Queen's University - Fall 2025

CISC 322

Assignment 1

Conceptual Architecture of Void

Group 7: The Trailblazers

Zijie Gan 21ZG6@queensu.ca

Nathan Daneliak 22TMX2@queensu.ca

Jinpeng Deng 22SS117@queensu.ca

Jeff Hong 22RQ20@queensu.ca

Emily Cheng 22TWS1@queensu.ca

Zipeng Chen 23QH10@queensu.ca

Abstract

Throughout its start in October 2024, Void has had four major upgrades. Void's conceptual architecture displays traits that can be tied to both layered and microkernel architecture styles. The major subsystems we found within Void are voidSettingServices, the LLM Messaging Pipeline, editCodeServices, and DiffZones. Void's dataflow when a developer changes their code is handled by an orchestrator to determine which tasks are important and pass through possible changes suggested by the AI. Concurrency is used within the system to help support stability, mainly through a multi-process model from its Electron framework. Two use cases are explored to understand how Void's conceptual architecture handles common requests. These two use cases are communicating with an AI API through a chat function and receiving code suggestions from an AI model. Finally, we discuss the lessons learnt through the writing of this report, and how it would be done differently in the future.

Introduction and Overview

Given the push for AI implementation in many industries in and out of the tech industry, the Void IDE is a good choice for analysis to get a deeper understanding of conceptual software architecture while keeping up with the direction of the modern tech world. After this short introduction paragraph outlining the structure and findings of the report, a quick overview of Void's functionality will be provided. Throughout this report, we start by exploring the evolution of the system and the numerous updates the Void IDE has gone through to reach its current state. In our research, we found that it went through 4 major updates after debuting in October 2024. Following is the high-level overview that goes through the outline of the architecture and how the different components interact. From analyzing the components and their interactions, we can derive the various architectural styles that make up the conceptual architecture of Void, settling on the main ones of microkernel and layered architecture. Afterwards, we analyzed how the data flows between the Void IDE and the user's chosen API and back to the IDE to respond. We found that most of this data flow was handled by the orchestrator, a key component of the system. Next was the developer division of responsibilities, where we looked at how the developers involved with the product participated throughout the life of Void. Concurrency was next, where we found that most of the concurrency through the Void IDE was due to it being built on top of VS Code, which inherited it from the Electron foundation. Finally, we analyzed the external interfaces of the IDE and two use cases to explore how the components work together.

System Functionality

Void editor is an open-source IDE built on the popular VS Code. Void allows users to code in a familiar environment with features they are used to, while combining the benefits of AI-assisted development. Void accomplishes this by integrating direct connections to any LLM the user wishes to use, whether it is locally run or from a provider [10]. This direct connection is better for users' privacy, as their data does not have to go through a private backend before reaching the AI model. With the integration of AI, Void can implement many functionalities that are beneficial to the development workflow of any project.

Autocomplete

Autocomplete is a feature that integrates AI seamlessly into a normal coding workflow [1]. Just by pressing tab, the user can accept a suggestion from the AI model that supports autocompletion, also called Fill-In-The-Middle (FIM), which uses the context of surrounding code to make a prediction of what comes next.

Quick Edit

Quick edit is a feature which allows users to highlight a section of code and ask or tell the AI model what should be done with that section [1]. This feature is more advanced than autocomplete since it has the ability to provide suggestions for additional code, refactoring code, optimizations, and error detection. The user then has the option to accept or reject the entire suggestion or just sections of the suggestion.

Chat

Void also provides a chat panel where the user can chat with the AI using one of three modes: Normal, Gather, or Agent [11]. Normal mode allows the user to chat with their model as they typically would, replying with suggestions or explanations depending on the request. Gather mode lets the model read the project files to give the AI context to better answer any questions about the project itself. Agent mode allows the AI to do nearly whatever the user themselves can do, like editing, creating, and deleting files, as well as terminal access and the ability to use built-in tools using model context protocol (MCP).

Architecture

System Evolution

Void debuted in October 2024. As an early version, it offered only basic commands like in-editor streaming of large language models, a customizable interface, history tracking, and slow application.

The beta version was released in January 2025, integrating Void into the VS Code repository and no longer providing the extension API. This update also introduced quick editing, auto-completion, a streaming interface with accept/reject controls, automatic Ollama detection, and a settings page for switching providers. That same month, after completing some bug fixes and algorithm updates, support for DeepSeek was added, and chat performance improved. By March 2025, Void launched Agent, Gather, and Chat modes, encompassing, but not limited to, file search, targeted editing, and terminal interaction. Auto-completion expanded to include more models, they introduced a quick application mechanism, and compatibility was extended to providers such as Claude and Gemini.

Through April and May 2025, the checkpoint feature launched, allowing developers to revisit historical edits. Meanwhile, upgraded tools improved application usability, and SSH/WSL support expanded deployment options. Automatic updates, streamlined onboarding processes, and Linux support further lowered the barrier to entry, while configuration sliders gave users greater flexibility in adjusting Void.

