Exploring the Conceptual Architecture of Void

Project: Void IDE - AI-Assisted IDE built on VS Code and Electron
Deliverable: A1 - Conceptual 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: October 10, 2025

Abstract

This report presents the conceptual architecture of Void IDE, an AI-assisted development environment built as a Visual Studio Code (VS Code) extension on Electron. Void integrates generative AI features such as chat based code generation, autocompletion, and natural language refactoring directly into the developer workflow, while preserving the modularity and extensibility of VS Code. Its significance lies in bridging conventional editors like VS Code and AI capabilities, which in turn results in faster iteration without leaving the IDE context.

This report's purpose is to document Void's conceptual architecture, and in doing so, emphasize its major subsystems, data and control flows, and architectural styles, the recovery process involved analyzing official VS Code and Electron documentation, studying Void's extension APIs, and deriving component interactions from use cases. The report distinguishes conceptual design from implementation detail, and focuses on interaction boundaries and evolvability.

At this level, Void employs a hybrid of client-server, and event-driven styles, which were chosen to balance responsiveness and modular growth. Its principal subsystems are the Platform Services, Editor Core, Workbench Framework, UI Components,AI Service, Provider Integration, Extension System, Database & Files. Control flows from user actions through the Void Extension to the AI Service Layer, while data moves asynchronously via IPC within Electron. This separation helps to ensure responsiveness, safety, and clear developer responsibilities.

To illustrate structure and behavior, the report includes UML components and dependency diagrams, as well as two sequence diagrams outlining our two chosen use cases. Overall, Void's architecture provides a scalable and evolvable foundation for AI-enhanced coding, with clean subsystem boundaries that support maintainability, testability, and extension to future LLM models.

---

1. Introduction

1.1 Purpose and Scope

The scope of study focuses on the relationships among three main systems: Electron (runtime), VS Code (core platform), and Void (extension and AI services), and treats the latter two as a system of interacting subsystems whose behavior can be described through sequence flows and UML component views, as depicted in sections below.

1.2 Background and Architectural Context

Electron Runtime Layer

Electron is a cross-platform desktop framework combining Chromium (UI renderer) and Node.js (system runtime). Its multi-process model separates a Main Process (application lifecycle and native OS access) from multiple Renderer Processes (UI windows), and communicates through asynchronous Inter-Process Communication (IPC) channels. This design provides fault isolation, concurrency, and portability across Windows, macOS, and Linux.

VS Code Core Layer

VS Code builds on Electron using a layered architecture.

Base Layer: shared utilities and UI frameworks
Platform Layer: dependency-injection services
Editor (Monaco): text buffers and language features
Workbench Layer: top-level UI and extension management

Core functionality is exposed through a stable Extension API, with all features, including language support, debugging, themes, implemented as plug-ins that execute inside an isolated Extension Host process. This microkernel pattern enables controlled extensibility while maintaining lightweight performance.

Void Integration/AI Extension Layer

Void extends this foundation by introducing an AI Extension Layer within the VS Code Workbench. It incorporates services implemented as singletons (registered via registerSingleton()), including:

voidSettingsService: provider configurations and model settings;
LLMMessageService: bridging local and remote language models via HTTP and WebSocket;
editCodeService: managing diff zones for AI-generated edits;
chatThreadService: streaming conversational state; and
toolsService: controlled file access and workspace operations.

These services form a client-server and event-driven microkernel hybrid, connecting the editor frontend to a Node-based AI backend. This architecture enables AI-assisted features while preserving VS Code's modularity and Electron's process safety.

1.3 Derivation Process

We derived the conceptual architecture through four phases:

Document and Reference Study: primary sources included Void's official documentation, the VS Code GitHub Wiki and DESOSA (2021) architectural analysis, and Electron architecture guides (LogRocket 2024; Medium 2023). These defined the baseline layered stack and concurrency model used for Void's integration.

Use Case Identification: two core workflows were selected to trace data and control flow: (a) Chat-Based Code Generation and (b) Autocomplete While Typing. Both illustrate how events propagate through the Void Extension Layer and AI Service Layer to modify the editor state.

