Gemini CLI Concrete 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)

Concrete Architecture Analysis

CISC 322 – Software Architecture

March 13th, 2026

0.0. – Abstract

This report covers our findings while investigating the concrete architecture of Gemini CLI's systems. Our newfound knowledge comes from examination of Scitools' Understand's analysis of subsystem dependencies, as well as the blame logs of GitHub.

Our understanding of the conceptual architecture begins partially changed, being shifted to fit the findings reported by the professor on OnQ. Our report analyzes the high-level architecture of Gemini CLI using the *Understand* tool, along with breaking down one subsystem (the "UI Terminal" component) into further subcomponents and closely examining the dependencies between them.

Upon reflection, we discuss how the systems are much more interlinked and communicative than we were led to believe by overview documentation and informed inference. This revelation was found by finding divergences between our conceptual and concrete architectures. Along with this, two use cases are presented, this time keeping track of which functions are used to pass information between components and layers.

Finally, our experience working on this report is recorded briefly. Throughout this research project, our team gained a deeper understanding and appreciation for the level of complexity that goes into a large-scale project and its organization. While certainly diagrams may become crowded and noisy no matter what, a solid hierarchy and flow of information as well as a cohesive architecture structure is absolutely necessary to keep things organized among hundreds of individual scripts.

Contents

1.0. – Introduction and Overview

The purpose of this report is to effectively communicate our findings from our detailed research and analysis on the concrete architecture of Google's open-source command line interface program powered by their Gemini large language model (LMM), formally known as Gemini CLI. This report aims to compare and contrast our new findings with our previous understanding of Gemini CLI's conceptual architecture, improving our overall knowledge of software architecture through a thorough analysis of Gemini CLI's architectural composition.

A major component of this document is our high-level analysis of Gemini CLI's concrete architecture through mapping it's components in SciTool's Understand, a piece of software provided for this course with a specialization in architectural analysis. This analysis will be complete with concrete box-and-line diagrams of Gemini CLI's components, as well as our updated conceptual box-and-line diagram from both before and after our analysis of the concrete architecture. Our original conceptual diagram has been updated to reflect the global feedback provided by the professor, aiming to provide full context for the journey taken between our last report and this one.

Another important section details our deep dive into Gemini CLI's UI Terminal subsystem with the same level of scrutiny as the high-level analysis. Specific details on the composition of the UI Terminal are provided, including an analysis of its subcomponents and their interactions. Through this honed-in subsystem investigation, harder details on how Gemini CLI displays visuals, handles user settings and authentication, and processes input can be provided.

In both our high-level and subsystem sections, we provide further subsections describing our reflexion analyses, that being how our understandings of concrete architecture have changed and evolved from our previous conceptual architecture report's understanding of how Gemini CLI functions. The first analysis provides a deep dive into the origins of the Agent Manager's deeply held dependency on the Tool Manager, despite them never directly interacting conceptually. The second analysis further expands on the differences between conceptual and concrete dependencies, reporting on the UI Terminal's excessive composition of almost entirely bidirectional dependencies between its subcomponents.

The two use cases we have provided are updated versions of the very same use cases in our conceptual architecture report, providing a less abstract view of the internal workings of Gemini CLI while in use. These use cases provide sequence diagrams referencing file names and function calls directly that pull information through Gemini CLI's subcomponents and eventually propagate back upward to complete a given task.

Our derivation process is explored through our use of Understand to create concrete box-and-line diagrams detailing both Gemini CLI's high-level architecture and its UI Terminal subsystem's architecture. Details on our sorting of subfolders into Gemini CLI's components, and individual files into the UI Terminal's subcomponents are explained. Ultimately, this section aims to provide a fuller understanding of our final Understand project files used to further our research into Gemini CLI's component interactions.

After providing a summary of our research conclusions, we provide more detail on the lessons our group has learned throughout the process of creating this report. Our struggles with Understand in utilizing its architectural representation tools and iterations on our group's organization strategies and division of work are reflected upon in this section.

Finally, the last sections of our report contain a compilation of naming conventions and key terms used throughout our report for easy viewing, as well as a report on our group's usage of OpenAI's ChatGPT-5.3 large language model in the creation of this report. To wrap it up, all external references used are neatly cited according to the APA format.

2.0 – Conceptual Architecture Breakdown

Our revised conceptual architecture will continue to utilize the Layered and Client-Server architecture styles as established in A1. After reviewing Professor Adam's global feedback on A1's box-and-line diagram shared on OnQ, we decided that our A1 conceptual architecture may have been too low-level and decided to simplify it. The results of this simplification can be seen in the below box-and-line diagram:

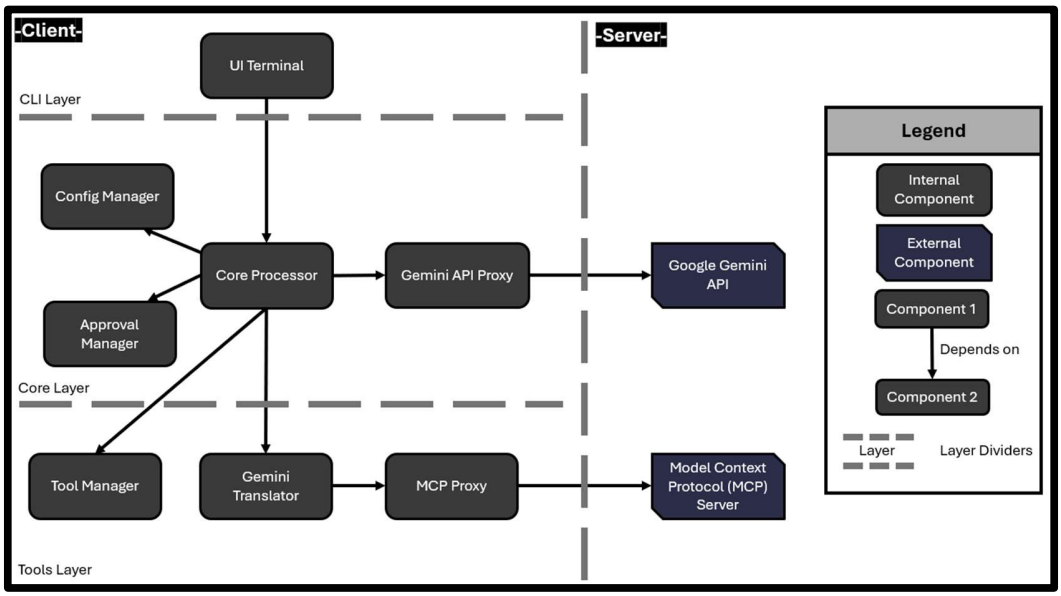

Figure 1. Updated Box-and-Line Diagram of Gemini CLI

We started revising the conceptual architecture by merging all the subcomponents of the more high-level view of the CLI layer as we noticed it acted very similarly to the [Terminal Interface] found in Professor Adam's diagram. Another area that was simplified was merging all the component tools within the Tools layer into the [Tool Manger] for a more streamlined process. We also modified the structure by making the [Core Processor] component dependent on the components of the Tools Layer

The [Gemini Translator] component has been relocated to the Tools Layer, functioning like a tool for the CLI to translate AI commands, along with the [MCP Proxy]. We believe our [Gemini Translator] operates similarly to the [External Integration Layer] in the professor's architecture as it is executing the tool from the client for the MCP servers. As such, the [MCP Proxy] now points to the external [Model Content Protocol (MCP) Server] component to connect with the optional MCP servers.

For the remaining server-side components, we have merged the [Gemini AI] and [Google Search Engine] components from our A1 conceptual architecture into the [Google Gemini API]

component as a higher-level view of these functionalities. Another change we made was moving the [Approval Manager] component from the Tools Layer to the Core Layer, as we believe based on the professor's feedback that it makes more sense for the [Core Processor] component to directly depend on [Approval Manager] rather than it needing to depend on it through [Tool Manager]. Lastly, we removed the [AI Agent Manager] component as the [Core Processor] component will handle the AI task delegations.

3.0. – High-Level Concrete Architecture

The concrete architecture of this program is divided into seven different parts. The UI Terminal is the frontend of the program. This subsystem is not dependent on any other subsystem, which aligns with our conceptual understanding. The remaining subsystems have a lot of bidirectional dependencies. This is not in line with our conceptual understanding, but the actual implementation of Gemini CLI causes these bidirectional dependencies to appear. This will be discussed further later in this section.

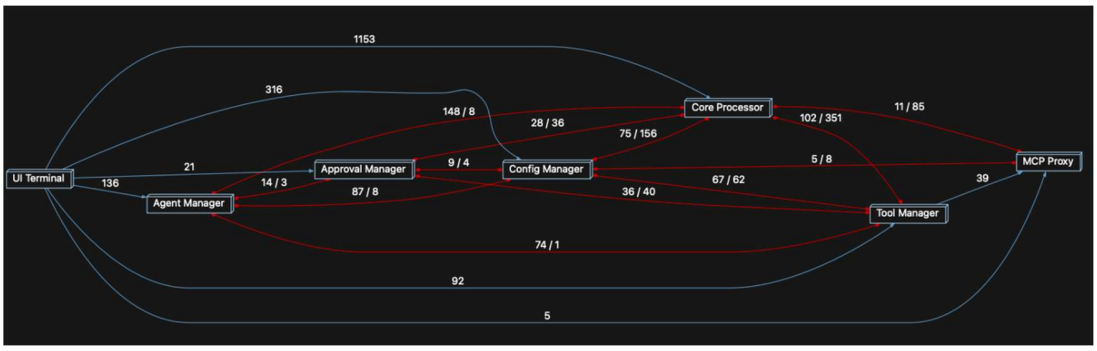

Figure 2. Concrete Architecture of Gemini CLI by subcomponent

We will discuss the six other components and what the dependencies between components can look like by analyzing the interactions between the UI Terminal component and each other component. In addition, this will demonstrate the purpose of the project's file structure, which we've mostly preserved within our subsystems and how the scripts within different folders interact with each other.

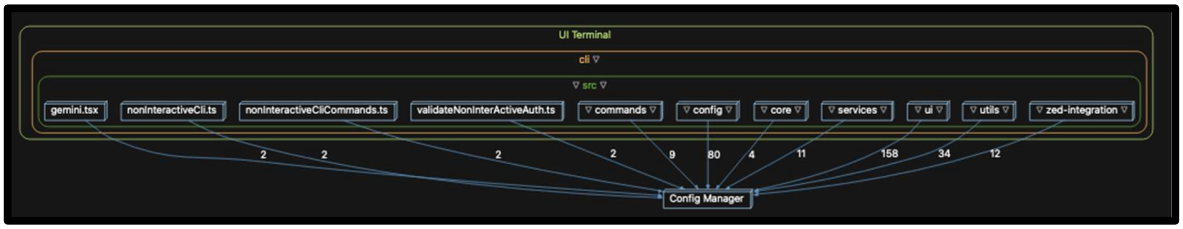

Figure 3. Dependencies between UI Terminal and Config Manager

The Config Manager stores backend-related settings and variables throughout the program. Its code is entirely contained within one folder, mostly contained within a single file. This file, named config.ts, is used throughout the UI Terminal component and throughout most of the subcomponents of this project. In the UI Terminal component specifically, it is used to set and get several different configuration variables throughout the subsystem.

