Gemini CLI Conceptual Architecture Analysis

Ananya Kollipara (22kc34@queensu.ca)

Arlen Smith (22htl2@queensu.ca)

Christian Fiorino (22bqs2@queensu.ca)

Lillie Amos (22ryw1@queensu.ca)

Maia Turner (22pwg6@queensu.ca)

Vivian Webster (22nvp2@queensu.ca)

Conceptual Architecture Analysis

CISC 322 – Software Architecture

February 13th, 2025

0.0. – Abstract

The following report is a detailed analysis on our findings when researching the conceptual architecture of Gemini CLI:

This product is a free, open-source tool that integrates an LLM (Large Language Model) directly into the user's command line interface. Given permission, it can access all the functionality built into the terminal and perform executive actions based on prompts and allowances typed into the CLI by the user. It does this using a combination of layered and client-server architecture.

The user interacts directly with what we've deemed the "CLI component" by writing in their terminal as if it were a chat log with Gemini itself. The CLI component is responsible for managing user input, displaying itself on the terminal in a visually appealing way, and providing feedback and further permission questions to the user upon interaction.

The input handled by the CLI component is sent back to the "core component". This is responsible for providing Gemini AI with the prompt and surrounding context to generate a response, formulating a plan and necessary permissions to generate a solution, and interacting with any necessary "tools" to complete the given task. It sends updates back and forth to the CLI component as progress is made to communicate with the user. The tools mentioned earlier are a suite of options the core component has access to that can directly affect files on the user's computer. The permissions asked for from the user are almost always to allow usage of write-capable tools. This is an important step to avoid misinterpretations destroying the user's computer.

We qualify this system as a sort of hybrid between layered and client-server architecture because it closely resembles both forms. While we are literally a client sending messages to a server to be processed and given output, the account-based nature and permission hierarchy of the CLI and core components as well as the underlying layer of tools present at the bottom is reminiscent of layered architecture.

The structure of Gemini CLI's architecture allows a high degree of modularity, transparency, and speed. It's an impressive fusion of two very powerful tools that allow easy file manipulation, script comprehension, and vibe coding. Incredibly useful, especially for people without formal coding knowledge.

Contents

1.0. Introduction & Overview 3

2.0. Gemini CLI Architecture 4

2.1. Architectural Overview 4

2.2. Components & Subsystem Interactions 3

2.3. Evolution 3

2.4. Concurrency 3

2.5. Responsibilities 3

3.0. External Interfaces 3

4.0. Use Cases 3

4.1. Use Case #1 3

4.2. Use Case #2 3

5.0. Derivation Process 11

6.0. Conclusion 12

7.0. Lessons Learned 12

8.0. Data Dictionary & Naming Conventions 13

8.1. Data Dictionary 13

8.2. Naming Conventions 13

9.0. AI Report 13

10.0. References 15

1.0. – Introduction and Overview

This CISC 322/326 group project report contains an exhaustive detailing of our group's research on the architecture of the open-source Gemini CLI application. This includes our full analysis of its systems, use cases, and external components, as well as an explanation of and reflection on our process for writing this report, with potential takeaways when proceeding with the next steps of our project.

The purpose of this report is to document our findings and communicate our collective understanding of the conceptual architecture of Gemini CLI through the analytical lens of the concepts presented in class. By the end of this report, a full understanding of both the conceptual architecture of Gemini CLI and the process used by our group to research and document our findings on the topic will be provided.

Google Gemini is a Large Language Model (LLM) developed and distributed for public use by Google, with an API accessible by those with a Google account. Gemini CLI is a free and open-source Command Line Interface (CLI) application which makes use of these API calls to integrate the Gemini LLM with a traditional CLI. This integration between the user's locally run application and Google's Gemini servers allows the user access to a variety of useful LLM-empowered functions for reading, writing and editing local files. To do so, the user writes prompts for the LLM instead of terminal commands, receiving responses and managing Gemini's agentic tasks on the user's files from within the CLI window.

The report will be organized into sections, beginning with a detailed report on the entire architecture of Gemini CLI, including its components and layers, interactions between internal subsystems, and the control flow of using the application. More report sections will also exist containing Gemini CLI's interactions with external interfaces and two use cases describing how the application would complete certain user actions. In addition to these reports on Gemini CLI's architecture, there also exists sections on how our group arrived at these discoveries, a summary of our conclusions, lessons we've learned during the process, and appendixes on our data and naming conventions, a report on the usage of generative AI's assistance in the creation of this report, and a final section with links to all references mentioned throughout the document.

The conclusions we draw rely upon the existence of two main packages: the CLI package, containing the user-facing portion of the application, and the core package, containing all the back-end logic and calls to the Gemini API. The conceptual architecture draws on a mix of both layered and client-server styles, with the CLI and core packages each representing layers on the client side, and the API calls to Gemini and various other Google services comprising the server side of the application. Gemini CLI has a variety of tools it can make use of for interacting with the user's files, and more can be added through third party extensions.

