A2: Concrete Architecture of Void

CISC 322

November 7th, 2025

Authors:

Matt Dobaj, Shaun Thomas, Babinson Batala,

Esmé Mirchandani, Sandy Mourad, Nihal Kodukula

0 - Table of Contents

1 - Abstract

This report explores the updated architecture of Void, an extension of Visual Studio Code, which includes AI-powered development features. By studying the methods and code with Understand, we found that Void's structure blends Layered, Client-Server, and Publish-Subscribe styles to keep the system organized, flexible, and easy to expand. The former Model Service has been replaced by a Common component that connects the user interface and platform layers, while new Base and Extensions components make it easier to add features and maintain the system. Communication between parts happens through lightweight, event-based channels, allowing smooth and responsive interactions. Overall, Void's architecture shows a clear focus on being modular, scalable, and adaptable.

2 - Introduction and Overview

Since VS Code, and Void by extension, are built on the Electron framework, their code is divided into two main processes, the Main Process and the Browser Process. The browser acts as the client to the main process's server. Both processes depend on a shared Base component that provides utilities accessible to either side. The browser process handles all user interface elements, while the main process manages the application's lifecycle and interactions with the host operating system [1].

Artificial intelligence is changing how developers write and interact with code, offering features like code completion, natural language prompting, and automated refactoring. Void, a fork of Visual Studio Code (VS Code), represents this shift by extending VS Code with AI-powered tools while maintaining its modular and efficient structure. Built on the same Electron framework, Void uses the Browser Process for user interactions and the Main Process for system logic and communication. Both rely on the shared Base component for coordination and resource management.

Our previous analysis outlined Void's conceptual architecture, identifying components such as the Void UI, VS Code Editor, Model Service, and Platform. This report expands on that work using the Understand tool to extract and analyze Void's concrete architecture. We found that the former Model Service was inside an even bigger component, which is the Common component, linking the UI and Platform layers. The new Common and Extensions components improve modularity and strengthen integration with VS Code's plugin ecosystem.

Overall, Void's structure blends Layered, Client-Server, and Publish-Subscribe architectural styles to achieve scalability, responsiveness, and maintainability. Its design shows that AI functionality is not simply added onto the system but is integrated into its core, creating a flexible and sustainable foundation for continued expansion.

3 - Derivation Process

Our derivation process for Void's concrete architecture involved looking at each folder that we could associate with each component in our conceptual architecture and inspecting its dependencies using Understand's butterfly and dependency visualizations. We also used the dependency inspector at the bottom of the screen to investigate individual file dependencies. When considering refactoring our architecture, we looked at subfolders for our components and tried seeing if separating them into two components would simplify the architecture, which did not often occur. For inspecting the Use Case's we followed the code along different function calls and IPC events to ensure we saw every detail in Void's process.

4 - Architecture

In our previous report, we split the Void specific components into 3 distinct layers, matching the base VS Code architecture. First, a Void UI component was responsible for hosting the LLM chat sidebar, pages, and potentially inline code completion. Second is a Model Service component, which is responsible for formatting an LLM message, gathering context from the user's repository, and interfacing with the Void platform component, which interacts with the LLM providers. We also believed the VS Code Editor was modified to depend on the Model Service component. Below is the diagram from our previous report:

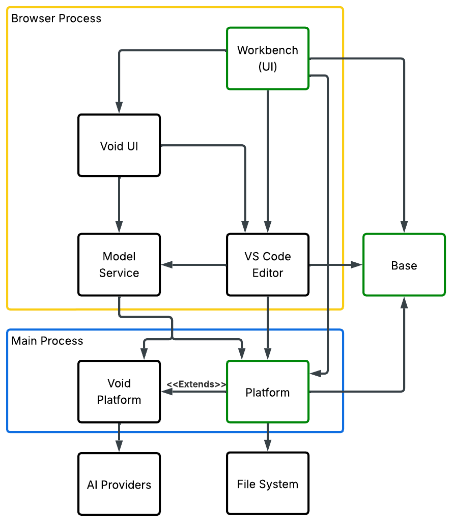


Figure 1: A diagram showing Void's conceptual architecture. Each arrow represents a dependency. Green boxes indicate original VS Code components.