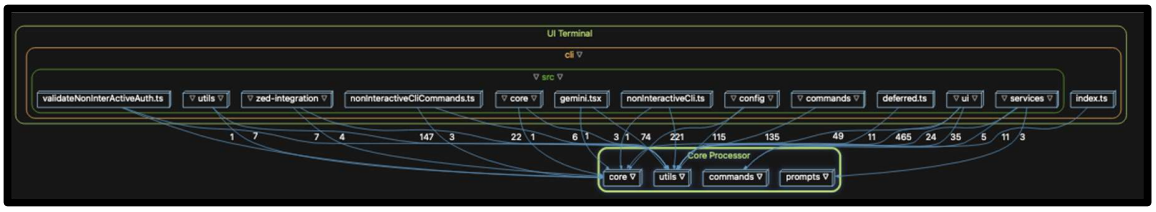

Figure 4. Dependencies between UI Terminal and Core Processor

The Core Processor component is a very large subsystem that acts as a central delegator to most other subsystems. The folders present within it are important to differentiate and help understand the subsystem as a whole. The “commands” folder deals with “/” and “@” commands present within a user’s prompt. The “services” folder checks when certain features are available and sends a message if they are not. The UI folder’s corresponding “commands” folder uses the core’s systems to perform several tasks, including listing extensions, initializing GEMINI.md files, or handling memory. The “core” folder is used by a variety of other folders for multiple tasks. It is used by the UI Terminal component’s “auth” folder to get the authentication status and type, its “commands” folder to log messages, its “components” folder to check token usage and limit, and its “contexts” folder to set the auth type. Additionally, the “hooks” folder uses the “core” folder to process commands, send prompts to the Gemini API, call tools, and handle sessions. The “prompts” folder provides information included in prompts that gives context to the AI model that is processing the prompt. The UI Terminal component’s “services” folder uses this information to help it understand various MCP Prompts and turn them into available commands. The “utils” folder is used in estimating token counts for actions, detecting programming language used, diffing files, among other tasks. These are used throughout all the subcomponents of the UI.

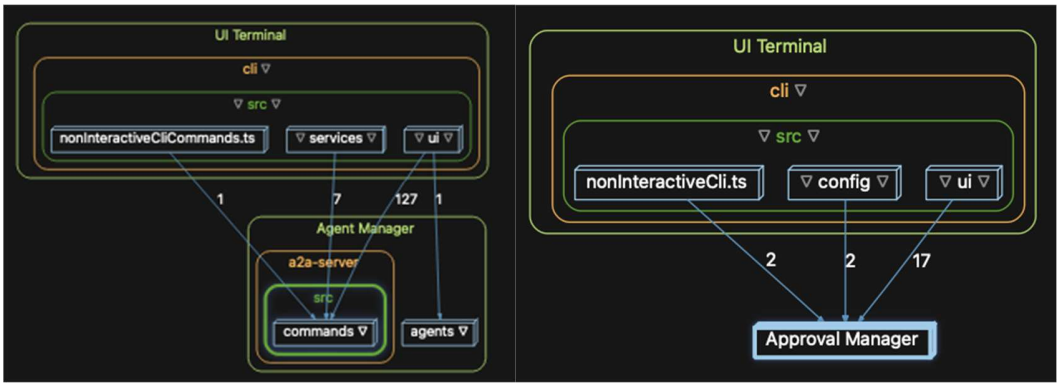

Figure 5. Dependencies between UI Terminal, Agent Manager and Approval Manager

The Agent Manager component’s Command and CommandContext interfaces are used by the UI’s “services” and “ui” folders. They are defined in “commands” folder.

The Approval Manager component is responsible for managing the lanes of communication used when asking the user for approval to use a tool. It will also send messages to the Core Processor component to tell it to execute tools that the user has approved. The “hooks” folder inside the UI

Terminal component will receive tool approval messages from the Approval Manager component.

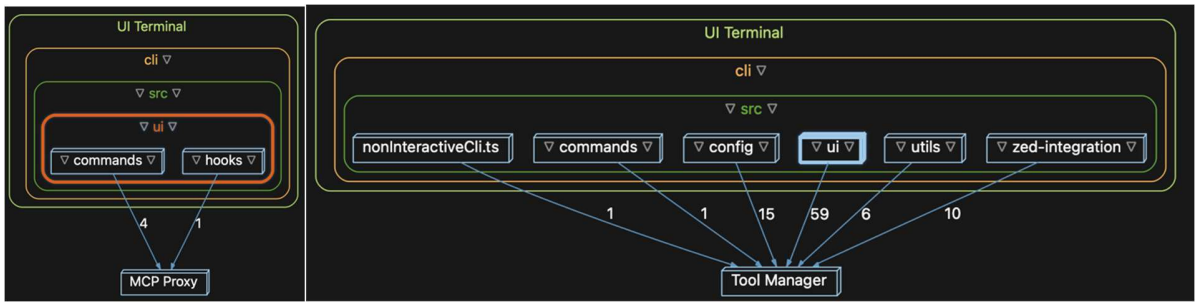

Figure 6. Dependencies between UI Terminal, MCP Proxy and Tool Manager

The MCP Proxy component handles the tokens, authentication and resources of MCP servers. The UI Terminal component's "commands" folder uses this component when listing available MCP servers or when connecting to a new MCP server. The UI Terminal component's "hooks" folder also uses this component when handling @ commands.