2.0. – Architecture

2.1. – Architectural Overview

We will be analyzing Gemini CLI through the architectural styles of client/server, and layered. Client/server is the clear choice to analyze the relationship between Gemini CLI, and Gemini itself. Gemini consists of servers which provide API calls for functions that each individual instance of the Gemini CLI calls. This is a clear server/client relationship. There are only a few specific API calls made to Gemini, so this distinction means we can keep the server's functionality quite abstract, only concerning ourselves with the fact that it has the ability to take a prompt and return a response, while getting into much more detail as to the functions and processes of the client, Gemini CLI. (*Gemini CLI Architecture Overview*)

The system is clearly divided into 3 distinct layers. There's user facing portion located at "packages/cli", which we'll be referring to as the CLI layer. The part which communicates with Gemini itself is located at "packages/core", and will be referred to as the core layer. The tools of the programs which safely execute Gemini's requests are located at "packages/core/src/tools", which makes up the tools layer. It's worth noting that our tools layer is contained within the directory core layer. Despite this, it's still clearly worth analyzing as a separate layer. This decision is made by the documentation itself, as well as more or less all analysis we could find, since the tools directory has a unique, understandable and important role in the execution of the program. (*Gemini CLI Architecture Overview*)

2.2. – Components & Subsystem Interactions

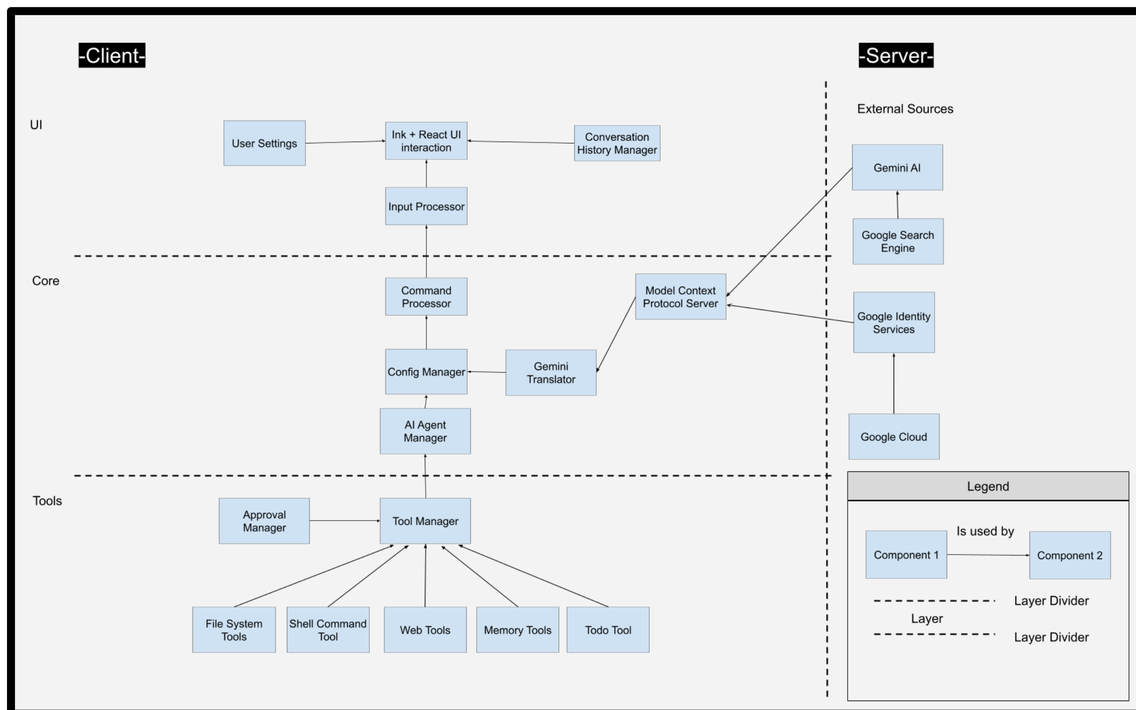

Figure 1. Box-and-Line Diagram of Gemini CLI

The user-facing portion, the CLI layer, located at the directory "packages/cli" processes your inputs and sends them to the correct place, usually the core so Gemini can process it, but there

are some exceptions, such as shell mode and shell commands, which would be passed directly to the terminal. It manages the history of these sessions of queries, to pass alongside each specific prompt, with options to rewind to earlier in the conversation, but sessions are also saved long term so they can be restored. It uses Ink, a React based renderer for command line apps, to render all program output that needs to be passed to the user, such as asking for different permissions, or the text outputted by Gemini. It gives options for different themes, so you can customize the UI to look how you want. It also can present the user with the possible settings to change and store their response. (*Unpacking the Gemini CLI: A High-Level Architectural Overview)*

