Conceptual Architecture of Gemini CLI

OMMKPN NotFound 404 Error

Ming Yuan (Team Lead)	–	22xb63@queensu.ca
Michael Liu (Presenter)	–	22jdn2@queensu.ca
Omar Kaddah	–	22chm1@queensu.ca
Enrong Pan (Presenter)	–	enrong.pan@queensu.ca
Nandan Bhut	–	22gp44@queensu.ca
Kosi Amobi-Oleka	–	22fnbn@queensu.ca

Abstract

Gemini CLI is an open-source AI agent developed by Google that brings the capabilities of the Gemini large language model directly into the terminal, enabling developers to perform tasks such as code generation, file manipulation, and project analysis through natural language commands. This report aims to recover a reasonable conceptual architecture of Gemini CLI, including identifying its main subsystems, analyzing the interactions and concurrency among them at a high level, and examining how the system handles both routine and security-sensitive operations. Through a process of reading official documentation and reasoning backward from use case sequence diagrams, we identified Gemini CLI as a hybrid client-server and layered architecture consisting of nine major components: the UI as the client, and the Agent Core Coordinator, API Messenger, Tool Scheduler, Toolbox, Extension Manager, Policy Manager, Risk Prevention, and Memory Manager on the server side. The architecture prioritizes extensibility and modularity, allowing new tools and security policies to be introduced without modifying core components. Two use cases, a read-only stock summary retrieval and a destructive file deletion, illustrate how these nine components collaborate and demonstrate the system's layered approach to security enforcement.

1 Introduction

1.1 Overview

The purpose of this report is to create a conceptual architecture of Gemini CLI, an open source AI agent developed by Google that brings the capabilities of the Gemini large language model directly into the terminal. Gemini CLI can understand and generate a variety of information such as videos, text, images, audios, and more. It is built as a multimodel AI which means it can reason and process across multiple types of data. Gemini CLI comes in variety of versions that are specialized for specific tasks (for example; powerful cloud models for complex reasoning and smaller versions for mobile applications). This report will be helpful for anyone interested in understanding the underlying conceptual architecture behind Gemini CLI. Rather than going into the implementation-level details of classes, variables, or specific code structures, this report focuses on the system at the subsystem and module levels. The target is to identify its most probably software architecture styles, show how the major components interact, why they exist, and how the architecture supports critical quality attributes such as modifiability, extensibility, security, performance, and developer productivity.

This report analysis and architectural reasoning tells us that the system is decomposed into nine main components: the User Interface, Agent Core Coordinator, API Messenger, Tool Scheduler, Toolbox, Extension Manager, Policy Manager, Risk Prevention, and Memory Manager. Each component has a clearly defined role that goes from building and external communication to tool execution, persistence, and security enforcement. Together, these components form a hybrid architecture that combines client-server and layered architectural styles.

In the end, this report shows the whole conceptual architecture analysis of Gemini CLI. This report describes what the system is, why it was built, how it is structured, and how its architecture provides support for manageability, extensibility, and testability. The report shall clearly outline how Gemini CLI works as an AI-assisted development platform through the examination of component responsibilities, control flow, performance considerations, and real world use cases. The findings will be important in showing how modern AI-enabled tools are reworking software architecture design, emphasizing the need for modular orchestration layers, distributed computation, and secure tool integration.

1.2 Derivation Process

Our investigation process began with reading the official Gemini CLI documentation from geminicli.com and google-gemini.github.io. Since this report concerns conceptual architecture, we deliberately avoided consulting the GitHub source code and relied solely on what the documentation described. Early on, we noticed the documentation organizes the system into three packages: CLI, Core, and Tools. We initially took this at face value and assumed it mapped to a layered architecture, but we later came to understand that this three-package breakdown reflects how the code is actually organized rather than the higher-level conceptual structure we needed to present.

With the architectural style still uncertain, we debated whether the system was best characterized as layered, repository, or client-server. Rather than forcing a decision prematurely, we chose to first analyze the system's major components and return to the style question once we had a clearer picture. This turned out to be harder than expected. Defining subsystems in isolation felt like guessing, because we could not determine what a component truly needed to do without understanding how it interacted with others. It was at this point that Professor Bram Adams suggested we try formulating use cases and drawing sequence diagrams alongside our component analysis. This shift made a significant difference. As we traced how a user request like "summarize a stock"

or “delete a file” would flow through the system, the necessary components and their responsibilities became much more apparent.

To make the most of this approach, each group member independently drew the same use case sequence diagrams. We anticipated that everyone would produce slightly different versions, and this turned out to be true. Some members missed interactions with the Policy Manager, others forgot that the Memory Manager needed to save session state at the end, and the ordering of steps between the Tool Scheduler and Toolbox varied across drawings. We cross-checked all versions during a group meeting and combined the strongest elements from each into the final diagrams shown in Figures 2 and 3. For the final versions, we began with hand-drawn drafts for quick peer review, finally moved to a digital sequence diagram tool.

The sequence diagrams ended up informing far more than just the use case section. They confirmed which of our proposed nine components were genuinely necessary, revealed concurrency considerations such as how the UI remains responsive while the API Messenger waits for the remote model, and exposed the precise message flows that defined each component’s responsibilities. Once we were confident in both the components and the use case details, we constructed the box-and-arrow system diagram in Figure 1. Every dependency arrow in that diagram corresponds to an interaction we had already traced in at least one sequence diagram, which gave us confidence that the three diagram types are consistent with one another. The architectural style debate also resolved naturally at this stage: the UI clearly operated as a client sending requests and receiving responses, while the remaining eight components worked together as a server with internal layered organization, leading us to adopt the client-server and layered hybrid style [1].

