Assignment 1: Conceptual Architecture of the Void Editor

October 10th 2025

Ibrahim Kettaneh (ibrahim.kettaneh@queensu.ca)
Sophie Liang (22whr@queensu.ca)
Noelle Morley (25gdb@queensu.ca)
Annika Tran (23LM5@queensu.ca)
Nicole Wu(22ll20@queensu.ca)
Joshua Zheng (23SBN1@queensu.ca)

Table of Contents

1. Abstract

This report concerns a conceptual analysis of the Void IDE, which is an open-source, AI-integrated fork of Microsoft's VS Code. Void expands on VS Code's architecture to enable seamless collaboration between software developers and AI agents within the same development environment. By usage of both a layered and implicit invocation communication style, Void achieves both modularity and flexibility, allowing AI-driven services to interact dynamically and responsively with a core IDE without compromising the security of the system.

The architecture was found to contain three primary layers: a Non-ML Foundation layer, which serves as the basic IDE editor, the bridge layer, which contains a series of components mediating the deterministic editor features, and the probabilistic AI features and the ML integration layer, which connects the editor to the AI agents through modular services. This design enables extensibility, concurrency, and security, offering developers freedom and customizability in integrating an AI model.

This report further examines Void's derivation from VS Code's source, outlines its core components, contains diagrams with key control flows, and presents use cases including Quick Fix automation and provider onboarding. Findings suggest that Void's architecture effectively balanced both AI augmentation and user agency, setting a precedent for potential future IDEs that integrate intelligent systems while maintaining transparency, performance, and developer trust.

2. Introduction and Overview

Void is a free-to-use, open-source AI code editor that is a fork of Microsoft's Visual Studio Code. It was developed in response to the growing demand for a transparent and customizable AI assistant in a development environment . It was modified to incorporate and designed to further integrate AI agents into a code base and project and allows users to select their own AI agent and have that agent perform edits and transform the code based on prompts and suggestions.

Void utilises an open source model, where the entire codebase is accessible and forkable online, and allows users to integrate their own chosen AI model and tokens to further tailor their development experience.

Having open source allows for more transparency for developers to inspect the code for a better understanding of the application and contribute to its development. Void also ensures that users have more control over their intellectual property by local processing information and avoiding information sent to private backend compared to other AI applications that usually put users' data into external servers to process for a more secure protection of users' privacy. Void also allows choices of any AI agent, where users can choose any model that best aligns with their goals and needs.

In the architecture style, it was decided that Void utilizes a stricter layered architecture and implicit invocation/pub sub. From each layer, each component was analysed for interactions on a layered and pub sub configuration. It can be concluded that there are three layers: the foundational core, the bridge layer, and the machine learning (ML) integration layer. By combining both layered and implicit invocation, Void would allow for both modularity and flexibility, which is key in terms of its goals of integrating AI into an IDE's workflow.

Driven by the issue that all major AI IDEs are closed source, the founders of Void decided something needed to change. The developers responsible for the implementation of Void are split into two main subteams: the core team and a network of contributors. The core team handles the decision making and overall direction of architecture as well as the bulk of the commits. The contributors, who were introduced later, focus primarily on improving user experience and refining existing features.

The layered architecture style of Void supports this decision by enabling the developers to work on different layers and components independently without interfering with each other. For example, a frontend team can focus on developing the UI components, while the backend team simultaneously works on components like the LLM connector and prompt engineering system. Despite this, there was a clear hierarchy in the delegation of tasks. According to the Void Roadmap found on its Github, the responsibilities are organized into five main categories, running from Improvements A, the most urgent and impactful, to Backlog, the lower priority changes to be addressed in the future.

The pub sub architecture style further reinforces the asynchronous yet collaborative workflow. Due to components communicating through publishing and subscribing to events, event triggers only affect components subscribed to the event, rather than every active module.

Overall, Void's organizational structure promotes collaboration and scalability by leveraging design patterns that support parallel development and maintain adaptability as new components and developers are introduced.