The CLI layer is split into Ink + React UI, which sends user inputs to the input processor and shows users the output of Gemini prompts and when approval is requested, Input Processor, which sends inputs from user to conversation history manager to be stored and command processor to be processed, Conversation History Manager, which stores and can send the history of messages that have been sent, and User Settings, which stores and retrieves user's settings and preferences in various files. (*DeepWiki Architecture Overview*)

The Gemini communication portion, the core layer, at the directory "packages/core" communicates with what we will be analyzing as our server, Gemini itself. Gemini, in this case, refers to the ability of Google's servers to take a prompt and return the generative AI's response to it in the specified manner. The core layer sends prompts to Gemini based on the current state and information. At first, it takes the user's prompt and session information from the CLI layer and creates the prompt for Gemini using additional context, such as the project structure and error details. Part of this additional information it provides is the tools which it has available to it, gotten at some point from the tools layer, and how its response can call those tools. When it receives a response, it sends user bound output to the CLI layer and checks for any of these tool calls. For any tool calls found, if necessary, it asks for user approval with a message specified by Gemini's prompt and communicates these calls to the tools layer accordingly. Once it receives the output of these tools, it can send a new prompt with these tool's outputs. If you deny permission to run the tool it returns this instead. This repeats until there are no new tool requests. (*Unpacking the Gemini CLI: A High-Level Architectural Overview*)

The core layer is split into Command Processor, which deals with cases such as shell mode commands and sends the result to the config manager, Config Manager, which determines what to run next, whether it be prompting gemini or using tools, AI Agent Manager, which organizes the use of an AI Agent, using tools if requested, Gemini Translator, which truncates and abstracts your prompt such that it can be parsed by Gemini, and the Model Context Protocol Server, which takes a query and context it was asked in and puts it in the industry standard format, sends it to Gemini. (*DeepWiki Architecture Overview*)

The tools layer at the directory "packages/core/src/tools" is responsible for keeping track of the available tools through a directory, and running a tool specified by the CLI layer. The way everything is structured makes it quite easy to add new tools, but there are a number currently provided. The file system tools let you read, search, and write to specified files or folders. There is a tool to run a shell command in the terminal and get the output. The web fetch and web search tools give information on a given URL or google search. The memory tools let you save and recall specific facts. The todo tool lets Gemini split up the given task and solve each one

individually while informing the user of its progress, showing which parts have been completed. (*Gemini CLI core: Tools API)*

The tools layer consists of a tool manager, which can list the available tools, and when called to execute a tool, asks Approval Manager for approval to run the tool if necessary and then runs it, the Approval Manager, which asks the user whether they want to run the specified tool under the specified conditions, with a custom prompt if provided, and the various tools, which have been described above. (*DeepWiki Architecture Overview*)

Externally, there is Gemini AI, which takes a query, if it is a search query it checks those resources, and then returns a response, Google Identity Services, which verifies your google account to see what data and features you have access to, Google Search Engine, which can return results of online articles in relation to a given query, and Google Cloud, which allows the program to engage with Google's different servers and services. (*DeepWiki Architecture Overview*)

2.3 – Evolution

The different components of Gemini CLI can evolve in different ways. Every layer of Gemini CLI has evolved and will continue to evolve over time through system updates. The core layer regularly has new features released and has bugs or issues fixed through these updates. The layer can also be updated to add new Gemini AI models when they are released. The CLI layer is additionally kept up to date whenever a new feature that requires it is implemented. This requirement is among the reasons for the client's monorepo structure. The CLI layer is also often updated to increase clarity while conveying information, to add new inputs, or to change functionality of existing inputs to further increase clarity. The tools layer is improved through adding new tools and by enhancing functionality of existing tools. (*Gemini CLI Github Releases*)

Outside of system updates, Gemini CLI also allows users to evolve the system themselves through its Extension system. These allow users to import user-defined prompts, MCP servers, custom commands, subagents, and specialized agent skills into Gemini CLI to expand its capabilities. Users can either create their own extensions either from scratch or by using built-in templates as a foundation for the type of extension the user wanted to make. These extensions are written in JSON, and users are expected to have a basic understanding of Node.js. Users can also download existing Extensions created by other users through an Extension gallery found online.

Additionally, users can extend the capabilities of Gemini CLI through specifying Agent Skills to the system. These allow users to have the system give specialized expertise, procedural workflows, and task-specific resources to AI models while interpreting prompts. Agent Skills can be either created automatically through Gemini CLI's skill creator or manually inside the system's directory. Unlike context files that provide persistent guidelines, Agent Skills represent on-demand expertise that Gemini AI models can call upon if it identifies a task that relates to the Agent Skill. These enable complex workflow packaging, repeatable workflow creation and resource bundling.