2 Conceptual Architecture

2.1 Architectural Style: Client-Server and Layered

2.1.1 Main Architectural Styles

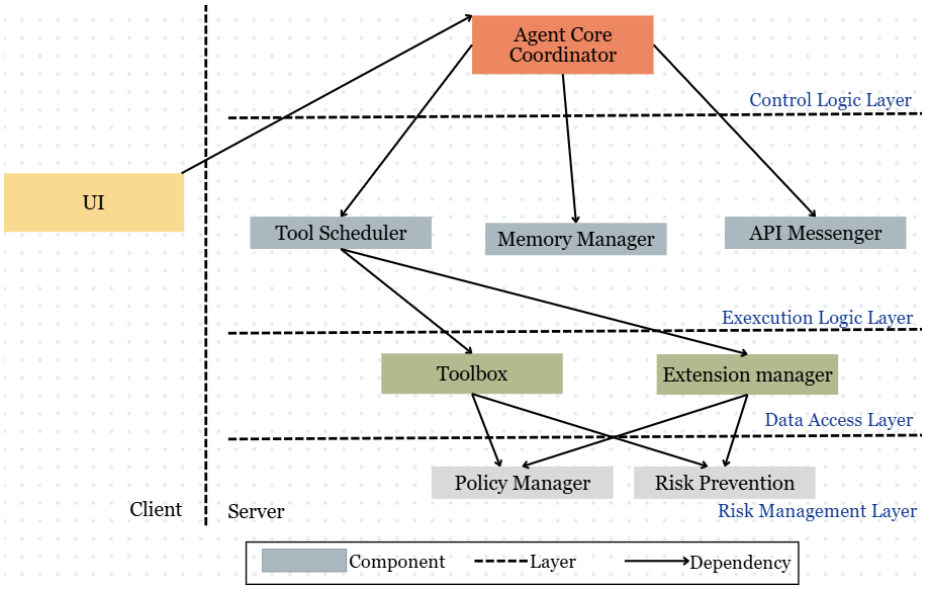

Figure 1: Gemini CLI Box and Line System Diagram

Gemini CLI adopts the mix of two architecture styles: **Client-Server** and **Layered** as showed in Figure 1 [2]. The overall system of Gemini CLI is divided into two main roles as in client-server style: client side and servers side. On the client side we have the UI component which takes care

all direct interaction with users like managing user input, output displaying, themes, and overall terminal experience. UI as client will send input from users as the requests to the server.

The server part of the Gemini CLI follows a Layered architecture style to provide service to request of client UI. The server part is divided into separate layers based on specific responsibilities. The layers are divided as: Control logic Layer, Execution Logic Layer, Data Access Layer, Risk Management Layer.User requests from client are first forwarded to the Control logic Layer, which contains the Agent Core Coordinator acting as the central control unit of the server which distribute the work down to the Execution Layer. Execution Layer then do the heavy lifting of communicating to API (API messenger), manages session memory through the Memory Manager, and schedule tool usage via the Tool Scheduler. The next layer handles data access and local system interactions, including execution of built-in tools provided by the Toolbox, and execution of extensions through the Extension Manager. The lowest layer provide services to Data Access Layer like checking safety and permission requirements stored and managed by Policy Manager, and validates potentially risky operations using the Risk Prevention component[1].

The main challenge is the latency caused by the "top-to-bottom" control flow in Layered style. In this layered server, a request like executing a tool must pass through every single layer and back again before the Agent Core Coordinator can respond to the client with the result. This multiple layered processing creates a bottleneck that slow down the overall system performance since it takes time to transmit the request and wait for response from lower layer.

Using a layered architecture inside a client–server system for Gemini CLI helps address one of the main weaknesses of a pure client–server style: the lack of clear internal organization and control flow on the server side. In a pure client–server model, the server may grow over time as more services and features are added. Without a structured internal design, the server can become chaotic with unclear division of responsibilities. This will makes the system harder to maintain, extend, or test, which is not ideal for Gemini CLI.

A layered architecture solves this issue by dividing the server into clear layers, each with a specific responsibility[2]. Each layer communicates only with adjacent layers, which creates a clear control flow and separation of concerns. This improves maintainability, since changes in one layer (such as the policy stored in PolicyManager) do not necessarily affect others. It also improves testability, as each layer can be tested independently.

However, the client–server style that the overall Gemini CLI system adopts still inherits some risks from a pure client–server architecture. The network connection between clients and the server remains a potential single point of failure. If the network is slow, unstable, or disconnected, clients may be unable to access services even when the server is functioning correctly. On the positive side, client-server division also made possible that client side can run separately even if the server side goes down because of point of failure.

Within the server, the layered architecture helps mitigate internal failures. If one layer encounters an error, the other layers are not affected unless they specifically depend on the faulty layer's services. This means that a failure in one component does not necessarily bring down the entire server. As a result, the layered design improves fault isolation and allows the system to handle internal failures more gracefully, even though system-wide availability still depends on the server and network connection.

In summary, layering server improves the internal organization and maintainability of the server in a client–server system, but it does not eliminate the inherent risk of single point failure.

2.1.2 Alternative Architectural Style: Publish-Subscribe

Building on the system's design goals, particularly the emphasis on extensibility[1], an alternative architectural approach worth considering is the Publish–Subscribe (or plugin) style[2], which can further enhance modularity and support dynamic extension of functionality. The Agent Core Coordinator acts as the core within the Gemini CLI. It does not do any heavy lifting job itself, instead it publish relevant work to relevant components for them to do the job and publish the results. At the same time, all other components acts as plug-ins.

However, we did not choose this architecture style since it introduces huge drawbacks to Gemini CLI. Because within this style the developer does not need to know the details of internal architecture or components, they then lack understanding of how other components work. As a result, the other components may not behave as expected when needed due to incorrect assumptions about execution order or system behavior. Also, this architecture style will not be able to enforce a specific order of execution of components due to its broadcasting characteristic. This make the result to the users unpredictable and uncontrollable. Therefore, the clear order of operation of the mix styles of Client-Server and Layered are the better choice for Gemini CLI.

2.2 Major Components Overview

While the official documentation groups the system into three high-level packages (CLI, Core, and Tools) [1], this decomposition understates the architectural complexity and conflates distinct subsystems with different responsibilities. As briefly noted in the layered-server description, the system can be decomposed into four layers comprising nine major components. The following discussion expands on these components, detailing their individual roles and illustrating how they interact within the overall architecture.

The **UI** handles all direct user interaction including input capture, response display, and visual theming. The **Agent Core Coordinator** serves as the central brain that orchestrates which components handle each part of a request and also takes care of converting the user-fed prompt to gemini-api-understandable messages. The **API Messenger** manages all external communication with Google's Gemini API servers. The **Tool Scheduler** determines which tools are needed for a task and coordinates their execution, deciding whether built-in tools suffice or extension-provided tools are required. The **Toolbox** contains all built-in tool implementations for file operations, shell commands, and web access. The **Policy Manager** enforces access control rules, determining whether operations can proceed automatically or require explicit user permission. The **Risk Prevention** subsystem provides execution isolation by testing potentially dangerous operations in a sandboxed environment before applying changes to the actual system. The **Extension Manager** discovers and coordinates external tools not included in the default installation. Finally, the **Memory Manager** handles session persistence and user-specific configurations such as prompting preferences and conversation history. In the following subsections, we provide a high-level discussion of each component's functionality, goals, and quality attributes, followed by an analysis of how these components interact to fulfill the system's requirements.

