GEMINI CLI

CONCEPTUAL ARCHITECTURE ANALYSIS

Anthony Qiu - Leader - anthony.qiu@queensu.ca

Aaron Ouyang - Presenter - 22sxb5@queensu.ca

Edward Li - Presenter - 22VB67@queensu.ca

Solomon Douglas - 21sd77@queensu.ca

Babacar Faye - 22bf4@queensu.ca

Teddy Wei - 22XM30@queensu.ca

February 13th 2026

Introduction

The purpose of this report is to present an in-depth conceptual architecture of the AI agent and developer helping tool Gemini CLI, which will consist of an analysis of its design, its architectural style and its components and their interconnections. This report will aid in giving a better understanding of the grand scheme of the coding agent for any user or stakeholder.

Gemini CLI functions as an intelligent development assistant capable of debugging, suggesting improvements, selecting and executing appropriate tools, and streamlining the overall development workflow.

Gemini CLI is an intelligent, terminal-first AI agent that assists developers by reasoning over their codebase, planning multi-step actions, and executing them using a variety of tools. Rather than being a simple question-answering tool, Gemini CLI acts as a collaborative agent that helps refine, develop, and execute the user's ideas efficiently within their development environment. Most importantly, it focuses on a high level of structure and reasoning patterns.

This report will first analyze the overall architecture of Gemini CLI by listing its ten major components and the way they interact with other moving parts of its system. The different parts of Gemini CLI will be ranked as order of importance for the system to understand where and why certain components have been optimized to be as performant as possible. Additionally, it will be important to know about the goals of such components and their interconnections while also mentioning the requirements that they must fulfill to please whatever developer may enquire about the services of Gemini CLI. The primary style used by Gemini CLI being the layered style architecture, it will be important to denote how some components are hierarchised and interact with others. The flow of data and the concurrency between the parts of Gemini CLI will also be observed to gain a better understanding of the intended design.

Subsequently, to help visualize our conceptual architecture analysis of Gemini CLI, we will use a sequence diagram and a box-and-lines diagram. It will be primordial to understand the basis of how components work in a simple manner to give a general idea about the system, however, a box-and-lines diagram gives an extremely vague and unconcise idea of the workings of Gemini CLI. Therefore, we will use sequence diagrams to showcase the temporality within which the components act together.

Furthermore, within this paper, we will explore the ways in which the system transmits and receives information. We will notice that Gemini can receive low level encoded information such as natural language from the developers as well as high level information from the files, network and overall environment of the developer. In return, Gemini can emulate and transmit natural human language to help the developer but can also communicate with tools and databases to help the developer in their tasks.

It will also be important to give examples of ways the developer may interact with the system as well as the way the servers for Gemini CLI respond and treat information that is shared. Moreover, since Gemini CLI is a highly used system, this part will help understand better the sequence diagram but also the requirements of the architecture to serve and aid as many users as possible. Concurrency will be one the most important part of our paper, showcasing the performance needed to run a system like Gemini CLI to serve as many users as possible within deadlines and appropriate response times.

Within this paper it will be important to explain the terms and conventions used to better explain the process of Gemini CLI to any stakeholder or probable user. Furthermore, the coding agent will also need to have a defined "personality" for the naming conventions which define it. It is important to note the ways and styles in which such a system is created to have a better understanding of its architecture.

Penultimately, we will explain the findings of this paper in a clear and concise manner as a sort of summary to understand why and how Gemini CLI works so well as a coding agent for developers. Furthermore, it is primordial to explain why this project has given us a better understanding of conceptual architecture as a whole

Finally, we will explain the ways in which the AI member of our group has helped us with this process. We used it not as a tool to do additional work but as an existing member that hashed out certain of our ideas.

Abstract

This paper analyses the conceptual architecture of Gemini CLI. There are ten major components listed with a main system style and some subsystem styles. The main architecture of the system will be a layered style architecture with components getting data and responses of components below and above them in the hierarchy, it is divided into four layers and some components will be wired to ignore said hierarchy for ease of information flow. We will also explore the way the systems control the flow of data and the concurrency within the system and different subsystems to treat demands as fast as they come while respecting performance requirements such as deadlines and response time but also other requirements the system may need to respect.

There are a total of four subsystems within Gemini CLI which are: Entry & Session Control subsystem, Core Reasoning Loop subsystem, Execution & Capabilities subsystem and the Extensibility & Tools subsystem. We used a box and arrow diagram as well as a sequence diagram to signify the interactions the user has with the system and how the different components work together to respond to demands and the timeframe within which each demand between user and system and between system and system is treated.