The latest update in June 2025 introduced MCP support, AI-powered commit message generation, and a visual diff view, replacing the plain text editing mode. Further expanded model support includes Claude 4, Azure, and Ollama, significantly enhancing the system's integration with external providers. Void appears to be designed as an open-source alternative, primarily aiming to address the privacy concerns, technical lock-in, and limited flexibility often associated with closed-source AI programming assistants. The project supports both local and remote models, offering developers the choice of keeping data locally on the device or leveraging external services. [10]

In this context, its design clearly prioritizes extensibility. New service providers and models can be integrated via adapters without requiring major system modifications. Notably, Void emphasizes control and transparency: features such as accept/reject buttons, visual diff comparisons, and checkpoints enable a human review process for suggestions before implementation. Thus, the Void model aims to provide developers with an AI-assisted development environment balancing flexibility, privacy, and code control.

High-Level Overview

Void is an AI-assisted code editor that builds upon a multi-layered foundation composed of Visual Studio Code and its low-level framework Electron. It is incorporated into the development environment following a layered, microkernel style. Like large “building blocks”, Electron provides the base desktop environment, VS Code is the extension framework, and Void introduces an intelligent orchestration layer on top. Each part of this code editor interacts to make features like tab autocomplete, inline editing, and custom LLMs possible [1]. From this high-level overview, one must understand its underlying components to explain exactly how Void integrates AI-assisted development into an existing IDE architecture.

Platform Layers and Component Interactions

Electron provides the framework foundation that allows VS Code and, by extension, Void, to function as a cross-platform desktop IDE. This free software is written in JavaScript and widely used for building cross-platform desktop applications, including VS Code [2]. No matter the OS, Electron supports the following Non-Functional Requirements:

- Portability/Reusability for developers and designers to express their interface vision in desktop UIs that work consistently across multiple platforms and operating systems.
- Interoperability through web technologies, as almost every platform supports the collaboration and exchange of information.
- Which corresponds to availability and accessibility in how Electron provides ample access to documentation resources [4 Why Electron].

Electron bundles the latest version of Chromium, V8, and Node.js directly into its application binary to ensure optimal security and performance while giving the developer greater control. Its Multi-Process Architecture includes mechanisms that also optimize and improve overall reliability. [4 Process Model].

- **Process isolation** means that each window or extension host runs in its own renderer process so if one crashes, then the rest of the app survives just fine.
- Because VS Code is built on Electron, its **Auto-Recovery** feature can automatically restart a failed extension host and preserve user sessions.

- With **Chromium Sandboxing**, renderer processes have limited OS access which reduces the chance that UI errors cause system crashes.
- **Inter-Process Communication** is used for message passing instead of shared memory to prevent corruption between modules.

By separating application logic from user interface rendering, the main process can be a central controller responsible for file operations, tabs, and OS interactions. Then, each window runs in its own renderer process powered by Chromium, which executes front-end code and requires IPC channels to communicate asynchronously [2]. Because VS Code and its user interface is built on Electron, this text editor adopts the same reliable system features and file system interactions. It can embed complex features like live code previews and multiple editor windows without risking a complete system crash if one fails. This same separation also allows VS Code to incorporate Extension Host processes dedicated to third-party modules without compromising the stability of its core editor [7].

VS Code Extension Host Process

Microkernel Architecture isolates a system's core services from optional components that act as separate modules to extend and enhance overall functionality. It is made of a core layer and an application microkernel from Electron to provide basic text editing capabilities, syntax highlighting, file management, debugging, and a framework for installing marketplace extensions [13]. This style is also known as the Plug-In Architecture because it provides the most essential services and fundamental aspects of a system while allowing plug-in modules to add new features. VS Code extensions are smaller code snippets designed to collaborate with other extensions and be set up independently by users without significantly impacting the core system.

Compared to a Layered Architecture, extensions make development simpler and efficient by isolating functionality, which in turn improves availability and reduces downtime [13]. It is also more testable because each part can be individually debugged. In applications not requiring high performance, this architecture performs best by allowing only necessary third-party customizations, even as unplanned integrations. VS Code extensions are loosely coupled with minimal dependencies on components to allow isolated changes while maintaining interoperability - the information exchange and collaboration of parts [8].

Void's Interacting Parts

Normal extensions in VS Code are lightweight and run using its official API inside the Extension Host process, which is isolated from the main editor GUI. In theory, they cannot modify the IDE itself; instead, they use safe JS APIs to extend features already offered, such as authentication, chat request handling, and code action providers [8]. Although Void behaves like an extension in architectural terms by borrowing VS Code's support for contract governance, plugin registries, and connectivity options, it is actually defined as a fork that modifies the editor at a deeper level. One may think of Void as a "custom version" with completely new capabilities embedded directly inside, like third-party contributors taking a microkernel and upgrading it before release.

Non-core functionality, such as Void, introduces its own orchestration and AI communication layers that interact with VS Code's microkernel interface. It therefore inherits the

editor's modular structure and collaboration mechanisms by relying on remote IPC channels between the Extension Host and Main Process [3]. Asynchronous connectors enhance responsiveness while data filtering supports privacy and security. Modular organization and clear process boundaries then improve its testability and maintainability. Finally, rather than existing as an external marketplace plugin, Void extends the VS Code core directly by introducing internal subsystems that expand its capabilities and preserve compatibility with Electron.