2.3 Subsystems and Inter-Component Interactions

2.3.1 UI (User Interface)

The UI subsystem serves as the client in the client-server architecture, handling all direct interaction with the user through the terminal [1]. This component exists because the system requires a dedicated boundary between human users and internal processing logic. When a user types a question or command, the UI captures this input and forwards it to the Agent Core Coordinator (the server entry point) for processing. When the AI generates a response, the UI renders the

text progressively as it streams. Beyond basic input and output, this subsystem manages visual elements such as color themes and syntax highlighting, keyboard shortcuts, and confirmation dialogs for sensitive operations.

The UI maintains a unidirectional connection with the Agent Core Coordinator, with user input flowing downstream and Agent Core Coordinator responses with return messages upstream. When the Policy Manager determines that an operation requires user approval, the Agent Core Coordinator signals the UI to display a confirmation dialog[3]. When users browse saved sessions, the UI requests them through the Agent Core Coordinator, which retrieves them from the Memory Manager. This separation supports testability, as UI components can be tested independently of server-side processing. A front-end developer with terminal UI expertise would own this component.

2.3.2 Agent Core Coordinator

The Agent Core Coordinator acts as the central orchestrator, functioning as the brain that determines which components should handle each aspect of a user's request [1]. This component exists because the system requires a single point of coordination to manage the interplay between user input, AI reasoning, tool execution, and response generation. Without such a coordinator, components would communicate directly with each other, creating a tangled web of dependencies. The Agent Core Coordinator implements the ReAct (Reasoning and Acting) loop that defines what the system fundamentally does: multi-step autonomous task completion where the AI reasons about a problem, takes an action, observes the result, and iterates until the task is complete.

When input arrives from the UI, the Agent Core Coordinator first consults the Memory Manager to retrieve conversation history and context from GEMINI.md project files. It then constructs a complete prompt by combining the user's input with this context and definitions of available tools, passing the prompt to the API Messenger for transmission to Google's servers. Upon receiving a response, it determines the next action: direct text is forwarded to the UI for display, while tool execution requests are delegated to the Tool Scheduler. Control flows outward from this component to all others, while data flows bidirectionally as context arrives from the Memory Manager and results return from tool execution. The Agent Core Coordinator also handles model fallback logic, switching from the Pro model to the Flash model when rate limiting is detected. This component requires a developer with deep understanding of AI agent patterns, serving as the integration point for other teams. [3]

2.3.3 API Messenger

The API Messenger handles all communication with Google's Gemini API servers, which host the AI model that powers the system [1]. This component exists because the system must isolate external network dependencies behind a single interface. Since AI inference occurs on Google's cloud infrastructure, centralizing network communication simplifies error handling, authentication, and testing. When the Agent Core Coordinator constructs a prompt, the API Messenger packages this request, transmits it, and receives the streaming response.

The API Messenger interacts exclusively with the Agent Core Coordinator, creating a clear boundary between internal logic and external dependencies. Responses stream back asynchronously, allowing the UI to display partial results. The API Messenger also tracks token usage for context window management and implements graceful handling of network errors and rate limiting. A developer specializing in network protocols would own this component.

2.3.4 Tool Scheduler

The Tool Scheduler coordinates tool selection and execution [4]. This component exists because the system needs a single point of control for all tool operations: maintaining the registry, validating

parameters, and routing requests. The Tool Scheduler maintains a registry of all available tools with their parameter schemas. When the Agent Core Coordinator receives a tool request, it delegates to the Tool Scheduler for validation.

After validation, built-in tools route to the Toolbox, while extension tools route to the Extension Manager. This routing is transparent to the Agent Core Coordinator. The registry-based architecture supports evolvability. A workflow orchestration developer would own this component.

2.3.5 Toolbox (Built-in Tools)

The Toolbox contains implementations of tools that ship with Gemini CLI, performing actual operations on the user's system [4]. This component exists because the system needs a home for core capabilities: file manipulation, command execution, web access, and memory persistence. File system tools enable reading, writing, and listing directories[3]. The shell tool executes terminal commands. Web tools provide internet access. The memory tool persists information by writing to GEMINI.md files through the Memory Manager.

The Toolbox receives requests from the Tool Scheduler. Before execution, the Toolbox consults the Policy Manager to determine whether the operation is permitted. Based on the decision, execution may proceed automatically, be blocked, or require user confirmation through the UI. For system-modifying operations, the Toolbox coordinates with Risk Prevention to execute within a sandboxed environment. Each tool operates independently with a well-defined interface, enhancing testability and evolvability. Multiple developers can work on individual tools in parallel.

2.3.6 Policy Manager

The Policy Manager implements rule-based access control governing what operations the system may perform [1]. This component exists because the system must balance powerful AI capabilities with user safety through centralized, configurable security. When the Toolbox or Extension Manager prepares to execute a tool, it queries the Policy Manager, which evaluates the request against rules: `allow` permits automatic execution, `deny` blocks the operation, and `ask_user` triggers a confirmation dialog.

Rules are defined in TOML files loaded through the Memory Manager, with three priority tiers: Administrator rules for enterprise enforcement, User rules for personal preferences, and Default rules where read operations are allowed while write operations require confirmation. The configuration-driven approach supports evolvability through policy changes without code modifications. A security-focused developer would own this component.

2.3.7 Risk Prevention (Sandbox Manager)

The Risk Prevention subsystem provides execution isolation to protect the user's system from unintended consequences [5]. This component exists because even with policy controls, an AI executing code presents inherent risks. When the Toolbox executes dangerous operations, Risk Prevention can wrap execution in a sandboxed environment without access to actual system resources.

The Toolbox passes operations with sandbox parameters, and Risk Prevention sets up the isolated environment, executes, captures results, and returns them. Docker and Podman provide container-based sandboxing, while macOS Seatbelt profiles offer a lighter alternative. Combined with the Policy Manager, this implements defense-in-depth security. A systems programmer with containerization expertise would own this component.

2.3.8 Extension Manager

The Extension Manager handles capabilities beyond built-in functionality, enabling the system to grow without modifying core code. This component exists because the system cannot anticipate

every tool users might need. It discovers, loads, and invokes external tools through MCP (Model Context Protocol) servers, which expose tool interfaces over a standardized protocol[6]. This component also handles extension lifecycle: installing, updating, and enabling extensions.

When the Tool Scheduler encounters an unknown tool, the Extension Manager discovers MCP servers from configuration, establishes connections, and prepares to execute[3]. Before execution, the Extension Manager consults the Policy Manager to determine whether the operation is permitted. The Extension Manager executes external tools itself rather than delegating to the Toolbox. Extensions can also register lifecycle hooks[1]. This architecture exemplifies evolvability through the open-closed principle. A plugin architecture expert would own this component.

