

A1: Conceptual Architecture of Void

CISC 322

October 10th, 2025

Authors:

Matt Dobaj, Shaun Thomas, Babinson Batala,
Esmé Mirchandani, Sandy Mourad, Nihal Kodukula

1 - Abstract

In this report, we investigate Void's conceptual architecture whilst noting key features and requirements. Because it is a fork of Microsoft's Visual Studio Code (VS Code), we determined whether or not Void's architecture style was similar to VS Code. Both projects are built using Electron, a framework for building desktop applications in javascript and typescript. It splits an app into a client, called a browser process, and a server, called the main process.

We discovered that Void does indeed have a very similar architectural style to VS Code, with its main architectural styles being Layered and Client-Server, through its logical hierarchy and multiprocess architecture, respectively. It uses all of the components found in VS Code, but adds / modifies 4 of its own, being the Void UI component, the VS Code editor component, the Model Service component, and a Void Platform component. These components are essential for Void to perform its tasks like inline code completion and a chat-based code generation interface.

The application is highly extendable thanks to its layered and client-server architecture and via its extensive extension system it inherited from VS Code. It is also able to avoid common issues with response time, a key performance requirement for the project, by using Electron's interprocess communication protocol instead of using the network whenever possible.

Void takes information from the user in the form of user prompts, inline completions, and editor commands, and returns information from its external AI Provider sources. It also reads files from the user's system for context generation.

Two key use cases were explored in the context of the project: inline code completion when the user is typing in the code editor, and a chat-based code generation prompt. Both use cases blend context from the user's project files and the prompt / written code to better inform the external AI providers in their responses.

2 - Introduction and Overview

AI is increasingly being built into modern developer tools, allowing programmers to use natural language, generate code automatically, and increase efficiency by accelerating development time and project completion. Void represents a new type of code editor that combines these AI capabilities, such as inline code completion, natural language prompts, and automated refining, all within a familiar coding environment.

Void is a fork of Microsoft's open-source VS Code and is built on the Electron framework, which runs applications using two main processes: a Browser Process (client) for the user interface and a Main Process (server) for the system logic and file operations. Because of this foundation, Void inherits much of VS Code's structure but extends it with new AI-focused components.

The purpose of this report is to analyze Void's conceptual software architecture, specifically how it introduced AI-driven functionality while maintaining the stability and extensibility of VSCode. The analysis focuses on several key areas:

- The main architectural components and how they interact.
- The use of Layered and Client-Server architectural styles to organize the system.
- How Electron's multi-process model supports responsiveness and scalability.
- The external interfaces that connect Void to users, extensions, and AI providers.
- Example use cases that demonstrate how the system behaves at runtime.

By examining these aspects, the report aims to show how Void achieves important quality attributes such as modularity, extensibility, responsiveness, and maintainability.

In summary, Void illustrates how AI-driven systems can evolve from existing architectures instead of being built from scratch. Its design combines innovative AI integration with conventional IDE principles to create a development environment that is adaptable, scalable, and built for expansion.

3 - Architecture

As previously mentioned, the Void Editor is extended from VS Code, Microsoft's open source code editor. VS Code is itself based on Electron, a framework that allows for cross-platform compatibility using its interpreter style architecture to execute javascript code.

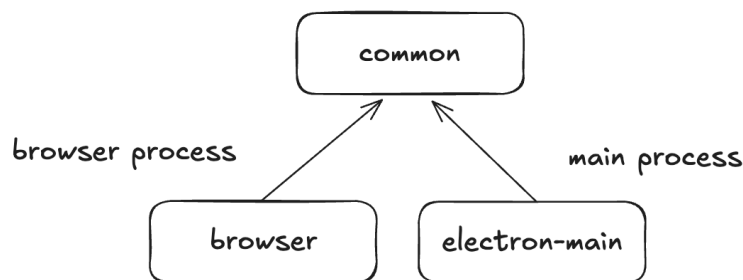


Figure 1: A diagram showing Electron's multi-process architecture [1].