Lastly, the Tool Manager component stores and executes the tools that are used in Gemini CLI. has access to. The component will also manage other processes treated as tools, such as Agent to Agent and MCP servers. Everything is stored in one tools folder, which we saw no need to divide into subsystems at this level. The UI Terminal component uses the Tool Manager component to get info from the GEMINI.md project file, to get info from MCP servers, to list available MCP servers, to look at tool request statuses, and to call specific tools.

The dependency graph we derived using Understand has a lot of bidirectional dependencies. We will be analyzing the 3 biggest examples of this to see why these bidirectional dependencies appeared. These can be understood looking at each subsystem as one entity, and so these connections will not be separated by folder.

The Approval Manager and Core Processor components have the first of these bidirectional dependencies. The Approval Manager component's dependency is present because it uses the Core Processor component's "utils" folder to convert json files into strings and to write to files. The Core Processor component's dependency is needed because it tells the Approval Manager component to ask for approval for the use of a given tool. The bidirectional dependency in this case is created by the use of the "utils" scripts. Clearly these utility functions wouldn't make sense to define twice and have basic defined purposes, so this doesn't pose a problem.

The Config Manager and Core Processor components also have a bidirectional dependency. The Config Manager component's dependency is present because the component needs to get a lot of information contained in the Core Processor component throughout its runtime. This includes the directory that Gemini CLI is running in, the Core Processor component's PromptRegistry class and createContentGenerator function, among other things. The Core Manager's dependency is present because values from the Config Manager component's classes are used throughout the core. The information the Config Manager component needs from the Core Processor component should be passed to it when it is called. This implementation was clearly done simply because it was easier and simpler to write this way.

Lastly, the Config Manager and Tool Manager component's bidirectional dependency is more straightforward. The Config Manager component depends on the Tool Manager component since it handles registering the tools, and the Tool Manager component depends on Config Manager component's information on directories and clients. Neither of these dependencies existed in our conceptual architecture, but they exist due to specific locations being chosen to store general information, which contradicts the strict layered structure present in our architecture.

3.1 – Reflexion Analysis #1

There were many discrepancies between our understanding of the conceptual architecture and the concrete architecture we analyzed through Understand. These changes included bidirectional dependencies and other new dependencies that didn't align with how we believed Gemini CLI to function. Any one of these relationships could have its own thorough analysis, though for brevity's sake we will stick to one key example we found the most interesting during our research. This inconsistency was the Agent Manager's dependency on the Tool Manager, a component which it should not have been able to access directly in our conceptual diagram.

Both the "agents" and "a2a-server" folders we had delegated to the Agent Manager contained many files referencing the "tools" folder within the Tool Manager. The quantity of these dependencies (63 in agents and 11 in a2a-server for a total of 74) immediately alerted us of a major divergence in the box-and-line layout of Gemini CLI's concrete architecture. Taking a deeper look at the files within the "agents" folder, we found many files heavily reliant or based entirely on the usage of tools within agents and subagents. For example, "delegate-to-agent-tool.ts", which alone has 14 references to the Tool Manager.

We turned to the Gemini CLI GitHub in search of answers to when and why this design decision was made, though it was impossible to track down one definitive commit, merge or answer due to the sheer volume of information archived within the repository. A common thread while looking through the GitHub's issues section was that having agents call upon tools was an incredibly well-known and commonly accepted feature of Gemini CLI's architecture, confirming our suspicion of this divergence being an intentional design decision.

A dive into the code blame for various files within the "agents" folder revealed a single commit from six months ago acting as the backbone of the modern agent infrastructure. This commit, titled "refactor(agents): Introduce Declarative Agent Framework #9778" by abhipatel12 details a significant refactoring of Gemini CLI's agent-related code, noting its most significant improvement being the ability to wrap and expose agents as tools to promote their reusability (abhipatel12, 2025).

While it is unclear if this refactoring is the origin of Agent Manager's referencing of Tool Manager, this commit at the very least marks a clear and intentional structuring of Gemini CLI's Agent Manager into a form that deeply relies on the ability to create and call specialized agents as if they were any other tool. A feature which justifies the linkage of these two components which once seemed separate conceptually into a clearly thought-out and well-designed dependency during development.

4.0 – Inner Subsystem

The UI Terminal component is a core part of Gemini CLI's operations, as it is the only component the user directly interacts with. Specifically, the UI Terminal displays all of Gemini CLI's visual elements, manages user authentication with the CLI's settings, handles the user's input to the LLM and displaying the response of the LLM to the user.

