**Recovering the Concrete Architecture of Void**

**Project:** Void IDE - AI-Assisted IDE built on VS Code and Electron
**Deliverable:** A2 - Concrete Architecture Report
**Course:** CISC 322
**Group Members:** Colin McLaughlin (Group Lead), Adrian Yanovich-Ghitis (Presenter), Thomas Schrappe (Presenter), Jaiman Sharma, Mantaj Toor, Lingwei Huang
**Date:** November 7th, 2025

---

## Abstract

This report presents the concrete architecture of the Void IDE, an AI-assisted development environment built as a Visual Studio Code extension on Electron. The study recovers the as-built structure of the overall system and of the Void subsystem, then evaluates that structure against the conceptual design from our Assignment 1 report. We apply static analysis with SciTools Understand to identify components, dependencies, call paths, and basic code metrics across VS Code core, the Electron host, the Monaco editor, and the Void extension. The top level view characterizes subsystem responsibilities and dependency directions. The subsystem deep dive details Void's internal services, provider adapters, UI integration points, and boundaries with VS Code APIs.

Behavioral analysis complements the structural view by showing how the system actually executes key interactions. Two representative scenarios are illustrated through concrete sequence diagrams that use real class and method names: one traces a user's prompt request as it moves through the Void extension to the AI provider and back, while the other captures the flow of a live inline suggestion within the editor. Following this, a two-level reflexion analysis compares the conceptual design with the implemented system; first at the overall level of VS Code and Void, and then within the internal structure of the Void subsystem. Each reflexion table highlights where the architecture aligns or diverges,and identifies matched connections, missing dependencies, and unexpected links, together with concise explanations based on design intent or technical constraints.

The results drive an updated conceptual architecture that preserves accurate A1 elements and revises connectors where the codebase provides evidence. The report highlights architectural styles in effect, noteworthy design patterns within Void, and implications for maintainability, testability, and future AI provider integrations. Overall, the document provides a precise and verifiable map from design intent to implemented code.

---

## 1. Introduction and Report Organization

### 1.1 Purpose and Scope

The purpose of this report is to document the concrete architecture of Void IDE, an AI-assisted development environment implemented as a VS Code extension built on Electron. Building on the conceptual model established in Assignment 1, this analysis recovers the as-built structure of the system through static analysis and compares it against the intended design. The study focuses on two levels of detail required by the assignment:

1. The top-level architecture, encompassing VS Code Core, Electron Runtime, and the integrated Void extension.
2. The subsystem-level architecture of Void itself, which contains its internal AI services, provider interfaces, and UI components.

The objective of this report is to determine how accurately the implemented system reflects the conceptual structure from A1 and to explain any discrepancies.

### 1.2 Architectural Context

To interpret the recovered concrete architecture, it helps to recall the main components established in Assignment 1: Electron (runtime), VS Code Core (editor and platform), and Void (AI extension). These layers define the foundation on which the as-built system operates and frame the analyses that follow.

#### Electron Runtime

Electron provides the runtime environment. It combines Chromium for interface rendering with Node.js for backend access. Its main process manages lifecycle and OS integration, while renderer processes handle UI and editor windows. Communication through asynchronous Inter-Process Communication (IPC) channels enables concurrency, fault isolation, and portability across operating systems.

#### VS Code Core

VS Code builds on Electron using a layered, service-oriented architecture with elements of client-server and implicit invocation. Core functionality includes base utilities, a dependency-injection platform, the Monaco editor, and a Workbench layer that manages user interactions and extensions. All extensions, including Void, run in a dedicated Extension Host that communicates with the core via defined APIs. This microkernel pattern supports extensibility while maintaining responsiveness and process safety.

**Void Extension Layer**

Void extends the Workbench to deliver AI-driven code generation, refactoring, and conversational support. Its key services include:

LLMMessageService – streams prompts and responses to external LLM providers.
editCodeService – manages AI-generated edits and diff zones.
chatThreadService – maintains conversational session state.
voidSettingsService – stores configuration and provider settings.
toolsService – executes controlled workspace actions.

Together, these services form a hybrid client-server and event driven architecture that links the editor frontend to a Node based backend. Their asynchronous interactions preserve modularity and maintain Electron's process isolation.

### 1.3 Derivation and Analysis Method

As discussed further in Section 2, the concrete architecture was recovered using SciTools Understand to parse source files, extract dependency graphs, and measure coupling and cohesion across modules. The tool output guided the identification of concrete subsystems and relationships. The recovered components were then compared with the conceptual architecture from A1 to locate alignment or erosion.

Two representative behavioral scenarios were modeled as concrete sequence diagrams:

1. A user prompt request routed through Void to an AI provider and back to the editor.
2. An inline suggestion generated while a user edits code in real time.

Each diagram uses actual class and method names to depict runtime interactions that reflect the system's real behavior.