Electron is made of two processes, a main process in the electron-main component and a browser process in the browser component. The browser acts as a client to the main process' server. These two processes are dependent on a common component that provides shared utilities either can use. The browser process contains all code relating to an app's user interface, with the main process containing all logic related to the application's lifecycle and interfacing with the host operating system [2].

VS Code follows a layered architecture style, segmenting its code into 4 fundamental components.

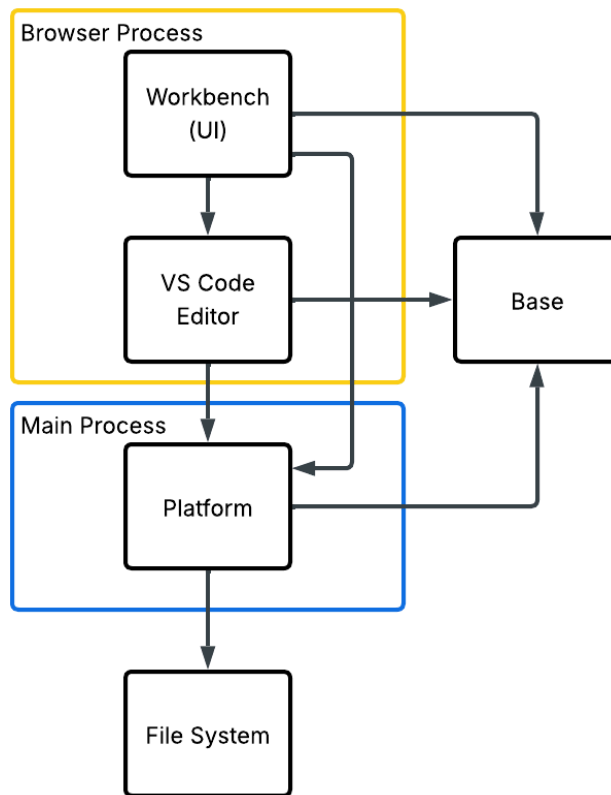


Figure 2: A diagram showing VS Code's conceptual architecture.

The workbench component, at the highest level of abstraction, contains the vast majority of the user interface elements of VS Code. This includes hosting the code editor, status bar, search bar, and elements of the file explorer. It is dependent on the VS Code Editor component, which contains all logic relating to the Monaco editor that is at the heart of VS Code [3]. These two layers are contained in Electron's browser process, shown in yellow.

The editor and workbench both interface with the platform component, which is the interface for all operating system services like reading and writing to the file system [3]. This component is contained in Electron's main process so that it has access to these special privileges, which is shown in blue.

All three layers of VS Code's logic are dependent on the Core component, which contains common utilities and building blocks that can be used anywhere across the app. Void provides several components that extend upon these VS Code building blocks.

