Void IDE: A Concrete Architecture Analysis

Mason Zhang
22yjz@queensu.ca

Jack Atkinson
22hrsd@queensu.ca

Ryan Sehgal
23sw2@queensu.ca

Jeremie Trepanier
jeremie.trepanier@queensu.ca

Duncan Mackinnon
duncan.mackinnon@queensu.ca

CISC/CMPE 322 – Software Architecture
Queen's University

November 2025

Abstract

In this report we present our study of the concrete architecture of Void IDE, an AI powered code editor based on Visual Studio Code. Building on our earlier conceptual model, this report looks at how the system's structure is shown in the actual code through its files, modules, and runtime links. Using the open source project as our main source, we sorted and grouped files into their main parts: the Workbench, Void UI, Model Services, VS Code Editor, Base, Platform, AI Providers, and File Systems.

Our analysis found that the general architecture stayed mostly the same, keeping the layered and client server style from the conceptual design. However, some new links and dependencies were found in the real code that did not appear in the conceptual view. We talk about these differences between the conceptual and concrete versions in our reflexion analysis.

We also explain how we built the concrete architecture for Void IDE. This includes how we read the source code, sorted files, and mapped them to their parts. We then looked again at two use cases with our new understanding of how the system works in practice. In the end, the report shares what our group learned from this process and includes a short look at the AI features in the system.

Introduction

As artificial intelligence becomes a larger part of software development, the tools that developers use are changing as well. Code editors now do more than just display and format code. They give real time feedback, suggest improvements, and allow users to interact with AI models directly inside the editor. Void IDE reflects this shift. It is an open source code editor that builds on Visual Studio Code and adds AI features into its structure and runtime.

Void IDE keeps all the main parts of Visual Studio Code, such as editing, extensions, and file management, but expands them with new parts for AI interaction and model communication. What makes Void IDE unique is how these new systems are built into the existing framework while keeping stability and performance. Since the project is open source, anyone can explore how its code works and learn how AI systems can be added to a large software project.

This report looks at the concrete architecture of Void IDE. Instead of only describing high level ideas, it studies how the design is shown in the actual code. Files and modules were grouped into main components such as the Workbench, Void UI, Model Services, VS Code Editor, Base, Platform, AI Providers, and File Systems. The analysis shows how these components connect to each other in practice, how their links are formed, and where the code differs from the original design plan.

The report also talks about the new links and dependencies found through reflexive analysis. It includes an explanation of how the concrete architecture was created through code study and mapping, followed by two examples that show how the system works when running.

By the end of this report, readers will understand how the conceptual design of Void IDE is expressed in its real code, how the system supports AI features, and what lessons can be learned from turning a design plan into a working program.

Derivation Process

To create the concrete architecture, we started by reviewing the public source code and documentation of VOID IDE available on GitHub, using SciTools Understand for code analysis. We first looked through the main folders and script names to get a sense of how the project was organized. Since large projects like VOID IDE tend to follow consistent folder and naming patterns, this helped us roughly place files into their likely components from our conceptual architecture. Each component from the conceptual architecture retained their original purpose, each handling a specific part of the architecture. We placed files in such a way so that the concrete dependency graph generated by understand roughly lined up with conceptual architecture with some obvious new dependency. After that, we confirmed our groupings by reading parts of the code, checking imports and the logic within each script. If a file did not seem to fit well in a chosen component, we reevaluated and placed it somewhere more suitable. Files that were not used during runtime, such as build tools or test scripts, were excluded from the concrete architecture. By the end, we established a clear and logical mapping of files to their respective components, giving us a complete and well-structured representation of the VOID IDE's concrete architecture.

High Level Architecture

After completing the high-level concrete architecture (see Figure 1), we found that Void IDE's structure closely mirrors the conceptual design from Assignment 1 (see Figure 3) [3]. The overall layered and client–server organization remains consistent, though the concrete analysis revealed several new dependencies that indicate tighter integration between components. No changes were made to the conceptual architecture for this phase, as it already aligned well with the system's actual structure no new components, renaming, or merging/spiting of components were required.

This high level concrete architecture was generated using the software understand by sci tools. This tool streamlined how the creation of dependency based on the actual source code of Void IDE. The dependency graph is what we based the concrete architecture on (see figure 2).