Furthermore, as discussed in Section 6, a two-level reflexion analysis is performed. The first level examines the overall integration of VS Code and Void. The second level analyzes the internal structure of Void and its component dependencies. Each reflexion table identifies matches, missing links, and unexpected dependencies, with concise explanations rooted in design intent, technical limitations, or architectural drift.

The outcomes of this analysis, discussed in Section 7, inform an updated conceptual architecture, and highlights how the implementation evolved and where improvements could increase modularity or alignment with the supposed original design goals.

---

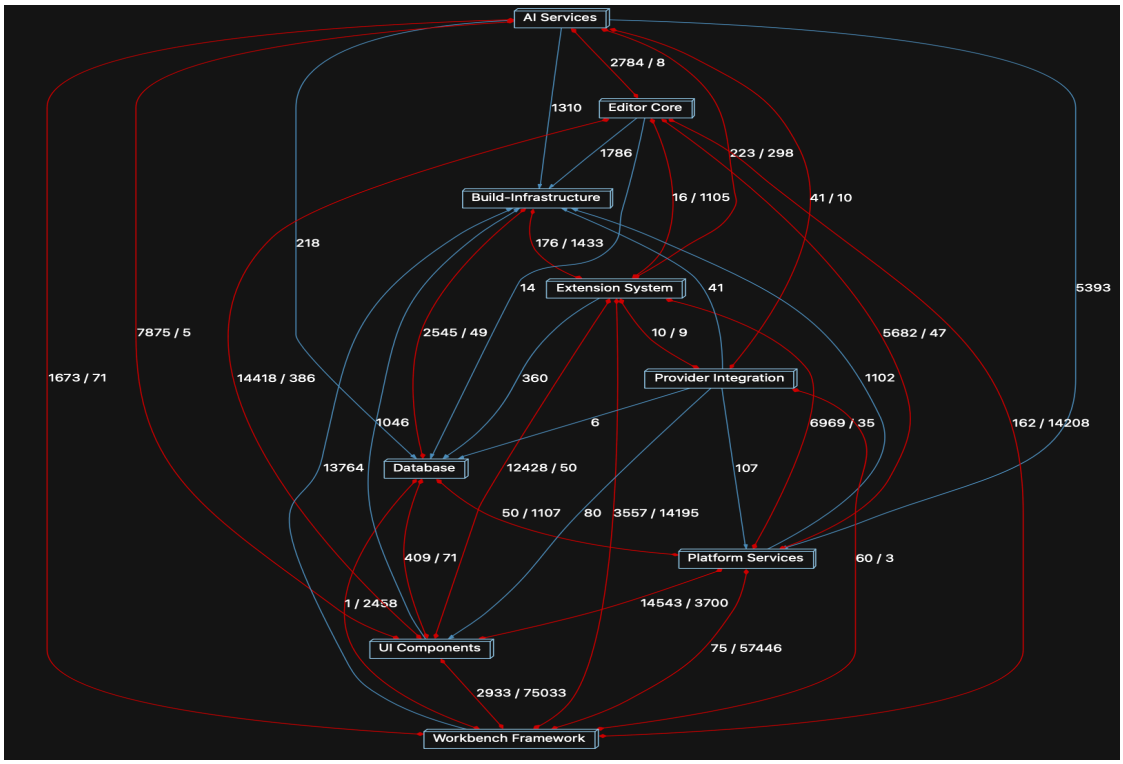
## 2. Methodology: Deriving the Concrete Architecture

To examine the concrete architecture, we started by downloading the source code of Void and opening the .und file within the prepared source code within Understand. We started with this in order to get a grasp on how major directories interact with each other and a rough purpose of each directory (more on this later). In order to determine the concrete architecture of Void, we next re-visited our already done conceptual model and determined what major subsystems we should focus on that made up Void's source code. Once we had determined what to look out for, we started by creating a dependency diagram of Void's subsystems and explored how each level is designed. When exploring the directories, we noted the similarities of our conceptual diagram to the directories names and purposes (most of the directories matched our pre-existing conceptual subsystem such as workbench, platform, editor, and extensions).

Once we had a good idea of how the directories are structured, we went forward with using the architecture designer tool in Understand to map major subsystems, directories and files together. Once we had gone through all the files we were confident could be mapped to the according existing subsystems, we double checked with the existing documentation (github, medium, zread), we realized the leftover files and directories did demand a new subsystem, we decided these files and directories (such as build, test, typings, server and bootstrap files) belonged under the new system "Build/Infrastructure". The new subsystem contains the test suite, resources for building the application (typescript to javascript, bundling, platform support), initializing the application (bootstrap), CI/CD (test suite) and other resource management. Most of the functionalities and lower level directories did not warrant multiple new subsystems at this stage, due to the fact that this content isn't large enough to warrant multiple new subsystems. Using Git Blame, we were able to locate the commit messages of Void and reason out why certain dependencies existed.

