Void IDE: A Conceptual Architecture Analysis

Mason Zhang
22yjz@queensu.ca

Jack Atkinson
22hrsd@queensu.ca

Ryan Sehgal
23sw2@queensu.ca

Jeremie Trepanier
jeremie.trepanier@queensu.ca

Duncan Mackinnon
duncan.mackinnon@queensu.ca

October 10, 2025

Abstract

In this report, we present our analysis of the conceptual architecture of Void IDE, a code editor built as a fork of Visual Studio Code with integrated AI features. The goal of this project was to understand how Void extends the traditional editor with AI capabilities and to create a clear conceptual model based on documentation and supporting sources. Research was conducted individually, then consolidated as a group to form a coherent architecture. Void IDE follows a layered design with three main layers: the NonML Foundation (Base Layer) for core editing and UI, the Bridge Components Layer for mediating between the editor and AI services, and the ML Integration Layer for AI processing and intelligent code assistance. We describe how information flows between layers and how this structure supports modularity and extensibility. This analysis provides a foundation for understanding Void IDE's design and guiding future exploration of its implementation and enhancements.

Contents

1 Introduction and Overview

As artificial intelligence becomes a bigger part of software development, the tools developers use are beginning to evolve. Today, code editors do more than just display and format code. They offer real-time suggestions, help developers write better code, and even answer questions through AI chat systems. One tool that reflects this shift is Void IDE, an open source code editor designed with AI features built directly into its core.

Void IDE is a fork of Visual Studio Code, one of the most popular open source editors. By starting with the strong foundation of VS Code, Void IDE brings in all the features developers are familiar with, such as editing, extensions, and file management, while extending it with smart AI capabilities. What sets Void apart is how it blends these new features into the editor in a thoughtful and well-structured way. Because it is open source, anyone can study how it works, suggest improvements, or adapt it for their own needs. This makes Void IDE a useful and important project for understanding how AI can be added to existing software in a clean and scalable way.

This report focuses on the conceptual architecture of Void IDE. That means it does not examine the source code or specific implementation details but instead looks at the system from a high-level design perspective. The goal is to understand how Void IDE is organized, how its major components interact, and how this structure supports the editor's functionality, especially its AI features. The architecture is broken into three main layers: a base layer for core editor functions, a bridge layer for coordinating with AI systems, and an AI integration layer for handling tasks like auto complete and chat. This layered approach allows the editor to stay organized, responsive, and easy to maintain or expand.

The rest of the report explains these layers in detail. It describes how information moves between them, how responsibilities are separated, and how the design supports modularity and extensibility. It also discusses how the system handles areas such as testing, performance, and evolution over time. Finally, the report includes two example use cases. One shows how code auto completion works, and another demonstrates the AI chat feature. These examples help show how the architecture supports real user workflows.

By the end of this report, readers should have a clear understanding of how Void IDE is structured on a conceptual level, and how that structure enables it to combine traditional code editing with modern AI support in a smooth and reliable way.

2 Architecture

2.1 Derivation Process

Most of our understanding of Void IDE came from the official documentation and other reliable sources. Since Void is a fork of Visual Studio Code, we first used the VS Code documentation to identify its underlying architectural style. After independent research, all members agreeing on the layered and server client approach, we collected information about the individual components that make up Void IDE from various sources. Once all the relevant details were gathered, we logically pieced the components together to form a sequence diagram and then a complete picture of the system's architecture though a box and arrow diagram.

2.2 Architecture Style

Void IDE is built as a fork of Visual Studio Code and follows a layered architecture style, dividing the system into three primary layers: the NonML Foundation (Base Layer), Bridge Components Layer, and ML Integration Layer. The Base Layer provides the core editing functionality inherited from VS Code, including text editing, file management, extension support, and model management through services that maintain text models and handle operations like undo and redo. React-based UI components in this layer render the editor, command bars, panels, and other interface elements. The Bridge Components Layer serves as an intermediary between the editor and AI services, extracting relevant code and context from the Base Layer, constructing prompts tailored for AI models, and integrating AI-generated suggestions back into the editor while maintaining data consistency. The ML Integration Layer handles AI processing, including connections to language models, request coordination, code editing, auto complete suggestions, conversational AI, and tool-based file operations. Data flows from the Base Layer to the Bridge Layer and then to the ML Layer for processing. Once AI responses are generated, data flows back through the Bridge Layer for integration and is applied to the editor in the Base Layer. This layered flow ensures responsiveness, clear separation of concerns, modularity, and extensibility, allowing new AI features or editor enhancements to be integrated with minimal disruption to the core system.