Component and Dependency Extraction: recurring entities from the use cases were grouped into conceptual modules: Electron Runtime, VS Code Core, Void Extension (front and backend services), and AI Provider Interfaces. Static relationships (e.g service dependencies, IPC links) and runtime flows (e.g streamed LLM responses) were mapped into component and sequence diagrams for structural and behavioral views.

Validation Against Reference Architecture: the resulting layer model was cross-checked with VS Code's official architecture to ensure conceptual consistency. This confirmed that Void's Extension Layer and AI Service Layer maintain proper separation of concerns and operate within the established processes client-server boundaries.

3.

3.1 Architectural Styles

Void is an AI enhanced code editor and IDE that utilises VS Code's Electron based architecture but also layers event driven logic for AI features. The main architectural style of Void is Layered, however, Implicit Invocation and Client Server are secondary architectural styles.

The main architectural style (Layered) is evident because of the interaction process of the major components (described below) and how they form a hierarchy of processes and responsibilities that provide services for one another. Void adopts VSCode's layered style of base layer -> platform layer -> editor layer -> workbench layer, it uses this separation of processes to organize code and only share what is necessary. This separation of responsibilities creates a clear flow of control between the layers. The base layer provides cross platform utilities, platform which delivers platform services, editor which contains the Monaco core and handles text editing operations, and finally workbench layer, which is the main user interface and app logic.

The secondary architectural styles are evident due to the structure of the Electron framework, which has a built-in client-server structure. In Electron, the primary processes are split up with each having its own role, such as the main process, which is responsible for managing the lifecycle of the application, creating application windows, managing applications, and acts as the application's entry point. There is also the renderer process, which is responsible for

generating/rendering web content using HTML, CSS and JavaScript, it also requests services from the main process using inter-process communication. The utility processes are responsible for generating child processes, which can be used to host untrusted services, resource intensive tasks and crash prone components, this stops the process from being hosted in the main process and causing issues. Communication between these utility processes occurs through message ports, reinforcing client-server ideas.

Finally, the Implicit Invocation style controls communication between the different layers. Instead of invoking components directly, the components in Void use an event driven system where components communicate through events rather than making direct calls to other components. By using an event bus, Void processes can "publish" events and utilize the "subscribe" system to ensure the necessary components listen to what is necessary.

3.2 Conceptual Diagram

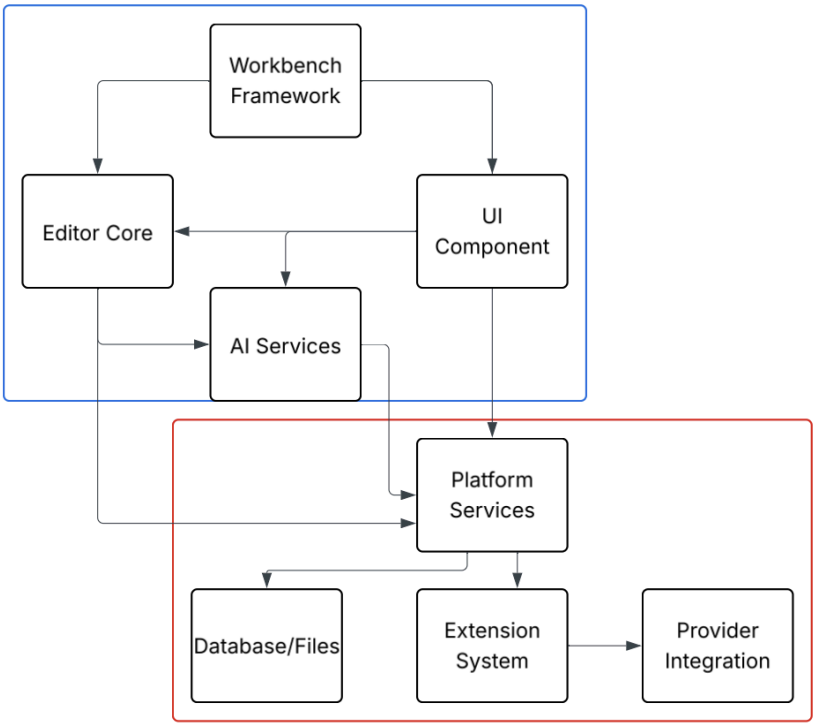


Figure 1: Diagram depicting conceptual architecture of Void

Legend:
Blue rectangle -> Browser Process
Red rectangle -> Main Process

3.3 Subsystem Descriptions

Platform services

The platform services subsystem is responsible for delivering core services and/or abstractions that form the foundation for all other components. In particular, this subsystem is responsible for configuration management (handling user settings and preferences), extension system (lifecycle and discovery), Inter-process communication operations, the theme system (for visual

customizations), and telemetry (for usage analytics). This subsystem was originally part of VS Code, but since Void is a fork of VS Code, it evolved into a major subsystem of Void, for example, the subsystem is now also responsible for handling AI provider settings. The Platform services subsystem is dependent on no other components, but rather it provides functionality and services to other components.