Once we analyzed the new concrete architecture, we came to the conclusion that the overall architectural style of Void is Layered due to the organization of the different layers that are responsible for their own functions, where upper layers only depend on the layers below them, this architectural style comes from VSCode (since Void is a fork of VSCode) and its directory level control (ESLint) of its workbench → editor → platform → base layers. In VSCode, the base layer is responsible for data structures, utility functions, and event control. The platform layer is responsible for OS abstractions (files and processes), command registry, logging, and configuration. The editor layer is responsible for text management, text manipulation, rendering, language features and editor behaviour. The workbench layer is responsible for the shell of the application (title bar, sidebar, panel, status bar etc.), some core IDE features such as file navigation, debugging UI, integrated terminal and extension and tool output.

The secondary architectural styles are client-server and implicit invocation. Client-server is a secondary architectural style in Void because of the structure of the electron framework, where the primary processes are assigned their own roles, such as the main process, which manages the lifecycle of the application and makes new application windows. The renderer process is responsible for rendering content using HTML, CSS, and JavaScript. It can also request services from the main process using interprocess communication. There is also the utility process, which is responsible for generating child processes that can host untrusted processes and resource-intensive tasks. The secondary architectural style Implicit Invocation style, controls communication between layers. The components in Void utilise an event-driven system where components communicate through events rather than making calls to other components.




---

### 3. High-Level Concrete Architecture: VS Code + Void

Based on the results of our derivation using Understand, we were able to create a high level concrete architecture using our previous conceptual diagram and the dependency diagram attained using Understand.

**Old Subsystem:**

UI Components

The UI Components subsystem renders Void's React-based chat interfaces and webview panels by using AI Services to display conversation threads and code suggestions, while also listening to platform services for theme updates and configuration changes to maintain visuals.

Platform Services

The platform services subsystem provides infrastructure services, including configuration management, storage, IPC, telemetry, and AI provider settings to all other subsystems through a dependency injection system where components request services using interfaces.

Editor Core

The editor core subsystem implements the Monaco text editor with syntax highlighting, IntelliSense, and modular editing features, using platform services for configuration and keybinding support.

Workbench Framework

The workbench framework subsystem acts as a core part of Void, managing services and coordinating feature contributions, using both Platform Services for infrastructure needs and editor core for text editing, while providing the container that binds all subsystems together and manages their lifecycle.

AI Services

The AI services subsystem contains the core intelligence of Void, managing LLM usage, conversation state, and inline code suggestions, receiving user requests from UI Components, requesting platform services for model settings, directly accessing Editor Core for code context and analysis, and sending requests to provider integration for actual AI model communication.

Database

The database/files subsystem provides file system operations through multiple providers supporting disk access and in-memory storage, with file monitoring capabilities that notify other subsystems of changes. It also manages persistent value storage using SQLite for AI Services to store conversation histories and model configurations, and for platform services to persist user preferences and application state, with bidirectional dependencies on platform services for the file system and change notifications.

Extension System

The extension system subsystem manages extension lifecycle and discovery by integrating with platform services for extension data and configuration, working with the workbench framework for extension host process management, and allowing the use of Void-specific extension points that allow third-party extensions to access AI services capabilities like chat agents and language model tools while maintaining VSCode's compatibility.

Provider Integration

The provider integration subsystem, implemented in the main process, acts as an abstraction layer that receives requests from AI Services and translates them into provider specific API calls

for OpenAI, Anthropic, Ollama, etc, handling custom endpoints for self-hosted models and sending responses back to AI Services in real time.

**New Subsystem:**

Build/Infrastructure

The Build/Infrastructure subsystem acts as a foundational utilities “library”, providing shared helper functions, common data structures, development tooling, application assets, test suite, and runtime server code that are used by other subsystems but do not warrant more than 1 subsystem. This subsystem acts as a utility subsystem; most of the other subsystems in some way depend on this subsystem in order to function correctly. This subsystem acts as a dependency for platform services, editor core, workbench framework, and other subsystems.