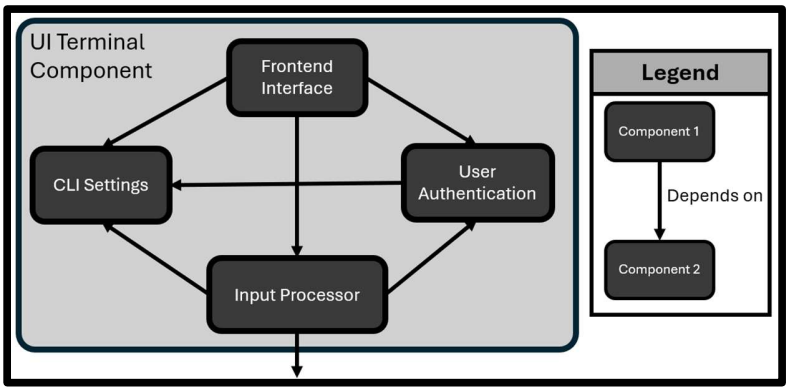

Figure 7. Proposed Conceptual Architecture of the UI Terminal component

In our conceptual architecture for the UI Terminal component, we found that there are four subcomponents. The Frontend Interface subcomponent displays the visual elements of the CLI using a combination of Ink and React. This subcomponent depends on information from all of the remaining subcomponents to display the correct information and visuals. The CLI Settings subcomponent applies the user's settings for Gemini CLI and handles any updates to those settings. The User Authentication subcomponent handles verifying who the user is and depends on the CLI Settings subcomponent to determine the details of the authentication, such as if the user's password will be saved for automatic logins. Lastly, the Input Processor subcomponent manages the received input and processing it to the "Core" layer. Input Processors depends on the CLI Settings subcomponent to determine if any user settings affect how the input will be processed. It also depends on the User Authentication subcomponent to ensure the user can access Google's APIs to call the Gemini LLM.

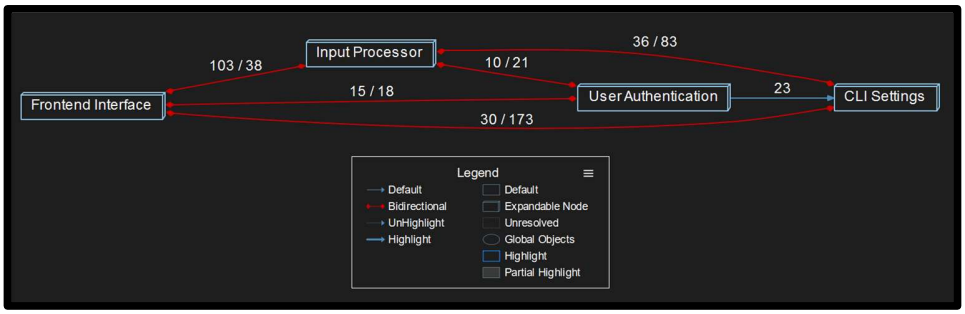

Figure 8. Concrete Architecture of the UI Terminal component found using *Understand*

Compared to our conceptual architecture, the concrete architecture reveals additional dependencies that change how the subcomponents interact with one another. In the conceptual architecture, the Frontend Interface subcomponent was reliant on all of the subcomponents without any of them relying on itself. Whereas the concrete architecture shows that the other

subcomponents do rely on Frontend Interface for additional functionalities and information. Additionally, the dependency between CLI Settings and Input Processor and the dependency between User Authentication and Input Processor have become bidirectional in the concrete architecture where they were once unidirectional in the conceptual architecture. The details of these findings will be explored in Section 4.1 of this report.

4.1 – Reflexion Analysis #2

Our chosen subsystem was the UI Terminal, which resides within the top layer of Gemini CLI's layered architecture. It deals with CLI settings, user authentication, input processing, and the frontend interface. Our conceptual architecture split the system into these four responsibilities with a tiered dependence system, where each subcomponent would rely on all others above it. These components are listed in descending order, meaning that authentication only relied on settings for example, while frontend relied on all three. Reading through documentation, we figured that these components would be able to keep a streamlined flow, since there seemed to be no reason that the authentication would be affected by frontend logic, for example.

In our investigation, we found that larger codebases with so many scripts and functions would tend to reference one another far more often than we thought. Except for the CLI Settings never needing to access the authentication process, every connection between components ended up two-way, as shown in Figure 10. These subcomponents all connected with each other in ways that were not covered in overview-style documentation, nor in our original considerations. Setup of the environment will change depending on the input of the user, moving things around and installing new features (extensionManager.ts in Settings uses envVarResolver.ts from Input). Settings like visual themes could be affected by logic internal to the frontend component (settings.ts imports default-dark/light.ts). It is interesting that CLI settings does not draw on authentication, though it makes sense since the interface for signing should be the same each time since no particular user has signed in yet, meaning the settings would have nothing to look at there.

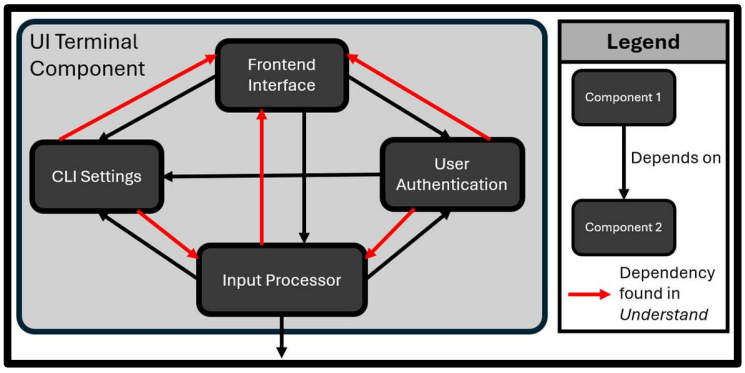

Figure 9. Box-and-line diagram of concrete architecture.