2.3.9 Memory Manager

The Memory Manager handles all persistent state and configuration within Gemini CLI [1]. This component exists because multiple components need access to persistent data, and centralizing access prevents scattered file operations. Conversations are automatically recorded with prompts, responses, tool executions, and token usage. Sessions are organized by project.

The Memory Manager serves as a data provider with simple interfaces. The Agent Core Coordinator retrieves conversation history and GEMINI.md context. The Policy Manager receives policy rules. The Extension Manager receives configurations. Configuration follows a hierarchical model with system, user, workspace, and project levels. A data persistence specialist would own this component.

2.4 Control Flows and Concurrency

2.4.1 The ReAct Loop and Global Control Flow

The Gemini CLI follows a centralized control flow where the Agent Core Coordinator manages the system through a ReAct (Reasoning and Acting) loop [1][7]. When the UI captures a user request, it forwards the input to the Agent Core Coordinator, which gathers relevant context from the Memory Manager, including conversational history, system instructions, and local project files. The coordinator then packages this context alongside the user query and passes it to the API Messenger for inference by the remote Gemini model. Upon receiving the model's response, the Agent Core Coordinator inspects whether the model has requested a tool invocation. If so, control returns to the coordinator, which delegates the request to the Tool Scheduler. The Tool Scheduler locates the appropriate capability within the Toolbox or, for third-party tools, within the Extension Manager. Before execution of tool proceeds, the Policy Manager verifies the action against its configuration-driven rules. Once approved, the tool executes and returns its result to the Agent Core Coordinator, which feeds the observation back to the API Messenger for further reasoning. This cycle repeats until the model produces a final answer, at which point the coordinator instructs the UI to render the response to the user.

This centralized design, where every interaction must pass through the Agent Core Coordinator, is the primary reason the system achieves high modifiability. New tools can be registered in the Toolbox or connected via the Extension Manager without altering existing communication paths, and security policies can be updated in the Policy Manager without requiring changes to the orchestration logic.

2.4.2 Performance Critical

The primary performance constraint occurs during the inference prefill phase managed by the API Messenger [1]. To generate a response, the remote Gemini model must process a substantial volume of data, including conversational history, system instructions, and local files retrieved by the

Memory Manager. As the context window scales, the prefill computation becomes increasingly resource-intensive on the server side, introducing latency that directly affects user-perceived responsiveness. The system's overall performance therefore depends on the balance between local processing speed and remote inference time: while the UI, Toolbox, and other local components operate at native machine speed, the API Messenger is constrained by network latency and server-side computation. This bottleneck is most apparent when the Memory Manager supplies large amounts of project context, as the entire payload must be transmitted to the remote API before the model can begin generating a response.

2.4.3 Concurrency and Responsiveness

To prevent the terminal from becoming unresponsive during slow network operations, the architecture employs an asynchronous pattern [1]. Although the Agent Core Coordinator acts as the central orchestrator in this client-server system, it does not block while waiting for the API Messenger to return a response. Instead, requests to the remote model are dispatched in the background, allowing the UI to remain interactive. Users can view streaming output as it arrives incrementally or cancel an in-progress request without restarting the application. For resource-intensive local tasks, such as the Toolbox searching through large directory structures, the system performs these operations independently from the main interaction flow so that they do not delay user input or output. Background operations, including automatic session saving by the Memory Manager, similarly proceed without interrupting user interaction. This approach provides effective concurrency, ensuring that the UI client remains responsive even when the server-side components are engaged in long-running operations.

2.5 Support System Evolution

The Gemini CLI's client-server architecture, with the UI as client and the remaining eight components forming the server, creates a natural evolutionary boundary [1]. The UI can be redesigned, replaced, or adapted for different platforms such as a graphical interface or a web frontend without modifying server-side logic, including prompt construction, tool execution, or security enforcement. Within the server, the layered organization ensures that orchestration (Agent Core Coordinator), external communication (API Messenger), tool execution (Tool Scheduler, Toolbox, Extension Manager), security (Policy Manager, Risk Prevention), and persistence (Memory Manager) evolve independently.

The Extension Manager enables third-party tool integration through MCP servers, allowing new capabilities to be developed and deployed without changes to core components. The Tool Scheduler's registry-based design allows additions to the Toolbox with minimal modification, as new tools register themselves through a standard interface. The Policy Manager's configuration-driven rules allow security policies to evolve without code changes, supporting different enforcement levels across deployment contexts. The Memory Manager's configuration hierarchy, where project-level settings override workspace, user, and system defaults, supports context-specific adaptation without architectural changes.

2.6 Division of Responsibilities Among Developers

The client-server boundary creates a natural team division: one team owns the UI client, while other teams work on server-side components without concerning themselves with presentation details [1]. Within the server, the layered structure further partitions work. An AI agent expert owns the Agent Core Coordinator. A network specialist owns the API Messenger. A workflow developer owns the Tool Scheduler. Multiple developers work in parallel on Toolbox capabilities. A security specialist owns both the Policy Manager and Risk Prevention. A plugin expert owns the Extension

Manager. A data persistence specialist owns the Memory Manager. This division enables parallel development with well-defined interfaces.

3 External Interfaces

Gemini CLI relies on multiple external interfaces to successfully provide its AI assisted functionality through a command line environment. Rather than operating on its own like a isolated system, it interacts with external components such as the user's terminal, local file resources, cloud based AI model services, authentication mechanisms, and network infrastructures to produce the output that the users are looking for. These interfaces act as channels through which information flows in and out of the system. This means that Gemini CLI acts as a bridge between user requests and the LLM used to create results and outputs.

3.1 External AI Model Interface (Gemini LLM)

Gemini CLI communicates with the Gemini LLM through its external cloud based API service [1] (API messenger). It is important to note that Gemini CLI does not perform any of its AI functionality locally; instead, it converts the user prompts and information into structured API requests (agent core coordinator). These requests are then fed into the large language model so that the appropriate output can be returned. This interface is essential to Gemini CLI's functionality, as it allows for the advanced AI capabilities to be accessed while minimizing computational load on local resources. Additionally, authorization to Gemini's services is also decided externally. The system confirms authorization by checking credentials (e.g., API keys and access tokens) which it finds through runtime environment variables or configuration files [1]. This process is a prerequisite for Gemini's operations, as identity verification and access authorization must occur before any AI functionality can be used.

3.2 External File System Interaction

Gemini CLI interacts with the user's local file system to allow for file based input and output operations as part of its execution workflow (toolbox). Through command line arguments, the system has the functionality to read external text documents, source code files, or other user provided resources from disk and convert their contents into prompts for the AI model to process (agent core coordinator). Similarly, the system also has the ability to write generated responses returned from the Gemini LLM directly into existing code and documentation (mediated by the host operating system) [1] (policy manager). Additionally, Gemini CLI can access locally stored configuration files to load user preference related settings that allow for consistent operation (memory manager). Altogether, the file system interface expands the CLI's capabilities beyond simple user input queries, allowing for consistent data exchange and integration into document driven workflows.