After exploring the code components using Understand, we discovered that there were many more dependencies between components, with many bidirectional dependencies. One major change is that what was previously called the Model Service has been renamed to the Common component to more accurately reflect its broader functionality. An interesting discovery was that every component in the top-level architecture is dependent on the base component from VS Code. Each component was also dependent on the extensions component, which we added after seeing its folder repeatedly during our exploration. Below is our derived concrete architecture from using Understand's code analysis:

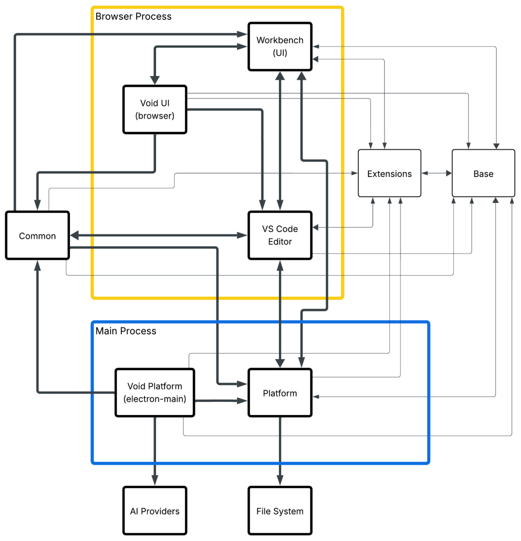


Figure 2: A diagram summarizing Void's concrete architecture. Each arrow represents a dependency, with double sided arrows showing components that are dependent on each other.

4.1 VS Code's Concrete Components

4.1.1 Extension

We decided to add a new Extension component after seeing that every other component is dependent on the extensions folder in the code base. Each component has an API for extensions to integrate with so that users can use different services. It itself is dependent on VSCode's Base, Workbench, and Editor components. These dependencies are likely so that extensions can borrow UI elements or interface using IPC via the base component. It is not a critical component as it is not necessary for Void to function, but it is useful as it houses all end-user modifications to any other component.

4.1.2 Base

The VS Code Base component seems to have more responsibility than previously assumed. After looking at its code, we determined that there are many UI elements included among its many utilities. Every component in the architecture is dependent on it, most likely because of these many tools, including IPC, and UI elements. Its code can also be found across the workbench and platform folders in their common folders.

4.1.3 Workbench

VS Code's Workbench component is similar to our previous report, with it containing most (but not all) of VS Code's UI elements. Many subcomponents, UI based or otherwise, had dedicated folders within a contrib folder in the workbench folder, improving extendability as the code is compartmentalized. This does blur the line between workbench code, base code, and platform code as there was platform and base code stored alongside the UI code in these contrib folders. In terms of dependencies, all of its dependencies changed to be bidirectional, likely as most components use UI components located in the workbench. There is also a new dependency from Void's Common component to the workbench.

4.1.4 Platform

VS Code's Platform also closely reflects the documentation we explored in the last report. It is responsible for handling interactions with the file system and running commands.. It is dependent on the VS Code Editor and Workbench components. After some code inspection we discovered these were for opening the UI and Code Editor windows.

4.1.5 Editor

VS Code's Editor is slightly different from our conceptual architecture. While it is dependent on Void's Common component, it is only for graphical purposes like drawing icons. Most of the interaction between Void and the Editor happens via InlineCompletion events which are already present in VS Code's stock form. It is also now dependent on the Workbench component for various graphical and configuration purposes. This means that this component is largely unmodified, which we didn't expect in our previous report.

4.2 Void's Concrete Components

4.2.1 Void UI

Void's UI component is largely the same as our understanding of it from our conceptual architecture. It contains all of Void's UI elements, and different pages for settings and configuration. While the logic behind the Inline Code completion largely lies in this component, it does not interface with a modified version of the Code editor but instead via APIs made accessible by VS Code. It is also dependent on the VS Code workbench as it builds off of UI components in the existing VS Code platform.

4.2.2 Common (Previously Model Service)

What we believed to be a component dedicated to LLM message processing and context generation was instead more of a shared space for both the UI and Platform components. It serves as an equivalent to While LLM message processing is still its primary focus, there are also subcomponents for storing Void's settings, various object types, and defining a service for allowing AI actions via the Model Context Protocol (MCP) [2]. One notable change is that it does not depend on the Void Platform directly, instead sending LLM Messages through the VS Code platform via IPC. These messages are relayed by the VS Code platform to Void's platform. It is also now dependent on the base component. We initially thought that it was dependent on the workbench component as it has file dependencies within the workbench folder. We later realized that these dependencies are only for the *common* folders inside, which would align more with the base component. Because code within this component can be run either on the browser or main process, we have moved it outside the bounds of either in our diagram. This can impact extendability as functions that might be expected in either the UI or Platform component are instead found in the Common component.