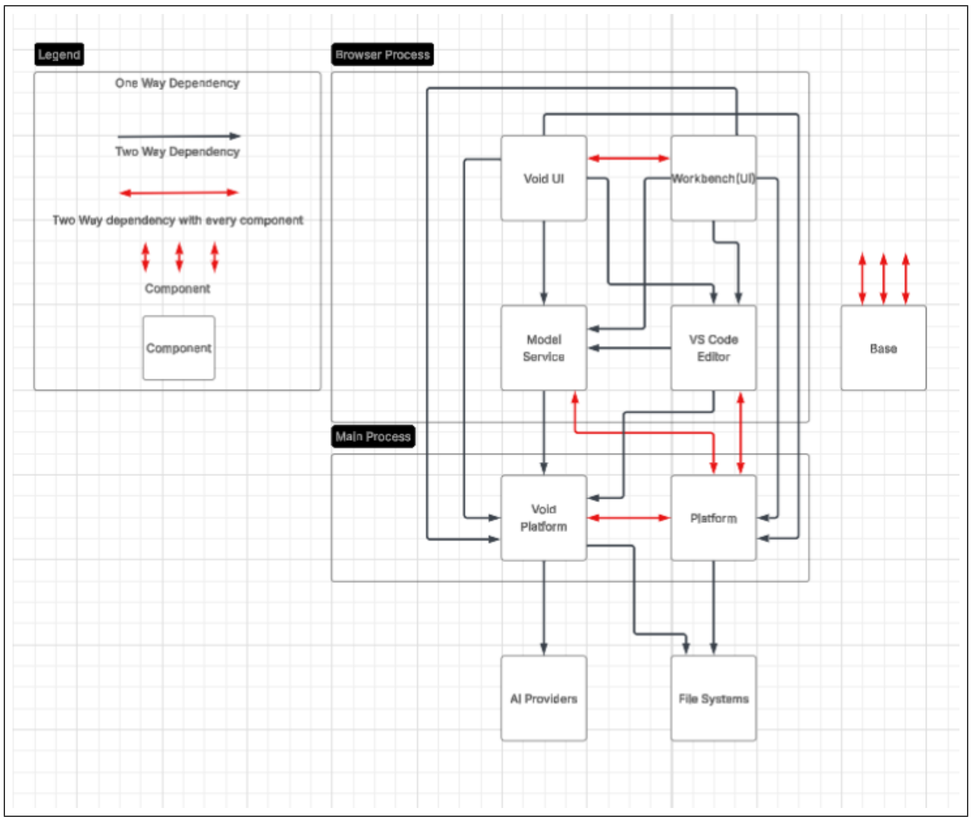

Figure 1: High-Level Concrete Architecture.

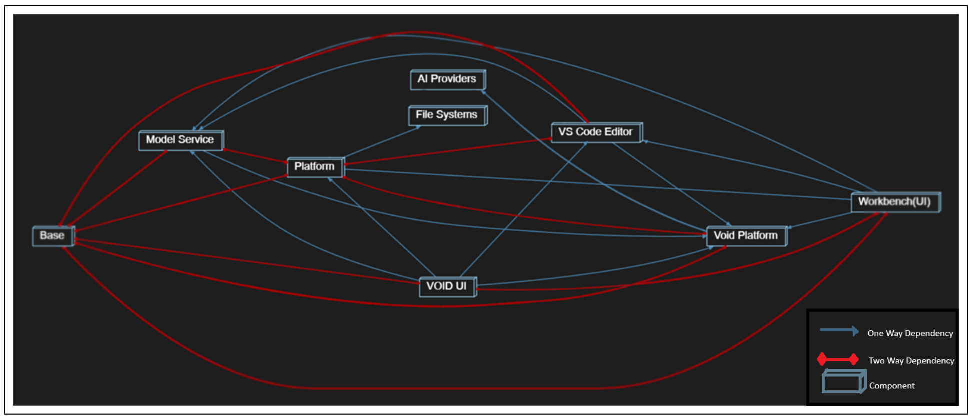

Figure 2: Dependency graph generated by the Understand tool.

Workbench Component

This part of the system holds most of the user interface elements that people see and use in VS Code. It manages things like the code editor window, the search bar, the status bar, and the file explorer. In simple terms, it is the main frame of the program that connects and shows all the tools the user interacts with [1].

VS Code Editor Component

This component controls how the Monaco editor works, which is the main text editor in VS Code. It takes care of features like syntax colors, cursor movement, code folding, and how text editing behaves. Basically, it handles everything that happens when you write or edit code[1].

Platform Component

The platform component works as a link between the program and the computer's operating system. It gives access to system services like reading and saving files, managing processes, and loading resources. This part makes sure the editor can work smoothly across different systems[1].

Void UI Component

This component is where most of VOID IDE's features are found. Built on top of the Workbench, it adds tools like inline code suggestions, a prompt page for writing to AI, a model picker, and a settings panel for managing Void features. It changes the normal VS Code look and adds AI features that help users code[2].

Model Service Component

This component handles how VOID IDE talks to AI systems. It collects and formats messages from the UI before sending them to the platform. It also grabs useful context from the editor, like what file is open or where the cursor is, so the AI can respond in a helpful way[2].

Void Platform Component

This component connects VOID IDE to different AI providers. It sends user prompts to the AI and returns the responses back to the program. It makes sure communication with the AI runs smoothly and lets new models be added easily without changing the rest of the system[2].

High Level Reflection Analysis

After completing our detailed high level concrete architecture, we found that the overall design of VOID IDE still followed the same structure as our original conceptual model (see figure 3). The system continues to use a layered architecture and a small part of the project uses a client-server setup, mainly for features that handle communication with large language models (LLMs).

This high level concrete architecture was generated using the software understand by sci tools. This tool streamlined how the creation of dependency based on the actual source code of Void IDE. The dependency graph is what we based the concrete architecture on (see figure 2).