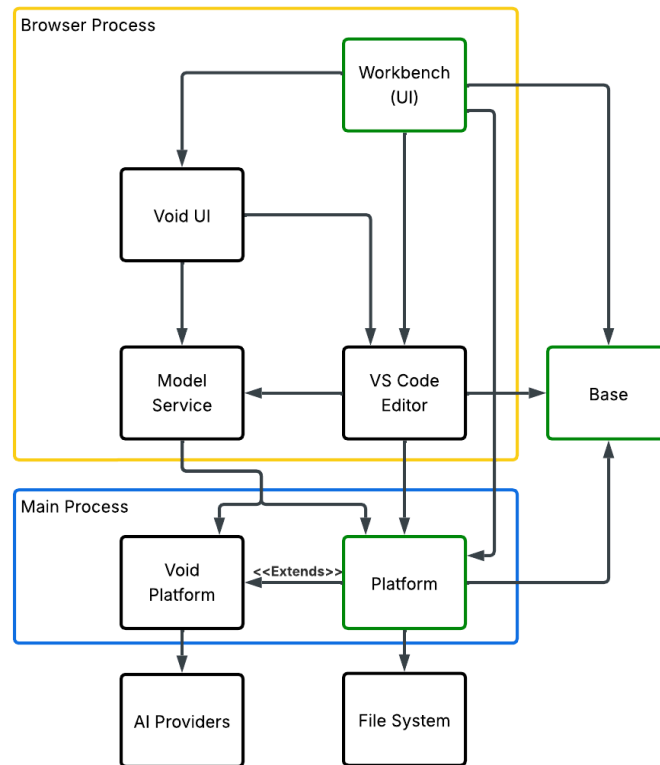


Figure 3: A diagram showing Void’s conceptual architecture.

3.1 Void’s Conceptual Components

Many of Void’s core components are dependent on their VS Code counterparts, shown in green. The majority of Void’s architecture lies in the browser layer of Electron’s framework, and is split into 4 key components.

3.1.1 Void UI

The majority of Void lies in its UI component, which VS Code’s Workbench component is dependent on for all of the Void elements the user directly uses. This includes inline code completion, a prompt page, the void settings panel, and model selection. This component is dependent on the modified VS Code Editor component to provide inline code suggestions. It is also dependent on Void’s Model Service component to interface with the Electron main process and prompt the user’s selected LLM. The Void UI also has an entry point for extensions.

3.1.2 VS Code Editor

Void also has a modified version of the VS Code editor. Like in VS Code, the main purpose of this component is to contain all of the logic for the Monaco code editor that Void uses. The Void team has extended this component to communicate with the Model Service component during the edit process to trigger the AI-based code completion functionality. This component depends on the VS Code platform in order to write files and trigger commands. It also depends on Void’s Model Service to perform the aforementioned code completion.

3.1.3 Model Service

The Model Service is responsible for formatting and preparing an AI message from the Void UI before being sent to Void's Platform component and ultimately the AI Provider. It is dependent on VS Code's core platform for obtaining important context to provide with the user's prompt [4], and Void's extension to the platform in order to prompt the AI providers.

While the Model Service is likely located in the browser process, there is conflicting information in Void's documentation. One source [5] places it in the browser process and another places it in the main process [6]. The component's location cannot be determined by documentation alone and may move once we extrapolate the concrete architecture from Void's source code.

3.1.4 Void Platform

The Void Platform component is responsible for directly prompting the various AI providers that Void is compatible with. It extends VS Code's native platform, and serves as the component where application commands execute, and has an entry point for extensions [7].

3.2 Extensibility

VS Code, which Void is based upon, has a highly extendable architecture. It has an extensive extension API which allows any of its primary components (being the Workbench, Editor, and Platform) to be extended with new functionality. This has allowed VS Code to thrive as a customizable development platform.

Given Void's conceptual architecture, it should also be easily extendable. Similarly to VS Code, Void provides an extension API which allows any of its core components (being the Void UI, Model Service, and Void Platform) to be extended by any third parties.

Void's components are also well laid out thanks to its layers of abstraction. If a new method of prompting was to be implemented, for example, only the Void UI component would likely have to be modified. In the rare instance where the Model Service would also need to be modified to support this change, it would be relatively minor. If support for a new AI model was to be added to Void, it would have the majority of its changes in the Void Platform component, with minor prompt changes and UI tweaks necessary in the Model Service and UI components, respectively.

3.3 Performance Requirements

Void does not have extremely high performance requirements as it is not a realtime application, however it does require a low response time for most of its functionality. The code editor, for example, needs to have its changes reflected in the source file relatively quickly so that development environments like development servers can update quickly. For Void's AI code completion suggestions, the response time should be low enough to prevent a mismatch between what the user is adding in realtime and what the editor suggests. This could be function declarations, variable assignments, or other tasks that can be done by the user relatively quickly.