Void IDE also uses a Client–Server architecture to connect the editor with externalAI systems. In this model, Void IDE acts as the client running on the user's device, whileLLM servers handle the AI processing. The client collects context from the user's codeand sends it to the server through secure APIs. The server processes the request usinglarge language

models and sends back results such as code suggestions or chat responses.

This setup keeps the editor fast and lightweight while allowing powerful AI modelsto run on remote servers. It also makes it easier to update or scale AI features withoutchanging the client itself. The Client–Server design works alongside the layered structureto support smooth communication, distributed processing, and reliable AI integration.

2.3 Dependency

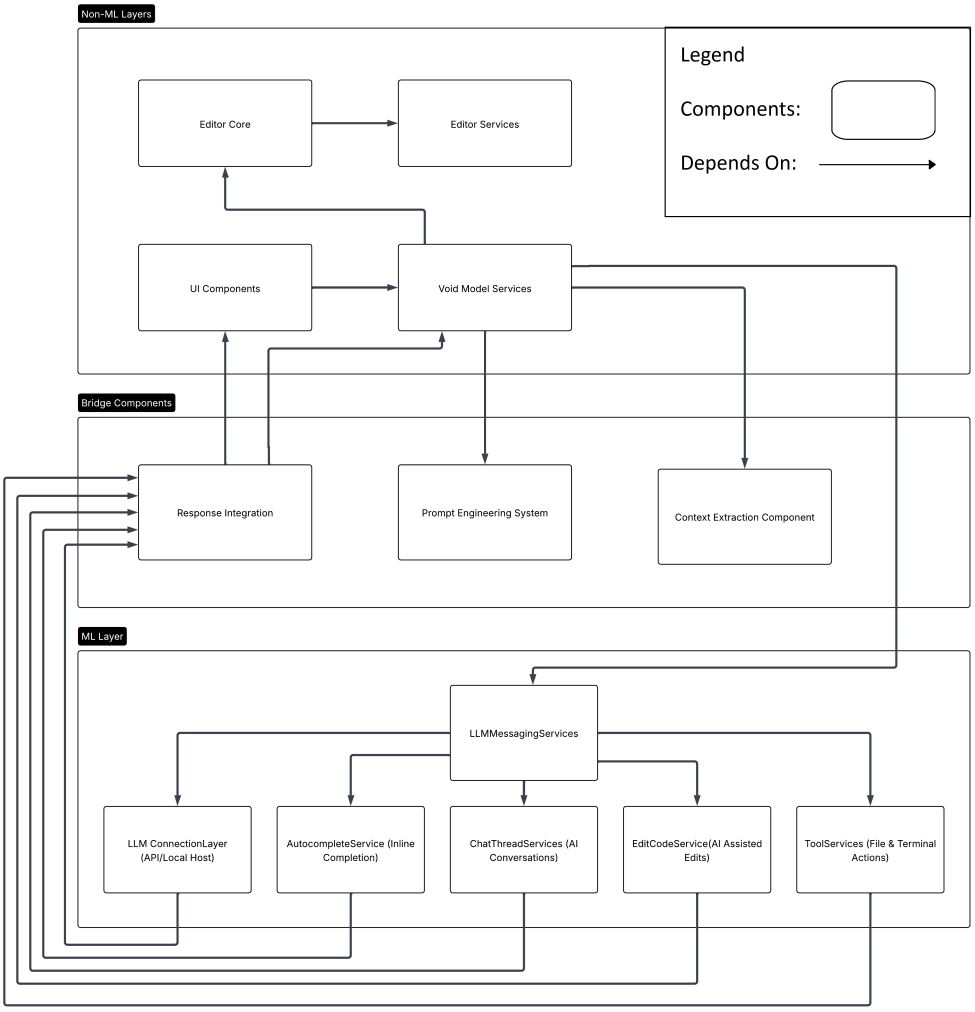

Figure 1: Dependency diagram showing relationships between Void IDE's major components and layers. Arrows indicate dependencies between subsystems.