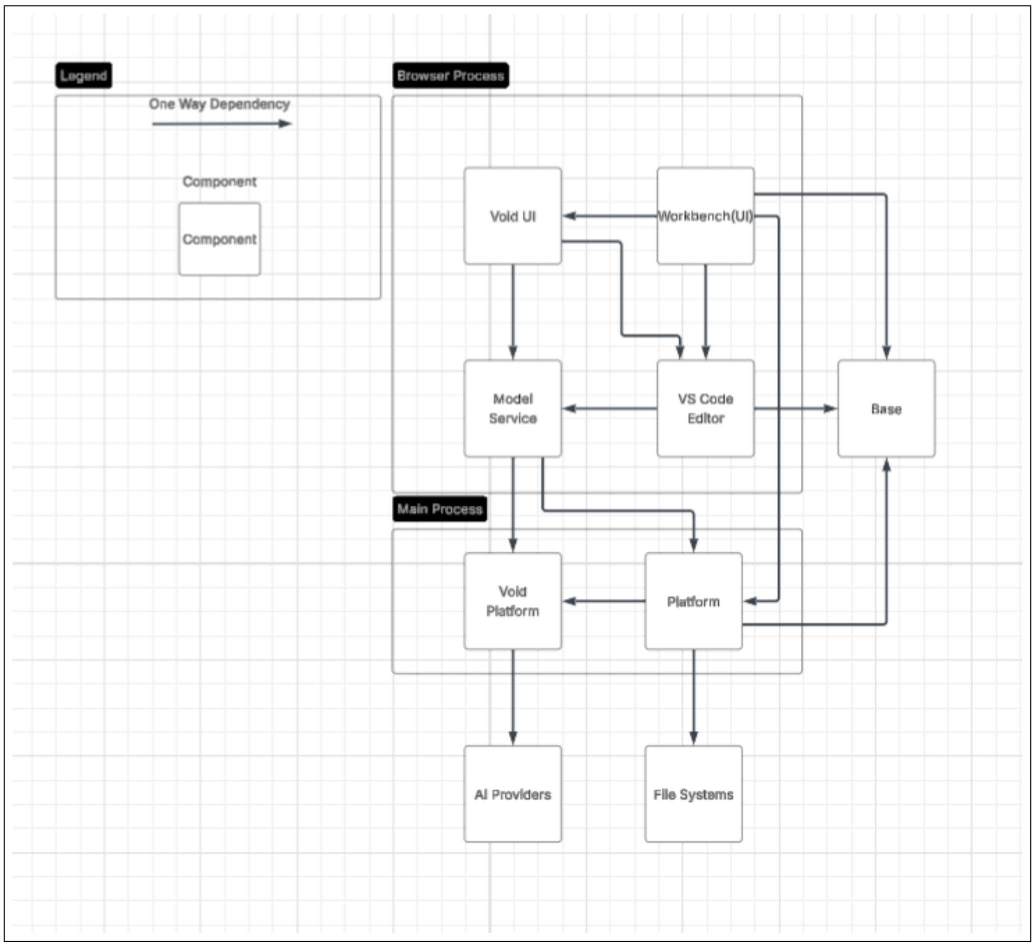

Figure 3: High-Level Concrete Architecture.

Workbench Component

This part of the system holds most of the user interface elements that people see and use in VS Code. It manages things like the code editor window, the search bar, the status bar, and the file explorer. In simple terms, it is the main frame of the program that connects and shows all the tools the user interacts with [1].

VS Code Editor Component

This component controls how the Monaco editor works, which is the main text editor in VS Code. It takes care of features like syntax colors, cursor movement, code folding, and how text editing behaves. Basically, it handles everything that happens when you write or edit code[1].

Platform Component

The platform component works as a link between the program and the computer's operating system. It gives access to system services like reading and saving files, managing processes, and loading resources. This part makes sure the editor can work smoothly across different systems[1].

Void UI Component

This component is where most of VOID IDE's features are found. Built on top of the Workbench, it adds tools like inline code suggestions, a prompt page for writing to AI, a model picker, and a settings panel for managing Void features. It changes the normal VS Code look and adds AI features that help users code[2].

Model Service Component

This component handles how VOID IDE talks to AI systems. It collects and formats messages from the UI before sending them to the platform. It also grabs useful context from the editor, like what file is open or where the cursor is, so the AI can respond in a helpful way[2].

Void Platform Component

This component connects VOID IDE to different AI providers. It sends user prompts to the AI and returns the responses back to the program. It makes sure communication with the AI runs smoothly and lets new models be added easily without changing the rest of the system[2].

High Level Reflection Analysis

After our group finished analyzing and formulating the concrete architecture through the examination of the source code, we proceeded to analyze the discrepancies between our conceptual (see figure 3)[3] and concrete architecture. In the following, we point out the reason for each divergence and what causes them to occur.