We used two use cases within the paper. The first one was for the user asking the Gemini CLI a question and the program answers. The second one is the user asking the bot to use a tool and the execution. In both use cases, we will analyze the goal, the steps taken by the program and the interconnections between components necessary to achieve the goal as well as the possible limitations and points of failures of the program.

We also made sure to include a dictionary for ease of understanding of certain terms within the Gemini CLI.

For this paper our 7th teammate was the ChatGPT chatbot 5.2. His main role was to bring clarifications over certain parts of our assignment. He was of great help during processes such as the architectural style clarification, the Subsystem responsibility explanation, the Control flow and concurrency clarification and the Validation and Quality Control Procedures. Its role was not to give out completely new information but to hash out our ideas and we estimate that around 15% of our work was possible with its help.

Architectural Style and Interacting Parts

Gemini CLI is a terminal-based AI assistant for development work. Its main function is to help users move from a request to an outcome in one continuous workflow: the user asks for analysis, code changes, or task execution; the system reasons over project context; and the result is returned in the terminal. It supports both interactive use and script-oriented non-interactive use, but the underlying purpose is the same in both modes: coordinate language-model reasoning with controlled tool use so that answers are not only descriptive, but actionable.

The main alternatives considered were repository and client-server, because both describe real parts of the system. Repository was considered because shared session and conversation state is central to continuity across turns. However, repository is better seen as a supporting data-organization pattern than the dominant system structure, since it does not explain the separation between user interaction, orchestration, execution control, and integration. Client-server was considered because Gemini CLI communicates with external services such as model APIs and MCP servers. However, that view mainly describes external communication boundaries, not the internal architectural organization of the CLI itself. For the system-level decomposition required in this section, layered architecture remains the strongest primary style.

Other styles are present in limited ways but were not selected as the primary architectural description. Repository style appears through shared conversation/session state, but state sharing alone does not define the whole system structure. Pipe-and-filter does not fit well because Gemini CLI is not a fixed linear transformation pipeline; it uses iterative loops with decision points and tool callbacks. Client-server applies to specific integrations such as model APIs and MCP servers, but it does not capture internal decomposition. Peer-to-peer is not suitable because control is centrally orchestrated rather than distributed among equal nodes. Object-oriented design is used in implementation, but that is a programming paradigm, not the dominant system-level style for this report. Implicit invocation is also used in event handling, but event mechanisms support the architecture rather than define its primary structure. For the overall system view required in this section, layered architecture is therefore the most accurate and defensible primary style.

At a high level, the architecture can be understood as ten cooperating components. The first group is the user-facing control path, which includes argument and settings resolution, interactive session control, and headless session control. These parts decide how the session should run, what capabilities are enabled, and how results should be presented. Their responsibility is not to perform reasoning itself, but to shape the operating context for the core system and keep input and output consistent for the user.

The second group is the core orchestration path, centered on runtime configuration, chat state management, and turn orchestration. Runtime configuration assembles the system's active services and policies for a session. Chat state management maintains continuity across turns so the assistant can make coherent decisions over time. Turn orchestration then drives each request cycle by preparing context, invoking the model, receiving streamed responses, and deciding when the loop is complete or when additional actions are needed. Together, these parts provide the stable control loop of the architecture.

The third group is the action path, formed by capability registration and policy-mediated tool execution. Capability registration defines which operations the assistant can request, such as file access, search, shell commands, and web-related operations. Policy-mediated execution determines which requested actions are allowed immediately and which require explicit user confirmation. This separation is important architecturally because it keeps “what can be done” distinct from “what is allowed right now,” which improves safety and keeps behaviour predictable.

The fourth group is the extensibility path, consisting of MCP capability integration and dynamic composition through extensions, skills, agents, and hooks. MCP integration allows external tool servers to add capabilities without changing the main loop. Extensions and skills provide structured ways to add reusable behaviour and domain-specific guidance. Agents and hooks allow specialized behaviours and lifecycle customizations to be introduced while still operating inside the same control structure. This means new capabilities can be introduced without redesigning the core architecture.