3.3 Networking and Communication Interfaces

Since Gemini CLI depends on remote AI services, it requires reliable network communication to support all of its operations. Requests are typically sent over secure HTTPS connections which ensure that prompt data and responses are exchanged securely between the CLI and external servers (API messenger). Through this interface, the system communicates user queries (converted into API requests) outward and receives model generated results in return. Additionally, the system relies on the MCP layer to structure how data, tool calls, and responses are transmitted between the ACC and external AI services. This protocol standardizes communication between internal components and external AI systems, ensuring consistent formatting and coordination. Network communication also enables the system to detect and report issues such as connection failures, service interruptions, or request limits. This means that network communication is a core interface;

and without HTTPS and MCP, the CLI would not be able to utilize AI resources or transfer data securely (agent core coordinator).

3.4 Integration with External Tools and Workflows

Gemini CLI is designed to integrate effectively with other command line utilities and development workflows through standard input and output interfaces. Given that it is a terminal based tool, it can accept input from standard streams allowing users to pipe text from other programs and applications into the system for processing. Similarly, the responses and outputs produced by Gemini CLI can be redirected into shell pipelines, files and scripts. This interface compatibility with external tools allows Gemini CLI to function as both an interactive assistant and as a modular component within external software engineering and architecture workflows.

4 Use Cases

4.1 Use Case 1: Summarize A Stock

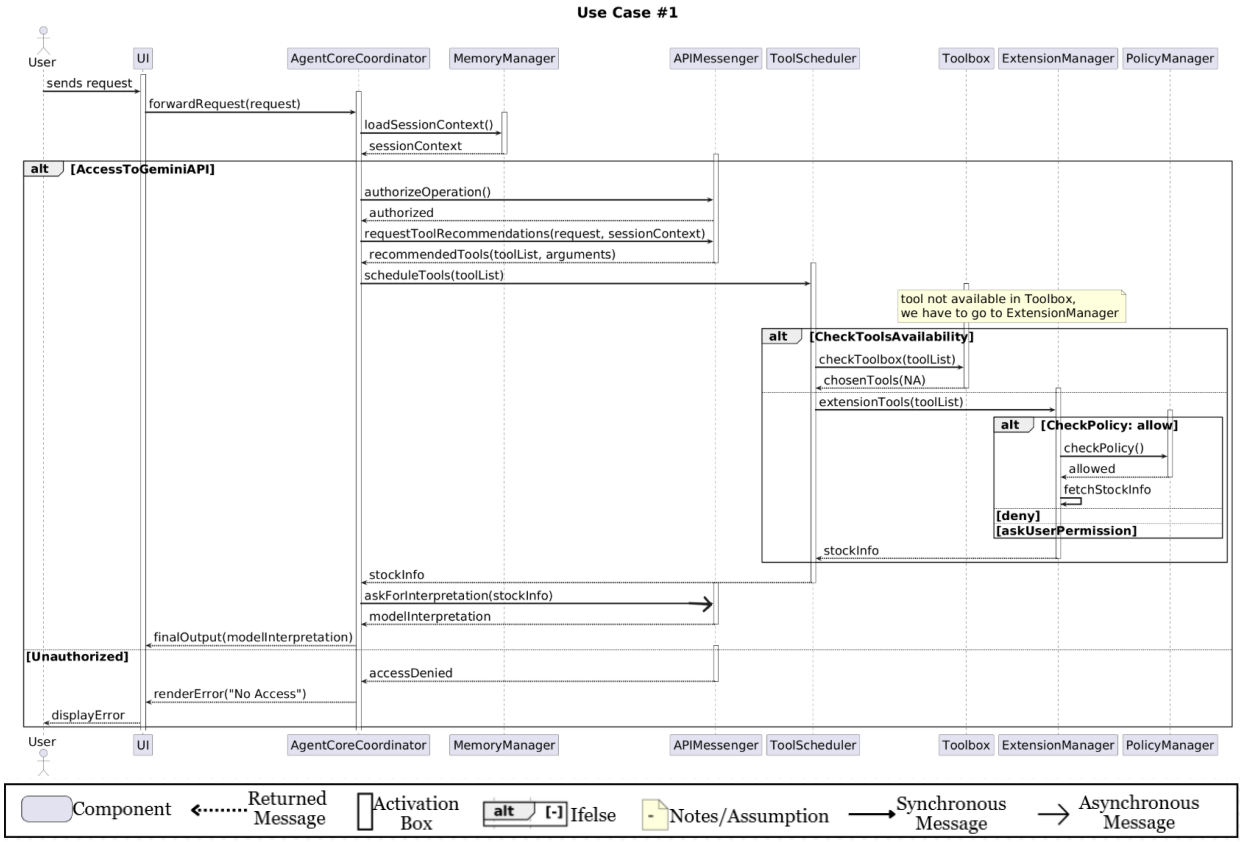

Figure 2: User asks Gemini CLI to summarize information of a chosen stock

To illustrate how the nine components within our conceptual architecture interact and their necessary existences, we have chosen two quite representative use-cases of Gemini CLI. The first sequence diagram Figure 2 represents our Use Case 1: **the user asks Gemini cli to summarize a chosen stock**. The diagram starts when the user initiates an operation by requesting a read-only operation (stock summary) through the user interface. The UI then forwards this operation request to the AgentCoreCoordinator (ACC). Once the ACC receives the request, it first communicates with the MemoryManager to load the current session context so that it can interpret the user's request more accurately. Before any external action is performed, the ACC checks with the APIMessenger to

determine whether the requested operation is authorized. This authorization phase simply checks if the user has the authority to use the external service (Gemini). If access is denied, the system returns an error to the user, stopping any further processing. If authorization is accepted, the ACC proceeds by contacting the APIMessenger to ask Gemini what tools or actions are necessary to satisfy the user's request. Gemini responds with a list of recommended tools and required arguments, allowing the ACC to determine how the request should be handled. The ACC then forwards these recommendations to the ToolScheduler which organizes a structured tool execution plan and checks whether the required operations can be handled by the Toolbox. If the Toolbox cannot support the requested operation (current assumption), the ToolScheduler consults the ExtensionManager to select the appropriate extension tools. After the extension tools are selected, the PolicyManager is consulted to check whether the requested operation is allowed (read-only operations will always be accepted). If the operation is allowed, the system proceeds to execute the operation through the extension tools; otherwise, access is denied and the process is terminated or user is asked for permission. After the operation is completed and the stock information is retrieved, it is returned back to the ACC through the ToolScheduler. The ACC then forwards the retrieved stock information to Gemini through the APIMessenger for interpretation. The message is sent **asynchronously**, as labelled in the diagram, so the user can view partial feedback on the UI. Gemini, the external interface responsible to the API Messenger, processes the information and generates a relevant response which is also returned asynchronously to the ACC. The ACC then instructs the UI to render the final response, and the result is clearly displayed to the user. Finally, the system updates the session state through the MemoryManager, ensuring that the interaction outcome is stored.

