redistributed responsibilities, prioritized core deliverables, connected with the teaching team, and relied on shared documentation to maintain consistency across diagrams and written sections. This experience reinforced the importance of clear task ownership, early coordination, and contingency preparation when working on complex technical projects. Overall, the project demonstrated how modular architecture and user-controlled execution enable practical integration of AI capabilities.

10. References

Gemini CLI Documentation. *Gemini CLI*, https://geminicli.com/docs/.

Google-Gemini. *gemini-cli*. GitHub, https://github.com/google-gemini/gemini-cli.

Google-Gemini. *ROADMAP.md*. GitHub, https://github.com/google-gemini/gemini-cli/blob/main/ROADMAP.md.

"Gemini CLI Demo." *YouTube*, uploaded by Van Amersfoort, YouTube, 2025, https://www.youtube.com/watch?v=QzJufbGhTeI.

"Gemini CLI Hands-On." *Google Codelabs*, https://codelabs.developers.google.com/gemini-cli-hands-on#0.

Jalateras, Jon. "Unpacking the Gemini CLI: A High-Level Architectural Overview." *Medium*, 2026, https://medium.com/@jalateras/unpacking-the-gemini-cli-a-high-level-architectural-overview-99212f6780e7.

Virtuslab Team. "How Claude Code Works: An Inside Look at the Textual UI Agent." *Virtuslab Blog*, https://virtuslab.com/blog/ai/how-claude-code-works/

Garlan, David, and Mary Shaw. *An Introduction to Software Architecture*. Carnegie Mellon University, Jan. 1994, CMU-CS-94-166.

11. Derivation Process

We began the architectural derivation by studying the system's domain and documentation to understand how Gemini CLI supports AI-assisted development workflows. From this analysis, we identified recurring user goals and operational patterns, which were then formalized into representative use cases. These use cases captured key system behaviors such as configuration management and AI-driven code modification. We translated each use case into a sequence diagram to model dynamic interactions between system components. Constructing these diagrams helped reveal which components were responsible for coordination, state handling, external communication, and tool execution, and clarified how information and control flow through the system during realistic usage scenarios.

Using the components and interactions identified in the sequence diagrams as a foundation, we synthesized a high-level box-and-arrow conceptual architecture. Components that repeatedly appeared across use cases were abstracted into top-level subsystems, while message exchanges were generalized into static dependency relationships. This process ensured that the final architecture reflects observed system behavior rather than assumptions about implementation. The layered organization emerged from grouping components according to their responsibilities and direction of dependency, while external interfaces were isolated based on their role as system boundaries. Through this iterative process—from domain understanding to dynamic modeling to structural abstraction—we derived a conceptual architecture that explains both how Gemini CLI operates and why its subsystem structure is organized as presented.

12. AI Collaboration Report

12.1 AI Model Selection

For Assignment 1, our team selected OpenAI GPT-5 (January & February 2026 version) as our virtual AI teammate. We considered many LLM models including Gemini itself but ultimately decided on GPT-5 for several reasons.

To start the selection process, we decided on three main AI companion options: Claude, ChatGPT, and Gemini. Our main concern was model power and the cooperability that the model may provide for our group. This was of utmost value because working coherently is the most important aspect of a group-based deliverable. If there are any miscommunications, disagreements, or inability to properly divide work, it can quickly become overwhelming and difficult. After review of the three different models, it revealed that ChatGPT offered collaborative group chats where all members of our group could interact with the same AI companion and utilize the same chatlogs as each other. Gemini and Claude did not offer a feature like this.

One of our members had access to the paid GPT-5 version of OpenAI's ChatGPT. This presents itself as higher usage limits during drafting sessions, support for document and image uploads, and improved handling of long-form structured outputs. Our group has also lacked two members since creation, so the extra performance provided by a paid LLM is valuable to our group. All our group members were familiar and comfortable with the use and capabilities of ChatGPT. These collective reasons of collaborative stability, structured drafting capability, accessibility,

and workflow efficiency led to the selection of OpenAI's GPT-5 as our AI companion for the project.

12.2 Task Allocation Between Humans and AI

Reorganization of rubric and template creation

Due to the awkward formatting of the rubric, it made it difficult to read. However, unlike humans, AI only takes in text, so it would easily be able to convert the rubric into something more readable. Building off of this, it is now in the perfect situation to help us create headers around this rubric and the given appendices.

Organization and delegation of tasks

Unlike a human, AI is much more capable of holding information and being able to plan ahead because of that. Due to this, it is appropriate to get our AI teammate to help organize positions and how work should be divided so that members are treated equally.

Analysis and suggestions on conceptual architecture choices

With the ability to insert our project deliverable documents and lecture texts, AI can be asked to use that specific information to help us make specific choices on our architecture with things such as picking certain architectural styles.

Review and formatting suggestions

Once again because AI can hold information like the rubric and appendices, it allows us to quickly scan our report document to find weak points or missing information that is suggested or required from us.

12.3 Interaction Protocol and Prompting Strategy

Overall, we decided to prompt our AI teammate as a collective. The group would briefly discuss the objective of every prompt to ensure alignment and avoid inconsistent outputs. Someone would come up with an idea for a prompt, another would write it out in the AI-based group chat, and the others would review and agree or make appropriate changes until the prompt was ready. A critical prompt we used was in conduction of project-objective document uploads: "Using the rubric, appendix, and course slides provided, help us determine key architectural styles used in Gemini CLI. Use logical justification." The biggest qualitative change we found in our prompts was not in changing the prompts themselves but providing context with the use of the course material and assignment rubric and appendices. As shown in the above example, without changing the prompt much, it provided GPT-5 with the context and information necessary to help us make critical decisions based on what we are expected to learn from and submit.

12.4 Validation and Quality Control Procedures

We took validation and quality control very seriously, as we know AI is prone to make errors, and these errors often look convincing. Although we had the AI aid us in giving suggestions of ideas and creating a template, we always made sure to cross check it. Because we submitted specific documents for GPT-5 to use, we compared the output GPT-5 provided us with the source material and primary sources, such as the rubric, appendices, and lecture slides. If we could not find a connection, and GPT-5 could not point us to where the information was found, we would completely void the prompt and output to try again.

12.5 Quantitative Impact of AI Teammate

We estimate that GPT-5 contributed approximately 40% of the drafting support for the deliverable, primarily through rubric reorganization, template structuring, brainstorming architectural style candidates, and formatting suggestions. However, all architectural decisions, subsystem definitions, dependency reasoning, and final written explanations were reviewed and approved by us. Overall, while GPT-5 accelerated drafting and organization, its overall intellectual contribution was closer to 15%.

12.6 Reflection on Human-AI Team Dynamics

Overall, our AI teammate positively contributed to our project through immense help in initial drafting and structuring our work around the deliverables. However, its involvement also required careful validation and prompt refinement. Some outputs were overly confident or slightly misaligned with course expectations, but these outputs were minimized with our strategy of the inclusion of course documents with every prompt. The group prompting process improved internal communication, since we clarified our reasoning before submitting prompts. We learned that AI is most effective as a structured drafting and brainstorming assistant, and at the end of the day, architectural judgment and final decisions are best done by us.