Major Subsystems

Void's architecture can be decomposed into several key subsystems, each responsible for a distinct aspect of its features. Together, these modules follow key principles of Process Separation, Service-Oriented Design, and Singleton Pattern. Besides the chat feature, Void's key subsystems can be abstracted as follows [12]…

1. Core services are encrypted internally within VS Code through voidSettingsService, which manages provider configuration and model settings. This subsystem acts as an implicit dependency that supports and is accessed by other core services below.
2. Communication between Void and various AI providers is handled by the LLM Message Pipeline.
3. editCodeService is responsible for autocomplete, managing where AI can modify code, processing responses, and visualizing differences.
4. Similarly, the Diff Zones subsystem visualizes and applies AI-suggested code changes to highlighted red/green regions.

Furthermore, components communicate primarily through asynchronous message passing, where the UI sends requests to an Orchestrator that routes them to the appropriate Model Adapter [4 Extension Development]. Streamed responses are returned incrementally to maintain responsiveness and ensure a non-blocking user experience.

Alternative Architectural Styles

Like most hybrid modern systems, Void doesn't use only one architectural style…

1. Main structure: Microkernel (Plug-in).
2. Communication: Event-driven/Publish-Subscribe.
3. Foundation structure: Layered Architecture.
4. Data management: Repository. An indexer refreshes data while the repository provides read access for prompt building, context selection, and checkpoints. Provider/model settings are handled by a central configuration service.
5. Processing flow: Pipes and Filters [5].

An alternative design could have embedded all functionality directly within VS Code's renderer to reduce IPC overhead but at the cost of modularity and evolvability. Void would be integrated monolithically where all components run in a single process with direct function calls instead of IPC or event buses [9]. One codebase has the advantage of easier development and deployment than microkernels (reduced overhead). Testing or debugging is also simpler because a central logging system contains all data processing. On the other hand, monolithic systems are tightly coupled which makes it difficult to integrate new technology. Scalability is also reduced since even minor changes may require dismantling and rebuilding the system [6]. This style suits small teams or prototypes prioritizing speed over long-term flexibility but Void's goals (extendability, portability, security) are better served by its current process-separated approach.

Control, Data Flow

When a developer enters code, requests a quick edit, or enters a command in chat, Void considers it a request. These requests are handled by the orchestrator, the core component of the system. The orchestrator decides whether to send the request to a local model on the developer's machine or to a remote provider in the cloud. The difference between the two is that the local model protects the privacy of the code, while the remote model can handle larger and more complex tasks. The orchestrator also manages the simultaneous processing of multiple requests. When developers type quickly, many small requests are created. To avoid wasted work, Void implements debouncing and cancellation features. Specifically, as developers continue to enter commands, older requests are canceled before the UI is displayed, ensuring that the most relevant and up-to-date suggestions are displayed only after the developer pauses.

Behind the scenes, the orchestrator works with the indexer and repository. The indexer continuously scans the workspace to update symbol, syntax, and embedding information, while the repository stores this information. The different components work together to provide a prompt for model building after the user makes a request: the prompt first passes through a security filter - removing sensitive information or blocking certain files, then the prompt goes to the model adapter, which is responsible for handling the connection to the local or remote model, and finally the model streams its output back, which is then converted into a difference by the editing application. The difference is displayed in the editor, and the developer can choose to accept, reject, or modify the changes [8].

Developer Division of Responsibilities

Analyst/Requirement Engineer

The Analyst is the primary owner of the Functional Viewpoint. Their responsibilities are foundational to the entire development process [12]:

- Elicit Demands: Gather raw needs and goals from stakeholders.
- Analyze & Formulate: Analyze these needs for consistency, feasibility, and completeness. They then translate them into formal requirements, defining the system's functional elements, their responsibilities, and their interactions as part of the specification [6].
- Validate: Ensure the specified requirements accurately reflect stakeholder needs, forming the basis for the system's intended behaviour.

UI/Frontend Developer

The UI/Frontend Developer operates within the Functional and Development Viewpoints. They are consumers of the core services and are responsible for bringing the user-facing aspects of the life application.

- Service Consumption: Import and utilize interfaces defined by the architect, such as the LLMMessageService.
- Implementation: Call service methods like sendLLMMessage with user input. Their main focus is implementing the logic for callback functions (onText, onFinalMessage, and onError) to update the UI components and display data streams to the user.
- Abstraction: They work against a defined contract (e.g., ServiceSendLLMMessageParams) and do not need to know the underlying implementation details of backend services. This is a key aspect of the Development

Viewpoint, ensuring the frontend can be built, tested, and maintained independently.

Core Logic/Backend Developer

The Backend Developer's work spans multiple viewpoints, as they implement the system's core engine.