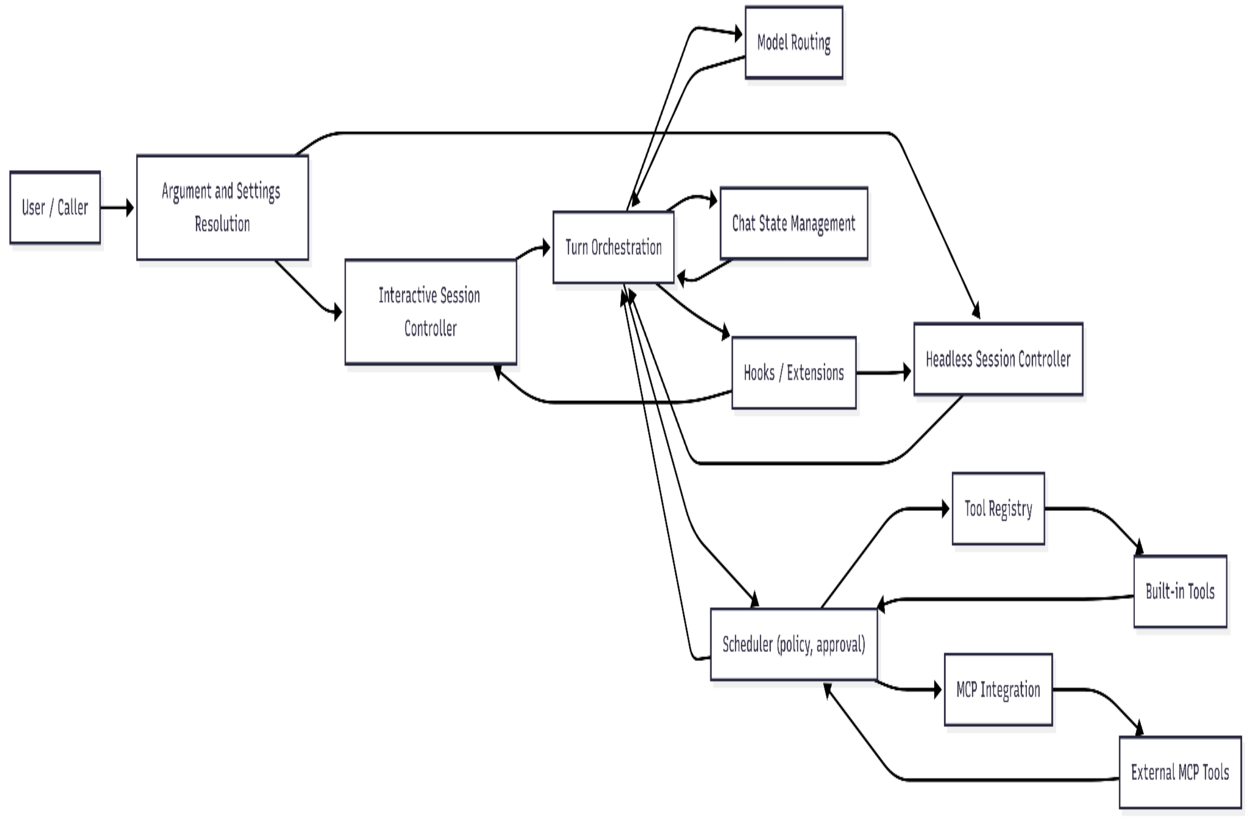
Interaction among these parts is iterative and feedback-driven. A user request enters through interactive or non-interactive control, passes into orchestration with current settings and context, and produces either direct model output or action requests. When actions are requested, the execution path evaluates and performs them under policy rules, then feeds the results back into the next reasoning step. The process repeats until the system reaches a final response or a controlled stop condition. Conceptually, this is a single loop with specialized responsibilities distributed across components rather than a set of disconnected pipelines.

From an evolution perspective, the architecture is designed to change by extension, not by replacement. Model-facing behaviour evolves through configurable model and routing policies. Action capability evolves through new registered tools and externally discovered MCP tools. behavioural capability evolves through skills, extensions, hooks, and specialized agents. Because these changes are incorporated through existing registries and lifecycle mechanisms, the system can grow in capability while preserving the same high-level operating model.

This evolution model matters because the system is expected to adapt in several directions at once: model options may change, external integrations may grow, and workflow expectations may become more specialized over time. A design that required central rewrites for each of these changes would become fragile. Instead, Gemini CLI evolves through additive mechanisms that are intentionally aligned to subsystem boundaries: model-related changes stay in model and routing configuration, capability-related changes stay in tool and integration layers, and behaviour-related changes stay in skills/agents/hooks. The architectural benefit is that growth in one area does not force instability in unrelated areas.