Base

Every component was found to have dependency to the base component and conceptually this makes sense especially since the base component holds foundational utilities used to make the software run smoothly. In the conceptual design, only the Workbench, Base, and Platform parts were shown to depend on Base. But in the real code, every part of the system depends on it. This happens because Base contains small but important tools used across all parts, such as event systems, cleanup helpers, and data utilities. These shared tools make coding easier and keep behavior consistent. The difference between the two views comes from the desire to keep the conceptual architecture simple. The conceptual view hides technical details to stay simple, while the real code must show how all parts use the same core building blocks from Base, thus the concrete has these new dependencies[4].

Void UI

In the conceptual architecture, Void UI depended only on the VS Code Editor, Void Model Service. In the concrete architecture, it also depends on Workbench, Void Platform, and Platform. This happens because Void UI features, such as chat windows, inline prompts, and AI panels, are built inside Workbench layouts and use its APIs to show and manage views. Void UI now connects to Void Platform for helpers like window control, webview management, and service routing. It also uses the base Platform for theming, workspace data, and file access. The difference appears because the real code integrates Void UI more tightly with the shell and host systems to enable smooth layout and shared services, even though the conceptual architecture kept it simpler and separate[4].

Workbench (UI)

In the conceptual architecture, Workbench depended on Void UI, VS Code Editor, and Platform. In the concrete architecture, Workbench also depends on Void Platform and Model Service. This change happens because the code that runs Workbench now manages both user interface elements and AI features. During startup, it imports registration files from Void UI and connects to services that process AI prompts and model outputs. It also calls utilities from Void Platform for data handling, webview control, and window management. The difference comes from the real need for coordination. The conceptual architecture separated the interface shell from logic and services, but in practice, Workbench must initialize and connect these parts for the IDE to function. This direct setup makes the system more integrated and faster[4].

Model Service

In the conceptual architecture, Model Service depends on both Void Platform and Platform. In the concrete architecture it only depends on the Void Platform component. This happens because many Platform utilities were moved inside Void Platform during development. The functions for communication, logging, and data handling now come from Void Platform rather than the general Platform layer. The change makes the design simpler. Model Service now focuses only on preparing and formatting model requests, while Void Platform handles the connection to providers and shared utilities[4].

VS Code Editor

In the conceptual architecture, the VS Code Editor component depends only on Model Service, base and platform components. In the concrete architecture, it also depends on the Void Platform component. This change happened because AI features like inline completions and model selection were built directly into the editor. These features use Void Platform for faster communication and data handling. The divergence comes from a need for performance and smoother user interaction. Instead of sending everything through the Model Service, the editor now connects directly to Void Platform to get quicker responses. This made the system faster but added an extra dependency not seen in the original design[4].

Void Platform

In the conceptual architecture, Void Platform depended only on AI providers. In the concrete architecture, it also depends on the file system and the Platform component. This happens because the real system must handle more than model communication; it also manages storage, data flow, and process control. Void Platform uses file system services to save model data, logs, and user settings. It connects to the Platform for features like window management, updates, and webview hosting. The difference appears because, in practice, Void Platform acts as a middle layer between AI providers and the environment, needing access to both runtime and file handling tools that were not shown in the conceptual architecture[4].

Platform

In the conceptual architecture, Platform depends on Base and the file system. In the concrete architecture, it also depends on the Void Platform. This change happens because some startup and environment logic that used to sit in Base has been moved into Void Platform. The platform now calls services from Void Platform to initialize AI systems, manage webviews, and handle cross process communication. The link to Base was replaced by a link to Void Platform because Void Platform wraps those core helpers and extends them with AI aware behavior. The difference comes from code integration needs. Instead of staying isolated, Platform now helps load and coordinate AI features through Void Platform, which causes the added dependency[4].

The ChatThreadServices Subsystem

Overview

ChatThreadServices is the coordinator that moves a user request through VOID's AI flow. It keeps the conversation organized across threads, asks the model at the right times, invokes tools when the model requests action, and presents diffs before any change is applied. Throughout, it exposes a small set of states so the UI can show progress and it places checkpoints around edits so work remains reversible [5, 8]. Our exposition follows the clean, prose-first style used in the exemplar architecture report while staying specific to VOID's services and terminology.

Conceptual Architecture

A request begins at the UI and is shaped before the model ever sees it. Context Extraction assembles the working set from what the developer is actually touching: open buffers, recent edits, diagnostics, and the project tree. When space is tight it trims or summarizes, and it tags each item with where it came from so later steps can explain why it was included [5]. Prompt Engineering then turns intent and that working set into a bounded instruction. It states the objective, clarifies constraints and success criteria, and declares which tools may be called, all within the model's budget [5].

With those inputs prepared, ChatThreadServices advances the conversation step by step. It sends the request through the messaging/connection layer to keep providers abstracted; the model streams partial text, final completions, or structured tool-calls back to the orchestrator. When action is requested, the orchestrator routes to ToolServices to read, search, write, or run a command, and then resumes the model stream with the result. When the output proposes code changes, the system surfaces diffs in the editor rather than applying them silently. Policy stays in the loop: risky operations and large edits are approval-gated, cost and rate are bounded, and model selection contributes capability hints such as context limits or tool-use support so the flow remains within safe, predictable boundaries [8].