4.2 Use Case 2: Delete A File

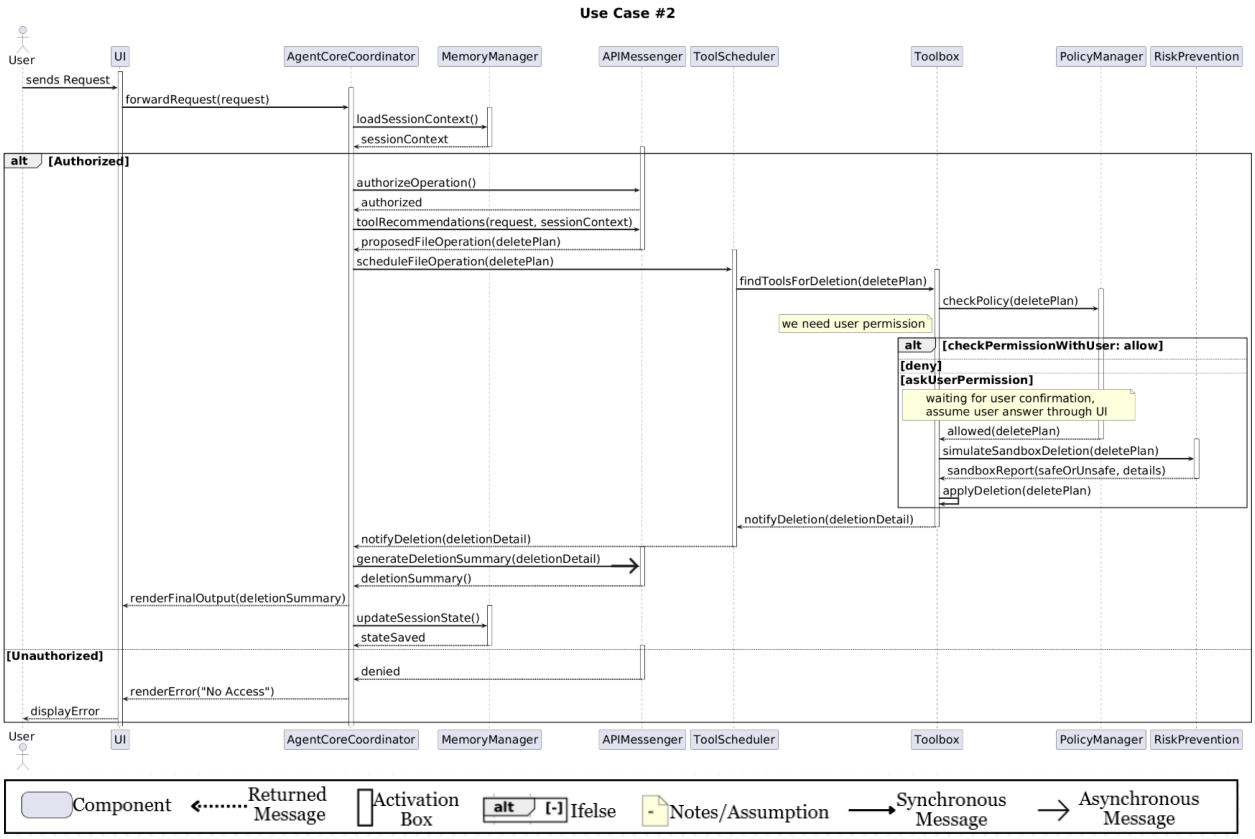

Figure 3: User asks Gemini CLI to delete a chosen file (dangerous action)

The second use case is **the user asks Gemini CLI to delete a chosen file**. This is not a read-only execution, so it might require user permission and security check. Up until the authorization step, this sequence diagram Figure 3 works exactly like Use Case 1. The new sequence events occur after the authorization status is returned. If the user is authorized, the ACC proceeds by consulting with the APIMessenger to determine what operation is required and what tools are needed. In this case, the APIMessenger returns a proposed deletion plan along with the recommended tools necessary for deleting the user's requested file. The ACC then passes this plan to the ToolScheduler to convert the operation into an executable workflow. The ToolScheduler then interacts with the Toolbox to verify that the required tools for the deletion are available. PolicyManager is also consulted to check whether the operation is automatically allowed or whether user permission is required. In this case changes are being made to the file so user permission will be requested. After the user grants permission through the UI, Toolbox sends the operation to RiskPrevention. The RiskPrevention component is the focal point of this sequence because it allows for an additional safety layer. Rather than applying the deletion immediately, RiskPrevention allows the operation to be simulated in a sandbox environment. This creates a virtual testing space to check whether the deletion is safe or unsafe, ensuring that potentially harmful outcomes are detected before affecting real data. If the sandbox simulation decides that the operation is safe, the Toolbox applies the deletion plan; otherwise the process is terminated immediately. After the operation is completed, deletion details are first returned to the ToolScheduler and then forwarded to the ACC. The ACC then **asynchronously** sends this information to Gemini through the APIMessenger to generate a deletion summary. The generated summary is returned to the ACC and presented to the user as the final output. Finally, the ACC updates the session state through the MemoryManager and makes sure that the outcome of the operation is stored.

5 Conclusion

In this report, we analyzed the conceptual architecture of Gemini CLI and demonstrated how its nine core components collaborate to enable multi-step autonomous task completion. At the center of the system is the Agent Core Coordinator, which orchestrates reasoning, tool invocation, and response generation through a ReAct loop. Surrounding this core are clearly defined components responsible for communication (API Messenger), workflow orchestration (Tool Scheduler), execution (Toolbox and Extension Manager), security enforcement (Policy Manager and Risk Prevention), and persistence (Memory Manager).

The architecture reflects strong separation of concerns and adherence to key software engineering principles, including modularity, encapsulation, and the open-closed principle. Each component has a well-defined responsibility and interacts through structured interfaces, enabling parallel development and independent evolution. The registry-based tool architecture, configuration-driven security policies, and MCP-based extension framework collectively ensure extensibility without requiring invasive modifications to the core system.