In contrast, the individual code prompt does not have these low response time requirements. The user simply inserts a prompt and waits for a response, and there is no parallel task completion like in the inline suggestion use case. While a low response time would be valued in this use case, it is not as critical.

3.4 Concurrency

Thanks to Void's multiprocess architecture, the browser and main process are able to run concurrently. For example, the user is able to navigate through Void's UI and edit code while the main process is waiting for a response from an external AI Provider. This ensures that the application remains smooth for the user during heavy server tasks like AI operations and file system read and writes.

3.5 Architectural Style

Void uses a variety of complementary architectural styles in its design. The primary architectural style present is likely layered, with elements of a client-server style as well.

3.5.1 Layered Architecture

Much like VS Code, Void uses a few key components of ascending abstraction to perform its tasks. Each component provides its services to the component above it, and uses the services exported from the component below it. For example, the Void UI should only use services from the Model Service component and the Model Service component should only use services from the Void Platform below it.

In addition to their ascending abstraction, these components are also laid out logically. The Void UI component only handles user interactions, the Model Service component only handles prompt formatting and processing, and the Void platform only handles external services like interfacing with AI providers. This logical hierarchy suggests that these components make up layers in a layered system.

Using a layered style is beneficial in terms of enhancement and reuse, where changes to a layer only require at most 2 layers to be modified, and different implementations of the same layer can be interchanged. If the Void UI were to change javascript frameworks, for example, this could be done largely without modifying the layers below it.

This style could also have potential drawbacks, where performance requirements in higher level components could cause them to require low level implementations. In this application, the higher level components such as the Void UI and the Model Service do not have strict performance requirements outside of maintaining a relatively low response time.

3.5.2 Why not Pipe and Filter?

When first outlining the components for Void, the layout suggested it may have implemented a pipe and filter style, where a prompt undergoes a series of processing steps before being finally passed to the AI provider. While from a simplistic standpoint Void would match the pipe and filter style, it does not align with any of pipe and filters requirements.

The Pipe and Filter style architecture requires that filters don't share their state or know the identities of each other, and requires filters to be able to move around within the architecture. Void's components have both a set order that they must go in and require the knowledge of each other in order to interface with their methods correctly. In conclusion, Void's components do not perform independent computations as the Pipe and Filter style requires, instead relying on a hierarchically abstracted structure.

3.5.2 Client - Server

Void, and by extension VS Code, is also logically split into two aforementioned processes, Electron's browser process and main process. The browser process requests services that the main process provides, be it file system access, running VS Code commands, or initializing extensions. The browser process acts as the client, requesting services from the main process, which is the server.

The UI and Editor components from either project are not required to run in the browser process of Electron. They can also be run in a browser tab or on separate machines for remote development purposes. When run as the Void desktop application, the two processes communicate using Electron's interprocess communication (IPC) but may use other protocols like SSH in remote instances [8]. As previously explored, Void is also very extendable through its extension APIs, and the separation between UI and Platform components mean that changes do not require large modifications across multiple components. Overall, Void's highly portable and extensible nature aligns with client-server's properties.

The client server architecture style also has a few disadvantages regarding performance. Most notably, the application's performance is dependent on the network performance, meaning that if a network is slow or inconsistent, the response time between UI actions and displaying results may be affected during remote development. Void, and by extension VS Code, only uses the network to communicate between its two processes when necessary and typically uses the aforementioned Electron IPC to communicate between processes. The Electron IPC is not dependent on network performance as it does not use the network to communicate. Void's architecture largely avoids the drawbacks of client-server by only relying on the network when absolutely necessary.

4 - External Interfaces

Void exchanges 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.

4.1 Graphical User Interfaces (GUIs)

Void's interactive front end is made up of the User Interaction Layer, which is implemented in Electron's browser process. It abstracts presentation issues from system logic by using modular interface components to record user input and display AI-generated feedback [5].

Electron's IPC protocol is used to transmit user commands and interactions to backend services, ensuring a distinct division between the interface and the main application processes.