2.4 – Concurrency

Overall, there is not too much concurrency present in the Gemini CLI system itself. Most of Gemini CLI's concurrency is done within the Gemini AI models while processing prompts, which is outside Gemini CLI's local architecture. However, Gemini CLI does demonstrate concurrency in handling multiple tool requests in calls from the Gemini AI model.

This behaviour is managed by a tool scheduler found within the core layer. When receiving multiple tool requests, the core will iterate through each tool and start each process without waiting for the previous tool process to finish. The scheduler will then bundle all the results from each tool in a single response after every concurrent operation completes. Once this is done, the consolidated response will be sent back to the Gemini AI model. This is done over executing each tool sequentially to increase the efficiency of processing multiple tool requests sent from the AI model. (*From Prompt to Code Part 1: Inside the Gemini CLI's Execution Engine*)

2.5 – Division of Responsibilities

The frontend and backend of the program are in different folders, and the tools layer is in a specific folder within the backend. The decision to divide the folders in this way servers two purposes. First of all, it means that developers of the frontend and backend of the software can work entirely independently of each other. It also allows for potential future use of the modularity of these programs. For instance, you could design an entirely new frontend which can prompt Gemini in the same way, or you could build an alternative backend utilizing a different Generative AI tool or prompting mechanism. While they are contained in separate folders, it utilizes a monorepo structure, meaning all of the Gemini CLI system we are looking at is contained within a single repository. This is because while independent development and modularity are important, at the end of the day this is a single product that is engaged with in one complete package, and changes to the frontend or backend usually need to be reflected in the other, so keeping one repository with everything makes it easy to ensure the end product the user downloads will have consistent frontend and backend functionality. In summary, the structure of this project enables developers to easily work on any of the layers of the system, but provides an opportunity for everything to be tested and confirmed to work before release. (Unpacking the Gemini CLI: A High-Level Architectural Overview)

3.0. – External Interfaces

Gemini CLI interacts with several external interfaces to generate responses to prompts, authenticate users with additional services, and search for information on the web. Most of the external services that Gemini CLI interacts with are all Google-related services that work alongside Gemini's AI models. To connect to any external service, Gemini CLI will establish a Model Context Protocol (MCP) server to act as a bridge between the CLI and any other external systems.

The most notable of external interfaces that Gemini CLI interacts with are the Gemini AI models that are supported. Any model from the current family of Gemini AI models are supported, and are what allow Gemini CLI to come up with responses from user input. Gemini CLI will transmit user prompts to a selected Gemini Model for it to interpret, along with any context files or agent skill files to the AI model to dictate its behaviour while generating a response. The model will then send back the generated response to Gemini CLI for it to display to the user.

Gemini CLI will also interact with Google Identity Services to authenticate users using the CLI. Users cannot use Gemini CLI without authenticating with a Google Account in some form. The easiest way for users to authenticate is to simply log into a Google Account, in which the CLI will open a web prompt for users to log into upon its first use. After the user logs in, the CLI will receive and cache login credentials locally to use in future sessions. Authenticating is required so that Gemini CLI knows what quotas, limits, and additional external Google services the user has access to.

Another external interface Gemini CLI interacts with is Google Cloud. After authenticating with a Google Account, users with Google Cloud subscriptions can use Gemini CLI to work within external Google Cloud projects. Within these projects users can allow Gemini CLI to access various services Google Cloud offers, such as virtual machines to deploy apps or cloud storage. This allows Gemini CLI to access Vertex AI, an AI development platform used to build, deploy and scale machine learning models.

Lastly, Gemini CLI will interact with Google Search through its web fetch tool. The tool essentially allows the user to make Google Searches within the CLI itself. It will take a search prompt imputed by the user and send the query to Gemini API. The API will then utilize Google Search to perform a web search, and return a generated response based on search results.

4.0. – Use Cases

4.1. – Use Case #1

The below diagram showcases the use case where a use is setting up a virtual environment for a new project. After starting the agent, they ask Gemini to set-up the project through a plain English message in the terminal. This message will be recorded by the CLI package, and isolated and sent to the core package. The core package will plug it into the Gemini AI as a prompt. Gemini then has a series of tools (managed by the core package) which replicate the functionality of the user's CLI, with these replicated tools requiring explicit permission from the user to run so that potentially harmful prompts don't destroy the user's computer. To do this, the core will format an ask for permission and pass it to the CLI package to be presented to the user in plain text. If given permission, the CLI package responds to the core package and the request is carried out. The agent creates the virtual environment successfully with the specifications asked of it, all without a single terminal-comprehensible line written.