Performance considerations, particularly remote inference latency and context window scaling, highlight the trade-offs inherent in cloud-based AI systems. However, the use of asynchronous communication patterns and streaming responses ensures user responsiveness even during long-running operations. Additionally, the layered security model combining policy enforcement with sandbox-based risk prevention demonstrates a defense-in-depth approach that balances AI autonomy with user safety.

Through the presented use cases, we illustrated how the system adapts to both read-only operations and potentially dangerous actions while preserving consistent control flow and security validation. This reinforces the architectural strength of central ized orchestration combined with modular

execution layers.

Overall, Gemini CLI exemplifies a well-structured client-server AI agent architecture designed for modifiability, scalability, and safe evolution. Its layered design and clear division of responsibilities position it as a robust foundation for future enhancements, additional tooling integrations, and expanded AI capabilities.

6 Lessons Learned

Throughout the development of this report, we encountered several challenges, but it was these difficulties that ultimately contributed to our learning and taught us valuable lessons. Perhaps the most instructive mistake we made was taking the official documentation's three-package structure at face value. Because the Gemini CLI docs present CLI, Core, and Tools as the top-level organization, we spent our first few meetings trying to map our conceptual architecture directly onto these packages. It took stepping back and asking what the system actually does, rather than how its code is organized, to realize that a conceptual architecture requires independent interpretation. The nine components we eventually proposed, such as separating the Tool Scheduler from the Toolbox or distinguishing the Policy Manager from Risk Prevention, are not visible in the documentation's package structure but are essential for understanding the system's behavior at the right level of abstraction. We ran into a similar issue when writing the control flow section, where our first draft discussed token limits and their effect on API Messenger processing speed. Reviewing the assignment guidelines reminded us that conceptual architecture stays above implementation details, and we rewrote the section to focus on the architectural pattern of asynchronous communication rather than specific resource constraints.

Our methodology also evolved through trial and error. We initially tried to define all nine components before writing any use cases, which felt like assembling a puzzle without seeing the picture on the box. Once all of us began working on use cases in parallel (following Professor Adams' suggestion), the sequence diagrams became our most powerful tool. Tracing a stock summary request from the UI through the Agent Core Coordinator, API Messenger, Tool Scheduler, Extension Manager, and Policy Manager made each component's role concrete in a way that abstract descriptions never could. Having every member independently draw the same diagram and then compare results was time-consuming, but it caught gaps and incorrect assumptions that no single person would have noticed on their own.

On the collaboration side, we learned that equal division of work does not mean each person retreats into their own section. Early in the project, some of us wrote our parts without fully reading what others had produced, and the result was noticeable inconsistency in terminology, level of detail, and architectural assumptions. For instance, one member described the Toolbox as consulting the Policy Manager before execution while another's section implied the Tool Scheduler handled permission checks. These contradictions only surfaced when we sat down and read the entire report end to end. From that point on, we made a rule that everyone should read all nine component descriptions before writing or revising their own sections. We also discovered that the introduction and conclusion, which initially seemed like simple bookend tasks, actually need to be written last because they must accurately reflect the final state of every other section.

The extended debate over architectural style, where we went back and forth between client-server, layered, and repository over several meetings, felt unproductive at the time but turned out to be one of the most valuable parts of our process. It forced each of us to articulate concrete reasons grounded in the component interactions we had mapped, rather than simply asserting a preference. More broadly, we found that allowing anyone to comment on or edit any section, rather than treating each part as one person's territory, significantly improved the final product. This required

a level of trust and openness that did not come automatically, but the willingness to give and receive honest feedback made the report stronger than any one of us could have produced alone.

7 Dictionary and Naming Conventions

Agent Core Coordinator: The central orchestrator component that manages the ReAct loop, delegates tasks to other components, and serves as the single point of coordination on the server side.

API (Application Programming Interface): Protocols and tools for building software integrations.

API Messenger: The component responsible for all communication between the system and Google's remote Gemini API servers.

CLI (Command Line Interface): Text-based interface for system interaction via commands.

Docker and Podman: Platforms for developing and running containerized applications.

Extension Manager: The component that discovers, loads, and executes third-party tools through MCP servers, as opposed to the Toolbox which handles built-in tools.

GEMINI.md: Per-project markdown context files loaded by the Memory Manager to provide project-specific instructions and context to the AI model.

HTTPS (Hypertext Transfer Protocol Secure): Encrypted protocol for secure web data transfer.

LLM (Large Language Model): AI model trained to understand and generate human-like text.

MCP (Model Context Protocol): Standardized protocol connecting AI models to external data sources and tools.

Memory Manager: The component that handles all persistent state including session history, configuration files, and GEMINI.md context loading.

Policy Manager: The component that enforces configuration-driven access control rules governing whether operations are allowed, denied, or require user confirmation.

ReAct Loop (Reasoning and Acting Loop): The iterative cycle where the AI model reasons about a task, requests a tool action, observes the result, and repeats until the task is complete.

Risk Prevention (Sandbox Manager): The component that provides execution isolation by wrapping dangerous operations in sandboxed environments. These two names refer to the same component.

Sandbox: Isolated environment for executing code safely without affecting the host.

TOML (Tom's Obvious Minimal Language): Minimal configuration file format designed for readability.

Toolbox: The component containing all built-in tool implementations such as file operations, shell commands, and web access, as opposed to the Extension Manager which handles third-party tools.

Tool Scheduler: The component that maintains the tool registry, validates parameters, and routes execution requests to either the Toolbox or the Extension Manager.

UI (User Interface): The client-side component handling all direct user interaction through the terminal, including input capture, response rendering, and visual theming.

8 AI Collaboration Report:

8.1 AI Member Profile and Selection Process:

We will be using the Gemini 3 Pro, Nov 2025 version as our extra AI member. Gemini uses the same infrastructure and it makes the most sense since we are studying Gemini CLI. Gemini is comparatively better to use in this case since it is integrated with google and can give real time and up to date information. Also since it is integrated with google it will also give out better and more accurate sources for certain prompts. Gemini is vastly better at image generation and interpretation than ChatGPT
Another aspect that Gemini would be better at is giving ideas and a short direct and effective answer which could be very beneficial for interpreting and understanding concepts related to the project. It could also be used to test our architecture drawing and get feedback on ways that it could be improved.