2.4 Alternative Architecture

Alternative Architecture Another possible architecture for Void IDE is the Repository architecture. In this style,all parts of the system share one main data store called a repository. Each part of the program, such as the editor core, the AI services, and the bridge components, reads from and writes to this common store instead of talking directly to each other. This approach could make data more consistent and make it easier to manage shared information like prompts, context, and AI responses. It could also help with global features such as version tracking and undo or redo actions. However, the repository style could make the system more dependent on a single data store and harder to change safely. In comparison, the layered architecture keeps the parts of the system more separate and easier to extend with new AI features in the future.

2.5 NonML Foundation (Base Layer)

The NonML Foundation forms the core of Void IDE, providing all standard features expected from a modern code editor. It includes text editing, file handling, extension support, and the main user interface components. Built on the Visual Studio Code base, it establishes a stable foundation that higher layers rely on for consistent behavior and predictable performance. This layer is critical for the editor's responsiveness, as it handles synchronous operations where needed, while supporting asynchronous event handling for input, file management, and other background processes.

2.5.1 Key Components

- **Editor Core**: Ensures stable editing operations, including text selection, tokenization, and rendering. Its consistent behavior allows higher layers, such as AI features, to safely rely on the editor for accurate context and stable positioning.
- **UI Components**: Include panels, overlays, status bars, and chat panels. They provide clear feedback for AI-driven suggestions, display origin trails for changes, and include user controls for approval and interaction.
- **Editor Services**: Provide shared utilities such as filesystem and source control access, diagnostics, configuration, and storage. By exposing clean events and operations, these services simplify testing and allow other layers to access core functionality without introducing complexity.
- **Void Model Services**: Track AI model availability, constraints, and capabilities.

They inform other components of context limits, streaming behavior, and tool-use support, enabling safe integration of AI features.

The Base Layer focuses on reliability and performance-critical operations, ensuring that the editor can handle large files, multiple open tabs, or simultaneous user actions without slowing down.

2.6 Bridge Components Layer

The Bridge Components Layer links the NonML Foundation to the AI systems. It ensures that data flowing to and from the AI models is properly processed, contextualized, and integrated into the editor in a consistent and readable way. This layer also supports traceability, modularity, and reversibility, ensuring AI suggestions do not disrupt the user's workflow. It operates asynchronously to maintain responsiveness, allowing the editor to remain interactive while processing AI requests.

2.6.1 Key Components

- **Context Extraction**: Collects relevant code snippets, recent edits, project files, and errors, summarizing or trimming information when needed. Each item is tagged with its origin for traceability and reliability. This component feeds the Prompt Engineering System and can respond to updated AI requests as the conversation evolves.
- **Prompt Engineering System**: Converts user intent and extracted context into structured instructions for AI models. It ensures prompts remain within model limits, defines the tools that can be used, and maintains consistent behavior across different AI providers.
- **Response Integration**: Processes AI outputs into editor-ready results. It converts streamed responses into diffs, terminal commands, or messages and attaches origin information so users understand suggestions. It ensures safe application of changes through checkpoints and feeds results into EditCode Service, Tool Services, and the user interface.

The Bridge Layer supports clear control flow between the Base and ML Layers, allowing data to move efficiently while preserving consistency, reliability, and modularity.

2.7 ML Integration Layer

The ML Integration Layer powers all AI-assisted functionality within Void IDE. It handles connections to language models, coordinates message streaming, and manages operations like auto complete, chat, and AI-assisted code edits. This layer uses asynchronous processing to maintain performance and prevent blocking the main editor, while handling retries, timeouts, and fallback scenarios for reliability.

2.7.1 Key Components

- **LLM Connection Layer**: Manages provider-specific credentials, transport, and routing. It reports provider health and allows the system to switch models or fall back as needed, supporting the system's evolvability.
- **LLM Messaging Services**: Handle live communication with AI models, streaming responses, marking partial and final states, and managing tool calls. It pauses output during tool execution and resumes afterward, ensuring seamless integration of AI results.
- **Auto Complete Service**: Delivers real-time, inline code suggestions. It updates dynamically with streaming responses from AI models while maintaining responsiveness and reliability. Fallback to standard suggestions ensures continuity when AI is unavailable.
- **ChatThread Services**: Manage AI conversations, track context and conversation state, handle approvals, and maintain reversibility of edits. It integrates with other ML components to deliver smooth, safe AI interactions.
- **EditCode Service**: Applies AI-generated code edits safely, supporting inline or batch application with undo/redo functionality. Changes spanning multiple files are summarized before application to maintain clarity.
- **Tool Services**: Execute AI-requested file and terminal actions under controlled rules, returning structured results for integration into the editor or Bridge Layer.