4.2.3 Void Platform

We believed that the Void Platform extended the VS Code Platform. This is not the case after inspecting the code base. We realized that the Void Platform simply hooks into the existing VS Code Platform by registering IPC and MCP channels. It is primarily responsible for interfacing with LLMs and handling lesser known MCP commands, where primary commands like add and edit are handled in the common component.

4.3 Void Platform Subcomponents

We have decided to investigate Void's Platform component as it depends on two key components, the Common and VS Code Platform components. We derived the sub components by looking at the files within this component and looking at their dependencies. Below is the derived architecture:

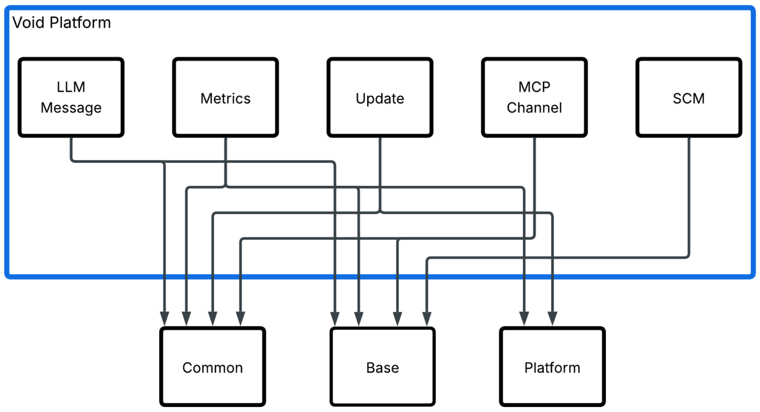


Figure 3: A diagram showing the subcomponents of the Void Platform component. An arrow represents a dependency.

4.3.1 LLM Message

This subcomponent is responsible for getting an LLM Message from the Common component via IPC, prompting the selected LLM, and returning the result after some processing. This processing can include extracting code and MCP commands from the LLM response. It is dependent on the Base component for IPC and the Common component for data types.

4.3.2 MCP Channel

This subcomponent is responsible for handling miscellaneous Model Context Protocol commands, which need to run on Electron's main process. The majority of the relevant protocols are handled in Void's UI component, which interfaces with the VS Code platform to perform its tasks. It seems that the remaining protocols are passed to this component.

4.3.3 Miscellaneous Subcomponents

The Metrics subcomponent is responsible for gathering statistics from other Void components via IPC calls. It relays this information online to Void's metric gathering service. The Update subcomponent acts as an abstraction for both the automatic and manual checks for Void Updates, which are based on the VS Code update service. Unfortunately, we were unable to find where the update service changed to check for Void's binaries over VS Code's. The Source Code Management (SCM) subcomponent is responsible for gathering information about the user's active git repository in order to automatically generate commit messages using the LLM. It is dependent on just the base component, likely to interface using IPC.

4.3.4 Architectural Styles

We believe that this component makes use of the Publish-Subscribe architectural style. Each component receives its commands via IPC channels, executes their tasks, and returns it to the function provided in the incoming IPC message. There are no code dependencies between them or to them from the rest of the project. This is inline with the Pub-Sub architecture style as the components are unaware of the identities of the components that send the events.

4.4 Performance Requirements

A primary concern we mentioned when deriving Void's conceptual architecture was response time, as tasks such as autocomplete require relevant suggestions to show and update with code edits relatively quickly. These requirements are met through various means.

Firstly, Void makes use of caching for autocomplete results. If a user is partially typing content that is already present in a previous autocomplete suggestion, the LLM service won't be prompted again and the remainder of the previous suggestion will be shown. This also solves the issue of the code completion having a mismatch with the user's input: if the user is typing along then the same suggestion will be shown, otherwise a new one will be made. The individual code prompt feature does not have these low response time requirements, so it does not have caching. While a low response time would be valued in this use case, it is not as critical.

4.5 Architectural Style

The two primary architectural styles mentioned in our conceptual architectural report continue to be present after analyzing the code base using Understand. We have also decided to add another architectural style after seeing the extensive use of events in the code base.

4.5.1 Layered Architecture

Void's concrete architecture retains its ascending layers of dependencies, but the order of components is switched around. The Common component, which was previously described as the middle layer, instead acts as the innermost layer. While there is no direct dependency between the Void UI and either Platform component, there is a loose connection via IPC. The chain of abstraction is as follows: the Void UI relies on the Void Platform loosely and they both rely on the Common component directly. For the VS Code portion, the Workbench component is largely dependent on the Editor and the Platform, the Editor is dependent on the Platform, and all of the components are dependent on the Base component.