8.2 Tasks Assigned to the AI Teammate:

- Brainstorming: We used AI to brainstorm ideas to get a start lead. AI can quickly process large databases of information and summarize all ideas related to the question clearly and fully with rarely missing information.
- Summarize resources: We went through long documentations with many new concepts and terms in order to identify the architecture. AI shortened this procedure by picking out the most important and relevant part of the documentation. As well as explaining the concept in simple language.
- Diagram transforming: We used AI to transfer our sequence diagrams from one website to another website to obtain a clearer view of diagram without worrying the language differences in two websites.
- Paragraph(s) refinement: Since AI can quickly detect grammar mistakes and inconsistency between the paragraphs in the long essay that a human detector may miss by checking every single word and connecting context.

8.3 Validation and Quality Control Procedures:

Gemini is an LLM model that is based on inference probability. To prevent Gemini from hallucinating we need to verify sources through credible links and references. One way to ensure validity of the LLM is through asking for citations and checking each hyperlink to ensure the data collected matches with the domain page. If Gemini fails to provide a hyperlink towards the information obtained then that signifies false information
Another possible method of validating the information obtained is through a Student-Teacher Relation. To mitigate hallucinations, we will use a Teacher Mode configuration. In this setup, the LLM is assigned the persona of a rigorous University Professor specializing in third-year Software Architecture. The model is instructed to be highly critical, evaluating outputs on a scale of 1–100. This process ensures the identification of irregularities and fabricated information by providing feedback for every generated segment. To ensure accuracy, a team member should cross-check each section of the LLM's output for syntactic and grammatical errors. Furthermore, prompts should be engineered to a professional standard: they must utilize correct grammar, provide direct instructions, and eliminate any indecisiveness to ensure high-quality generation. In each output of the LLM model, certain key words and phrases may show and validation can be checked through Google Scholar. If zero matches are returned, that can be used as a huge indication of a hallucination created by Gemini.

8.4 Interaction Protocol and Prompting Strategy:

Since everyone in the group using and creating prompts can lead to disagreement of even unorganized work. Choose to create a guideline on what we will use Gemini for and how it will be used to prompt for certain things. As a group we created the guideline but we will be designating a single member to use and create prompts.

Workflow structure followed a simple basic set of steps; firstly, we would define the question and the desired outcome. Then, create a rough prompt of the task we wanted to be completed. Third, Prompt the AI and record the answer and analyze it. Finally, it is always a good idea to refine the prompt further and that way it would give a more specific and refined outcome which would be more useful. This process would make sure that the AI gives the best possible answer with the specific characteristics that we would be looking for.

Example: Role-play prompt "You are an AI team member of the group project for Software architecture. We are working on this conceptual architecture report for Gemini CLI. You are given the task of refining the following paragraphs and check the consistency between the them: [paragraphs here]" "Redo the refinement without changing the ideas in the paragraphs"

8.5 Reflection on Human-AI Team Dynamics:

Integrating a virtual AI teammate into our group workflow had a strong impact on our overall collaboration, communication, and productivity. It helped us articulate ideas more effectively and clarified confusions we had during discussions. This made the meetings and decision making process smoother and allowed us to work more efficiently as a team. Throughout the project, we found the AI to be especially helpful in clarifying key concepts, strengthening our understanding of project expectations, and confirming the quality/accuracy of our explanations and drawings. It also supported us when it came to brainstorming ideas for the report and ensuring that our work aligned with the marking rubric. Additionally, it is important to mention that in the beginning, the integration of AI caused a few disagreements. During our discussions, we were split between which AI model to use and the appropriate situations which it should be used for. But through open communication during meetings, we were able to reach an understanding which solved both problems.

Although the AI's involvement introduced some additional work in the form of drafting a usage guideline, creating prompts, and validating outputs (ensuring the LLM is not hallucinating answers); the time saved through AI integration greatly outweighed the time spent managing its use. By reducing the time and energy spent on understanding the concepts and deliverable expectations, we were able to focus more of our energy on the more difficult parts of the project.

Overall, our team learned that collaboration with AI can be a powerful tool for efficiency; allowing us to allocate more time and effort towards more demanding tasks like creating diagrams, architectural explanations, and strengthening the final presentation of our work. These lessons influenced all our members in a similar way, inspiring us to integrate AI as a collaborative tool during future projects moving forward.

References

[1] Google Gemini Team. "Gemini cli architecture overview." Official Gemini CLI Documentation, Accessed: Jan. 28, 2025. [Online]. Available: https://geminicli.com/docs/architecture/.

[2] D. Garlan and M. Shaw, "An introduction to software architecture," Carnegie Mellon University, School of Computer Science, Tech. Rep. CMU-CS-94-166, 1994. [Online]. Available: https://userweb.cs.txstate.edu/~rp31/papers/intro_softarch.pdf.

[3] J. Alateras. "Unpacking the gemini cli: A high-level architectural overview," Medium, Accessed: Feb. 6, 2026. [Online]. Available: https://medium.com/@jalateras/unpacking-the-gemini-cli-a-high-level-architectural-overview-99212f6780e7.

[4] Google Gemini Team. "Gemini CLI core: Tools API," Accessed: Feb. 10, 2026. [Online]. Available: https://geminicli.com/docs/core/tools-api/.

[5] Google Gemini Team. "Sandboxing in the Gemini CLI," Accessed: Feb. 10, 2026. [Online]. Available: https://geminicli.com/docs/cli/sandbox/.

[6] Google Gemini Team. "MCP servers with the Gemini CLI," Accessed: Feb. 10, 2026. [Online]. Available: https://geminicli.com/docs/tools/mcp-server/.

[7] S. Yao et al., "React: Synergizing reasoning and acting in language models," *arXiv preprint arXiv:2210.03629*, 2022. Accessed: Feb. 6, 2026. [Online]. Available: https://arxiv.org/abs/2210.03629.

[8] Google, *Gemini cli*, version 1.0, Nov. 18, 2025. [Online]. Available: https://github.com/google-gemini/gemini-cli.

[9] G. DeepMind. "Gemini 3 technical report. "[Online]. Available: https://deepmind.google/models/gemini/.

[10] D. Allemang. "Avoiding llm hallucinations: Data vs. metadata. "[Online]. Available: https://medium.com/@dallemang/avoiding-llm-hallucinations-data-vs-metadata-11829f116f16.

[11] Google Gemini Team. "Welcome to gemini cli documentation." Official Gemini CLI Documentation, Accessed: Jan. 28, 2025. [Online]. Available: https://google-gemini.github.io/gemini-cli/docs/.

[12] Google Cloud. "Gemini cli | gemini for google cloud," Accessed: Feb. 6, 2026. [Online]. Available: https://docs.cloud.google.com/gemini/docs/codeassist/gemini-cli.

[13] Google Cloud. "Best practices with large language models (llms)," Accessed: Feb. 6, 2026. [Online]. Available: https://docs.cloud.google.com/vertex-ai/generative-ai/docs/learn/prompt-best-practices.