The ML Layer is designed for high performance and concurrency, using asynchronous and streaming methods to maintain editor responsiveness. Its modular design allows new AI models or services to be added without changing the Base or Bridge Layers, ensuring future growth and maintainability.

2.8 System Flow and Extensibility

Information flows upward from the Base Layer through the Bridge Layer to the ML Integration Layer, and responses return downward to the editor. This structured flow ensures predictable interactions, clear separation of responsibilities, and consistent performance. Performance-critical components such as Auto Complete and Messaging Services are optimized for real-time feedback, while asynchronous execution prevents blocking the editor. Each layer's clear boundaries support independent testing, modular upgrades, and future expansion. New AI models can be integrated entirely within the ML Layer, while editor improvements can occur in the Base Layer without affecting AI functionality. The layered design balances reliability, responsiveness, and adaptability, allowing Void IDE to deliver intelligent assistance while remaining stable and extensible.

2.9 Testability

Void IDE is easy to test thanks to its layered design and clear separation of parts. Each layer—the Base, Bridge, and ML Integration—can be tested independently, helping isolate issues. The Base Layer can be checked with unit and UI tests, the Bridge Layer by simulating context extraction and response integration, and the ML Layer with mock AI responses. Its predictable asynchronous operations also make concurrency issues easier to detect. Overall, the system's structure and clear interfaces make it maintainable and reliable.

2.10 Concurrency

Void IDE uses Electron's dual-process model, with the Renderer handling the UI and the Main process managing I/O and provider calls. They communicate asynchronously, keeping the interface responsive. AI responses stream in, letting the UI update chat and code previews while the Main process continues work. Edits are applied safely, one file at a time, through undo/redo with checkpoints. ChatThreadServices tracks ongoing actions, limits rapid triggers, and checks risky operations. This setup prevents UI freezes, avoids conflicts, and keeps edits consistent.

2.11 Evolution

Void IDE began as a VS Code fork, with the NonML Foundation providing the editor core, services, and UI. An AI pipeline was added on top, including bridge components and the ML Integration Layer. Its evolution moved from auto complete and inline edits to chat workflows and diff-first edits with checkpoints. The LLM Connection Layer allows multiple

AI models without affecting the editor core. This layered design keeps Void modular and easy to extend with new features.

3 External Interfaces

Void IDE interacts with several external sources to support editing and AI features. Through its graphical interface, users type code, enter commands, and interact with AI, receiving suggestions, diagnostics, and messages in return.

The system reads and writes files from the local file system, allowing users to open, edit, and save their work. For AI features, Void IDE communicates with external large language models (LLMs) over the internet. It sends prompts containing code and context, and receives responses such as code completions, explanations, or edit suggestions. These messages are exchanged in structured formats but are handled behind the scenes to keep the experience smooth.

Void IDE can also connect to version control tools like Git to track code changes and manage commits. These external interfaces help the system provide intelligent support while staying closely connected to the developer's workflow.

4 Use Cases

4.1 Use Case 1: Code Auto Completion

The diagram above illustrates how the auto complete function of the Void IDE works. When the user requests auto complete, the Void Model Services first gather the raw code and send it to the Context Extraction component. Using the extracted context, a prompt is generated by the Prompt Engineering System. This specially designed prompt is sent via the LLM Messaging System to the Auto Complete Service, which utilizes a large language model (LLM) to generate a relevant code suggestion. The generated code suggestion is then previewed in the UI Component. If the user approves the suggestion, it is passed to the Editor Core, where the code is updated and integrated accordingly.