In the conceptual architecture report we discussed the logical separation of components, where the UI is contained in the UI component, the LLM message formatting and processing is entirely contained in the Common component, and the LLM interaction and file commands is contained in the Void Platform component. This remains true for most of Void's code, but the MCP commands and LLM message formatting process is split across all three components. This impacts readability and makes interchanging the implementation of the layers slightly more complicated, taking away from some of the Layered architectural style's advantages.

The Layered style drawbacks of performance requirements due to abstraction remain unfounded as the majority of the latency regarding LLM use is found in the interactions with the LLM itself. There are not enough layers to cause a serious latency or throughput issue when compared to the latency of the LLM.

4.5.2 Client - Server

The client server architecture style can be seen throughout Void's code base, as browser process components like VS Code's Workbench and Editor components or Void's UI component request services that the Platform components on the main process provide. This includes file system access, LLM communication, or executing other commands. In this instance, the browser process components act as the client, with the Platform components acting as the server.

The extensive use of IPC to execute commands also means that Void is highly extendable through the addition of extensions or new IPC channels. The client-server code separation also means that new features do not necessarily require changes across multiple components. Overall, Void's requirements of portability and extensibility align with client-server's advantages well.

In the previous report we mentioned disadvantages to the client-server architecture style regarding performance. Performance could be impacted either via network performance or via abstraction, where data needs to be formatted into a message on client side to then be unpacked on the server side. In our code analysis we did not see strict requirements regarding message formatting, so Void largely avoids this issue.

Regarding network performance, Void's IPC protocol does not seem to use the network. The only time the network would be used to communicate is for remote development purposes, where network performance issues are impossible to avoid. Overall, Void seems to largely avoid the drawbacks of the client-server architecture.

4.5.3 Publish Subscribe / Implicit Invocation

As mentioned in the Void Platform subcomponent analysis, most of the interactions between the UI process and the Platform components are done using IPC. The interaction between the autocomplete service and the VS Code Editor also uses IPC. The IPC protocol used in VS Code, and by extension, Void, is written using listeners, which follow different channels for events and are triggered by new messages to the channel. As per the Pub-Sub requirements, that means the various components are only loosely connected, and there are no file dependencies from Common to the Void Platform, where the general flow of data occurs.

The use of the Pub-Sub architectural style has some associated pros and cons. The primary advantages are reuse and evolution. Because there is a loose connection between components, components can be added or replaced without affecting the interfaces of others. The primary disadvantage is that a message is not guaranteed to have a consistent order of responses or even a response at all. For a project like Void, which aims to be highly extendable through extensions, the advantages definitively outweigh the disadvantages.

4.5.4 Pipe and Filter?

Because a prompt undergoes a series of processing steps before being finally passed to the AI provider, one might assume that Void follows a Pipe and Filter architectural style. While some of the filters also don't share their state or know the identities of each other, not all of them do, meaning that the system fails Pipe and Filter's invariant. The filters are also not able to be rearranged as they do not have standardized inputs and outputs.

5 - External Interfaces

Void continues to exchange information through several key interfaces that connect its user-facing components, AI backends, and extension frameworks. These interfaces define how data flows into and out of the system, ensuring consistent communication between the browser and main processes while maintaining modularity and platform independence.

5.1 Graphical User Interfaces (GUIs)

Information sent to the Platforms:

- User prompts, inline completions messages and the selected model
- File requests, system commands

Information received from the Platforms:

- Finalized AI responses (code completions / changes)
- Contextual updates from the Platform

5.2 AI Provider Interfaces

Information sent to AI providers:

- Structured prompts including user input, message history, code context, and metadata
- Model configuration parameters (provider ID, token limits, temperature, top-p)

Information received from providers

- Model responses code edits, service requests

5.3 File and Platform Interfaces

Information Transmitted

- File and workspace operations (open, save, rename, delete) from the Platform.

6 - Use Cases

We have kept our same Use Cases but have redrawn the sequence diagrams for them to better reflect how Void is implemented. The lines are labelled to represent the file being referenced rather than the direct method called.

6.1 Use Case 1

Use Case 1 investigates how Void handles inline code completion. We explore the User typing in the editor and how Void automatically suggests an inline code completion.