Information sent to the system:

- User prompts, inline completions, and editor commands
- File selections, cursor positions, and active project metadata from the Editor Core
- Config Parameters such as model selection, temperature, and context length

Information received from the system:

- Streamed AI responses (completions, explanations, or refactors) from the Void Process
- Contextual updates from the Platform

4.2 AI Provider Interfaces

The main process of Electron contains the Void process, which controls all interactions with outside large language model (LLM) providers [5]. In order to ensure seamless integration between Void and different AI backends, it is in charge of creating requests, sending prompts, and receiving model responses. The layer preserves consistent behavior across inference workflows while abstracting the complexity of various providers.

Information sent to AI providers:

- Structured prompts composed by the Bridge and Context Components, including user input, code context, and metadata
- Model configuration parameters (provider ID, token limits, temperature, top-p)
- Session identifiers for continuity and state management

Information received from providers

- Streamed model responses (tokens, code edits, or text)
- Inference metadata such as latency, completion status, and error messages

4.3 File and Platform Interfaces

The Void Platform, extending VS Code's Platform layer, handles all operating system and file-level communication. Running within the main process, it provides access to the file system, workspace, and system resources while maintaining compatibility with VS Code's APIs.

Information Transmitted

- File and workspace operations (open, save, rename, delete) from the Editor Core
- Project metadata, workspace context, and configuration data sent to the Bridge and Context Components.
- Session logs and cached model responses stored by the Void Platform component.

These exchanges use Electron's IPC protocol locally or SSH during remote development [8].