In a system with dozens and dozens of scripts, there's likely to be a lot of inter-communication between subcomponents. While it would be convenient to be able to separate these so cleanly, a great deal of information must be shared around at each step, connecting the subcomponents in several unforeseen ways. The degree of interactive complexity between scripts is likely far too

great to accurately display in overview-style documentation. All in all, we've learned that works of this scale are certain to be far more interconnected than initially expected.

5.1. – Use Case #1

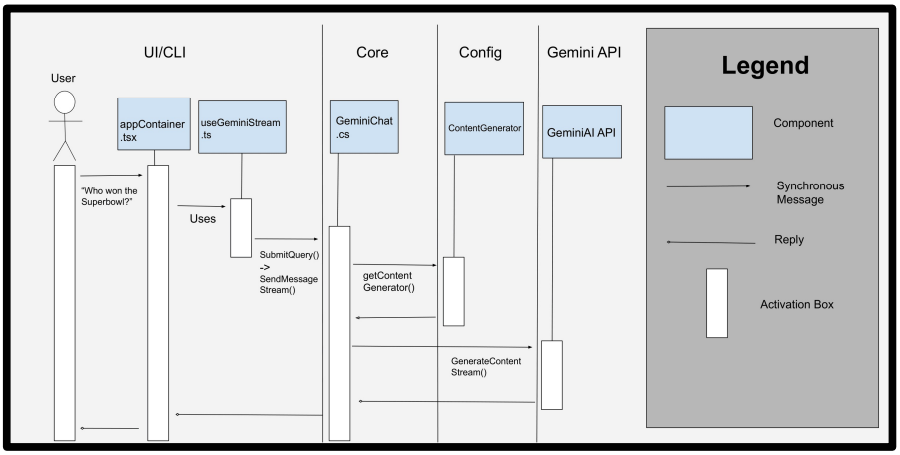

Figure 10. Sequence Diagram of Gemini CLI receiving an input question from the user to be answered by Gemini AI and responding. An example prompt being "Who won the Superbowl?"

The user's question will be collected through the terminal into the CLI layer's UI/CLI/src/UI/appContainer.tsx. This uses UI/CLI/src/UI/Hooks/useGeminiStream.ts (also in CLI) to call the core layer. Specifically, its function SubmitQuery() calls SendMessageStream() from the Core Manager's GeminiChat.cs. SendMessageStream() then gets an instance of a ContentGenerator object from the Config Managerlayer's getContentGenerator() and feeds it the message stream, then calls the ContentGenerator's internal method GenerateContentStream(), which feeds the message to the GeminiAI API as a query as well as records the response. UI/CLI/src/UI/appContainer.tsx will now retrieve the information of who won the Superbowl as answered by the AI agent and can display this to the user.

5.2. – Use Case #2

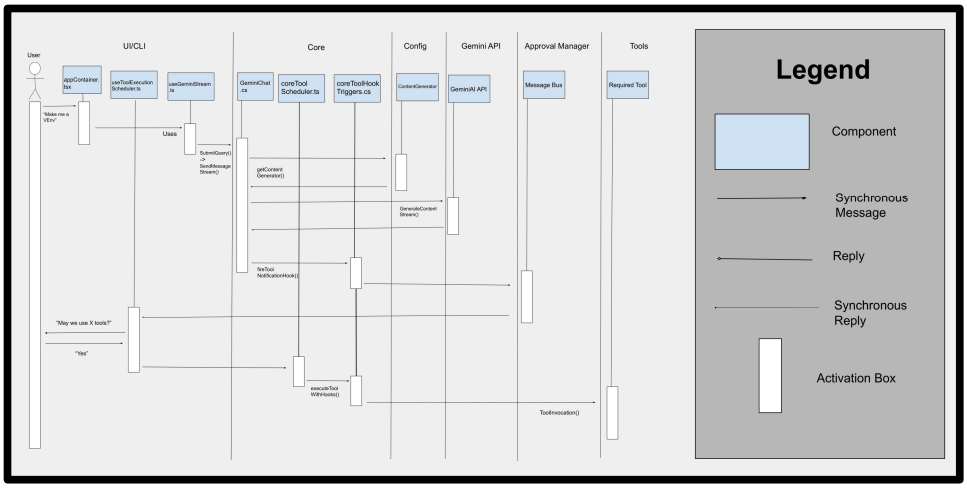

Figure 11. Sequence Diagram of Gemini CLI receiving an input command prompt to execute some function on the user's device. It responds by asking permission and acting accordingly given an affirmative response. An example of this being the user asking Gemini CLI to set up a virtual environment for a new project.

The process of Use Case 1's scenario is followed until the SendMessageStream() function gets a response, where it is told that GeminiAI has asked to execute a function using a tool that requires user permission. This causes it to call the fireToolNotificationHook() function in core/src/core/coreToolHookTriggers.cs, which sends a message requesting the specific tool to a MessageBus shared by it and cli/scr/ui/hooks/useToolExecutionScheduler in the ApprovalManager layer, which relays the ask to the user. On an affirmative answer, a script in the Core layer named coreToolScheduler.ts calls a function named executeToolWithHooks() from core/src/core/coreToolHookTriggers.cs. This then executes the ToolInvocation() function within the Tools layer to complete the desired outcome.

6.0 – Derivation Process

The derivation process for determining Gemini CLI's concrete architecture started by us looking back at our conceptual architecture. Based on global feedback version of the conceptual architecture provided on OnQ, we met as a group to restructure our conceptual architecture to be a bit more high-level. Additionally, we reviewed the Gemini CLI documentation to confirm that our revised conceptual architecture still had accurate dependencies and components. But once these modifications were finalized, we constructed a new box-and-line diagram to properly display the revised architecture.