3. Derivation Process

The research process began with examining Void's GitHub source code page. An initial review of the project's file structure, configuration files and documentation provided crucial insight into the overall concrete organisation and functionality of the software. Then, using the codebase guide document in the repository along with external sources such as Kumar's and Ahwan's respective articles on Void, we deduced the architecture's layered style, specifically how the architecture contained several layers emerging upwards from its VS Code origins.

A more focused analysis of the source code was conducted, looking at individual code files and the relationship between them. This along with the codebase guide helped us derive the key components of the architecture. At this stage we also identified pub/sub as another style that inspired the architecture of Void.

With key components determined we brainstormed possible use cases, eventually landing on Quick Fix and provider set up as the two cases most relevant to the components selected. With the use cases determined, sequence diagrams were created for each case using what was observed from our in-depth analysis of the codebase as a whole as well as the architecture styles determined as reference.

Finally, using the sequence diagrams, we created the box and arrow diagram as a high-level representation of the two diagrams.

4. Architecture

4.1 Styles

4.1.1 Layered Architecture

Void's architecture is organized into three distinct layers: the foundational core, the bridge layer, and the machine learning (ML) integration layer. These layers combine the robustness and conciseness of a typical IDE with the dynamic abilities of ML components to deliver a new modern coding experience. Layered separation also facilitates concurrency. Since layers are loosely coupled, tasks can be processed independently. For example, the foundation layer could be running editor actions while ML features are run by the ML layer.

4.1.1.1 Non-ML Foundation Layer

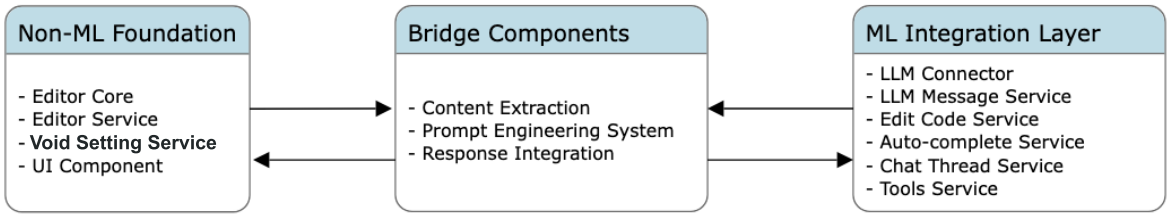
The lowest layer is the non-ML core, which contains the majority of the central code from the original VS Code code. It is mostly unmodified and serves as a basis for several essential functions. However, there is one key addition: the Void Model Service. This component gives ML models increased access to the user's codebase.

4.1.1.2 Bridge Components Layer

The bridge exists to connect the ML Integration Layer and the IDE Foundation. It contains components that mediate between human actions and machine-derived operations, ensuring that AI agents behave in accordance with the user's intent and that communication between the two layers remains contextually accurate and efficient. The Bridge Layer also serves as a translation and safety boundary between deterministic editor operations and probabilistic machine learning responses.

4.1.1.3 Machine Learning Integration Layer

The third layer is known as the ML Integration layer and contains key functionality that binds the ML functionality of the program directly into the workflow of an end user. This contains the code that contains interactions between the users' code and AI agents.



Three-layer architecture of Void, with each box showing its key components: the Non-ML Foundation, the Bridge Components , and the ML Integration Layer

4.1.2 Implication Invocation (Pub/Sub) Style

Alongside the layered stack, Void relies on VS Code's built-in core event bus to keep components in sync without binding them tightly together. Services publish notifications whenever their state changes, and any interested collaborator subscribes to those events rather than calling the service directly.

For example, the chat thread service listens for streaming tokens emitted by the LLM message channel, diff zones raise events that drive the approval UI, and provider settings broadcasts prompt-dependent services to refresh their configuration. This implicit invocation backbone makes it straightforward to bolt on new capabilities-whether that is a metrics hook, a tooling

endpoint, or an additional UI surface-by wiring fresh listeners into existing channels instead of reworking every layer in the system. This style of architecture also has decoupled components allowing for concurrency.