- Functional Viewpoint: Implement the server-side listeners (e.g., for sendLLMMessage) and the business logic required to fulfill the feature requirements.
- Information Viewpoint & Security Perspective: Manage the static data structures and information flow, including handling sensitive data like API keys. This involves applying Information Security principles like access control to protect credentials [5].
- Concurrency Viewpoint & Performance Perspective: Handle the complex logic of managing simultaneous connections and data streams from LLM providers. This directly addresses Concurrency Performance, involving the coordination of shared resources, queuing requests, and preventing blocking to ensure a responsive system.
- Operational Viewpoint: Implement robust error handling for network failures and other runtime issues to ensure system availability and stability in a production environment.

Software Architect/Integration Lead

The Software Architect is responsible for the overarching design, defining the "seams" between different parts of the system and considering all viewpoints and perspectives to ensure long-term health [12].

- Defining Contracts: Design and define the critical interfaces (ILLMMessageService) and data transfer objects (DTOs) that constitute the API contract. This work within the Information Viewpoint establishes clear boundaries for information flow.
- Functional Evolution Perspective: By designing flexible interfaces and extension points, the architect ensures the system is adaptable and can evolve. The design of the message service is a critical architectural artifact that enables future expansion to new LLM providers with minimal friction.
- Concurrency Viewpoint: Make key architectural decisions to manage concurrent processes, such as implementing a UUID-based request tracking system to handle multiple, simultaneous LLM requests safely and efficiently.
- Cross-Cutting Concerns: Ensure non-functional requirements, such as security and performance across all viewpoints, are met. They design the communication protocol to be both efficient (Performance Perspective) and secure (Security Perspective), enabling parallel team development.

Concurrency

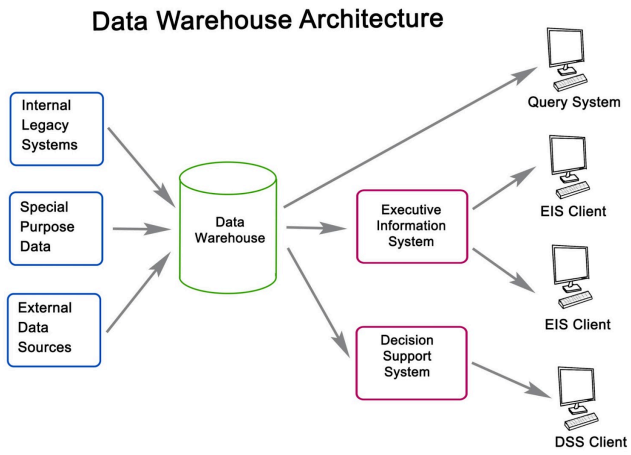
Concurrency is a critical aspect of the Void system, ensuring a responsive and stable user experience. The approach is not based on a single mechanism but is a multi-layered strategy that aligns with the **Concurrency Viewpoint** of the software's architecture. The two core principles of this strategy are process-level isolation and asynchronous, event-driven communication [14].

1. Process-Level Isolation for Stability and Performance

At the highest architectural level, Void utilizes a multi-process model inherited from its Electron foundation. This is the cornerstone of its stability and resilience, creating a clear separation of concerns between core logic and the user interface.

- Conceptual Structure: The application is divided into a single Main Process (handling backend logic, network requests, and file system access) and one or more Renderer Processes (each responsible for a distinct UI window). This separation creates a hardware-enforced boundary, preventing a crash or performance issue in a UI window from affecting the application's core functionality [12].
- Communication Model: Interaction between these processes is handled exclusively through controlled Inter-Process Communication (IPC) channels. While IPC introduces overhead compared to direct function calls, it is essential for maintaining the integrity of the isolation boundary. This design choice prioritizes system stability and predictable performance, especially in an application that makes frequent I/O calls [12].
- Architectural Example: Asynchronous IPC Invocation When a user acts as the UI, such as sending a query to an LLM, the Renderer Process dispatches an asynchronous "fire-and-forget" message to the Main Process via an IPC channel. The Renderer does not block or wait for a full response. Instead, it immediately receives a unique request identifier and continues its operations, ensuring the UI remains fluid.
  To manage concurrent requests and their corresponding responses, the system employs a Publisher-Subscriber (Pub/Sub) pattern.
  - The Renderer Process subscribes to specific event channels, registering its interest in responses tied to the unique request identifier.
  - The Main Process, after handling the request (e.g., communicating with the LLM API), acts as a publisher, emitting events back to the Renderer Process. These events contain the data (such as streaming text or a final result) and the original request identifier.

This event-driven model, managed through unique identifiers, allows the system to handle multiple, interleaved, long-running tasks without confusing their responses. It is a direct application of the Concurrency Performance perspective, focusing on coordination and non-blocking operations to serve the user efficiently [14].



2. Asynchronous Patterns for a Non-Blocking UI

Within each process, and especially in the Renderer, Void relies on an event-driven, non-blocking concurrency model native to the JavaScript/TypeScript environment. This ensures that the user interface is never frozen by long-running tasks [14].