5 - Use Cases

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

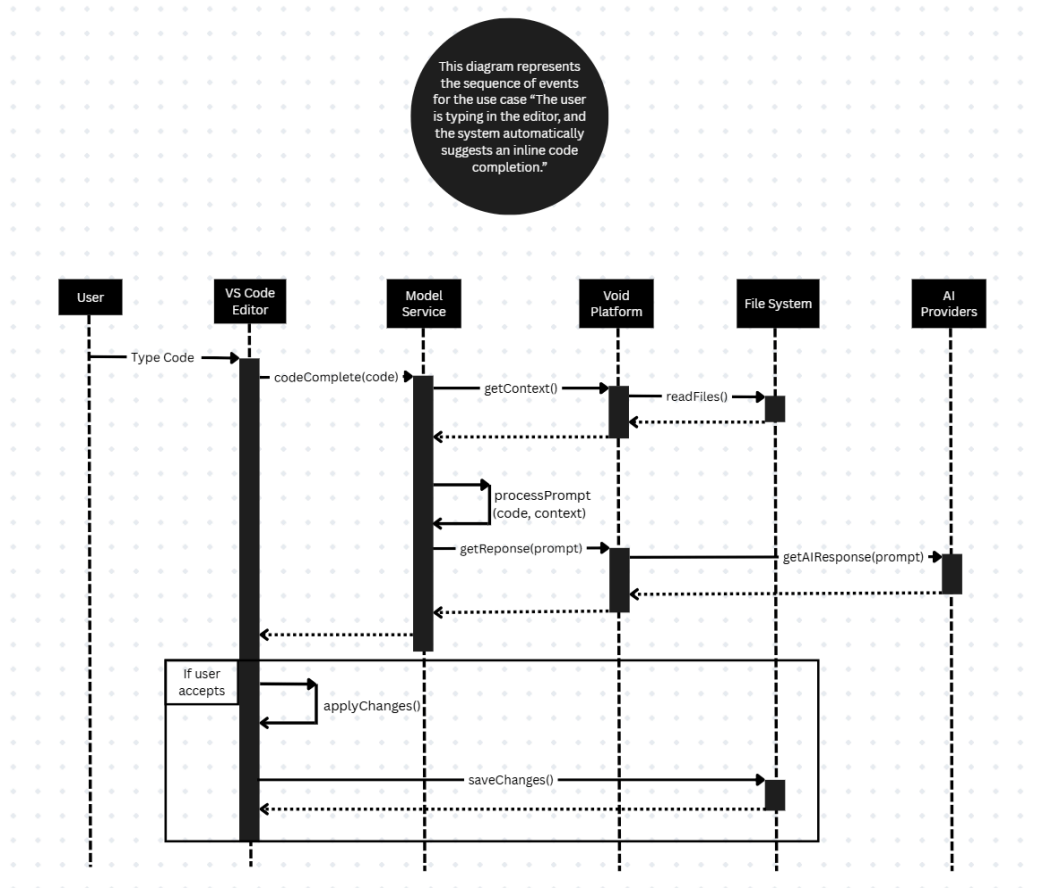


Figure 4: Sequence diagram showing the events of use case 1.

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 multiple architectural layers. It follows this sequence of events:

1. The User types code in the VS Code editor, which triggers a `codeComplete()` function.
2. The Void Platform queries the File System for project context.
3. The Model Service processes both the user’s input and the retrieved context, forming a prompt.
4. The AI Provider receives this prompt and generates a completion suggestion.
5. The Model Service sends the response back to the Void Platform and then the Model Service, which inserts the suggestion inline in the editor.
6. If the user accepts the suggestion, the editor applies and then saves the changes in the File System.

Use Case 2: “The user types a natural-language request in the chat window, and the AI generates new code, which the user can apply.”

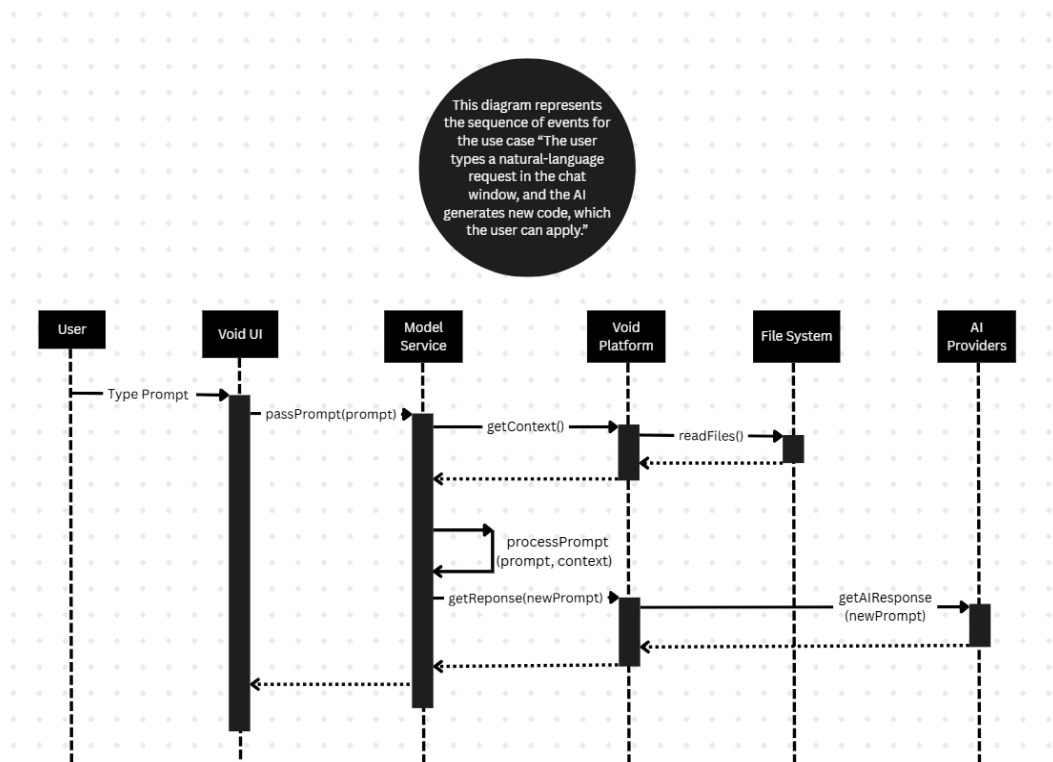


Figure 5: Sequence diagram showing the events of use case 2.