**High-Level Concrete Architecture:**

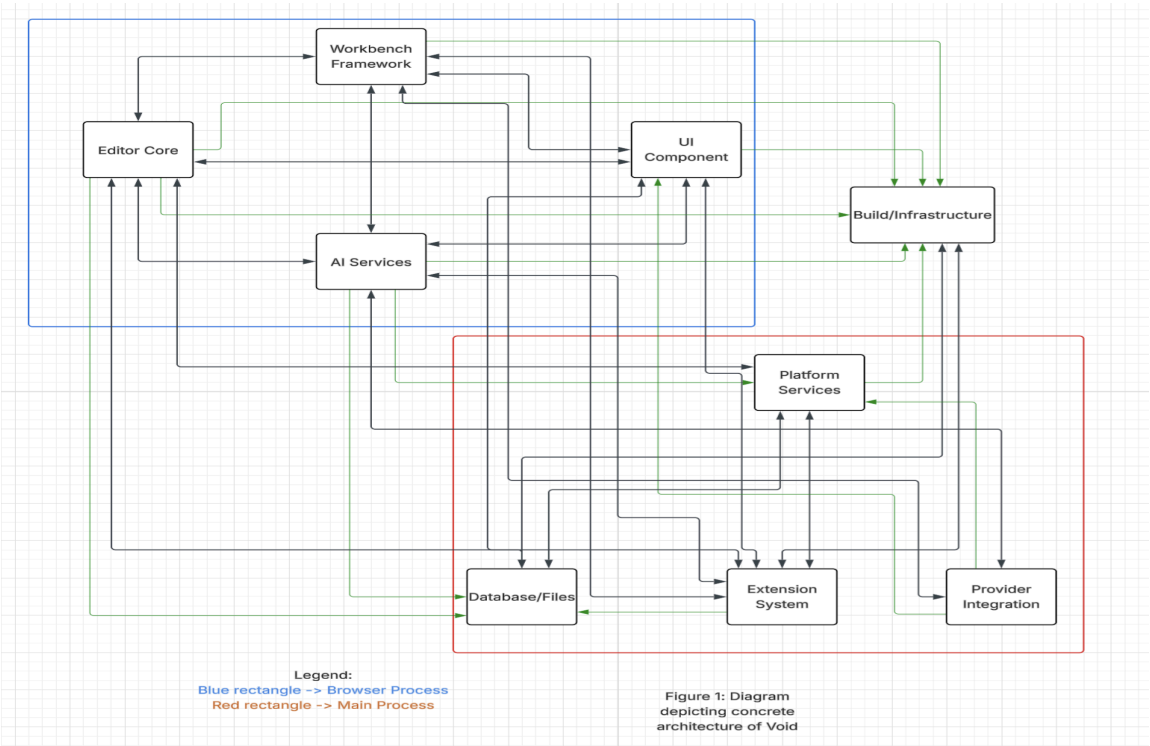



Figure 1: Diagram depicting concrete architecture of Void

---

## 4. Subsystem Analysis: Void (Second-Level)

### 4.1 Purpose and Responsibilities (Conceptual vs Concrete).

In the conceptual architecture breakdown provided through A1, the Void subsystem was described as the AI integration layer within the IDE. Its purpose was to interpret and communicate information between the user through chats, prompts, and settings with lower-level system services such as the editor, terminal, and workbench. In addition, the Void subsystem was responsible for coordinating AI services, context collection, and interaction delivery, ensuring a

seamless exchange between the developer and the AI tools provided through Void. From the breakdown provided in A1, here are 4 of the sub-components of Void we investigated:

- AI Services: managing language model requests, message formatting, and inference routing.
- Provider Integration: connecting the IDE's internal services to external AI backends.
- UI Components: responsible for user interface elements like tooltips, settings, model dropdowns, and chat box.
- Workbench and Platform Ties: integrating the AI features into VS Code's extension and workbench infrastructure.

| Sub-Component | Concrete Files | Responsibility |
|---|---|---|
| AI Service | convertToLLMMessageService.ts, chatThreadService.ts, aiRegexService.ts | Handles prompt construction, AI response evaluation, and context gathering. |
| Provider Integration | contextGatheringService.ts, fileService.ts, toolsService.ts | Bridges between IDE services, such as editor, terminal, and file system, with Void's AI runtime. |
| UI Components | void-settings-tsx/Settings.tsx, ModelDropdown.tsx, WarningBox.tsx, void-tooltip/VoidTooltip.tsx | Manages the visual presentation of AI-related UI, including settings, warnings, and in-editor tooltips. |
| Workbench Framework | void.contribution.ts, voidOnboardingService.ts, voidCMSService.ts | Registers Void as a workbench contribution, binding it to VS Code's platform APIs, telemetry, and lifecycle events. |

Through SciTools Understand, we can see how the concrete architecture of Void shares strong similarities with the conceptual architecture we proposed in A1.

**4.2 Design Patterns and Styles.**

**Layered Architecture**

The Void subsystem structure is organized in a Layered Architecture that is evident in the manner in which its files and dependencies are arranged. At the top layer, there are React-based .tsx components, including Settings.tsx, ModelDropdown.tsx, and WarningBox.tsx, which are engaged in user interaction and interface implementation. Below this, integration and service

layers in the form of files such as chatThreadService.ts, contextGatheringService.ts, and toolsService.ts are used to coordinate the flow of data and AI logic, whereas the platform layer (void.contribution.ts, voidCMSService.ts) takes care of registration in the VS Code environment. Information passes downward, where the user inputs are converted into model requests, and upwards, where responses are made to update the interface. Therefore, through the concrete architecture provided the flow of information and separation of processes provides evidence of a layered architectural style.

**Client-Server Architecture**

The Client-Server style is found in Void through both internal and external communications. Internally, the subsystem functions as a client that requests information from "server-like" services such as fileService.ts, editorService.ts, and contextGatheringService.ts, which provide contextual data from VS Code's core. Externally, the AI Service Layer acts as a network client that sends prompts and contextual data to remote large language model providers and asynchronously receives responses. This request–response flow mirrors the client–server model we outlined in the conceptual design, positioning Void as a mediator between the editor (VS Code) and remote AI inference systems. Through how data is exchanged via requests and responses, sometimes from external models, the subsystem demonstrates a client-server relation, which is further backed up by the concrete architecture that outlines how it makes requests and how they travel.