- **Conceptual Structure**: All operations that involve waiting for an external resource—such as API calls, file I/O, or database queries—are handled asynchronously using patterns like Promises and async/await.
- **The Event Loop Model**: Unlike traditional multi-threaded models, JavaScript uses a single-threaded event loop. When an asynchronous operation is initiated, it is handed off to the underlying environment to be processed. The event loop is then free to continue executing other code, like responding to user input. Once the asynchronous task is complete, its result is placed in a message queue to be processed by the event loop only when the main execution thread is idle.

This model is fundamental to the application's design, guaranteeing that I/O-bound operations do not compromise the responsiveness of the user interface, which is a primary goal of the system's operational design.

External Interfaces

User Interface/Frontend

As we can see from the GUI, users interact with the system in multiple ways. In general, Void takes input from the user based on what they are doing in the editor, whether it's typing code directly into the editor, using quick edit, typing in the chat panel (input depending on the chat mode selected), creating files, writing commands, or accepting AI suggestions. Forms of output include normal IDE functions such as syntax highlighting, but the majority comes from the output of the AI model, such as suggestions, explanations, and generated code.

APIs and Servers

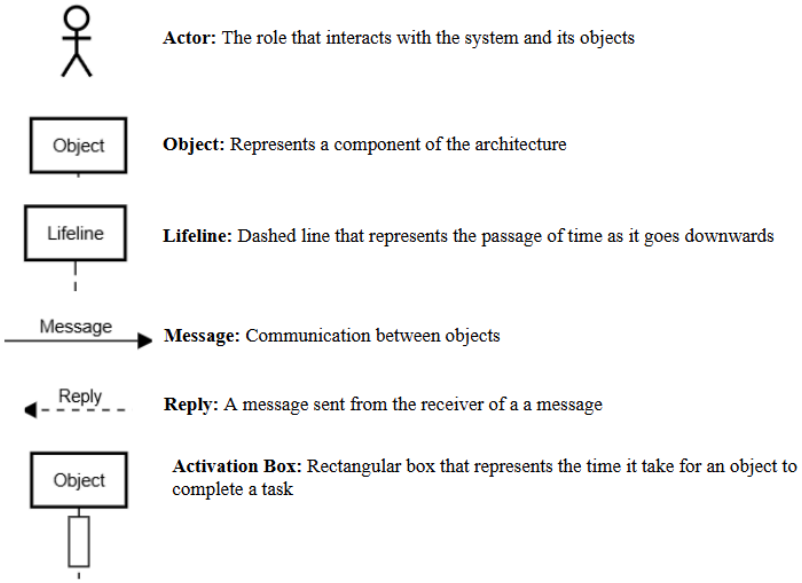
When users request a task from an AI (from the user interface), they have the option to use a local model or a direct connection to any provider [12]. The communication is handled by an LLM message pipeline through which the chosen model receives the prompt as well as any context required and outputs the text response. In a recent addition to Void, a new form of input are checkpoints for LLM changes, where APIs will receive data from the system about the acceptance or rejection of previous AI outputs and will produce new suggestions based on that new context. Aside from AI, Void also offers the same features of VS Code, including extension APIs, where data about files and the editor itself will be exchanged depending on the extension functionality.

Files and Data Sources

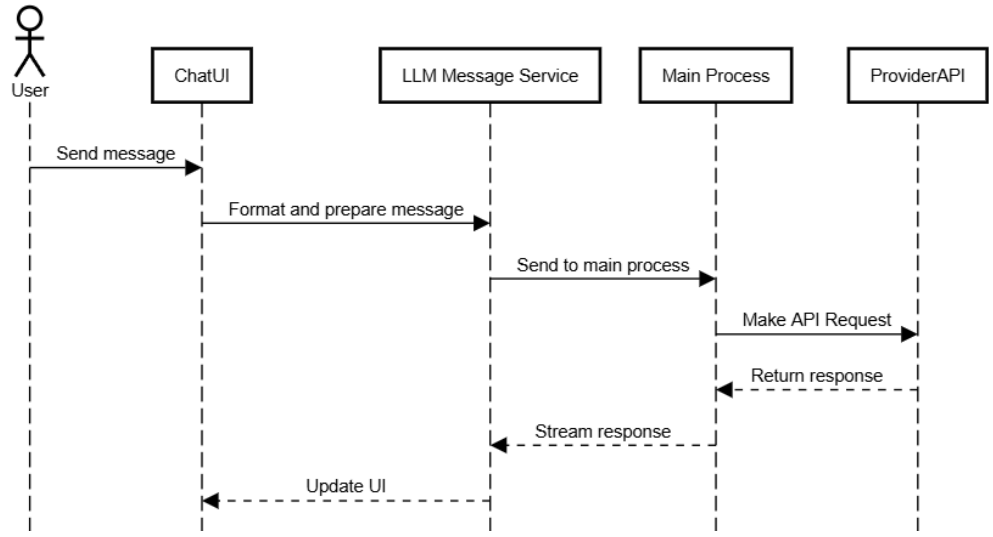
Being a fork of VS Code, Void inherits the same external interfaces that VS Code originally had access to. Like any IDE, Void writes and reads straight from your local file system, saving any changes made by the user or the AI to the source file and loading the contents of a file to edit. Similarly, Void saves settings, themes, keybinds, and any other user preferences, which can initially be set by transferring those preferences from a previously used IDE such as VS Code, Cursor, or Windsurf [10]. In addition to local data, Void also has support for Git, giving the ability to push up and pull down from repositories as well as view diffs and repository status. Finally, Void gives and receives information from the OS through the built-in terminal,

allowing the user or AI models to compile, test, run, or perform similar operations through commands.