The user interacts with the AI assistant via a chat interface to generate or modify code. In this use case, the user submits a natural-language request and the system produces new code that can be previewed and applied. It follows this sequence of events:

1. The User types a natural-language prompt into the Void UI chat window.
2. The Model Service receives the prompt and asks the Void Platform for relevant code context.
3. The Void Platform retrieves necessary files from the File System and sends them back.
4. The Model Service processes the combined data (prompt + context) and forwards it to the AI Provider.
5. The AI Provider returns the generated code suggestions.
6. The Void UI displays these suggestions to the User, which can then be applied.

6 - 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.

Extension API: Programming interface that allows developers to create plugins or extensions for Void and VS Code.

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.

Monaco Editor: The open-source code editor used in VS Code and Void for syntax highlighting and editing.

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

7 - Naming Conventions

We intended to name our architectural components in a similar fashion to the components from VS Code's architectural overview [3] in order to have an easy to follow and consistent naming scheme when explaining how Void's components interact with VS Code components. Functions in our use case diagrams were written with camelCase and aim to clearly state the purpose of each function call. For example, the function `getContext()` will aim to get the relevant code context from the file system. If it is not clear, the sequence of events is also listed below each diagram. We use two abbreviations consistently but they are fairly straightforward: UI, meaning the user interface, and AI, meaning artificial intelligence.

8 - Conclusion

Through this analysis, we demonstrated that Void's conceptual architecture is a robust evolution of Visual Studio Code's design, combining layered and client-server architectural styles to integrate advanced AI capabilities without compromising modularity or performance.

Void has introduced new subsystems such as the Void UI, Model Service, and Void Platform largely by extending existing VS Code components. These components help enable real-time inline code completion, chat-based code generation and a very seamless interaction with external AI providers.

Clear, logical, separation of responsibilities allows for minimal impact when extending or modifying components. The performance impact of client-server interactions is largely mitigated by leveraging Electron's IPC protocol. The application also supports third-party extensions and can easily extend to add future AI model integrations.

In summary, Void demonstrates how AI-enhances developer tools can be built on stable, well understood architectural foundations. With extensible design, layered abstraction, and strong integration points make it well positioned to support future growth in AI functionality, improved performance, and greater adaptability to emerging developer needs.

9 - Lessons Learned

Our team learned a lot about communication and teamwork in software architecture projects while working on this report. We realized that early and consistent communication would have made the process more seamless by allowing all team members to collaborate efficiently.

We realized that because the majority of the report relies on having a complete architecture finalized, we should work collectively as a team to ideate and later finalize this architecture so that the rest of the report can be extended from an agreed upon and solid foundation. This would allow all team members to have a shared understanding of the architecture and would lead to less information conflicts in individual work. Overall, we realized that some situations do not require task delegation but rather task collaboration.

Next time we will schedule more frequent but shorter check-ins to coordinate our progress and make sure the tone and content of each section flow together. Additionally, we will work on the report more sequentially, with all group members targeting the same section together simultaneously. All things considered, this experience has improved our ability to work together and helped us better understand how to strike a balance between consistent group output and individual contributions. We hope that lessons learned from this report will make a significant impact in the reports to follow.