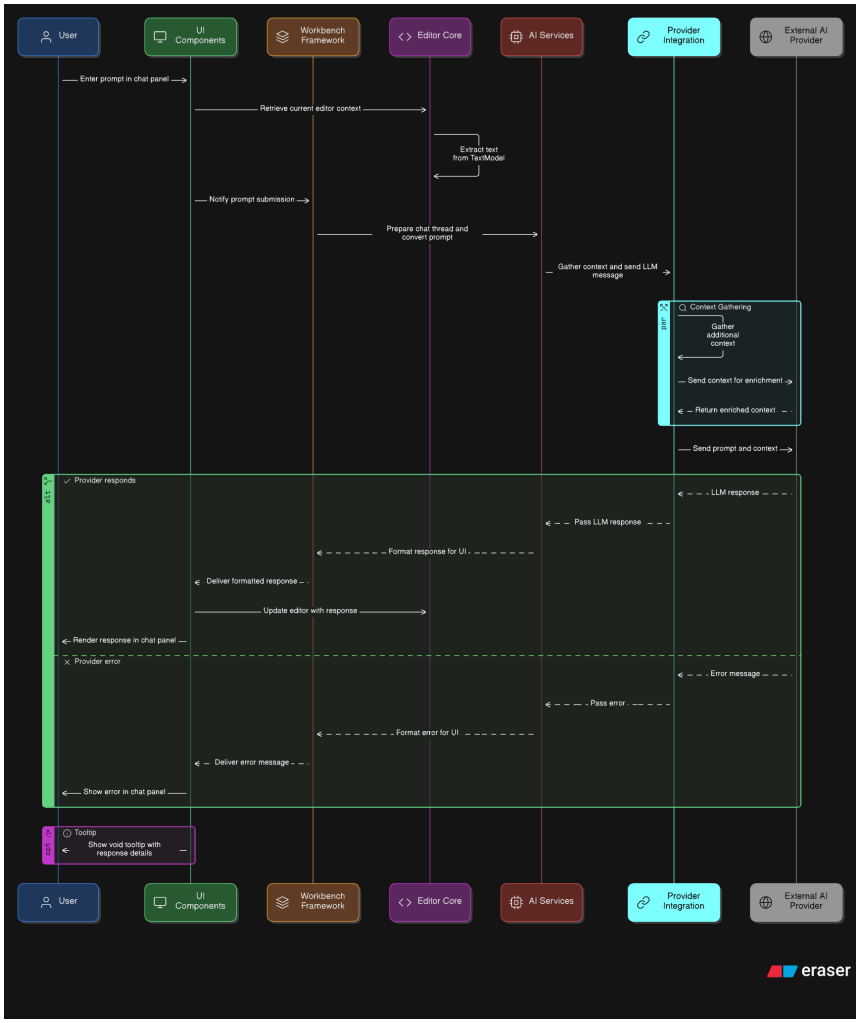
**Implicit Invocation Architecture**

The style of Implicit Invocation exists in the Void in its responsive and fragmented structure. Communication is not done with direct and synchronous calls but with emitted events and subscribed listeners between layers. As an example, UI elements, including but not limited to ModelDropdown.tsx and WarningBox.tsx, would automatically respond to updates or events emitted by background services, like chatThreadService.ts and convertToLLMMessageService.ts. The interaction with the AI is most significantly an event-driven one, as a user can provide a prompt, and the system sends an asynchronous event that calls the AI service. The request is processed by the service, and only when a response or chat message event is returned does the UI react and update. The mechanism can be considered an example of implicit invocation where modules are triggered by published events, not by explicit invocation, and it can be seen that the concrete implementation of Void is a match to the hybrid architecture of Layered, Client-Server, and Event-Based architectural composition as suggested in the conceptual model.