Editor core

The editor core subsystem is indispensable for containing the Monaco editor core (Microsoft's code editor powering VSCode), which provides text editing, syntax highlighting, language features/services (IntelliSense), code validation and a contribution system for editor extensions. This subsystem was also originally part of VS Code, but since Void is a fork of VS Code, it evolved into a major subsystem of Void. The editor core subsystem's dependencies are platform services.

Workbench framework

The workbench framework subsystem is responsible for the main user interface (VS Code) and the application logic behind many services in Void. Some of the responsibilities include managing 80+ services, modular feature contributions and the API definitions. In particular, it uses a dependency injection system, through this it can provide AI enhanced services such as the chat thread service, but also provides core application services such as managing editor instances and providing terminal functionality. This subsystem was also originally part of VS Code, but since Void is a fork of VS Code, it evolved into a major subsystem of Void. The workbench framework subsystem's dependencies are both the editor core (Monaco and editor services) and platform services (IPC communication, extension management).

UI Components

The UI Components subsystem is responsible for providing AI enhanced user interfaces that integrate the AI features of Void into VS Code's environment. The main responsibilities of this subsystem are rendering the chat interface and any other void specific features/menus. The UI subsystem is essentially just a humanized interface to present the AI features within Void. The UI components subsystem's dependencies are AI services (updating chat) and platform services (updating themes, config settings)

AI Services

The AI Services subsystem is responsible for being the "brains", allowing users to interact with an AI, it works by receiving user requests for certain AI features, providing a workflow and managing the requests and operations. In particular, it is responsible for analyzing code content, organizing chat sessions, providing real time code suggestions, processing general AI responses, managing other AI dev tools and more. The AI services subsystem's dependencies are platform services (model settings, api keys etc.) and provider integration (sends requests to providers and receiving real time responses)

Provider Integration

The Provider Integration subsystem exists to unify/connect different AI providers, it's meant to create an interface for AI operations and handle multiple different API's. Some of the responsibilities include supporting custom endpoints (for self hosted models), provider specific features, model capability detection and configuration management (for all providers).

Extension System

The extension system is vital in maintaining compatibility with VSCode while adding AI features. Specifically, the extension system allows for configuration, viewing of extension points. It essentially acts to allow Void to stay a VS Code fork while adding AI features. Some of its main responsibilities include Lifecycle management, providing AI specific extension points (allows extensions to access the AI features within Void). This subsystem was also originally part of VS Code, but since Void is a fork of VS Code, it evolved into a major subsystem of Void.

Database/Files

The Database/files subsystem is responsible for data access operations, handling file operations, and potential storage needs. Some of the core responsibilities are file operations (read, write, create, delete) and storage management. In the case of Void, databases/files are a very important subsystem because the system needs to store conversation histories, potential model configurations and even slight user preferences (in terms of AI chats).

3.4 Control & Data Flow Among Parts

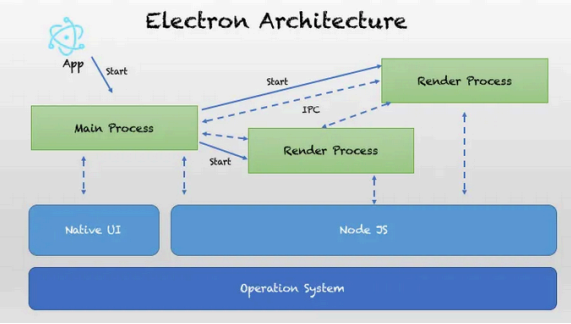


Figure 1
Electron Framework Architecture illustrating the Main Process, Renderer, and IPC model. Adapted from "Electron: Things to Watch Out for Before You Dive In," by Vishal Dwivedi, 2018, Medium.
https://medium.com/@vishaldwivedi13/electron-things-to-watch-out-for-before-you-dive-in-e1c23f77f38f

Void software's control and data flow follow a structure that is mostly reliant on the existing infrastructure created by Visual Studio Code (VS Code) and the Electron framework, while providing added layers for integrating AI into the user's workspace.