With the conceptual architecture updated, we began to work on the concrete architecture. We decided to split the group into two subgroups to tackle the concrete architecture, where one group worked on assigning the code files for the components in the "CLI" and "Core" layers of the conceptual architecture while the other group worked on assigning the code files for the "Tools" layer components of the conceptual architecture. Once each subgroup completed their sections of the concrete architecture, we reconvened to combine our two portions of the architecture into one. This ensured that there were no inconsistencies between our two portions (such as a file being in two components at once). We assigned the code files using the Understand software, which allowed us to easily identify divergences between our conceptual and concrete architectures.

Our methodology for assigning which code files were placed in which components was based on analyzing the individual code file's purpose in the architecture and placing it accordingly. This analysis was done by reviewing the file's comments for information on its operations along with identifying what files it imports and where else it is being used in the software. Additionally, if the file itself didn't provide enough information, we would consult the documentation to get a better understanding of how some files functioned and infer the locations for the file based on their file position in the software and their naming convention. Once all the code files were placed in their respective components, we reviewed their placements and repositioned them if needed.

This repositioning is synonymous with the alternative concrete architectures we considered. One of the more prominent examples of this is within the “UI Terminal”. Initially, we placed many of the files relating to the visual themes of Gemini CLI in the “Frontend Interface” subcomponent. However, after a group discussion and getting a better understanding of what each file did, we decided that it made more sense for these files to be placed in the “CLI Settings” subcomponent. As users are able to change the visual theme through the settings, which aligns more with CLI Settings rather than Frontend Interface. After the repositioning of the files was completed, we used the finalized concrete architecture to perform reflexion analysis and identify the divergences from the conceptual architecture.

7.0. – Conclusions

Our understanding of Gemini CLI’s conceptual architecture has been greatly improved, with this updated knowledge reflected in the updated diagrams and throughout our analysis of Gemini CLI’s concrete architecture. These new revelations allowed for a file-by-file level knowledge of intersystem function calls in various use cases, as well as a more comprehensive understanding of software architecture through the lens of a product in continued development.

At a high level, we made good use of Understand and GitHub documentation to discover flaws in our previous understandings and to correct our previous misinterpretations. This took the form of merging and separating various architectural system components, as well as researching and reflecting on the many bidirectional and seemingly incorrect implementations of dependencies we discovered. Many of these discoveries came from our process of mapping the concrete architecture’s box-and-line diagrams in Understand.

As for the inner subsystem analysis, we found many of the same inconsistencies in our mapping of the UI Terminal’s subcomponents. We discovered how and why bidirectional dependencies are an important piece of Gemini CLI’s subsystems, despite our hesitancy to accept them as necessary for the functioning of concrete architectures.

The future of this project will be fueled by our new hands-on experience with the codebase of Gemini CLI. In the next and final report, we plan to continue refining our research methods and solidifying our understanding of Gemini CLI’s architecture and general best practices in software architecture. Through this, we can devise an architectural enhancement that meshes with the intentions of Gemini CLI as a user-first productivity tool.

8.0. – Lessons Learned

Throughout this assignment, we had to determine a concrete architecture for Gemini CLI while using the Understand tool. This required us to learn how Understand works, how to form an architecture using it, and how to properly interpret the information we came to. One key distinction we came to learn about while using Understand was the difference between an architecture and an implementation. When using Understand to create a higher-level architecture diagram, we ended up having several bi-directional dependencies that were very unevenly weighted. These dependencies would have dependency spreads of 90/3 as an example. We came to realise that this was because Understand would build an architecture for us using the exact

implementation of the system instead of based on its architecture. Since we only wanted a high-level abstraction of the architecture rather than its exact implementation, we learned to treat some of these dependencies as single-directional dependencies if the difference of weights between dependencies was large enough. This allowed us to determine a simpler concrete architecture using Understand for higher level systems and subsystems.

Another lesson we learned while working on this project was the significance of dividing workloads effectively throughout a project. Learning from mistakes we had made in our last report, we started work on this report much earlier than our last report. In one of our earlier meetings, we decided to split into two different groups to work on different major sections of the report. One group was tasked with deducing and creating the high-level concrete architecture of the system, and the other was tasked with choosing a subsystem, creating a conceptual architecture for it, and creating the concrete architecture of the subsystem alone. Dividing up the workload in this way helped to spread responsibilities of tasks between group members.

9.1 – Data Dictionary

- **Gemini AI | Gemini LLM** – Refers to one of Google's LLM models.
- **"core"** – package def
- **"cli"** – package def
- **"tools"** – package def

9.2 – Naming Conventions

- **CLI**: Command Line Interface
- **LLM**: Large Language Model
- **AI**: Artificial Intelligence
- **API**: Application Programming Interface
- **MCP**: Model Context Protocol
- **UI**: User Interface

10.0 – 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.

Before deciding on an LLM model, we came together as a group to reflect on where the LLM could have been used within the previous assignment. We decided that we still want to complete the more intensive tasks of the assignment (such as assigning the files to each component for the concrete architectures) without the LLM's involvement to better learn the concepts and prevent potential hallucinations from the LLM. However, we also realized that the LLM could boost our group's efficiency and accuracy by completing the following two tasks:

- **Presentation Generation** – Given a tailored-made version of our report, the LLM will generate a rough draft of our slideshow that contains all the required sections and

information for the presentation along with speaker notes for the video recording. This is an ideal task for the LLM as it can not only perform this task faster than an individual member of our group can, but it is also good at summarizing content, such as summarizing the report's content into speaker notes.