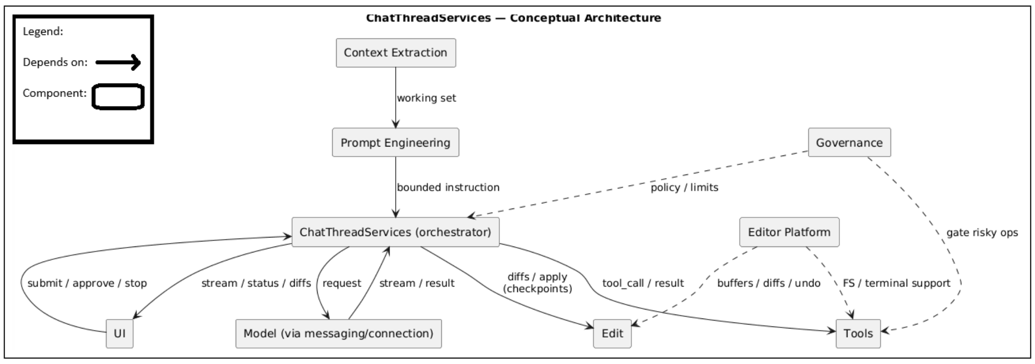

Figure 4: Low-Level Conceptual Architecture.

Concrete Architecture

In the workbench code, the same flow appears as a compact set of cooperating parts. A per-thread state model holds the active thread identifier, message history, and the current step; it persists through storage so switching threads reconciles in-flight work and the chat surface stays in sync [6]. Message submission enters a dispatch loop that delegates to the provider-neutral messaging layer, streams tokens, and pauses cleanly when the model issues a tool-call. Tool results are fed back and streaming resumes until completion or user stop [7]. A lightweight tracker marks transitions—idle, streaming, tool-running, awaiting approval—and raises events that the UI consumes for progress, abort, and status without guessing [7].

Actions run behind an approval-aware path. Tool requests are validated and—when required—prompts are shown inline to Approve or Deny. On approval, the operation executes (file or terminal), returns a structured payload, and the conversation continues. On denial, the action is halted with a clear policy reason and the thread remains consistent [8]. Around edits, the service asks the editor to create snapshots so apply/undo remain exact rather than approximate; larger or cross-file proposals are summarized before apply to keep the user in control. A small public surface—create or switch a thread, submit a message, stop a stream, answer an approval, navigate checkpoints—backs these behaviors, and corresponding events keep the chat panel, thread switcher, and status indicators current [6, 7].

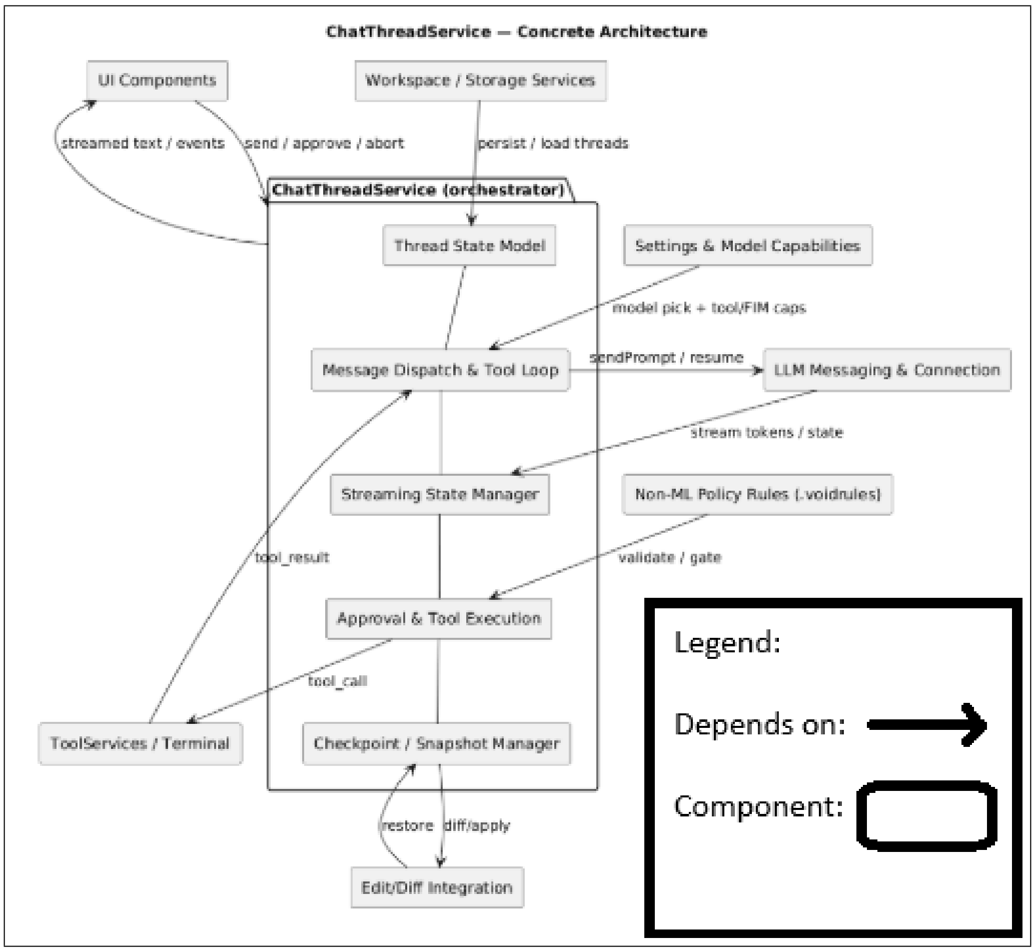
ChatThreadService — Concrete Architecture
UI Components
Workspace / Storage Services
streamed text / events
send / approve / abort
persist / load threads
ChatThreadService (orchestrator)
Thread State Model
Settings & Model Capabilities
model pick + tool/FIM caps
Message Dispatch & Tool Loop
sendPrompt / resume
LLM Messaging & Connection
stream tokens / state
Streaming State Manager
Non-ML Policy Rules (.voidrules)
tool_result
validate / gate
Approval & Tool Execution
tool_call
ToolServices / Terminal
Checkpoint / Snapshot Manager
restore
diff/apply
Edit/Diff Integration
Legend:
Depends on:
Component:

Figure 5: Low-Level Concrete Architecture.

Interfaces and Data

From the UI, the subsystem receives intent and selections; from Context Extraction and Prompt Engineering, it receives a compact bundle and a bounded instruction. Toward the model, all traffic moves through the messaging/connection layers to avoid embedding provider logic in the orchestrator. When tools are needed, parameterized file and terminal actions are issued and results return as typed payloads suitable for reinjection or display. When edits are proposed, diffs are produced and accepted through the editor's standard apply/undo paths; checkpoints bracket these steps. Policy outcomes and model capabilities influence the path taken through the loop but do not change the public surface of the subsystem [5, 6, 7, 8].

Reflexion Analysis

Comparing the two views, the conceptual loop that the reader reasons about—shape intent, ask a model, optionally act, and show diffs—maps directly to concrete responsibilities in the workbench. The thread state model anchors continuity; the dispatch loop alternates between streaming and tools while keeping providers abstracted; the state tracker makes progress legible; the approval path and checkpoints make changes safe and reversible. This matches VOID's documentation that emphasizes provider-agnostic messaging, explicit approvals for risky operations, and diff-first edits as the foundation for trust [5, 8].

textbfUse Cases

Use Case 1: "The user is typing in the editor and the system automatically suggests an inline code completion."

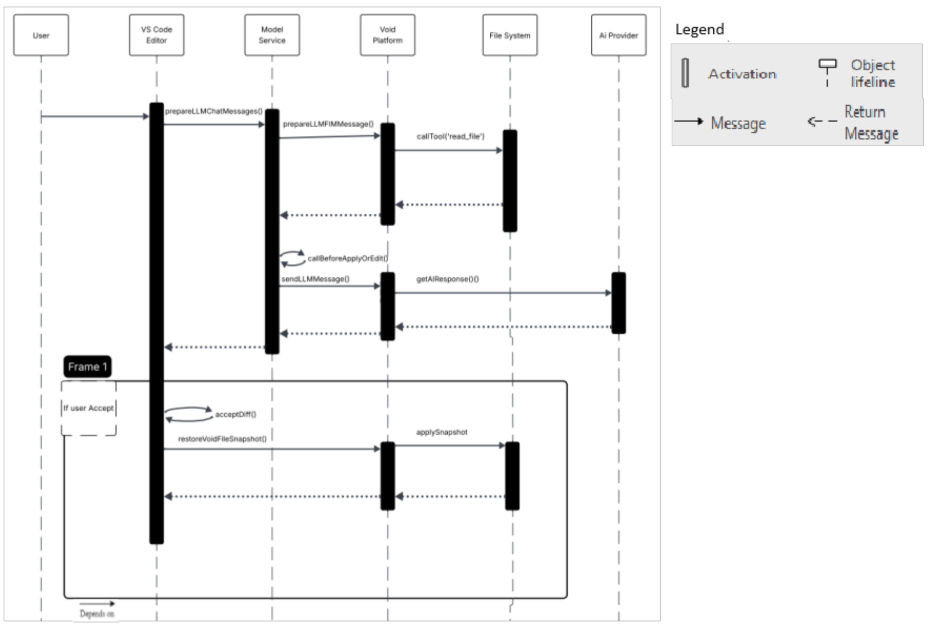

Figure 6: Sequence diagram showing the events of Use Case 1 in the concrete architecture.

When the user is typing in the code editor, Void automatically provides an inline code suggestion powered by an AI model. This use case highlights how the system responds to user input and coordinates across the same conceptual layers used in Assignment 1, now implemented by concrete services inside the code base. It follows this sequence of events[4]:

1. User → VS Code Editor – The user types code in the VS Code editor, triggering the `IEditCodeService.startApplying()` method through an event listener in `EditCodeService`.

2. VS Code Editor → Model Service – The Void UI forwards this edit event to the Model Service, implemented by `IEditCodeService`, which requests additional project context.