## 5. Behavioral Views: Two Concrete Sequence Diagrams

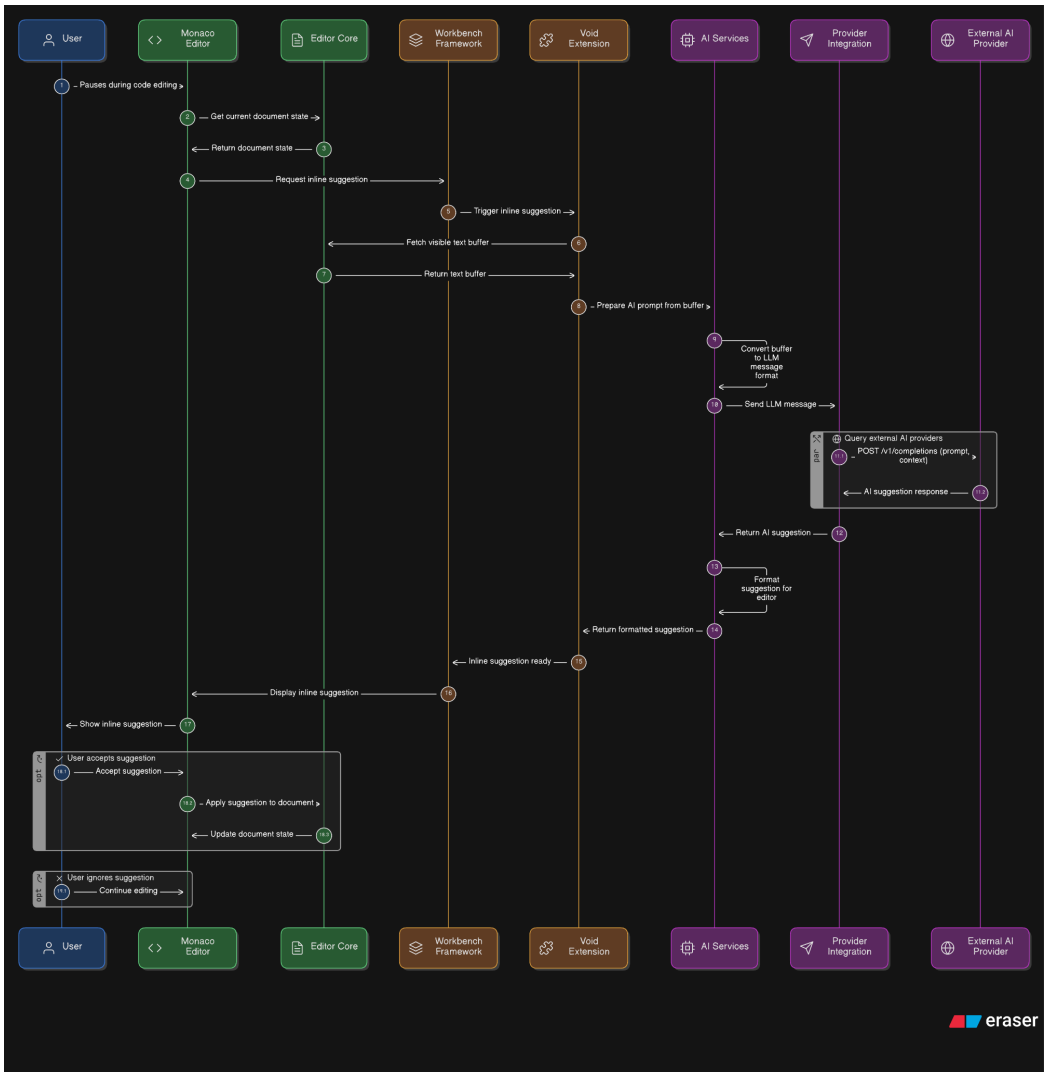
### 5.1 Use Case A: “Prompt request to response render.”

The user submits a prompt through the chat interface, which the UI Components pass to the Workbench Framework. The Workbench grabs the current editor context—active file, cursor position, selected code—from the Editor Core to give the AI proper context. It then sends both the prompt and context to AI Services, which formats everything into a structured message and updates the chat thread history. AI Services forwards this to Provider Integration, which builds the actual API request and sends it to the external AI provider. The response streams back in chunks through the same path: Provider Integration receives each chunk, passes it to AI Services to append to the response, which notifies the Workbench, which tells UI Components to render the growing answer in real time.



### 5.2 Use Case B: “Inline suggestion during edit.”

As the user types, Monaco Editor fires a content change event that flows through Editor Core to the Workbench Framework. The Void Extension listens for these changes and checks if conditions are right for a suggestion—right cursor position, enough context, user paused typing. If so, it grabs the surrounding code from Editor Core and sends it to AI Services, which formats a quick completion request with low temperature and token limits for speed. Provider Integration sends this to the AI provider and streams back the suggestion. Instead of inserting it directly, Void Extension renders it as gray "ghost text" through Editor Core. The user can accept it with Tab to make it real code, or dismiss it by continuing to type or pressing Escape.




---

## 6. Reflexion Analysis

In our conceptual architecture, the intended design of how the AI services interact with different components, such as the Workbench Framework and UI components, was meant to be that the components had to go through the AI Services component in order to interact with the Provider integration. However, in the concrete architecture, we found that the Workbench Framework and certain UI modules bypass this layer and directly reference provider-related logic through shared common services. We found this in some browser-side files like chatPanel.tsx and editCodeService.ts import utilities from sendLLMMessageService.ts and modelCapabilities.ts. Essentially, instead of AI Services being the sole connection, both the Workbench and UI components are directly communicating with the provider functionality.