4.2 Use Case 2: AI Chat

The diagram above showcases how chatting with a large language model (LLM) functions in the Void IDE. When the user initiates a chat, the prompt is already provided by the user. However, contextual information is still needed to generate a relevant and accurate

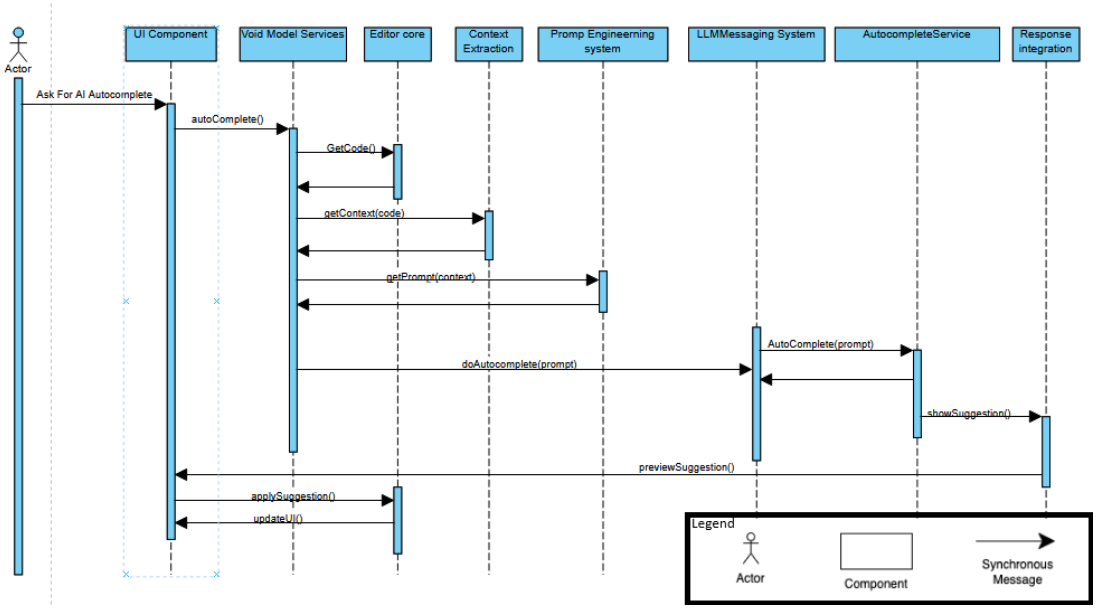

Figure 2: Sequence diagram illustrating the code auto completion use case.

response.To gather this, the Void Model Services first request the current code from the Editor Core. This code is then passed to the Context Extraction component, which analyzes it and generates a context. The context is returned to the Void Model Services, which now have both the user prompt and the relevant context. These are sent to the LLM Messaging System, which forwards the data to the Chat Thread Services. The LLM processes the request, and the response is formatted by the Response Integration component. Finally, the reply is displayed in the UI Component for the user to view.

5 Data Dictionary

- **Void IDE**: The open-source code editor based on Visual Studio Code, enhanced with AI-powered features.
- **Visual Studio Code (VS Code)**: The original open-source editor that serves as the foundation for Void IDE.
- **LLM (Large Language Model)**: External AI systems trained on large datasets to generate code suggestions, chat replies, and edits.
- **Base Layer**: The core part of the architecture responsible for basic editing functions, file handling, user interface, and extension support.
- **Bridge Layer**: The middle layer that connects the editor with AI services, handling