3. Model Service → Void Platform – The Model Service calls `IConvertToLLMMessageService`, implemented within `ConvertToLLMMessageService.ts`, to construct an AI-ready message.

4. Void Platform → File System – `IConvertToLLMMessageService` invokes `IToolsService.callTool(` or `callTool('get_dir_tree')` to gather relevant file contents and directory structure.

5. Model Service → Void Platform – The Model Service calls `IConvertToLLMMessageService.prepare` to build a Fill-In-Middle (FIM) prompt.

6. Void Platform → AI Provider – The AI Provider returns a completion suggestion via `getAIResponse()`.

7. VS Code Editor – `IEditCodeService.acceptDiff()` applies the inline suggestion and updates the code buffer.

8. VS Code Editor → Void Platform – Upon acceptance, `restoreVoidFileSnapshot()` is called on the Void Platform.

9. Void Platform → File System – The Void Platform invokes `applySnapshot()` to save changes.

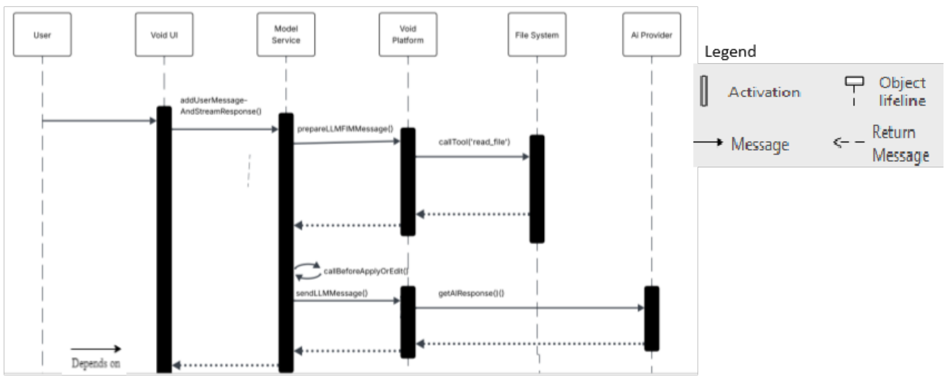

Figure 7: Sequence diagram showing the events of Use Case 2.

When the user interacts with Void's chat interface, the system processes a natural-language request through the same conceptual layers—Void UI, Model Service, Void Platform, File System, and AI Provider—each implemented by specific services in the code base. It follows this sequence of events[4]:

1. User → Void UI – The user enters a natural-language prompt into the chat window.

2. Void UI → Model Service – The `ChatThreadService` forwards the prompt to `ConvertToLLMMessageS`
3. Model Service → Void Platform – The Model Service calls `IToolsService.callTool('read_file')` or `callTool('get_dir_tree')`.
4. Void Platform → File System – The VS Code File System API reads project files and returns results.
5. Model Service → Void Platform – The Model Service calls `prepareLLMFIMMessage()` to construct the FIM prompt.
6. Void Platform → AI Provider – The AI Provider returns a completion suggestion via `getAIResponse()`.

Conclusions

This report gives a detailed look at the real structure of Void IDE and how it matches the ideas from the original conceptual architecture. It explains how the source code was studied, how the files were grouped into main parts, and how their connections were checked using the Understand tool. Each part, including the Workbench, Void UI, Model Service, and Void Platform, etc. was examined to see how they work together while the program is running.

The comparison between the conceptual and the actual code showed where new links appeared and why they were needed when the system was built. The examples of user actions also helped show how these parts interact in real use, especially when working with the artificial intelligence features.

In the end, this study shows that Void IDE keeps a clear and balanced structure while adding new features that use machine learning. The system is organized, flexible, and built in a way that will allow it to keep improving as development tools and AI continue to grow.

Limitations and Lessons Learned

This study gave us a good understanding of Void IDE's structure, but we faced some limits. Since Void IDE is a large version of Visual Studio Code with new AI features added, it was hard to track every link between components. Some files are not clear on what they actually do, in cases like these best judgment an asumptions based on file paths or file name is used.

While working on this report, we learned how hard it can be to match design ideas with what is really in the code. We saw that large projects like Void often mix layers when new features are added, which creates links that do not appear in the conceptual architecture. By checking both the code and the project files, we built a clearer view of how the system fits together. Overall, this work taught us to be careful when studying real software and to focus on clear proof rather than assumptions.

Data Dictionary

- **Void IDE:** The modified version of Visual Studio Code that includes artificial intelligence features such as chat prompts, inline code suggestions, and model selection. It serves as the main subject of this study.