This structure addresses the required architectural questions for this part of the assignment. The system's functionality is clearly defined as request-to-outcome orchestration in the terminal; the major parts are identifiable at subsystem level; interactions are described as a controlled reasoning-and-action loop; and evolution is explained through compositional extension points rather than low-level implementation rewrites.

Box and arrow diagram



Flow, Control, and Data Movement

At a conceptual level, Gemini CLI operates as an agentic interaction loop in which control moves across the ten cooperating components previously identified. Rather than executing actions directly from model output, the system routes all reasoning, configuration, and capability execution through a mediated orchestration pipeline. Each interaction turn begins at the CLI entry layer, passes through configuration and orchestration subsystems, and returns structured output through either the interactive or headless controller. This layered flow preserves a clear separation between reasoning, execution, and extensibility while maintaining a unified control path.

Data flow:

1. User input and runtime initialization
   (Argument & Settings Resolution → Interactive Session Controller)
   When running in interactive mode, argument and settings resolution establishes execution constraints before control is passed to the interactive session controller. The controller mounts the terminal interface, manages streaming UI state, and forwards structured requests into the backend orchestration pipeline.

2. Headless execution initialization
   (Argument & Settings Resolution → Headless Session Controller)
   In headless mode, argument and settings resolution routes execution to the headless session controller instead of the interactive controller. The headless controller executes the same agent loop but produces structured output rather than maintaining persistent terminal state. Despite different entry behaviour, both controllers delegate to the same backend components for reasoning and tool execution.

3. Configuration and context assembly
   (Session Controllers → Runtime Configuration Kernel → Chat State Management)
   Regardless of entry mode, the runtime configuration kernel provides active registries, policy infrastructure, and routing services. Chat state management then maintains conversation history and contextual information required for the upcoming reasoning turn.

4. Turn preparation and orchestration
   (Chat State Management → Turn Orchestration)
   Turn orchestration constructs a model request using system instructions, contextual history, and schemas exposed through capability registration.

5. Model interaction and event generation
   (Turn Orchestration → Model Routing → Turn Orchestration)
   The orchestrator sends structured requests to the model and receives streamed content or tool-call intents, which are converted into typed events that determine subsequent control flow.

6. Policy evaluation and control transfer
   (Turn Orchestration → Scheduler)

When tool execution is requested, the scheduler evaluates approval rules, applies trust constraints, and determines whether execution proceeds automatically or requires confirmation through the interactive controller.

7. Capability execution
(Scheduler → Capability Registration / MCP Capability Integration)
Approved built-in tools execute through registered capabilities, while external capabilities are accessed through MCP integration. Both remain mediated by the scheduler.

8. Iterative reasoning loop
(Scheduler → Turn Orchestration → Chat State Management)
Tool results return to orchestration as structured responses and are incorporated into the next reasoning step until completion or policy stop conditions are reached.

9. Streaming output rendering
(Turn Orchestration → Extensions/Hooks → Interactive Session Controller)
During interactive sessions, lifecycle hooks may intercept streamed events before output is rendered incrementally in the terminal interface.

10. Structured output finalization

    (Turn Orchestration → Extensions/Hooks → Headless Session Controller)
    During headless execution, lifecycle hooks process final events before structured output is returned to the calling environment.

External Interfaces:

Gemini CLI interacts with several external entities to fulfill its functional requirements. These interfaces define the boundaries of the system:

1. User Terminal (Standard I/O): This is the primary interface for user interaction. The system reads natural language prompts from Standard Input (stdin) and renders streamed text responses, loading spinners, and interactive confirmation prompts to Standard Output (stdout).
2. Local File System: The system requires Read and Write access to the local file system. This is used to read project files to provide context to the AI model and to write changes to code files when directed by the user.
3. Gemini API (Network): The system communicates with Google's backend servers via a secure HTTP/gRPC interface. This interface is used to send context windows and prompts to the large language model and receive reasoning tokens and tool calls in return.

4. MCP Servers (Network/IPC): The system interfaces with external Model Context Protocol (MCP) servers. This allows the CLI to discover and execute tools that are not native to the CLI, expanding its capabilities through a standardized protocol.
5. Operating System Shell: The system maintains an interface to the host operating system to execute shell commands directly, allowing it to perform system-level tasks such as installing dependencies or running tests.

Concurrency:

From the user’s perspective, Gemini CLI behaves as a single interaction loop; however, several conceptual forms of concurrency arise from streaming communication and extensibility mechanisms.