Divergences were also found in the subsystem architecture. In the conceptual architecture, the Build/Infrastructure component was left out. It was thought that it was handled inside the Platform Services layer, supporting deployment and dependency management. However, we found that in the concrete architecture that was made through the understanding tool, Build/Infrastructure appeared to be a distinct top-level subsystem with its own dependencies and responsibilities. This is confirmed through the presence of a dedicated build/ directory, Electron packaging scripts (webpack.config.js, gulpfile.js), and CI pipeline in .github/workflows/build.yml. These reveal a formalised build process responsible for compiling, packaging, and distributing the Void application. Essentially, through the use of the understanding tool in making the concrete architecture, we were able to discover a new subsystem that was passed over in the conceptual architecture.

---

**7. Updated Conceptual Architecture (Post-Reflexion)**

| Conceptual Components | Roles and Rationale | Changes |
|---|---|---|
| Platform Services | Responsible for delivering core services and/or abstractions that form the foundation for all other components. | Now depends on Provider Integration |
| Editor Core | Contains the Monaco editor core, which provides text editing, syntax highlighting, language features/services, code validation, and a contribution system | Now depends on Database/Files and AI services, |
| Workbench Framework | Responsible for the main user interface (VS Code) and the | Now depends on new component |

| | | |
|---|---|---|
| | application logic behind many services in Void, | Build/Infrastructure |
| UI Components | Responsible for the main user interface that integrates the AI features of Void into VS Code's environment. Renders the chat interface and any other void specific features/menus. | Now relies on new component Build/Infrastructure |
| AI Services | Responsible for being the "brains" of the system. Allows users to interact with an AI, providing workflow and managing requests and operations. Also analyses code content, organizing chat sessions, providing real time code suggestions, processing general AI responses, managing other AI dev tools and more. | Now depends on the new component Build/Infrastructure and on Database/Files component. |
| Provider Integration | Responsible for connecting different AI providers. Meant to create an interface for the AI operations and handle multiple different API's. It supports customs endpoints, provider specific features, model capability detection, and configuration management. | Now depends on Platform services and UI components |
| Extension System | Responsible for maintaining compatibility with VSCode while adding features. Specifically, the extension system allows for configuration and viewing of extension points. | Now depends on Database/Files component |
| Database/Files | Responsible for data access operations, handling file operations, and potential storage needs. | Now depends on AI Services and Editor Core components. |

## 8. Conclusion & Lessons Learned

The recovery of the concrete architecture of the Void IDE confirmed the soundness and modularity of its layered design. At the top level, the overall structure of Electron, VS Code Core, and the Void extension reflected the expected service-oriented pattern described in our conceptual architecture. The Extension Host boundary and asynchronous messaging mechanisms were verified as essential to maintaining responsiveness and process isolation, validating the design emphasis on extensibility and safety. Within the Void subsystem, the recovered code followed a consistent layering of UI, AI Services, Provider Integration, and Workbench Ties, showing that the implementation aligns closely with the conceptual intent from Assignment 1.

The process of deriving the architecture through SciTools Understand provided valuable insight into how conceptual structures manifest in code. The tool was effective for identifying subsystem boundaries and mapping dependencies, but it required careful human interpretation to confirm relationships and avoid misreading auto-generated graphs. This demonstrated that automated analysis can accelerate discovery but cannot replace architectural reasoning and manual validation.

Throughout the project, the team gained practical experience in bridging conceptual and concrete architecture views. Redrawing diagrams, naming components consistently, and organizing tables and legends improved both readability and traceability. Comparing our A1 and A2 results reinforced the importance of documenting design intent clearly and revisiting it as systems evolve. Overall, the assignment strengthened our understanding of software architecture recovery and the discipline required to maintain structural consistency between design and implementation.

---

## 9. AI Collaboration Report

Our team mainly used OpenAI GPT-5 (October 2025), Anthropic's Claude Sonnet 4.5 (within Cursor), and eraser.io throughout this assignment. GPT-5 supported structured writing, rubric alignment, and editing consistency, while Sonnet 4.5 handled most of the file-system parsing and interpretation of data extracted from SciTools Understand and the Void GitHub repository. Compared to Assignment 1, AI tools were used slightly more extensively, as this phase required closer analysis of source code structure, dependency graphs, and subsystem mappings. These models were chosen for their strong reasoning on large code contexts, accuracy in summarizing dependencies, and reliability when interpreting source-level relationships. Other AI providers and models such as Gemini 2.5, Grok 4, and o3 were also considered and tested, but they were not selected as they seemed to be less reliable in interpreting source code structure and

dependency graphs across large contexts compared to Claude Sonnet 4.5. Lastly, eraser.io was used to aid in drafting the sequence diagrams, as in Assignment 1.

### 9.1 Tasks Assigned to the AI Teammate

**Task:** Drafting and refining report sections, improving clarity and academic phrasing
**Reason:** GPT-5 maintains a consistent structure and tone suitable for technical documentation

**Task:** Parsing and organizing repository files and dependency graphs (Cursor: Sonnet 4.5)
**Reason:** Efficient for traversing large codebases and summarizing complex folder hierarchies

**Task:** Interpreting, understanding metrics, and dependency outputs
**Reason:** Helped translate extracted data into subsystem mappings used for diagrams and tables

**Task:** Converting our use cases into diagram templates (Eraser.io)
**Reason:** Aided layout and labeling before manual refinement

### 9.2 Interaction Protocol & Prompting Strategy

Work was coordinated in shared Google Docs. Each prompt followed three stages:

1. Provide technical or contextual background (for instance, subsystem, metric, or graph).
2. Specify the desired deliverable (summary, mapping, or explanation).
3. Iterate through group feedback until agreement on accuracy and tone.