- **Visual Studio Code (VS Code):** The open-source editor created by Microsoft that forms the base structure of Void IDE. It provides the original layers, services, and editor components.
- **Workbench:** The main user interface framework of the editor. It manages windows, panels, menus, and the layout where other tools and features appear.
- **Void UI:** The set of new user-facing features added by Void IDE. It includes the chat interface, inline AI prompts, settings, and model selection panels that connect the user to AI functions.
- **Model Service:** The internal system that prepares and sends messages from the user interface to the AI models. It formats user input, gathers context, and builds structured prompts for model requests.
- **Void Platform:** The communication layer that connects Void IDE to external AI providers. It manages requests, handles responses, and coordinates how data moves between the editor and the model.
- **Platform:** The layer that links the editor to the computer's operating system. It provides access to files, processes, and system resources while ensuring compatibility across environments.
- **Base:** The shared foundation that provides common utilities and tools used across all components. It includes functions for events, file handling, data structures, and background services.
- **AI Provider:** The external service or model (such as a large language model) that generates text, code suggestions, or answers based on user prompts.
- **File System:** The layer responsible for reading, writing, and saving user files within the editor. It supports the storage and retrieval of data for both user actions and AI processing.
- **ChatThreadService:** The system that manages AI conversations inside Void IDE. It tracks message history, tool requests, and approvals, keeping the chat flow organized and consistent.
- **SciTools Understand:** The analysis tool used to study code dependencies and generate the concrete architecture diagrams for this report.

Naming Convention

In this report, names and terms were written to stay consistent with the style used in the Void IDE codebase and related documentation. Component names such as Workbench, Void UI, Model Service, and Void Platform are capitalized to show they represent major parts of the system. When referring to smaller parts or internal files, lowercase words and code-style formatting were used, such as `editCodeService` or `convertToLLMMessageService`.

File and folder names follow the same naming patterns found in the repository. This keeps the report accurate to the real structure of the code. System terms like "platform," "editor," and “service” describe function rather than location, while “component” is used only for large, independent sections of the architecture.

Throughout the document, both conceptual and concrete layers were named using the same format to make it easier to see how ideas from the design phase appear in the actual implementation. This approach keeps the report clear, consistent, and aligned with the naming logic used in the source code of Void IDE.

References

[1] Microsoft, “Source Code Organization,” *GitHub*, Oct. 11, 2025. [Online]. Available: https://github.com/microsoft/vscode/wiki/Source-Code-Organization

[2] “Void Overview,” *zread.ai*, 2025. [Online]. Available: https://zread.ai/voideditor/void/1-overview

[3] M. Dobaj, S. Thomas, B. Batala, E. Mirchandani, S. Mourad, and N. Kodukula, “A1: Conceptual Architecture of Void CISC 322 Authors,” 2025. [Online]. Available: https://babinson2005.github.io/CISC322/

[4] voideditor, “GitHub - voideditor/void,” *GitHub*, 2024. [Online]. Available: https://github.com/voideditor/void/

[5] A. Kumar, “VOID IDE: The Comprehensive Guide to the Open-Source Cursor Alternative,” *Medium*, 2024. [Online]. Available: https://medium.com/@adityakumar2001/void-ide-the-comprehensive-guide-to-the-open-source-cursor-alternative-2a6b17cae235

[6] voideditor, “VOID Codebase Guide,” *GitHub*, 2024. [Online]. Available: https://github.com/voideditor/void/blob/main/VOID$_{C}ODEBASE_{G}UIDE.md$

[7] voideditor, “ChatThreadService,” *GitHub*, 2024. [Online]. Available: https://github.com/voideditor/void/blob/main/src/vs/workbench/contrib/void/browser/chatThrea

[8] “VOID Architecture Overview,” *zread.ai*, 2024. [Online]. Available: https://zread.ai/voideditor/void/9-architecture-overview

AI Report

AI Model Selection

We used ChatGPT (GPT-5, November 2025) as our primary AI collaborator, selected for its strong reasoning and familiarity with the previous A1 phase. Alternative tools like Claude and Gemini were evaluated but not chosen due to less reliable formatting and code analysis output.

Role and Task Assignment

AI tasks were supportive and analytical, not generative. Specifically:

1. Helping format and grammatical consistency

2. Reviewing sequence diagrams for logical consistency and dependency validation.
3. Suggesting improvements in phrasing, section structure, and format of the report and slides.
4. interpolate the purpose of code/files if their purpose is not clear

Interaction Protocol and Prompting Strategy

We followed a structured prompting strategy:

- **Context-first prompting:** Each prompt began with A1 architecture and the at the time available sections of A2. Also attached was the Void IDE's GitHub repository.
- **multiple iteration:** Every prompt was asked by two group members and ensured their answer was simar so any chance based hallucinations can be reduced.
- **Human verification loop:** Each AI-assisted explanation was reviewed by two members who fact checked the AI answer with either documentation or source code.

Quality Control and Validation

- **Source verification:** Every AI-supported claim was verified via Understand output.
- **Version tracking:** Git logs cross-checked AI summaries for accuracy.
- **Peer review:** Two members validated each AI-suggested phrasing or structure.

Quantitative Impact

ChatGPT contributed approximately 20% of the overall deliverable, mainly through verification, summarization, and documentation refinement. All architectural and analytical work remained human derived.

Reflection on Human–AI Collaboration

Compared to A1, our AI interaction was deeper but more structured. GPT-5 accelerated repetitive analytical tasks, letting us focus on interpreting divergences and linking evidence from Understand. However, we learned that AI often generalized relationships and required careful correction—reinforcing that AI works best as an analytical amplifier, not an architectural authority.