1) Streaming model output

Turn orchestration emits incremental events that session controllers render progressively. Rendering, lifecycle hooks, and reasoning may therefore occur simultaneously within a single interaction turn, introducing pipeline-style concurrency.

2) Tool execution and external processes

Scheduler-mediated execution may spawn subprocesses or communicate with MCP servers operating independently from the main agent loop. These concurrent activities allow model interaction, network communication, and tool execution to overlap while remaining coordinated by orchestration.

3) Extensions and lifecycle hooks

Hooks introduce asynchronous behaviours that operate alongside the primary control path. Because they modify lifecycle events rather than execution logic, they do not block interaction completion or shutdown.

Implications for Division of Responsibilities:

<u>CLI Controller Responsibilities</u>
Developers responsible for interactive and headless controllers manage input handling, streaming output, and confirmation workflows while remaining decoupled from orchestration logic.

<u>Core Configuration and State Responsibilities</u>
Contributors maintaining the runtime configuration kernel and chat state management ensure consistent registries, policies, and session continuity across turns.

Orchestration Responsibilities
Developers working on turn orchestration coordinate model communication, lifecycle events, and control transfer between reasoning and execution phases.

Capability and Tool Responsibilities
Capability registration and tool developers define schemas, validation behaviour, and functional logic while relying on scheduler mediation for execution safety.

Policy and Scheduling Responsibilities
Scheduler developers enforce approval rules and maintain safe integration of new capabilities within a centralized governance layer.

Extensibility Responsibilities
Developers working on MCP integration, extensions, skills, agents, and hooks expand system behaviour without modifying the core agent loop.

Use Cases, Data Dictionary, and Sources

Use Case 1: User asks Gemini CLI a question and gets an answer

Goal: Get an answer from Gemini CLI for a question.

Trigger (what starts it): User types a prompt or question into Gemini CLI and presses Enter.

Before it's used, you need: Gemini CLI installed, internet connection, and valid authentication (API key or login).

Steps:

1. User opens a terminal (Command Prompt, PowerShell, or Terminal).
2. User runs Gemini CLI and types a prompt.
3. Gemini CLI reads runtime settings (for example: model and configuration).
4. Gemini CLI sends the prompt to the Gemini model.
5. The model generates a response and sends it back.
6. Gemini CLI renders the answer in the terminal.

Result (what the user gets): The answer appears in the terminal interface.

What can go wrong:

- No internet connection, so the model cannot be reached.
- Missing or invalid API key/login token, so the request fails.
- Rate limit or server error, so a response is not returned.

<u>Use Case 2: User asks Gemini CLI to use a tool and gets an answer</u>

Goal: Get an answer that requires Gemini CLI to invoke a tool (for example: reading a file or running a command).

Trigger (what starts it): User sends a request that requires tool execution and presses Enter.

Before it's used, you need: Gemini CLI installed, internet connection, valid authentication, and permission to access the requested tool.

Steps:

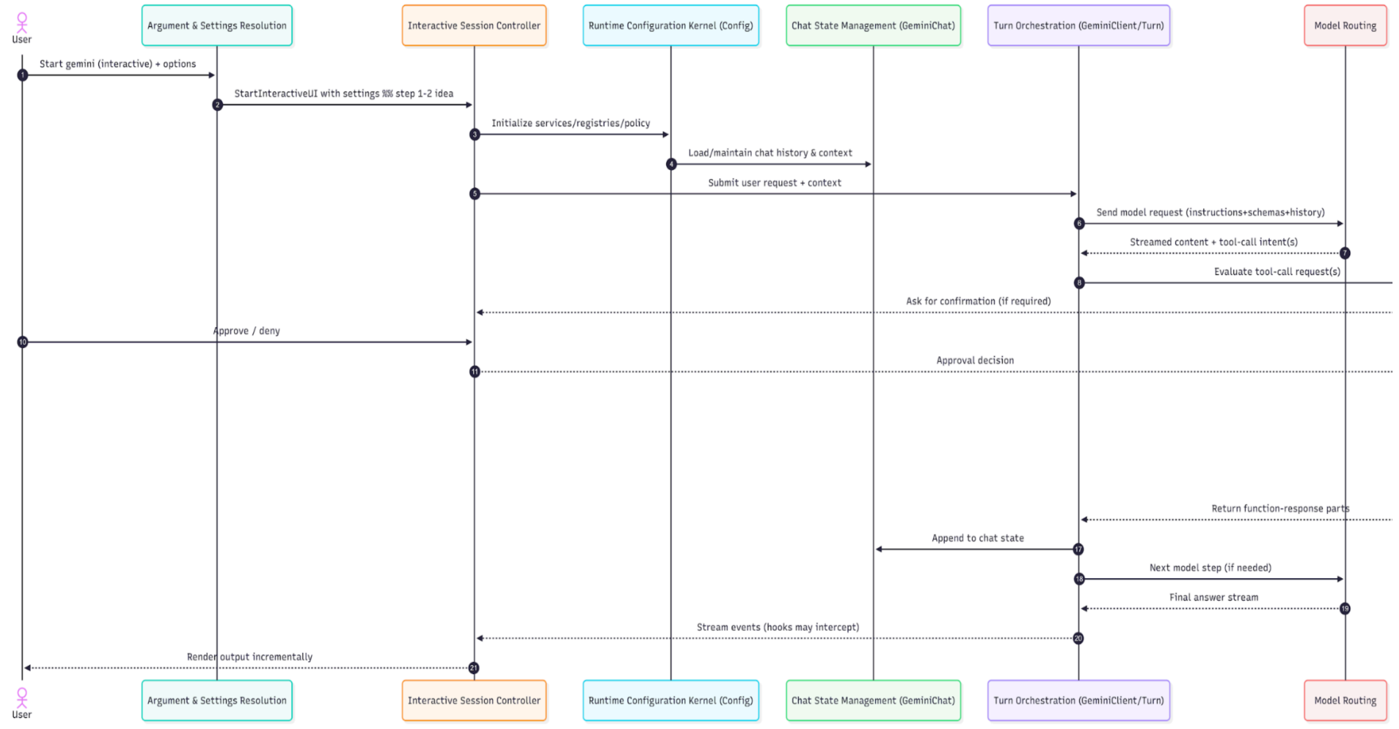
7. User opens a terminal and runs Gemini CLI.
8. User enters a request that needs a tool (for example: summarize a file).
9. Gemini CLI forwards the request to the Gemini model.
10. The model decides a tool is required and issues a tool call.
11. Gemini CLI executes the tool and returns the tool output to the model.
12. The model generates the final answer using tool output, and Gemini CLI prints it.

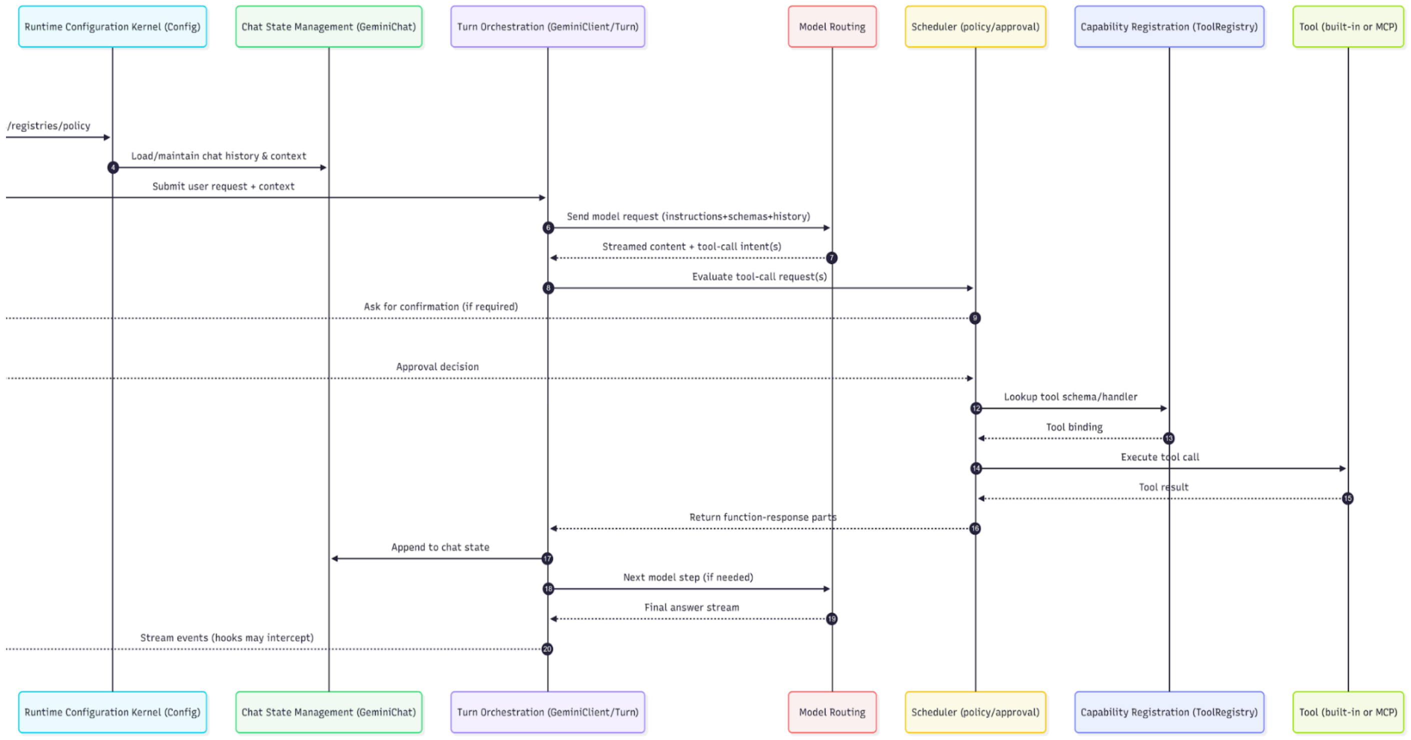
Result (what the user gets): A final answer that includes tool-derived information.