The Main Process in Void manages window creation and handles any changes made during runtime like crashes, closing and opening windows, and launching the app. It also facilitates

communication between the user interface and background services that manage the actual connections to AI providers (Electron, 2018; Perkaz, 2021).
Each Renderer Process is responsible for running the user interface and responding to user input. These renderer processes operate independently, ensuring that user interactions occur without blocking other parts of the application (Dwivedi, 2018). From the perspective of Void, this mainly refers to displaying the interactions you have when chatting with an AI model of your choosing.

The Extension Host is a separate subsystem where Void's AI-based features run. It processes language model requests, context extraction, and code suggestion generation. When using any of these features, data flows from the Renderer (once it provides an initial command), through the Main Process, and to the Extension Host, where it completes the task if it is a part of its capabilities. Once the task is processed, it sends the result back to the UI through the Main Process from the Extension host, finally arriving at the Renderer, where the user can view the response in the UI. (Pareles & Pareles, 2025; Kumar, 2025).

Communication between the Main Process, Renderer, and Extension Host in Void happens through Inter-Process Communication (IPC), which is built on top of Chromium's message-passing system. This allows the different systems to send messages to each other using Electron's asynchronous IPC channels. When a user begins to interact with the UI, the Renderer sends the message to the Main Process, which assesses the request and delegates it to the Extension Host if needed for AI operations (Electron, 2018; Chromium, 2025).

Overall, control and data begins primarily with the Renderer via the UI and then to the Main Process for system logic where its possibly sent to the Extension host based on the kind of request and finally returned back through the all points to the Renderer so the user can see the response through the UI which is all sent using IPC which is built using Chromium's message-passing system. This accurately details the control and data flow among parts within the overall system.

3.5 Concurrency
Void supports and uses various degrees of concurrency including the Electron framework and asynchronous operations. Architecturally, at the process level, the architecture of Electron allows concurrency through executing the Main Process, several Renderer Processes, and the Extension Host as separate processes. All the processes operate on their own threads simultaneously, which eliminates freezing or latency during code generation and situations where the AI model that was selected might require thinking before generating an output (Electron, 2018; Perkaz, 2021).

On a more fundamental level in each of the processes, Void will take advantage of the event loop that is provided by Node.js, allowing it to be asynchronous to leave intensive operations, including answering prompts and code-generating, to background threads. This will enable such operations to run in the background and the main thread to service other interactions or updates to the UI. This means that users can still carry out other tasks such as editing files or any user settings, even as AI generation or file modifications are taking place. This non-blocking design makes Void receptive and efficient at resource-intensive operations (Kumar, 2025; Perkaz, 2021; Electron, 2018).

Moreover, the AI services embedded in Void enable parallel AI requests, i.e. a query in one feature (e.g., an update of chat context, suggestions in an autocomplete, and analysis of errors) can be processing concurrently in another function (e.g., an image recognition query). Reverse responses are subsequently sent back to the Renderer and can be displayed in the UI in the order that they are received (Kumar, 2025; Pareles & Pareles, 2025).

Finally, the process communication is concurrent because the messages may be received and transmitted simultaneously between the Main, Rendering and Extension Hosts, synchronized by the asynchronous event listeners. The illustration of concurrency exhibited by Void enables a faster response time and more effective communication (Electron, 2025; Dwivedi, 2018).

3.6 Evolvability & Modifiability

Void uses a modular and extensible architecture that allows for easy integration of new LLMs and features. Because of this, it allows for model providers such as OpenAI and Ollama to implement a shared provider interface which standardizes how prompts, parameters, and responses are exchanged. This type of architecture provides a simple way for providers to be added by creating a new module that follows the same interface, without altering the existing system logic. In addition, the system inherits the VS code extensions framework, enabling developers to use optional extensions that allow developers to add new commands, UI components, or analytical tools. Because of the extensible and modular architecture, it gives developers the flexibility to build off and expand the system's capabilities without changing its fundamental architecture.