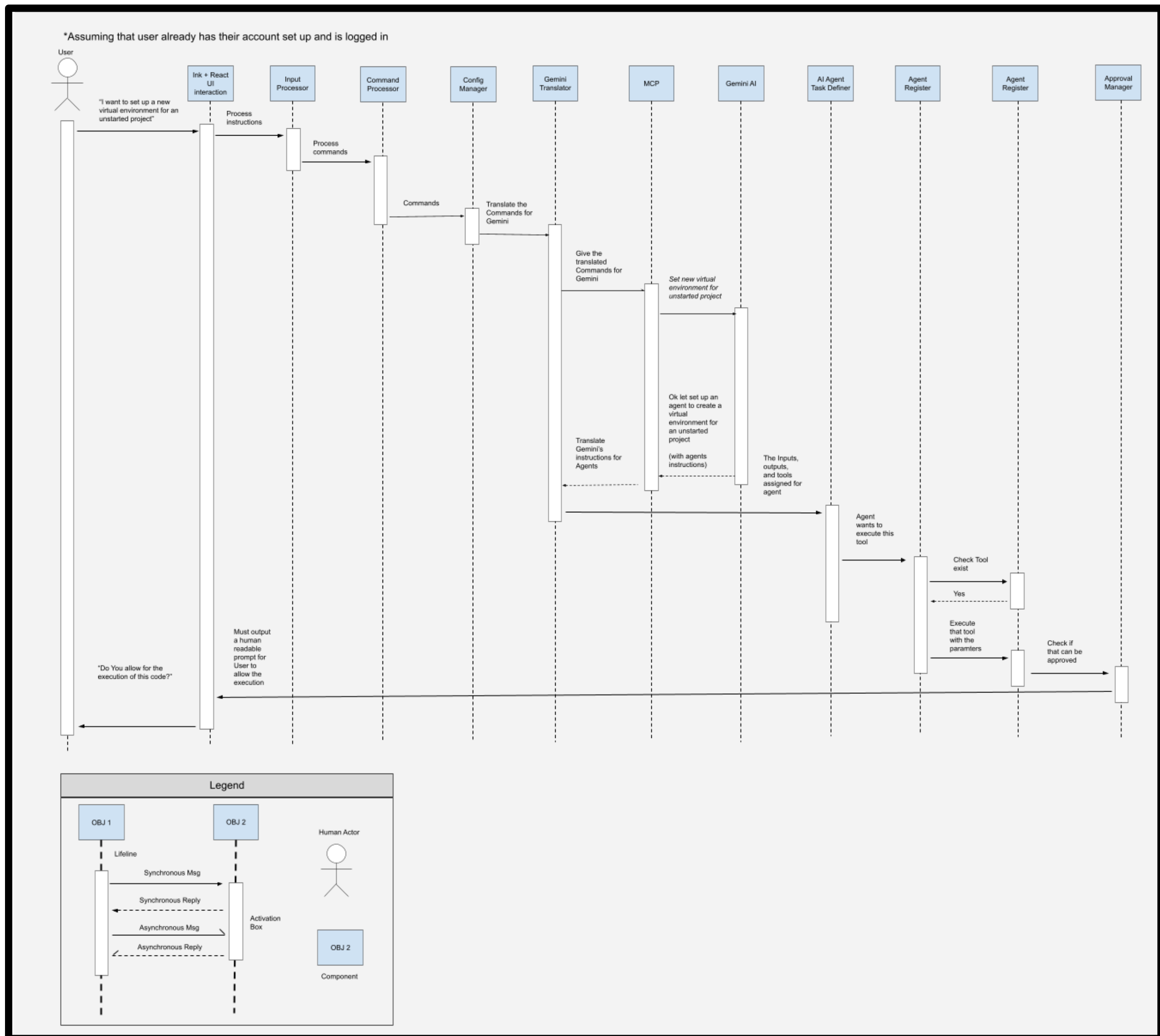

Figure 2. Sequence Diagram to create a virtual environment with Gemini CLI

4.2. – Use Case #2

Gemini CLI can be used to research a concept without the user needing to leave the terminal and open a separate application with a search engine. Let's say the user wants to know "who won the Superbowl?" is. They can ask the Gemini CLI to ask the AI to inform the user what it is. Without leaving the interface the user now knows who won the Superbowl.

The below diagram showcases the use case where Gemini CLI can be used to research a concept within the terminal.