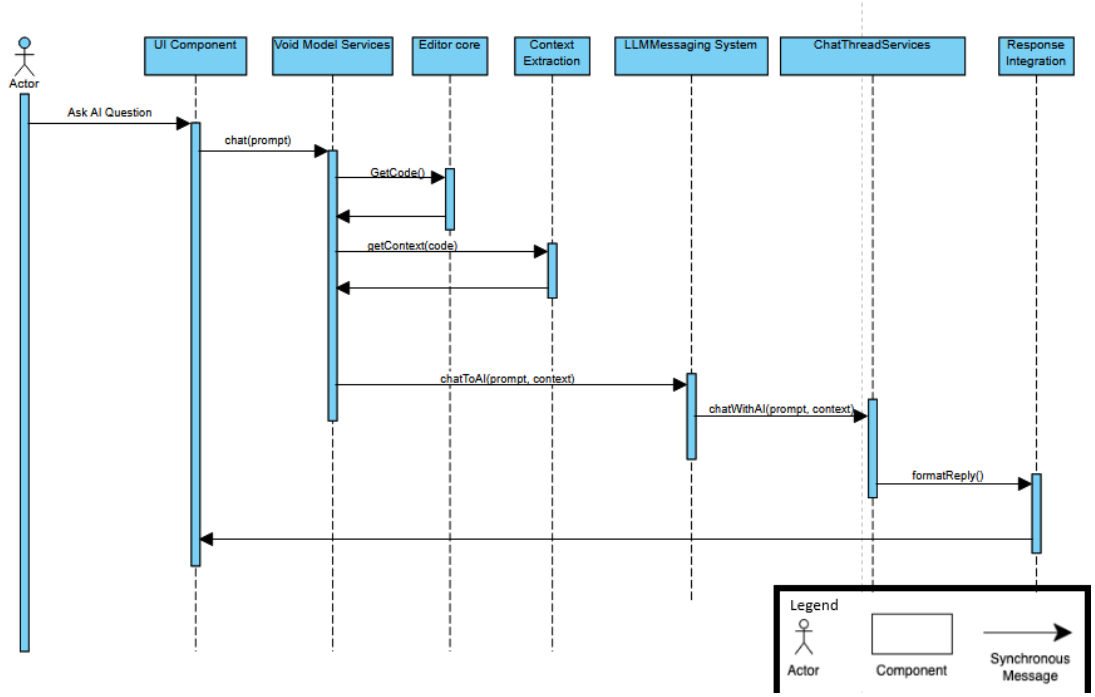

Figure 3: Sequence diagram illustrating the AI chat use case.

extraction of context, prompt creation, and integration of AI responses.

- **ML Integration Layer**: The layer managing all AI-related processing, including communication with LLMs and coordination of AI features such as auto complete and chat.
- **Context Extraction**: Component that collects relevant code snippets and project information to provide useful input for the AI.
- **Prompt Engineering**: Component responsible for transforming user inputs and code context into formatted prompts for AI models.
- **Auto Complete Service**: Feature that offers real-time AI-based code completion suggestions.
- **ChatThread Services**: Component that manages AI chat conversations and maintains context during interactions.
- **EditCode Service**: Component that safely applies AI-generated code changes, supporting undo/redo operations.
- **Tool Services**: Component that runs AI-triggered tools like file operations and ter-

minal commands under controlled conditions.

- **Renderer**: Electron process handling the user interface and frontend display.
- **Main Process**: Electron process managing background tasks such as file input/output and communication with AI providers.

6 Conclusions

This report outlines the conceptual architecture of Void IDE, an open-source editor that builds on Visual Studio Code by adding AI-powered features. The design is organized into three main layers: the Base Layer providing essential editing and UI functions; the Bridge Layer connecting the editor to AI services through context extraction and prompt preparation; and the ML Integration Layer handling all AI interactions, including communication with large language models.

This layered structure promotes modularity and clear separation of concerns, allowing each part to evolve independently while maintaining performance and stability. The system supports responsive user interactions even as AI features like auto complete and chat provide powerful coding assistance.

Being open source, Void IDE's architecture offers flexibility for further enhancements and integrations. It creates a strong foundation for future development, encouraging contributions from the community to expand AI capabilities and improve the overall coding experience.

7 Lessons Learned

1. **Balancing Research and Teamwork**: Combining individual research with group discussions helped us understand the system better. Some parts of the architecture were not clearly documented, so working together helped us fill in the gaps.
2. **Clear Task Division**: Assigning tasks early and checking each other's work improved productivity and ensured accuracy. This was especially important when using AI tools.
3. **Using AI Effectively**: AI tools helped speed up writing and idea generation. However, their suggestions needed careful checking. AI was helpful, but humans still made the final decisions.
4. **Understanding Layered Architecture**: Studying the Base, Bridge, and ML Integration layers helped us see how the system is organized. It clarified responsibilities,

data flow, and how parts interact.

5. **Technical and Team Challenges**: Some technical details were hard to understand at first. Regular team discussions and reviews helped us solve problems and improve our understanding.

Overall, this project improved our teamwork, understanding of software architecture, and ability to use AI tools in a structured way.

8 References

1. Void IDE Codebase Guide: `https://github.com/voideditor/void/blob/main/VOID_CODEBASE_GUIDE.md`
2. Medium Article on Void IDE: `https://medium.com/@adityakumar2001/voididethecomprehensive`
3. DeepWiki Architecture Guide: `https://deepwiki.com/voideditor/void/3coreservicesandarchi`
4. Void IDE GitHub Repository: `https://github.com/voideditor/void/tree/main`
5. VS Code Wiki: `https://github.com/microsoft/vscode/wiki/`
6. Void IDE Design Documentation: `https://github.com/voideditor/void/tree/main`
7. VS Code API Documentation: `https://code.visualstudio.com/api`
8. Layered Architecture Pattern. Garlan and Shaw.