3.7 Alternative Architectural Styles Considered

Another architecture that was considered for Void was the pipe-and-filter architecture, as the data of the AI Extension Layer is processed by several steps including user input, context extraction, model inference, response processing, editor update, and a series of filters converting data. Nevertheless, pipe-and-filter is more appropriate to linear or batch processing, and Void operations are highly interactive, event-driven and concurrent. There can be several AI queries at once, which means that they need to be answered asynchronously and added to the UI in real time. This renders pipe-and-filter inappropriate in dealing with dynamic and non-linear flows in Void. Therefore, a hybrid one, which is a combination of layered, client-server, and event-driven style, is more conducive to modularity, asynchronous communication, and extensibility, all of which are essential to the functionality of an AI-assisted IDE.

---

4. System Functionality & Use Cases

4.1 Core Functionality Overview

Void IDE extends Visual Studio Code's Electron-based architecture with an AI Extension Layer that introduces intelligent, LLM driven developer features while preserving VS Code's modular microkernel design. Its functionality centers on three user-facing capabilities:

Feature	Description	Subsystems/Services
AI Autocomplete	Predicts and inserts code completions as the developer types, using context from the current buffer and cursor.	editCodeService, sendLLMMessageService, VS Code Language Server, AI Services Subsystem
Chat-Based Code Generation / Refactor	Generates or refactors code from natural-language prompts entered in the chat panel.	chatThreadService, sendLLMMessageService, editCodeService, toolsService, AI Services Subsystem
Model Configuration & Tool Management	Lets users select model providers (OpenAI, Ollama), manage API keys, and control file-access tools.	voidSettingsService, toolsService, Workbench Settings UI, AI Services Subsystem

These capabilities operate asynchronously across Electron's Main -> Renderer -> Extension Host processes using inter-process communication (IPC). The AI Extension Layer bridges developer intent (that being the natural language input) with editor actions, which helps to maintain responsiveness and modularity.

4.2 Primary Actors

Actor	Role	Responsibilities / Notes
Developer	Primary user	Triggers AI features through typing or chat.
Void UI (Frontend)	User interface layer	Built in SidebarChat.tsx; handles input/output rendering.
AI Service Layer (Backend)	Core AI processing tier	Includes sendLLMMessageService, sendLLMMessage.impl, modelCapabilities; handles prompt construction, provider calls, and streaming.
Post-Processor / Edit Code Service	Code integration layer	editCodeService and toolsService; merges model output as diffs.

VS Code Core (Editor)	Host platform	Monaco Editor and Language Server Protocol (LSP) for completions and edits.

4.3 Use Case 1: Chat-Based Code Generation

Goal: Generate or refactor code from a natural-language prompt.
Actors: Developer, Void UI, AI Service Layer, Post-Processor, VS Code Core.

Main Flow (see Figure 5.1):

1. The Developer enters a prompt in the Void UI chat panel (SidebarChat.tsx).
2. chatThreadService forwards the prompt to the AI Service Layer (sendLLMMessageService).
3. The AI Service Layer packages context (file snippet, cursor state) and calls the Model Inference Engine (sendLLMMessage.impl, modelCapabilities).
4. The model streams tokens iteratively (loop).
5. Each chunk is sent to the Post-Processor (editCodeService) for diff formatting.
6. The Editor receives the patch via VS Code's edit API and updates the buffer.
7. Alternative flow: short snippets may be inserted directly before UI confirmation.

Result: Generated code appears inline, conversation history is saved.

Architectural Insight: Demonstrates event-driven coordination across chatThreadService, sendLLMMessageService, and editCodeService, aligning with Void's microkernel and client-server styles.

4.4 Use Case 2: Autocomplete While Typing

Goal: Provide real-time, AI-driven completions as the developer types.
Actors: Developer, VS Code Core (Editor and Language Server), Void Extension (editCodeService), AI Service Layer (sendLLMMessageService).