10 - References

[1]

A. Pareles, “Void Codebase Guide.” [Online]. Available:
https://github.com/voideditor/void/blob/main/VOID_CODEBASE_GUIDE.md

[2]

“Process Model.” [Online]. Available:
<https://www.electronjs.org/docs/latest/tutorial/process-model>

[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>

[5]

“Architecture Overview.” [Online]. Available:
<https://zread.ai/voideditor/void/9-architecture-overview>

[6]

“Remote Development.” [Online]. Available:
<https://zread.ai/voideditor/void/14-remote-development>

[7]

D. Schipper, J. Faber, R. Proost, and W. Spaargaren, “Visual Studio Code.” [Online]. Available:
<https://delftswa.gitbooks.io/desosa-2017/content/vscode/chapter.html>

[8]

B. Pasero, “Migrating VS Code to Process Sandboxing.” [Online]. Available:
<https://code.visualstudio.com/blogs/2022/11/28/vscode-sandbox>

11 - AI Report

For this deliverable, our team used ChatGPT (GPT 5, August 2025 version) as a virtual AI collaborator primarily to assist with verification, clarification, and formatting. We had already completed our architectural design for the Void IDE before engaging the AI. ChatGPT was selected after comparing it with other models such as Claude and Gemini, based on its strong natural language understanding, consistent responsiveness to technical prompts, and ability to generate clear and structured explanations that supported our manual verification process.

Unlike other tools, ChatGPT demonstrated reliable performance when reviewing layered and client server architectures, clarifying subsystem responsibilities, and cross checking terminology for technical accuracy. We did not rely on the AI for original architectural decisions; rather, we used it to confirm our human derived conclusions and improve the clarity of our documentation.

We assigned only verification and support tasks to ChatGPT. These tasks were chosen to improve our final report preparation while keeping all critical analysis, architectural reasoning, and final decision making strictly human made. The tasks asked of ChatGPT included:

1. Term verification: Ensuring consistent use of terms (e.g., client server, publish subscribe) across the report.
2. Concept validation: Verifying our chosen architectural style (Layered + Client Server) to confirm it aligned with software architecture principles.
3. Clarity and structure checking: Suggesting ways to make technical descriptions more concise and readable.
4. Formatting assistance: Helping refine report structure and phrasing for academic readability.
5. Technical cross checking: Providing short summaries of mechanisms such as VS Code's client server model, which we then fully cross checked using official documentation.

For example, after deciding on our architectural style (Layered + Client Server), we used ChatGPT to verify our decision. The model suggested that an alternative (Microkernel) could also be suitable, but after all of us independently reviewing its feedback and researching the Microkernel style, we confirmed that our original decision remained the most appropriate for the project.

We made sure our team maintained a structured and transparent workflow for all AI interactions.

1. Context first prompting: Prompts were crafted only after internal team discussions, ensuring that the AI had full context about what was being verified or clarified.
2. Iterative review: Initial outputs were reviewed and refined by the full team to ensure they met the report's academic expectations.

3. Feedback incorporation: When AI outputs were unclear or tangential, we further adjusted our prompts or for the most part, dismissed the outputs altogether.

Because AI generated material can contain inaccuracies, we followed strict validation steps before including any information in our report:

1. Source verification: Every AI suggested claim was verified against official documentation, such as Electron's process model and VS Code's architecture guides.
2. Peer review: At least two team members reviewed each AI assisted section together for clarity, accuracy, and alignment with course expectations.
3. Cross referencing: AI generated text was compared with primary source materials and our class notes before being incorporated.

In our group, the AI member served only as a supplementary reviewer, not as an author or decision maker, to maintain the academic integrity of our work.

We estimate that ChatGPT contributed approximately 10% of the overall deliverable, mainly through verification, phrasing refinement, and structural editing. All the technical reasoning, architecture selection, and analysis were conducted manually by the team.

Integrating ChatGPT as a verification partner streamlined our workflow by helping us double check concepts and maintain report consistency. It reduced time spent on phrasing and surface level edits (missed punctuation, etc.), allowing us to focus more on analysis and drawing diagrams.

However, effective use required careful prompting and continuous reviews. The model occasionally produced basic general explanations or suggested alternative architectures that did not align with our team's judgement, underscoring the need for human judgment.

Overall, the experience reinforced that AI tools are most effective when used as assistants instead of decision makers.