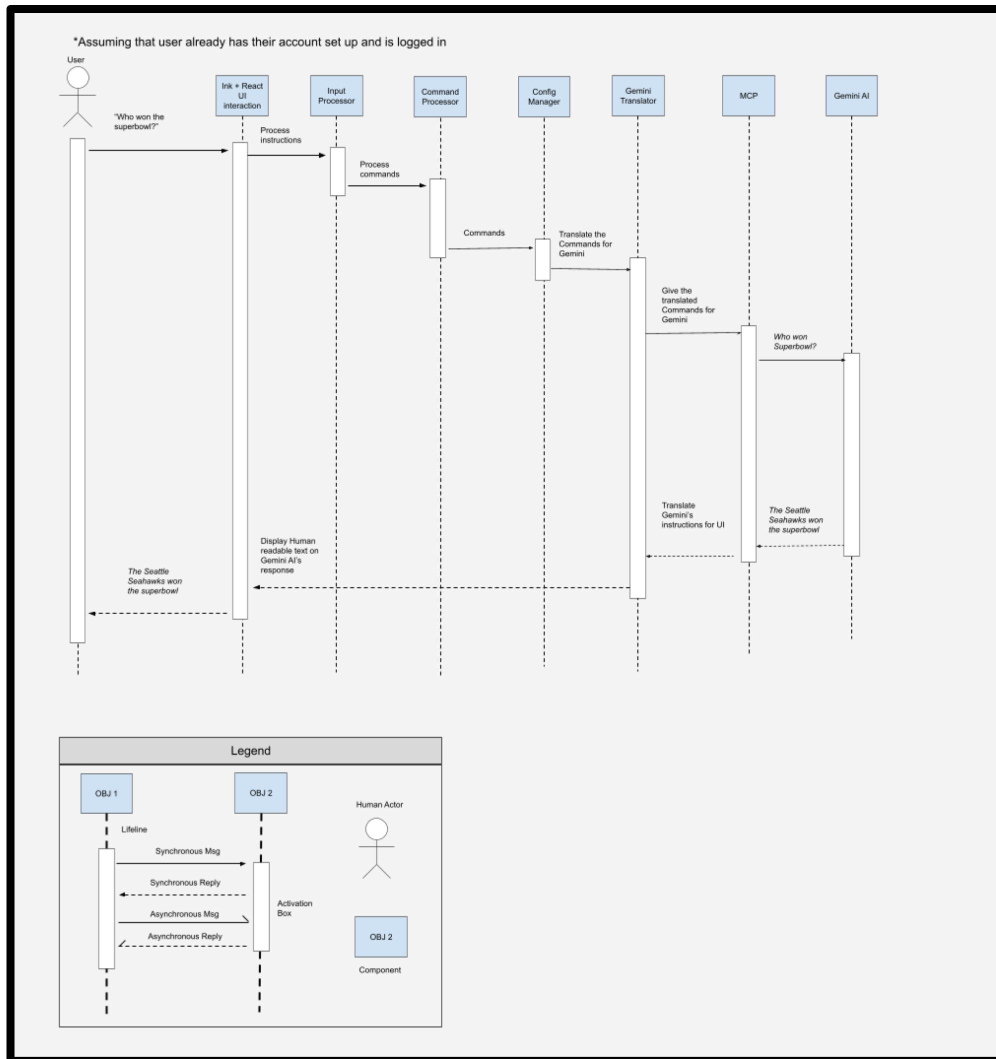

Figure 3. Sequence Diagram to search using Gemini CLI

5.0 – Derivation Process

The derivation process for determining the architectural styles of Gemini CLI involved a significant amount of research and iteration before we settled on the client-server and layered styles. We started this process by gathering a list of resources to consult. This included the official Gemini CLI documentation along with additional resources posted through OnQ. We also utilized our virtual member to have it search for more resources we could review. Once the list of resources was finalized, we collectively analyzed them and derived the main packages of Gemini CLI. Based on these packages and their interactions found through the resources, we individually created use cases and rough sequence diagrams for using Gemini CLI, which we then used to draw a rough box-and-line diagram to better visualize the system's structure. From here, we met as a group to discuss the implications of the diagrams and finalize our decisions on what the architectural styles could be.

Through this process, we noticed that the core components could be partitioned into several different “layers”. Such as a “cli” layer and a “core” layer. This implied that Gemini CLI used a layered style in its architecture, as the structure had clearly separated certain responsibilities from one another, and because each component (on its separate layer) relied on services provided from components on a lower layer. Additionally, a layered style allows Gemini CLI’s architecture to be more modular, which enables one of Gemini CLI’s key design principles of allowing users to develop their own extensions for the system. (*Gemini CLI Architecture Overview*)

Next, we considered how users would connect to and interact with Gemini CLI, which led us to considering a client-server style as the main style Gemini CLI uses. This was because the “core” component uses an API to communicate with the Google Gemini API, implying that Gemini CLI needs an internet connection to operate, and thus uses a network to connect components. Furthermore, Gemini CLI has a “cli” component that handles the user input and is separated from the more complex back-end operations of the CLI, indicating a divide between client and server that would suggest a client-server style. Additionally, a client-server approach would allow a more straightforward distribution of the data provided by the user, allowing Gemini CLI to provide responses in a shorter amount of time. (*Gemini CLI Architecture Overview*)

While we decided that the client-server and layered styles were the most prominent, we considered a few alternative architectural styles. We initially considered that Gemini CLI could use a repository style, as the Gemini LLM model could act as a repository of information that the system can access to respond to user prompts. The repository style could also be used to retain information from previous prompts for longer periods of time, allowing for better responses from the LLM. However, we decided against this approach as using a repository style would require more robust infrastructure to support the system and make it less modular, slowing Gemini CLI’s performance and going against one of its core design principles.