What can go wrong:

- Tool failure (for example: file not found, permission denied, command error).
- Missing or invalid authentication, so the model request fails.
- Tool access disabled by policy, so execution cannot proceed.

Sequence diagram


User
Argument & Settings Resolution
Interactive Session Controller
Runtime Configuration Kernel (Config)
Chat State Management (GeminiChat)
Turn Orchestration (GeminiClient/Turn)
Model Routing
1 Start gemini (interactive) + options
2 StartInteractiveUI with settings %% step 1-2 idea
3 Initialize services/registries/policy
4 Load/maintain chat history & context
5 Submit user request + context
6 Send model request (instructions+schemas+history)
7 Streamed content + tool-call intent(s)
8 Evaluate tool-call request(s)
Ask for confirmation (if required)
10 Approve / deny
11 Approval decision
Return function-response parts
17 Append to chat state
18 Next model step (if needed)
19 Final answer stream
20 Stream events (hooks may intercept)
21 Render output incrementally


Runtime Configuration Kernel (Config)
Chat State Management (GeminiChat)
Turn Orchestration (GeminiClient/Turn)
Model Routing
Scheduler (policy/approval)
Capability Registration (ToolRegistry)
Tool (built-in or MCP)
/registries/policy
4 Load/maintain chat history & context
Submit user request + context
6 Send model request (instructions+schemas+history)
7 Streamed content + tool-call intent(s)
8 Evaluate tool-call request(s)
9 Ask for confirmation (if required)
Approval decision
12 Lookup tool schema/handler
13 Tool binding
14 Execute tool call
15 Tool result
16 Return function-response parts
17 Append to chat state
18 Next model step (if needed)
19 Final answer stream
20 Stream events (hooks may intercept)

Data Dictionary (important data the system uses)

User Prompt - The natural-language request entered by the user.

CLI Command - The command used to run Gemini CLI in the terminal.

Config/Settings - Runtime options such as model selection and behaviour settings.

API Key/Login Token - Credentials used to authenticate requests.

Model Request - Structured request sent from Gemini CLI to the model.

Model Response - Response returned by the model.

Tool Call - Model request instructing Gemini CLI to invoke a tool.

Tool Name - The specific tool being called.

Tool Input - Inputs supplied to the tool (for example: path, arguments, query).

Tool Output - Result returned by the tool.

Final Answer - Final response displayed to the user.

Error Message - Message returned when any stage fails.

Naming Conventions:

To ensure consistency and readability across the architecture, the system adheres to the following naming conventions:

1. Subsystems: Subsystems are named using Nouns that describe their primary responsibility. Examples include the Orchestrator, Scheduler, and Controller.
2. Interfaces: Interface definitions are named based on the capability they provide. Examples include CapabilityRegistry and ModelProvider.
3. Events: Lifecycle events follow a strict pattern using the prefix "on_" followed by the event name. Examples include on_turn_start, on_model_response, and on_tool_execution.
4. Components: Internal components use PascalCase naming (e.g., SessionController), while configuration keys and internal event identifiers use snake_case (e.g., max_output_tokens).