Use Cases

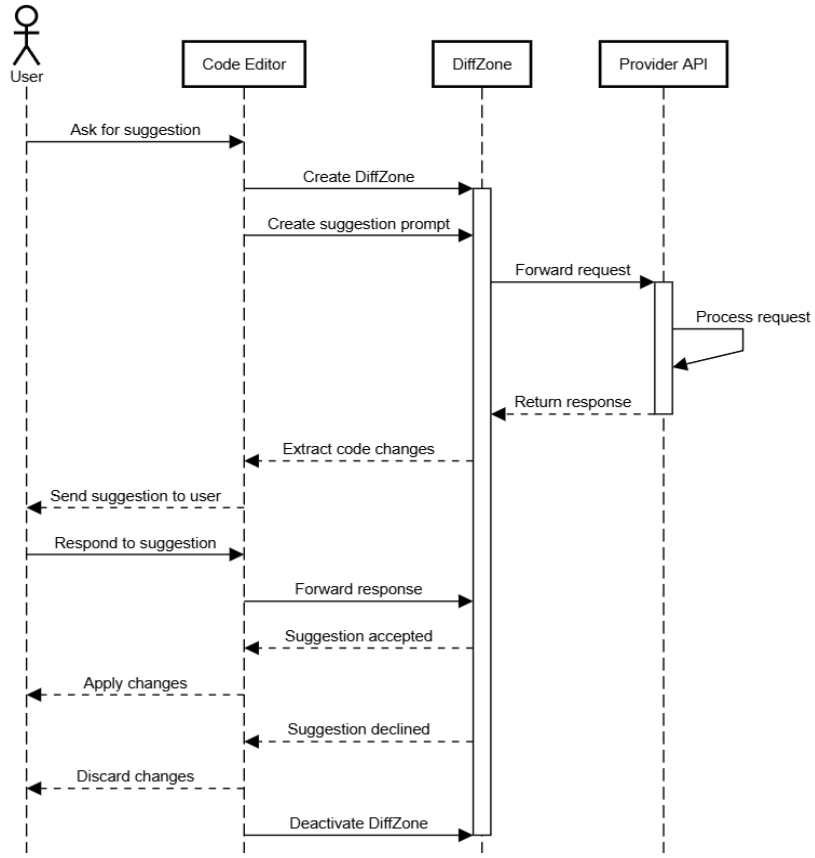


Use Case 1: Communication between Void and AI Providers



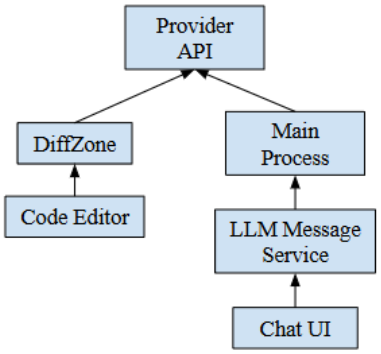
The user would open the chat function through the Void IDE and send a message through the chat UI. The chat UI would format and prepare the message to be sent to the built-in LLM messaging service. The LLM messaging service would send the formatted message to the main process, where it is forwarded to the user's chosen API provider, such as ChatGPT or Deepseek [12]. The user's chosen API would generate a response for the main process, which passes it on to the LLM messaging service. Finally, the LLM messaging service updates the chat UI within the IDE to show the user the generated response.

Use Case 2: Void AI Code Editing Service



When coding with the Void IDE, the user can get AI suggestions on how to complete code. The code editor would create a temporary "DiffZone", an area where the AI can provide suggestions and edit code. The code editor would then send the code to the user's chosen API, where it processes the request and returns a possible way to finish or improve the code to the DiffZone [12]. The code editor would get the code changes from the DiffZone and show them to the user, who would then respond to the AI's suggestion. If the suggestion is accepted, the code editor applies the changes to the user's code; if it's declined, the code editor discards the AI's suggested changes. Regardless of the user's choice, the code editor then deactivates the DiffZone.

Box-and-Line Diagram



Data Dictionary

- Requests – Created when a developer enters content, requests edits, or uses chat. They contain user input and code context.
- Orchestrator – A central controller that routes requests to local or remote models, cancels outdated requests, and manages concurrency.
- Indexer – Continuously scans project files to track symbols and syntax, and stores the latest information in the repository.
- Repository – Stores indexed data, used to build prompts for the models.
- Security Filters – Check inputs and outputs, removing potentially sensitive or unsafe content.
- Model Adapters – Provide a unified interface for different local or remote models and stream the results back.
- Diff/Applier – Converts model outputs into patches and displays them in the editor for review and acceptance.