Along with a repository style, we also considered an object-oriented approach for Gemini CLI. This was because object-oriented components can have their implementations changed without causing inconsistencies with other clients. This allows the components to be more modular, which works well with Gemini CLI’s design principles. However, we decided that this was mainly encompassed in the backend components of Gemini CLI (such as “core” and “tools”) rather than the whole architecture. Which the client-server and layered architecture styles do a better job of representing.

6.0. – Conclusions

The conceptual architecture of Gemini CLI utilizes a layered and client-server style. It also prioritizes modularity such that users can create their own external extensions to alter or improve on Gemini CLI’s behaviour. The CLI interacts with Gemini’s API through a controlled system that sends all required context from the user’s prompt to the file(s) being asked about in one big prompt for Gemini to respond to with sequential agentic tasks to be executed. These tasks can make use of tools that interact with the user’s files on their local computer through reading, writing, editing, or other means, with potentially destructive actions needing express consent from the user prior to their execution. Overall, Gemini CLI acts as a bridge between the classic command line terminal and the modern LLM prompting interface, allowing the complex execution of terminal commands through plain English prompting.

7.0. – Lessons Learnt

During this project, we learnt more about the architecture of Gemini CLI, but one of the key takeaways was the importance of security in an architecture. This was showcased with the "tools" package, which was used to ensure that the Gemini AI's outputs wouldn't negatively affect the user's environment. Going into this project, we thought that it would be very difficult and dangerous to have an LLM work within a terminal without it causing damage to the user's systems. But through learning about this "tools" package, we understood that there were effective ways of protecting the user's systems from the unpredictability of an LLM's outputs.

Additionally, one of the biggest lessons we took away from this project was the importance of starting projects early and understanding all the requirements of a project. Initially, we did not prioritize working on this project partially due to other assessments, but primarily because we thought we had a good understanding of the project requirements and thought we were in a good position to complete it. However, as we began to prioritize this project, we realized that our understanding of its requirements was either incomplete or incorrect, causing a rushed completion of the assignment. Additionally, we also learnt that our approach to this assignment was not as structured as it could have been, as we didn't set any soft deadlines for when certain sections were to be completed by.

If we were to do things for this assignment differently, we would start researching and writing sections of the report much earlier. This would not only allow us to better understand what needed to be completed for the project, but working on the project earlier also allows us to ask the professor or teaching assistants any questions in case we run into roadblocks on our projects. Along with this, we would also have scheduled a large initial meeting to review all of the project's requirements, ensure everyone is on the same page, and to assign tasks and set soft deadlines. This meeting would have improved the structure of how we approached the project and would have allowed for a smoother workflow when working on it.

8.0. – Data Dictionary & Naming Conventions

8.1 – Data Dictionary

Include a glossary that briefly defines all the key terms used in your architecture, giving when appropriate, the "type" of the item being explained. Additionally, list any naming conventions used in the described architecture. Explain any abbreviations that you use.

- **Gemini AI | Gemini LLM** – Refers to one of Google's LLM models.
- **"core"** – package def
- **"cli"** – package def
- **"tools"** – package def
- **Monorepo Structure** – A project contained within one repository

8.2 – Naming Conventions

- **CLI**: Command Line Interface
- **LLM**: Large Language Model
- **AI**: Artificial Intelligence

- **API**: Application Programming Interface
- **MCP**: Model Context Protocol

9.0 – AI Report

The following section discusses our use of a virtual AI teammate (referred to as "the LLM" going forward) during this project, our methodology when using the LLM, and how it affected our group's workflow.

The first decision made regarding the LLM was which model would be used. This was a group decision, and we decided on using Gemini v3 Fast as our virtual member. This decision was based primarily on what tasks we believed the LLM would be required to do for the project, which we decided on by reviewing the project requirements and discussing between ourselves what the LLM could optimize. From these discussions, we decided that there was only one task appropriate enough for the LLM to perform. That being gathering resources to help us understand Gemini CLI's architecture. (Also Google Searches)

We believed this was the best use for the LLM as it could scour the internet for dozens of possible resources exponentially faster than any member of the project could. It was also easy to validate this task and ensure that the LLM wasn't hallucinating, as discussed later in this report. Along with this, we primarily wanted to complete the more intensive tasks ourselves (such as creating the use cases or drawing the sequence diagrams) so that we can better learn the concepts those tasks taught, which limited the types of tasks the LLM could perform to be relatively smaller tasks such as searching the internet for resources. As this was not an extensive task that required a significant amount of power, we decided to use one of the fastest currently available LLM models, that being Gemini v3 Fast. Additionally, we decided on using a Gemini model specifically as we believed, due to its relationship with Gemini CLI, that it would be able to provide the most accurate information on the system's structure.