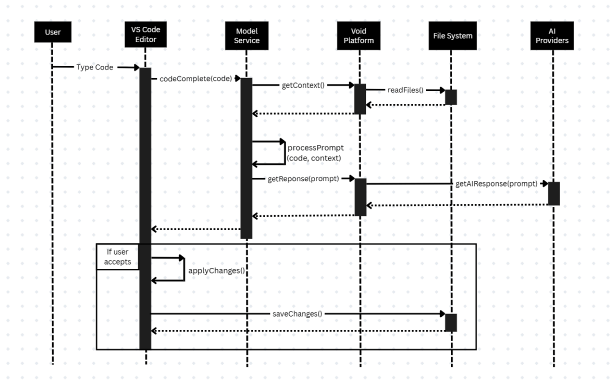


Figure 4: Sequence diagram of Use Case 1 based on Void's conceptual architecture.

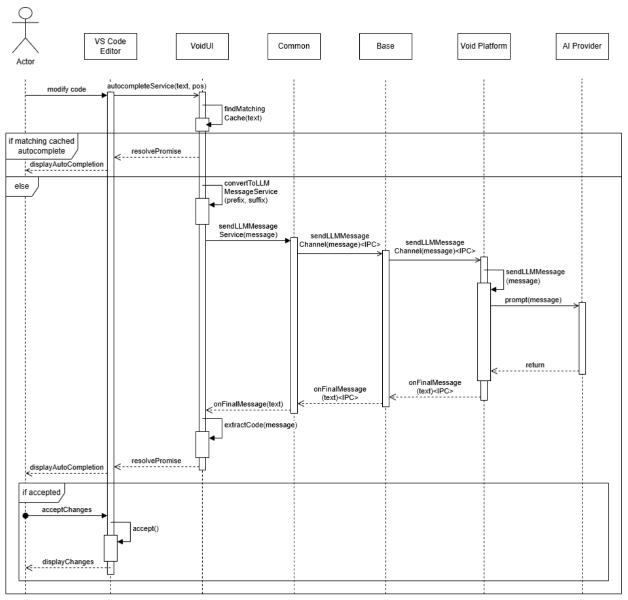


Figure 5: Sequence diagram of Use Case 1 based on Void's concrete architecture.

When the user types in the code editor, VS Code sends an event to all registered inlineSuggestion handlers. In the Void UI component a handler is registered to this event and is triggered by the user's input. Before prompting the LLM, the UI component checks to ensure that a previous autocomplete that matches the text in the user's editor already exists. If so, then this result is displayed to the user.

In our conceptual architecture report we believed that code context based on the user's project files would be automatically discovered and added alongside the user's currently open file. It appears that this functionality was once present in Void, but all references to it have been commented out. This means that no additional context other than the text in the open document is provided to the LLM.

Next, the UI component formats the message using the convertToLLMService.ts file. It then uses the sendLLMMessage service to send the message via IPC through the base component to the Void Platform component, where the LLM is communicated with. A function call passed along with the message is executed when a response is received by sending another IPC call back to the Common component.

After receiving the LLM response, the VoidUI component extracts the code and displays the inline suggestion. If the user accepts this suggestion, instead of writing directly to the file as we previously believed, the editor simply adds the text without writing anything to the open file.

6.2 Use Case 2

Use Case 2 investigates how Void handles requests through its LLM chat window. We explore the User sending a message and how Void responds.

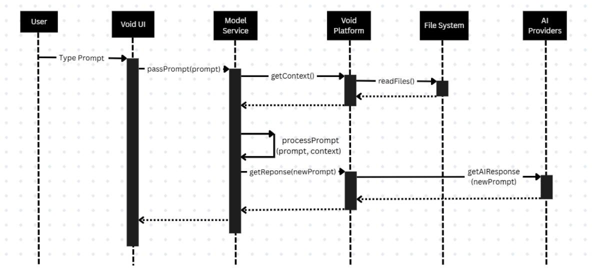


Figure 6: Sequence diagram of Use Case 2 based on Void's conceptual architecture.