Main Flow (see Figure 5.2):

1. The Developer types in the Editor (Monaco).
2. The Language Server detects a completion trigger and invokes the Void Extension's editCodeService provider.
3. editCodeService extracts context (prefix and suffix) using voidPrefixAndSuffix() and sends a structured autocomplete request to sendLLMMessageService.
4. The AI Service Layer forwards the context to the model (modelCapabilities), which iteratively produces tokens (AI Completion loop).
5. The predicted tokens and confidence scores are returned to the Language Server, which formats and ranks suggestions.
6. The Editor displays the suggestion popup (Suggestion Display).

7. The Developer accepts or dismisses the completion; accepted snippets are committed to the buffer via VS Code's API.

Result: Low-latency, context-aware suggestions appear seamlessly as the developer types.

Architectural Insight: Mirrors the diagram flow (Editor -> Language Server -> AI Completion -> Suggestion Display) while grounding each step in Void's subsystems (editCodeService, sendLLMMessageService, modelCapabilities). It shows tight integration between Void's Extension Layer and VS Code's LSP mechanism and is consistent with its event-driven architecture.

---

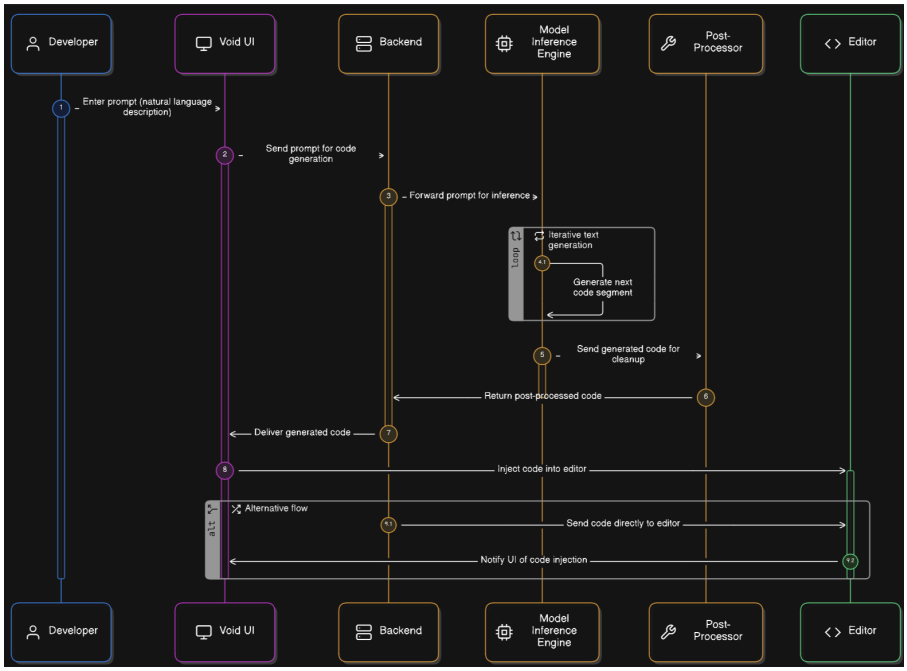
5. Sequence Diagrams for Use Cases

5.1 Use Case 1 – Chat-Based Code Generation

Developer -> Void UI -> AI Backend -> Response -> Editor Update.

This diagram shows how a developer interacts with the chat based code generation. The general flow of it is the developer enters a prompt into the Void UI, the void UI sends that to the backend, which processes the request in a loop (likely a transformer so iteratively generates text). After code is generated, the backend sends the data to the frontend, which injects the same code into the editor. An alternative would be sending the code to the editor then the UI, either would work from my point of view. Here's the diagram:

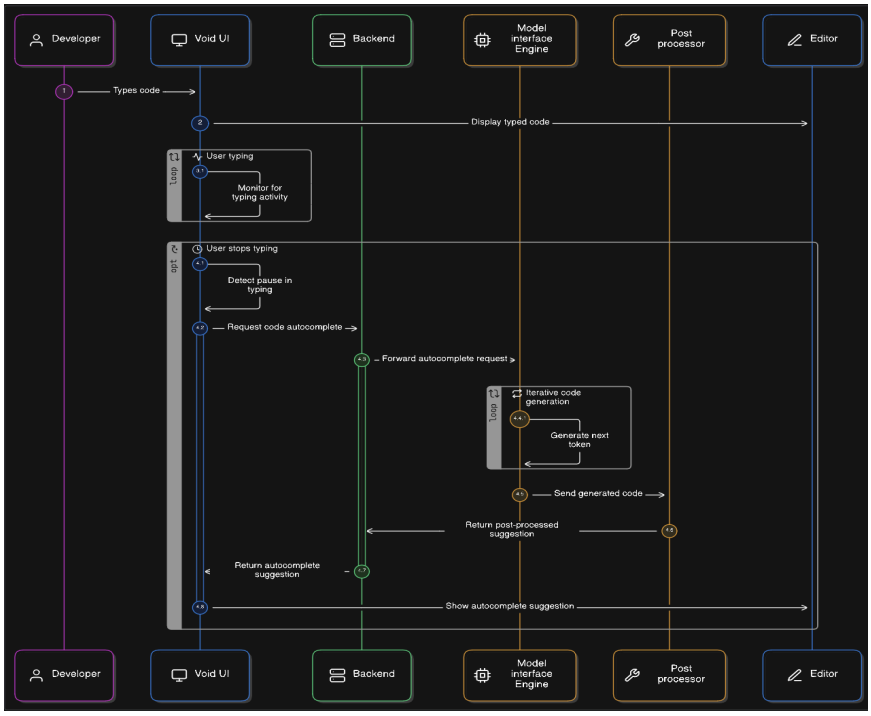
5.2 Use Case 2 – Autocomplete Request Flow



Editor -> Language Server -> AI Completion -> Suggestion Display.

This diagram shows how an editor would utilize the AI autocompletion feature. The flow of the diagram is that the autocompletion feature would trigger as they type into the editor, since it was not specified when I just left it vague. After the trigger is sent to autocomplete a request for autocompletion is sent to the language server which would send the code to the AI completion software, again likely a transformer that would iteratively create text so it loops there until it is done completing the text. It would then send completion suggestions to the language server along with confidence scores, the server would send suggestions to the editor, at which step the user can choose whether to keep the suggestions or not.

Here's the diagram:



---

6. External Interfaces

Void exchanges information with external interfaces through two main channels. The first one is through user components such as GUI components like the chat panel, completion list, and the settings view. Through these interfaces the system is able to collect text, documents, configuration preferences, and code fragments that it can use to complete the tasks it is assigned to do. The second channel involves Void communicating with AI model providers such as Open AI and Ollama via HTTP and WebSocket protocols using JSON-formatted messages. These exchanges contain user prompts, conversation context, and generated code completions or explanations returned by the AI services. Essentially, through user input, and communication with external AI model providers, Void is able to take in information from the user and generate responses according to what the user asks for.

---

7. Implications for Division of Responsibilities (Developers)

A key component of the system's development is the allocation of duties among developers. It enables us to share duties fairly and plan development work effectively, ensuring smooth work flow. The system is broken down into multiple subsystems. Each subsystem is owned by a dedicated developer team which is specialised in that area. E.g. a team of front end engineers could be in charge of developing and implementing the UI, while a team of AI/ML engineers would be responsible for the LLM message pipeline ensuring stable communication between the system and the external AI model. Essentially, by assigning ownership of different subsystems to specific teams, it reduces chances of conflict during integration, ensures smoothness in the development progress, and fosters accountability.

---

8. Conclusions & Lessons Learned

Analyzing Void's architecture reinforced that modular, layered design enables extensibility without sacrificing stability, particularly by separating UI, services, and AI providers into distinct layers that isolate concerns and allow parallel development. We learned that offloading computationally expensive operations like LLM inference to cloud providers is essential for maintaining IDE responsiveness, and that supporting multiple AI providers through a unified integration layer grants users freedom of choice while preserving architectural modularity. The iterative process of tracing data flows and revisiting diagrams revealed how asynchronous communication patterns directly impact user experience, and that architectural analysis requires constant interpretation as systems evolve. Moving forward, validating conceptual models against actual implementations and leveraging automated documentation tools would help maintain alignment between design intent and code as projects scale.

---

9. AI Collaboration Report

Our team used OpenAI GPT-5 (October 2025), Claude Sonnet 4 (inside Cursor), and ZRead.ai (GLM 4.5). GPT-5 helped us handle structured writing and cross referencing with sources, Sonnet 4 helped us with code analysis (inside the source code repositories), and ZRead.ai clarified implementation details from Void’s documentation. We chose these tools over others like Gemini 1.5 Pro and Mistral Medium for their stronger long context reasoning, writing quality, and overall technical accuracy.