- **Report Evaluator** – Given our report and the assignment rubric, the LLM will evaluate the report. Specifically, it ensures that all the required sections are present in the report and then grade each section's quality based on its respective description in the rubric. Which is an ideal task for the LLM as they are good at logical comparisons and advanced reasoning, allowing it to make accurate comparisons and provide effective feedback.

With these two tasks defined, we needed an LLM that had a significant amount of power to generate the slideshow presentation (ideally a PowerPoint presentation as that's what our group is most familiar with). The LLM also needed to have high advanced reasoning and logical skills for evaluating the report. As such, we decided to use one of the most powerful LLMs currently available to satisfy these requirements, that being OpenAI's GPT-5.3. As GPT-5.3 can perform advanced reasoning tasks accurately, and is able to generate downloadable PowerPoint files, which is ideal for our presentation.

After some discussion between the whole group, we decided to repeat our approach from A1 and assign one member as the "prompt engineer". We decided to keep this approach as it simplified the process of working with the LLM and allowed other members to dedicate time to other tasks during A1, so we believe that it will provide similar benefits with A2. Like with A1, the prompt engineer's job is to create the prompts for the LLM, fine-tune the prompts if needed, and return the LLM's outputs to the rest of the group.

Each prompt designed by the prompt engineer assigned a persona to the LLM, such as assigning it the role of "an experienced presenter well-versed in creating slideshows". Due to these personas, a new version of the LLM model was used for each task the LLM needed to complete. Each prompt provided context to the LLM by uploading a tailor-made report and the assignment rubric to the LLM so that it can accurately perform its tasks. Lastly, outputs for each prompt of the LLM were specified in a certain format, such as one of the prompts specifying that the LLM returns "a PowerPoint file (i.e., a pptx file).". These prompts required minimal refinements to improve the LLM's quality (these refinements mainly related to how the LLM formatted the slideshow by explicitly stating what should be put on the first two slides). The finalized version of the prompt used for generating the rough presentation document is displayed below:

> *You are an experienced presenter well-versed in creating slideshows. Your task is to create a presentation that contains a slide for each of the sections in the provided Word document. Each slide should contain either a few bullet-points that are each one sentence long describing the section's contents, an image that is taken from the respective section in the Word document, or a combination of both. The first slide of the presentation should be a title slide with the name "CONCRETE ARCHITECTURE OF GEMINI CLI" as the title and "Group #8 - The Teamm8tes" as the subtitle. The second slide of the presentation should be an agenda outlining all the sections in the presentation. As for the design of the presentation, please use one of PowerPoint's default themes, and customize it relate it to technology and computer science. Lastly, this presentation file should be outputted as a PowerPoint file (i.e., a (.pptx) file).*

Due to the outputs' relation to the assignment's requirements, we validated the LLM's outputs by cross-referencing the results with the assignment rubrics to ensure that what the LLM produced was accurate to the assignment. Additionally, for the LLM's slideshow and its generated text, we had the prompt engineer proofread the AI-generated text to ensure consistency and accuracy with the report's content. If the text had any issues, it would be rewritten to be more accurate and consistent with the report. If no issues were found, then we believed that the AI-generated text was accurate. However, as discussed below, we ended up not validating the slideshow's generated text as we did not use the LLM's generated slideshow.

Overall, we found that the LLM's contributions didn't provide too much value to the final deliverable. As the LLM failed to make an effective PowerPoint presentation after several attempts at prompting it, causing us to have to make the PowerPoint ourselves. Additionally, while the LLM provided a good and fair evaluation of our report, many of the key improvements it suggested were aspects it couldn't identify, such as the sequence diagrams. Thus, reducing the impact that the LLM's evaluation had on the report. As such, we will grade the LLM's overall contribution as a 5%, as it did not contribute effectively to the final deliverables.

However, the lack of contributions from the LLM is partially a result of us not integrating the LLM into our group's workflow and tasks. As we only had the LLM work on tasks that were relevant to the end of the assignment's completion. Although, this doesn't mean that the LLM saved a lot of time. Due to the LLM not creating sufficient presentations despite several different prompts, it ended up creating more work for us rather than boosting efficiency. Whereas the LLM's task of evaluating our report didn't create any new work outside of creating and altering the initial prompt. Additionally, due to how late in the project's timeline the LLM was used, the LLM did not influence our team's decision-making processes. Especially as, regarding the LLM's rubric-based evaluation, the issues were presented were aspects it could not identify like the sequence diagrams. Luckily, this reduced influence on the team meant that the LLM didn't cause any challenges within the group besides its incapability to create a sufficient PowerPoint presentation. Based on all of this, we've learned that to get the most out of the LLM, we need to integrate it more into the main components of the assignment rather than saving its usage until the end of the project. As such, going into the final assignment we plan to use the LLM to complete tasks directly related to Gemini CLI's architecture, rather than being hesitant to allow the LLM to work on such tasks, to allow it to have a stronger impact on our group's work

11.0 – References

abhipatel12. (2025, September 25). *Refactor(agents): Introduce declarative agent framework #9778*. GitHub. https://github.com/google-gemini/gemini-cli/pull/9778

Adams, Bram. (2026, February). *Global Feedback about A1*. OnQ. https://onq.queensu.ca/d2l/le/content/1132545/viewContent/6711621/View