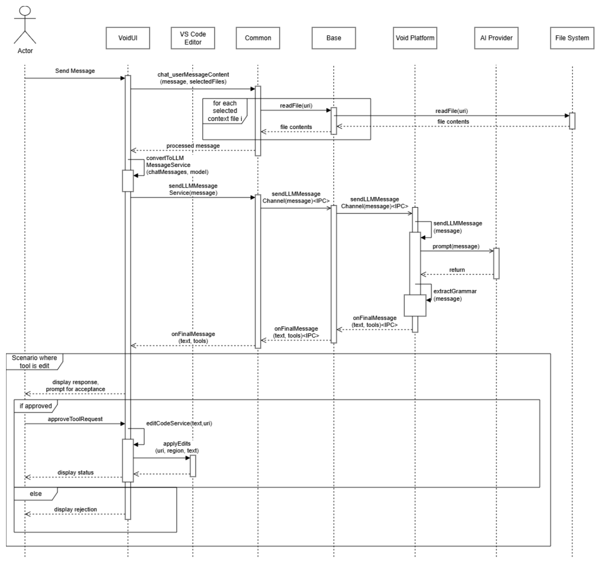

Actor
VoidUI
VS Code Editor
Common
Base
Void Platform
AI Provider
File System
Send Message
chat_userMessageContent (message, selectedFiles)
for each selected context file i
readFile(uri)
readFile(uri)
file contents
file contents
processed message
convertToLLM MessageService (chatMessages, model)
sendLLMMessage Service(message)
sendLLMMessage Channel(message)<IPC>
sendLLMMessage Channel(message)<IPC>
sendLLMMessage (message)
prompt(message)
return
extractGrammar (message)
onFinalMessage (text, tools)<IPC>
onFinalMessage (text, tools)<IPC>
onFinalMessage (text, tools)
Scenario where tool is edit
display response, prompt for acceptance
if approved
approveToolRequest
editCodeService(text,uri)
applyEdits (uri, region, text)
display status
else
display rejection

Figure 7: Sequence diagram of Use Case 2 based on Void's concrete architecture.

When the user sends a message in Void's chat sidebar in the UI component, the chat_userMessageContent function in the Common component is called. This function also does not implement automatic context generation, but it does add the contents of files that the user has explicitly selected to the message for context.

After receiving the processed message with some added context, the UI component will then convert the message into an LLM message before sending it via the sendLLMMessageService.ts file in the Common component. As with the previous use case, this function will send a command to the Void Platform via IPC through the Base component. The Void Platform then communicates with AI as it did before, but before calling the onFinalMessage function via IPC, it calls its own extractGrammar file to extract the code content and the requested tool, or MCP command, from the LLM response.

If the scenario is an edit to a file, the UI with ask the user if they approve of the changes. If true, then the text will be given to the VS Code Editor component, which will tentatively apply the edits to the file. It does not directly write to the file, instead waiting for the user to save the changes, matching its behaviour with the previous use case.

7 - Data Dictionary

AI Provider: Third-party model (e.g., OpenAI, Anthropic) that processes user prompts and returns code completions or code suggestions.

Browser Process: Client-side Electron process that runs Void's user interface and captures user interactions.

Electron Framework: Cross-platform system for building desktop apps using web technologies; used by both VS Code and Void.

Inter-Process Communication (IPC): A protocol used to pass information between multiple processes.

Large Language Model (LLM): Model that generates human-like text or code from natural language input.

Main Process: Backend Electron process controlling app lifecycle, system logic, and external communication.

Model Context Protocol (MCP): A standard for connecting AI applications with external systems.

Source Code Management (SCM): A methodology to organize, track, and log changes to source code over time (i.e. Git).

Prompt: The text or code query written by the user and sent to an AI model for code completion or code suggestions.

8 - Naming Conventions

When talking about function names, we chose to use the same names as those found in Void's code base to allow readers to easily follow along in the code should they wish. Components follow the naming scheme outlined in VS Code's architectural overview [3] and Void's overview [4]. Sometimes, like in the case of Void's Common component, we chose to name it after the file where the code was located.

9 - Conclusion

To better understand Void's concrete architecture, we used Understand to analyze its main subsystems. This revealed several differences from the conceptual design. The Common component, replacing the old Model Service, now serves as a shared layer between the UI and Platform while depending on the Base and Extensions components. The Extensions subsystem also plays a larger role than expected, linking multiple modules to support third-party integrations.

The results show that Void's architecture relies on Electron's IPC-based communication, following an event-driven Publish-Subscribe pattern. The Layered and Client-Server structures remain, with the browser and main processes loosely connected through shared utilities and asynchronous messaging.

Overall, the Understand analysis clarifies how Void's modules interact and depend on one another, confirming that its design supports scalability, responsiveness, and modular AI integration. Future work could explore smaller dependencies and how they evolve with new AI updates.

10 - Lessons Learned

The assignment taught us how different a system looks once you compare the conceptual idea to the real code. One thing we learned quickly was that some assumptions from A1 didn't match the actual implementation. For example, we originally treated the Model Service as a clean, standalone layer, but once we dove into the repository using Understand, we realized the real logic was mixed together inside the larger Common component.

We also learned the value of starting with a broad overview before digging into individual files. At first, we focused too much on low-level details, which made everything feel more complicated. Once we looked at Understand's big-picture diagrams, the structure became clearer, and then the smaller pieces made sense.