AI Collaboration Report

AI Member Profile and Selection Process

We picked OpenAI GPT-5 September 2025 because it excels at creating well-organized writing exactly what we needed for reports alongside technical guides. It also handles requests from many different areas so we used it for everything from coming up with concepts to polishing how we describe systems, even checking writing. We chose it since it was clearer, more adaptable, easier to use than other tools like Gemini and DeepSeek which we tested and they most of the time took too long when they didn't need to and provided inaccurate information more often. It also gave way to quicker responses and the ability to alter how long it spends thinking on answers to be more careful when we need it to be.

Tasks Assigned to the AI Teammate

Our team handed off specific jobs to an AI helper. We picked these duties – things needing quickness, precision, or lots of options – since that's where the AI shines. However, we still oversaw and carefully judged results, then made the calls in the end.

- Getting the first versions of documents written basing it off of a previously written assignment in order to get the format and overall concept fully down.
- It gave us somewhere to begin with things like how the system works, why it was built this way, likewise evaluating different options.
- Polishing descriptions, making them clearer.
- The AI polished them, smoothing out wording yet keeping the original ideas intact. It was a helpful extra eye to catch anything we missed. Thinking up ways to check if something works. We considered potential problems an AI flagged, subsequently confirming those scenarios aligned with what the project needed.

Interaction Protocol and Prompting Strategy

We had one member be designated for working with the AI but everyone used it here and there in order to get work that needed to be done faster. Then they shared the information with the designated member. We focused on clear instructions, background details, and limitations. To get initial documents, we'd ask something like this: Give us a summary for how the subsystems would work. And we would give it a link to the GitHub folder. The initial request went through several changes. We set length restrictions, told it to take a lot more time when coming up with answers and to double check them with the resources provided. Through repeated adjustments like these, the results got better because they aligned more closely with what we needed. It first created a nice organized chart listing out all of the subsystems that were written into the handbook as well as their purposes and interactions with other elements. We compared it to our sequence diagram that we made and we noticed that the AI had missed a few components. We explained what it did wrong and it was helpful at helping us compare what we thought the components functions were compared to the AI's view.

Validation and Quality Control Procedures

Because machine-made responses sometimes miss the mark we carefully checked every bit of text from that system. Here's how we did it:

- Verify statements by comparing them to what Void's code guidebook, class materials, or some of the websites that were linked in the projects page. Checking summaries against the actual sources so they don't change what was originally said.
- The team checked each other's work – drafts created with assistance received a look over from at least one person who ensured things were technically sound, felt right, and also matched what we aimed to do.
- We checked everything by hand, step by step, to be certain nothing created by a machine slipped through unnoticed. Relying on those results straight away? Not good enough nor did we think it was right.
- We went through each component in the guidebook in order to ensure that the AI's description of it was perfect and improved any error or made up answer that it gave.

Quantitative Contribution to Final Deliverable

We would say overall the AI did 20% of our work in total just like a normal team member would have done. For generating concepts: roughly 30% of the work. The first bit of writing for some of the sections is about 15% complete. A little polish about 5% of the work goes toward making sure everything reads smoothly. People on our team did most of the work – about 80%, covering everything from choices to tech stuff, visuals, then polishing it up. Artificial intelligence helped speed things along, yet humans always led the way, making calls and ensuring quality.

Reflection on Human–AI Team Dynamics

Working alongside an AI changed how our team operated. Initially, it meant quicker first drafts consequently, we could dedicate ourselves to complex code and crucial technical choices. Moreover, brainstorming sessions blossomed; the AI offered ideas beyond what we typically conceived. Yet, using artificial intelligence wasn't without its snags. Occasionally, outputs just went on too long so considerable editing became necessary. Sometimes, this bothered everyone, requiring checks and improvements. So, artificial intelligence both streamlined things yet also generated additional duties. Whenever the AI wasn't able to find an answer it would make up something close or similar to it which made the fact checking more difficult but all the more vital. In order to help getting specific with requests really helps. Well defined instructions yield far better outcomes. Checking things over matters a lot. Ultimately, having an AI assist us boosted how much we got done, made our writing better, yet still needed someone watching closely.