Lessons Learned

One of the most significant lessons learned from analyzing Gemini CLI's architecture is the value of centralized orchestration in managing complex, multi-component systems. Although the architecture contains numerous subsystems the Turn Orchestration component provides a clear structural centre that coordinates reasoning, tool execution, and system feedback. By acting as a single control point, orchestration reduces the cognitive overhead required to understand how decisions propagate through the system. Instead of each subsystem independently determining execution flow, orchestration enforces a consistent interaction loop where model reasoning, policy checks, and tool results are integrated in a predictable sequence. This design not only improves traceability of reasoning loops but also demonstrates how architectural cohesion can emerge from a carefully designed coordination layer rather than from simplifying or reducing the number of components.

Another key lesson was that recovering the architecture required focusing on how control moves between subsystems rather than on the structure of individual files or modules. Initially, examining isolated components provided only a fragmented understanding of the system; the architectural intent became clearer only after mapping the flow of execution across session controllers, orchestration, scheduling, and capability layers. This shift in perspective highlighted that architecture is fundamentally defined by interaction patterns. By tracing these flows through sequence diagrams and box-and-arrow models, our group was able to identify meaningful subsystem boundaries and understand the rationale behind their separation.

AI Collaboration Report

AI Member Profile and Selection Process

For this deliverable, our group selected **OpenAI ChatGPT (GPT-5.2, February 2026 version)** as our virtual AI teammate. Assignment 1 required us to recover and explain the conceptual architecture of Gemini CLI based on documentation, which involves identifying subsystem responsibilities, architectural styles, and control flow relationships. ChatGPT was well suited for assisting with structured architectural explanations and improving the clarity of technical descriptions.

We considered other AI tools such as Google Gemini and Anthropic Claude. However, ChatGPT was selected because it consistently provided better subsystem decomposition and more structured outputs. The AI was used as a collaborative assistant rather than an authority, and all final architectural interpretations and conclusions were made and verified by human team members.

Tasks Assigned to the AI Teammate

Architectural style clarification: The AI helped explain why layered architecture is the most appropriate primary architectural style and how supporting styles such as repository and client-server fit into the system.

Subsystem responsibility explanation: The AI assisted in clearly explaining the roles of orchestration, scheduling, capability registration, and session control subsystems at a conceptual level.

Control flow and concurrency clarification: The AI helped structure explanations of control flow, reasoning loops, tool execution, and conceptual concurrency mechanisms such as streaming and asynchronous tool execution.

These tasks were assigned because they involved explanation and structuring rather than architectural decision-making. All architectural interpretations were independently verified by the team.

Interaction Protocol and Prompting Strategy

Multiple team members interacted with the AI using iterative prompting. We provided detailed architectural context and requested conceptual-level explanations suitable for formal documentation.

An example prompt used was:

“Explain the conceptual architecture of a terminal-based AI assistant that uses orchestration, tool execution, and extensibility mechanisms. Focus on subsystem responsibilities, control flow, and architectural style without discussing implementation details.”

Follow-up prompts were used to refine explanations and improve clarity. This iterative approach ensured that the outputs aligned with assignment requirements and conceptual architecture principles.

Validation and Quality Control Procedures

All AI-generated content was carefully validated before inclusion in the report. Validation included:

- Independent review by multiple team members
- Ensuring explanations remained conceptual and did not include implementation-level details
- Editing and refining AI outputs to ensure correctness, clarity, and consistency

No AI-generated content was used without human review and modification. This process ensured that the AI functioned as a support tool rather than a primary source of architectural decisions.

Quantitative Contribution to Final Deliverable

We estimate that the AI contributed approximately 15% of the final deliverable. This contribution consisted primarily of explanation refinement, structural guidance, and clarification of architectural relationships.

The remaining 85% of the work including but not limited to: architectural interpretation, diagram creation, use case development, validation, and final writing was done by team members.

Reflection on Human-AI Team Dynamics

The AI teammate improved efficiency by helping clarify architectural concepts and improve documentation quality. It allowed the team to focus more on understanding and analyzing the architecture rather than struggling with explanation clarity. However, working with AI required careful validation and refinement to ensure correctness. This demonstrated that AI is most effective as a support tool rather than a replacement for human reasoning. The AI also helped identify gaps in explanation clarity, which improved the overall quality of the report. Our team learned that effective prompting, validation, and human oversight are essential when working with AI. Overall, integrating an AI teammate improved productivity, enhanced clarity, and supported our understanding of conceptual architecture while maintaining full human responsibility for all architectural conclusions.

Sources / Links

- Google Gemini CLI GitHub repository: https://github.com/google-gemini/gemini-cli
- Google AI for Developers documentation (Gemini API): https://ai.google.dev/