Naming Conventions

- OS: Operating System.
- HTML: Hypertext Markup Language.
- CSS: Cascading Style Sheets.
- JS: JavaScript.
- APIs: Application Programming Interface.
- VS Code: Visual Studio Code.
- IPC: Inter-process Communication.
- IDE: Integrated Development Environment.
- LLM: Large Language Model.
- AI: Artificial Intelligence.
- FIM: Fill-In-The-Middle.
- MCP: Model Context Protocol.
- GUI: Graphical User Interface.

Conclusion

Void IDE is a flexible, extensible, high fault-tolerance, and privacy-focused AI-assisted development environment, built upon VS Code and Electron. It allows users to choose their own LLM assistant and communicate directly with it without the need for a third party to ensure security. Its conceptual architecture blends microkernel and layered styles, coordinated by an intelligent orchestrator that manages AI-driven tasks across key subsystems. The system ensures responsiveness and stability through process isolation and asynchronous communication, inherited from Electron's multi-process model.

Void's evolution demonstrates a clear commitment to developer control and transparency, with features like visual diff views and checkpointing enabling manual review of AI suggestions. By supporting both local and remote models, Void effectively balances flexibility, privacy, and code control. As an open-source project, it stands as a more secure alternative for modern AI-assisted programming.

Lessons Learned

After completing this study, we have gained the following insights and lessons learned. We understand that modern systems usually do not adopt a single architectural style, but rather a hybrid style. Void IDE uses a layered architecture as its foundation, inherited from Electron/VS Code and superimposes a microkernel/plug-in style on it to achieve flexibility, high fault tolerance and extensibility.

We understand that the key to the Void system's excellent concurrency and stability lies in the multi-process model of its underlying framework, Electron, especially the process isolation mechanism, which allows the main application to continue running even if a rendering process crashes. We understand the importance of asynchronous inter-process communication (IPC) and the publish-subscribe model (Pub/Sub) for maintaining UI responsiveness, and how to safely handle multiple concurrent requests through UUID-based tracking.

In terms of lessons learned from group management, setting an earlier deadline provides more time to edit for grammar, write the introduction and conclusion sections, and format the presentation.

AI Report is the last 2 pages below References.

References

[1] A. Kumar, "Void IDE: The Comprehensive Guide to the Open-Source Cursor Alternative," *Medium*, Mar. 24, 2025. [Online]. Available: https://medium.com/@adityakumar2001/void-ide-the-comprehensive-guide-to-the-open-source-cursor-alternative-2a6b17cae235.

[2] A. Perkaz, "Advanced Electron.js architecture," *LogRocket Blog*, Oct. 5, 2023. [Online]. Available: https://blog.logrocket.com/advanced-electron-js-architecture/.

[3] C. Jesse, "From Learner to Contributor: Navigating the VS Code Extensions Structure," *Medium*, Nov. 2, 2024. [Online]. Available: https://medium.com/@chajesse/from-learner-to-contributor-navigating-the-vs-code-extensions-structure-ed150f9897e5.

[4] Electron, "Electron Documentation," *Electron*, [Online]. Available: https://www.electronjs.org/docs/latest/.

[5] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 3rd ed. Boston, MA, USA: Addison-Wesley, 2012.

[6] M. Kleppmann, *Designing Data-Intensive Applications*. Sebastopol, CA, USA: O'Reilly Media, 2017.

[7] Microsoft, "Extension Host," *Visual Studio Code*, [Online]. Available: https://code.visualstudio.com/api/advanced-topics/extension-host.

[8] Microsoft, "Visual Studio Code Extension API," *Visual Studio Code*, [Online]. Available: https://code.visualstudio.com/api/references/vscode-api%5C.

[9] P. Powell and I. Smalley, "What is monolithic architecture?," *IBM Think*, [Online]. Available: https://www.ibm.com/think/topics/monolithic-architecture.

[10] Void Editor, "Void Official Website," *Void Editor*, [Online]. Available: voideditor.com.

[11] WorldofAI, "Void IDE: 100% FREE AI Agentic IDE - Cursor + Windsurf Alternative! FREE API! (Opensource)," *YouTube*, May 14, 2025. [Online]. Available: https://www.youtube.com/watch?v=nBWONO1Nnsw.

[12] Zread.ai, "Architecture Overview | Voideditor/Void," *Zread*, Jun. 25, 2025. [Online]. Available: https://zread.ai/voideditor/void/9-architecture-overview.

[13] Z. Tavargere, "Microkernel Architecture: How It Works and What It Offers," Zahere, Feb. 26, 2023. [Online]. Available: https://zahere.com/microkernel-architecture-how-it-works-and-what-it-offers.