Another takeaway was that VS Code's architecture is far more connected than the official documentation suggests. There were more two-way dependencies than we expected, and components lived in places we didn't anticipate, which forced us to rethink some earlier assumptions.

Finally, we realized how much smoother the work became once we divided the analysis instead of overlapping. Early on, multiple people inspected the same folders, but once we assigned sections and compared findings, everything moved faster.

Overall, this project showed us how important it is to validate architectural ideas with real code and how helpful tools like Understand are for navigating large systems.

11 - References

[1] "Process Model." [Online]. Available: https://www.electronjs.org/docs/latest/tutorial/process-model

[2] "What is the Model Context Protocol." [Online]. Available: https://modelcontextprotocol.io/docs/getting-started/intro

[3] "Source Code Organization." [Online]. Available: https://github.com/microsoft/vscode/wiki/Source-Code-Organization

[4] "Void Overview." [Online]. Available: https://zread.ai/voideditor/void/1-overview

12 - AI Report

For this deliverable, our team used ChatGPT (GPT 5, August 2025 version) as a virtual AI teammate to assist with basic code understanding, clarification, and verification during our analysis of Void's concrete architecture.

After completing our initial exploration, we engaged ChatGPT to help interpret TypeScript files and dependencies that were difficult to understand from the raw code alone. ChatGPT was selected based on its consistent accuracy with technical explanations and its ability to simplify large code segments into understandable summaries.

Unlike in A1, where ChatGPT was used just for verification and formatting, in this phase it also supported us in understanding the real implementation of the system. We used it to understand the purpose and functionality of specific files.

For example, when investigating the VoidModelService, SCMService, and LLM message handling files, we used ChatGPT to understand how the code managed model initialization, IPC message passing, and Git operations. This understanding helped us describe the relationships and responsibilities of these subsystems in the report.

We assigned ChatGPT clearly defined support tasks that focused on understanding and validation, but not decision making. These tasks included:

1. Code explanation: Interpreting TypeScript files and summarizing their purpose.
2. Subsystem clarification: Explaining the role and dependencies of components.
3. Architecture connection: Helping link implementation details to the styles observed in the system (Layered, Client-Server, Pub-Sub).
4. Documentation support: Helping rephrase technical content for consistency.

We made sure our team maintained a structured and transparent workflow for all AI interactions. To ensure the accuracy of AI generated content, we followed strict steps:

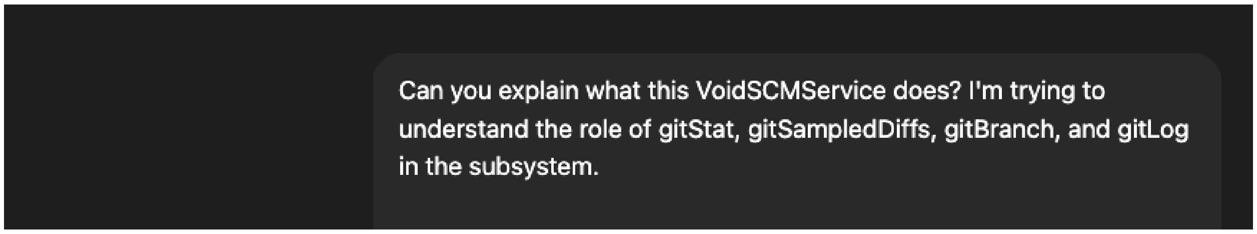
1. Cross checking: Every explanation from ChatGPT was verified using the dependency graphs and the actual codebase.
2. Peer review: At least two team members reviewed AI generated content to ensure correctness and clarity. We reviewed and discussed each AI explanation as a group before including it in the report.
3. Revision and refinement: Any unclear or inaccurate responses were revised, clarified, or discarded.
4. Manual confirmation: When ChatGPT's interpretations were vague or too general, we manually traced the code until the behavior was confirmed.

ChatGPT's contributions mainly improved the efficiency of the analysis process. It helped us save time understanding complex dependencies and allowed us to focus on architectural reasoning instead of getting stuck on low-level code interpretation. However, we made sure that all conclusions, diagrams, and architectural analysis were based on our own review and team discussion. We also noticed that ChatGPT sometimes gave vague answers at first, which required manual confirmation and extra prompting, but this taught us how to ask clearer, more focused questions.