When it came time to interact with the LLM, we decided it would be simplest to assign one member as the "prompt engineer". The engineer's job would be to create prompts for the LLM, fine-tune the prompts to generate better results, and return the LLM's outputs to the rest of the team.

Due to the limited scope of tasks the LLM was assigned, we only had one key prompt that was used for getting resources from the LLM. The prompt initially assigned the LLM a persona as a "researcher for Gemini CLI" and asked the LLM to output its results as a list of resources with specific details pertaining to each source. When using this prompt, few refinements needed to be made to improve the quality (which were mainly related to the types of resources the LLM found, such as ensuring it didn't return AI-generated resources). The finalized version of this prompt is presented below:

> *You are a researcher for the application Gemini CLI. Your task is to find resources that contain information on the conceptual architecture of Gemini CLI. This conceptual architecture specifically includes the general architecture of the application, the architecture styles (such as repository or pipe-and-filter) that Gemini CLI uses, and the global flow of information within the application.*

You are to return a list of resources that contain explanations for the above information. For each resource, you should include where the resource was found, who wrote the resource, what information it provides relating to Gemini CLI's architecture, a link to the resource, and why you believe it is a credible resource. The list of resources should contain at least seven resources. These resources must NOT be AI-generated articles.

The above information should be organized in a tabular format, where each column represents one of the above pieces of information. Do you understand your task?

To validate the LLM's outputs, we first refrained from having the AI generate any text or images we would use in the report. This would reduce the possibility of hallucinations from the LLM affecting the report's quality. Secondly, when prompting the LLM for resources, the prompt engineer asked the LLM to provide links to the resources it found. From here, the engineer fact-checked the LLM's claims by reviewing the resource and cross-referencing the resource with the LLM's claims. If there were contradictions, the engineer would either ask the LLM for clarification to hopefully resolve the inconsistency or restart the LLM with the initial prompt. If the LLM's description of the resource was accurate to what the resource actually stated, then we believed that the LLM's information was safe to consider.

Overall, we found that the LLM's resource searching only provided a small contribution to the final report. As while the resources it found were useful for reinforcing our understanding of Gemini CLI, it often provided information that we were already aware of. As such, we will grade the LLM's overall contribution as 10%, as it was not able to effectively contribute to the final report.

However, we are aware that we did not fully integrate the LLM into our group's overall dynamics. As we had assigned a single member to work with the LLM and restricted the tasks the LLM would perform. The sole task we had the LLM perform prevented it from saving us a lot of time, but it also didn't create new work outside of creating and tweaking the initial prompt. The small workload as well as the LLM only interacting with one member also caused it to influence our team's decision-making processes to an insignificant degree, if it influenced them at all. Luckily, this does mean that the LLM caused no challenges while working on the report, but it also means that we did not use the LLM to its fullest potential. From this report, we've learned that in order to effectively collaborate with the LLM, we not only need to allow it to provide more value to the report, but we also need to have more group members collaborate with the LLM. As such, in future projects (specifically A2 of this course) we will try to assign the LLM more tasks so that it leaves an impact on our group's dynamics, along with allowing the whole group to interact with the LLM instead of a sole prompt engineer.

10.0 – References

Alateras, J. (2025, July 8). *Unpacking the gemini CLI: A high-level architectural overview*. Medium. https://medium.com/@jalateras/unpacking-the-gemini-cli-a-high-level-architectural-overview-99212f6780e7

Architecture Overview: Google-gemini/gemini-CLI. DeepWiki. (2026, February 6). https://deepwiki.com/google-gemini/gemini-cli/1.1-architecture-overview

Datta, P. (2025, June 30). *From Prompt to Code Part 1: Inside the Gemini CLI's Execution Engine*. The AI Positive Substack. https://aipositive.substack.com/p/from-prompt-to-code-part-1-inside

Gemini CLI Architecture Overview. Gemini CLI. (n.d.). https://geminicli.com/docs/architecture

Products and Services | Google Cloud. Google. (n.d.). https://cloud.google.com/products

What is the Model Context Protocol (MCP)?. Model Context Protocol. (n.d.). https://modelcontextprotocol.io/docs/getting-started/intro

Gemini CLI Github Releases (n.d.). https://github.com/google-gemini/gemini-cli/releases

Gemini CLI core: Tools API (n.d.). https://geminicli.com/docs/core/tools-api/