[14] A. M. Boljam *et al.*, "Impact analysis of generative AI," *ICCTDC*, 2025. [Online]. Available: ieeexplore.ieee.org/document/10895857

AI Collaboration Report: OpenAI, ChatGPT 5, August 2025

Reason for choice

GPT-5 is currently one of the best LLMs in the world. It boasts powerful natural language processing, text summarization, creative writing, grammatical review and correction, and logical reasoning capabilities. We primarily use GPT-5 for three tasks. The first is to help us check our reports for grammatical, factual, and logical errors. The second is to help us clarify issues and ensure that our reports meet the required standard. The third is to use GPT-5 to help us introduce and understand various architectures or word concepts. GPT-5's powerful natural language processing, grammar review, and correction capabilities demonstrate its ability to serve as an excellent reviewer, helping us check the facts, grammar, and logic of our reports. (Task 1) GPT-5's powerful reasoning and natural language processing capabilities helped us clarify questions, understand the concepts of structure and vocabulary, and run logic, ensuring we understood and correctly wrote what we were supposed to write. (Tasks 2 and 3) GPT-5 also had a lower hallucination rate than other LLMs, making us more willing to trust GPT-5's output.

Work with GPT-5

We don't have a dedicated person who asks questions to GPT-5. Everyone writes down their questions, motivation and records the answers from GPT-5. At the end, one person is responsible for compiling and writing the AI collaboration report.

Task we give to GPT-5

1. Suggesting division of group member roles based on original requirements to ensure fair work distribution and a clean report structure (Question and Answer).

Motivation: The amount of work required for each part of the report varies depending on the task. Our team had no experience writing reports for similar tasks, so we couldn't determine the amount of work required for each part. To ensure that everyone completes a similar amount of tasks, we gave the task description to GPT-5 and asked it to assign the tasks for us.
Improve the quality of GPT-5 output: Tell GPT-5 the details of the task and the number of people in the team (provide background information). Ask GPT-5 to provide a fair division method so that each member has a balanced workload (clarify the task and requirements).

2. Before writing our own section, clarify how a layered architecture works in general, especially in systems that use Electron and VS Code (Question and Answer).

Motivation: Using GPT-5 to explain how the layered architecture works in the context of the Void IDE helps us understand why Void's developers chose this implementation approach and its advantages.
Improve: Provide clear requirements for GPT-5 and provide a layered architecture operating environment (background information).

3. Reviewing sentences for grammar only, without rewriting the meaning. Only giving pointers on how to improve, fix clarity and flow. Example, "Like any IDE, Void writes and reads straight from your local file system, saving any changes made by the user or the AI to the source file and loading the contents of a file to edit" (Question and Answer).

Motivation: Use GPT-5 check the text for grammatical problems and whether the clarity and fluency of the text can be improved.

Improve: Clarify our requirements and needs, ask GPT-5 not to re-interpret the text meaning and only check the text grammar (clear requirements), and put forward that we only want to improve clarity and fluency and suggestions for improvement (state requirements).

Critical prompt

Task 3 is an example of us using a critical prompt to let GPT-5 give the answer we want. That critical prompt is “Please don’t rewrite the meaning” and “Only give us some pointers on how to improve”. All LLMs diverge unconsciously. When your input is not clear enough/the prompt words are not enough, LLM will do a lot of unnecessary or even wrong things that you did not ask it to do. If we give GPT-5 a passage and ask it to improve clarity and fluency, it will add unnecessary embellishments, sentiment, and interpretations to its response. These word changes often cause the modified sentence to deviate from the original meaning. Therefore, we must tell GPT-5 “Please don't rewrite the meaning”, and “Only give us some pointers on how to improve” in the input to ensure that the response does not deviate from the original meaning.

Validate GPT-5's output

Task 1: We assigned tasks to each person according to the GPT-5 division suggestion. We held another group discussion three days after the assignment to ensure that everyone felt that the assigned tasks were reasonable.
Task 2: After getting GPT-5's answer, we searched for the definition on Google and compared both of them with the answers of other LLMs (Deepseek v3, Gemini 2.5 flash) for verification.
Task 3: We asked multiple members to read the original input and GPT-5 output and choose which output was better.

Quantitative Contribution to Final Deliverable

GPT-5 didn't play a significant role in our report development. Our team was reluctant to rely heavily on GPT-5. We viewed it primarily as a reference, checklist, and guidance tool. We used GPT-5 to explain unfamiliar concepts and help us better understand them (saving time). We also used GPT-5 to review our reports and provide suggestions (improving the quality and accuracy of our reports). We didn't always accept GPT-5's suggestions, and sometimes we disagreed with its responses. Therefore, we agree that GPT-5's contribution to our reports was not significant. Therefore, we believe GPT-5's overall contribution to the final deliverables was approximately <5%.

Reflection on Human-AI Team Dynamics

GPT-5 has indeed helped us save time and improve the quality of our reports. It helps us understand new knowledge and concepts more quickly, and it allows us to quickly review grammar and provide suggestions. GPT-5 hasn't caused any disagreements or challenges within the team, nor has it impacted our brainstorming and decision-making processes. Because writing reports is already a familiar task for our team members, we can make many decisions without consulting GPT-5.

Our team has learned from working with GPT-5 that AI is a very limited tool. AI is better suited to helping us quickly understand new knowledge and new architectures. Its strengths include natural language analysis (grammar checking). However, we cannot fully trust its output, and LLM illusions remain a serious problem.