9.1 Tasks Assigned to the AI Teammate

Task: Drafting and editing core sections, style and clarity revisions (including citations)
Reason: Strong summarization and tone consistency, uniform academic phrasing

Task: Organizing subsystem descriptions and tables
Reason: Efficient at repetitive, structured formatting

Task: Interpreting source code repository structure (Cursor + Sonnet 4)
Reason: Ideal for tracing dependencies in large codebases

Task: Clarifying implementation details from Void docs (ZRead.ai – GLM 4.5)
Reason: Supplemented understanding of system design and architecture

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

9.2 Interaction Protocol & Prompting Strategy

All members used GPT-5 collaboratively. Work was done in a shared Google Doc, where prompt drafts were iteratively reviewed by the team. Our typical prompt structure was as follows:

1. Provide context
2. Specify deliverable scope
3. Iterate on through group feedback.

Example prompt:

"Search the Void repository for where the LLMMessageService handles message streaming. Identify which files define its initialization and how it connects to the chatThreadService. Summarize the control flow between these components."

9.4 Validation & Quality Control

To ensure accuracy and originality, we followed a defined validation pipeline:

1. **Cross-checking** all technical statements with the official Void and VS Code documentation and our compiled reference document.
2. **Human proofreading** by at least two members for clarity and tone.
3. **Consistency review**: ensuring component names and terms matched across diagrams and text.
4. **Manual code confirmation**: validating Sonnet 4 outputs by looking at the cloned repositories ourselves.

9.5 Quantitative Contribution

Category	AI Contribution	Human Contribution
Drafting & structural organization	~40 %	60 %
Source interpretation (Cursor + Sonnet 4)	~15 %	85 %
Diagram preparation	~10 %	90 %

Editing & integration	~10 %	90 %
Overall Estimated Impact	**≈ 30 %**	**≈ 70 %**

9.6 Reflection on Human/AI Team Dynamics

Though integrating AI helped us to streamline drafting and improve organization, everything still required close human oversight. GPT-5 accelerated writing and clarity, while Cursor + Sonnet 4 deepened technical insight by helping us map out subsystem relationships. Having to design prompts and validate them added some overhead, and minor disagreements on when to trust AI outputs, especially when Sonnet 4 conflicted with documentation, really highlighted the need for human judgment. Overall, precise prompting, iterative review, and balanced human-AI collaboration enhanced efficiency and understanding. These are lessons we will apply in future projects within and beyond this course.

---

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/

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)

Naming Conventions

- camelCase
- UpperCamelCase
- UPPER_SNAKE_CASE

Abbreviations

- API: Application Programming Interface
- UI: User Interface
- UX: User Experience
- AI: Artificial Intelligence
- IDE: Integrated Development Environment
- VS Code: Visual Studio Code
- JSON: JavaScript Object Notation
- HTTP / HTTPS: Hypertext Transfer Protocol (Secure)
- NPM: Node Package Manager
- CFG: Configuration
- Msg: Message

The UI Components subsystem renders Void-specific React-based chat interfaces and WebView panels by consuming AI Services to display conversation threads and code suggestions, while also listening to Platform Services for theme updates and configuration changes to maintain visual consistency. The AI Services subsystem contains the core intelligence managing LLM orchestration, conversation state, inline code suggestions, and code transformations, receiving user requests from UI Components, querying Platform Services for model settings and API keys, and sending formatted requests to Provider Integration for actual AI model communication. The Provider Integration subsystem acts as an abstraction layer that receives standardized requests from AI Services and translates them into provider-specific API calls for different AI backends, handling custom endpoints for self-hosted models and streaming responses back to AI Services in real-time. The Extension System subsystem manages extension lifecycle and discovery by integrating with Platform Services for extension metadata and configuration, coordinating with Workbench Framework for extension host process management, and exposing Void-specific extension points that allow third-party extensions to access AI Services capabilities while maintaining VSCode compatibility. The Database/Files subsystem provides file system operations through multiple pluggable providers supporting disk, browser IndexedDB, and in-memory storage, with file watching capabilities that notify other subsystems of changes, while also managing persistent key-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.