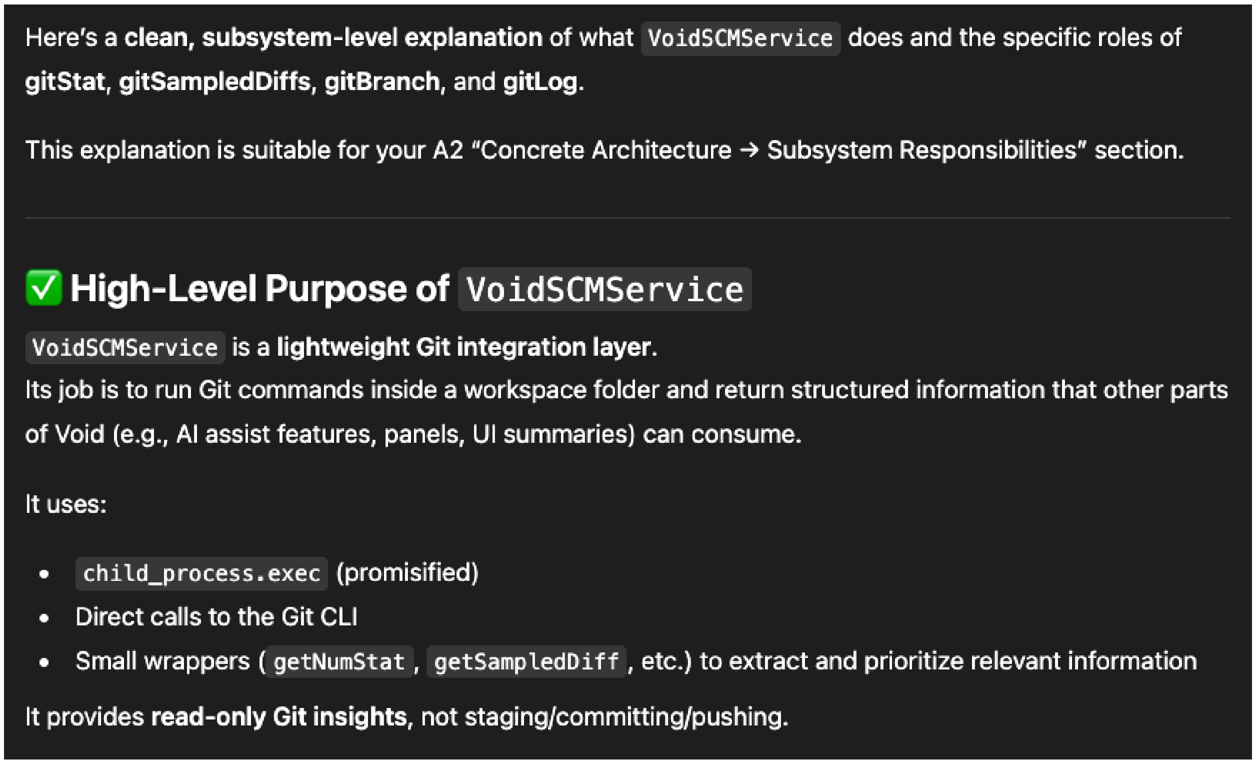
We estimate that ChatGPT contributed roughly 15% in the form of idea support, clarification, and conceptual guidance, rather than producing content. The remaining 85% including all architectural reasoning, dependency mapping, and final writing was completed manually by our team.

Overall, integrating ChatGPT into this phase allowed us to work more efficiently and with better understanding of the real system implementation. It clarified the link between the conceptual and concrete architectures and helped us accurately describe how Void's components interact. Like in A1, ChatGPT served only as a supportive tool. All analysis and conclusions remained entirely a group decision.

Example Usage:



Can you explain what this VoidSCMService does? I'm trying to understand the role of gitStat, gitSampledDiffs, gitBranch, and gitLog in the subsystem.



Here's a **clean, subsystem-level explanation** of what `VoidSCMService` does and the specific roles of **gitStat**, **gitSampledDiffs**, **gitBranch**, and **gitLog**.

This explanation is suitable for your A2 "Concrete Architecture → Subsystem Responsibilities" section.

✅ High-Level Purpose of `VoidSCMService`

`VoidSCMService` is a **lightweight Git integration layer**.
Its job is to run Git commands inside a workspace folder and return structured information that other parts of Void (e.g., AI assist features, panels, UI summaries) can consume.

It uses:

- `child_process.exec` (promisified)
- Direct calls to the Git CLI
- Small wrappers (`getNumStat`, `getSampledDiff`, etc.) to extract and prioritize relevant information

It provides **read-only Git insights**, not staging/committing/pushing.