Example Prompt:

"Analyze the Understand dependency graph for LLMMessageService. Identify its direct imports and any outbound references to chatThreadService. Summarize the relationship and note unexpected dependencies."

### 9.3 Validation & Quality Control

To maintain accuracy and originality, the team followed a structured review process:

1. Cross-checked all AI-derived findings with official VS Code and Void documentation.
2. Two members manually confirmed Sonnet 4.5 outputs by inspecting the corresponding files in GitHub.
3. Verified consistent naming of classes, methods, and subsystems across diagrams and text.
4. Proofread by multiple team members for clarity, conciseness, and academic tone.

### 9.4 Quantitative Contribution

| Category | AI Contribution | Human Contribution |
|---|---|---|
| Drafting & structural organization | ~45% | 55% |
| Repository and dependency parsing (Cursor + Sonnet 4.5) | ~25% | 75% |
| Diagram preparation | ~10% | 90% |
| Editing & integration | ~10% | 90% |
| **Overall Estimated Impact** | **≈ 35%** | **≈ 65%** |

### 9.5 Reflection on Human/AI Team Dynamics

AI tools played a slightly larger role in this phase than in Assignment 1. Parsing the Void repository and Understand exports required careful inspection of dependencies and file structures, areas where Sonnet 4.5 in Cursor provided significant assistance. GPT-5 continued to ensure organization, flow, and consistency across sections. Human oversight remained essential, and some minor disagreements still occurred: several AI summaries overlooked indirect relationships or generalized dependency directions, which we corrected manually through cross-checking and repository validation. Overall, the collaboration between human analysis and AI assistance enabled deeper technical insight and more efficient synthesis of results, strengthening both the quality and accuracy of the final architectural documentation.

---

## 10. References


Dwivedi, V. (2018, April 25). *Electron: 4 Things to watch out for before you dive in*. Medium. https://medium.com/@vishaldwivedi13/electron-things-to-watch-out-for-before-you-dive-in-e1c23f77f38f

electron. (2025, February 13). *GitHub - electron/electron: :electron: Build cross-platform desktop apps with JavaScript, HTML, and CSS*. GitHub. https://github.com/electron/electron

*Extension API*. (n.d.). Code.visualstudio.com. https://code.visualstudio.com/api

Hijdra, R., Geffen, H. van , Petrescu, S., & Yarally, T. (2021). *VSCode - From Vision to Architecture - DESOSA*. Desosa.nl. https://2021.desosa.nl/projects/vscode/posts/essay2/

*Inter-process Communication (IPC)*. (2025). Chromium.org. https://www.chromium.org/developers/design-documents/inter-process-communication/

Kumar, A. (2025, March 24). *Void IDE: The Comprehensive Guide to the Open-Source Cursor Alternative*. Medium. https://medium.com/@adityakumar2001/void-ide-the-comprehensive-guide-to-the-open-source-cursor-alternative-2a6b17cae235

microsoft. (2019, November 22). *Home*. GitHub. https://github.com/microsoft/vscode/wiki
Pareles, A., & Pareles, M. (2025). *Overview | voideditor/void | Zread*. Zread. https://zread.ai/voideditor/void/9-architecture-overview
Perkaz, A. (2021, July 14). *Advanced Electron.js architecture*. LogRocket Blog. https://blog.logrocket.com/advanced-electron-js-architecture/
Rascia, T., & Nolan, T. (2021, September 27). *Understanding the Event Loop, Callbacks, Promises, and Async/Await in JavaScript | DigitalOcean*. Www.digitalocean.com. https://www.digitalocean.com/community/tutorials/understanding-the-event-loop-callbacks-promises-and-async-await-in-javascript
voideditor. (2024). *GitHub - voideditor/void*. GitHub. https://github.com/voideditor/void/

---

**Appendix A. Data Dictionary / Glossary**

- LLM: Large Language Model
- LSP: Language Server Protocol
- Extension Host: Process running extensions in isolation
- Renderer: UI process handling user input/output
- Main Process: Central Electron process managing windows and IPC
- Void UI: Frontend interface of Void (SidebarChat.tsx)

**Appendix B. Naming Conventions**

- camelCase
- UpperCamelCase
- snake_case
- UPPER_SNAKE_CASE

**Appendix C. Abbreviations**

- API: Application Programming Interface
- UI: User Interface
- AI: Artificial Intelligence
- IDE: Integrated Development Environment
- VS Code: Visual Studio Code
- IPC: Inter-Process Communication
- OS: Operating System
- CI/CD: Continuous Integration, Continuous Deployment