4.1.3 Alternative Styles

4.1.3.1 Client/Server Architecture

In a client/server style, Void's IDE would be the client that would send codes and other resources to ML functionalities, which act as remote servers. This can be beneficial as it reduces resource demands on the user's local device and facilitates scaling of backend AI services. On the other hand, sending commands to remote servers increases latency, which makes AI integration less responsive. Furthermore, sending users' codes to remote servers poses security concerns.

4.1.3.2 Object-Oriented Architecture

If Void is designed with an object-oriented approach as opposed to pub/sub, all functionality would be encapsulated in classes and objects. Some benefits to this would be that it promotes reusing and simplifies unit testing. However, compared to pub/sub, this style will lead to tight coupling between editor, ML, and bridge logic, making extensibility more difficult.

4.2 Components

4.2.1 Editor Core

This component serves as a backbone for the non-ML foundation layer. This component is responsible for most of the functionality of the IDE as a piece of software. Most of the editor core is code retained from the original VS Code IDE, and contains very rudimentary operations such as inputting and displaying text, as well as essential code for starting up the software.

4.2.2 Editor Services

Another component in the non-ML foundation layer. A natural extension of the editor core, it consists of a series of functions related to editor core, such as copy-paste, search, and syntax awareness. These functions become extremely useful for later ML operations to take place, such as inserting machine-generated code into an existing source.

4.2.3 UI Component

A component encompassing all UI functionality that also exists in the non-ML foundation layer. It is a modified version of the original VS Code UI with new utilities such as text boxes and prompts that fit into an ML-enriched environment. This component is responsible for drawing the actual software on the screen.

4.2.4 Void Setting Service

The Void Setting Service acts as a centralized configuration manager within the foundational layer of the architecture. It governs how user preferences, model credentials, and runtime parameters are stored, retrieved and propagated throughout the system. This services provides a secure place for user credentials to be access across the system.

4.2.5 Content Extraction Subsystem

A component in the bridge layer that continuously analyses the user's workspace and editing state to determine which parts of the codebase are most relevant for AI updates and edits. It selectively gathers snippets, dependencies, and semantic cues to ensure the AI receives significant enough context without overwhelming the model or exceeding token limits.

4.2.6 Prompt Engineering System

Building upon the content extraction subsystem, the prompt engineering system allows for the creation of dynamically constructed prompts based on both user input and extracted context. This system leverages templates, heuristics, and model-specific optimisations to ensure that each AI query is both well-formed and effective. It acts as a linguistic and logical bridge between user intent and machine interpretation.

4.2.7 Response Integration Subsystem

Once an AI response is received, the response integration subsystem processes and merges it back into the editor. It validates syntax, detects conflicts, and ensures that the modifications are compatible with the IDE's version control and undo mechanisms. This makes a seamless workflow where AI-suggested edits are just as smooth as user edits. It is another component in the bridge layer.

4.2.8 LLM Connector

The primary component in the ML layer. It is what enables specific AI agents to be engaged. This component contains multiple backends to separate APIs. This service handles authentication, request throttling, error handling, and many miscellaneous tasks involving direct calls to an ML service.

4.2.9 LLM Message Service

The data coming to and from the MLs themselves is routed by the LLM Message service, which acts as a source of directions between the editor, the bridge, and external AI systems. This service ensures that prompts, responses, and transformation requests are all handled consistently and accurately, and that AI agents operate within their authorisation limits.

4.2.10 Edit Code Service

The Edit Code Service utilises this messaging framework to implement live AI-assisted editing. It handles functions such as streaming code suggestions, code generation, and patch implementation. Functions in this component also attempt to maintain the editor's original text and style. This feature is a potential bottleneck and is designed with this in mind, as this component interacts in real time with the user.

4.2.11 Autocomplete Service

An advanced line completion agent. It merges outputs from the MLs with contextual information from the source, allowing for fast and instantaneous suggestions.

4.2.12 Chat Thread Service

Conversational interaction with the AI agents is managed by the Chat Thread Service, which allows users to maintain ongoing, multi-turn discussions with an AI agent. Contextual relevance is stored within this, so the AI knows what the user is focusing on at this moment. The human-level, plain language interface allows users to directly administer commands to the ML services.

4.2.13 Tools Service

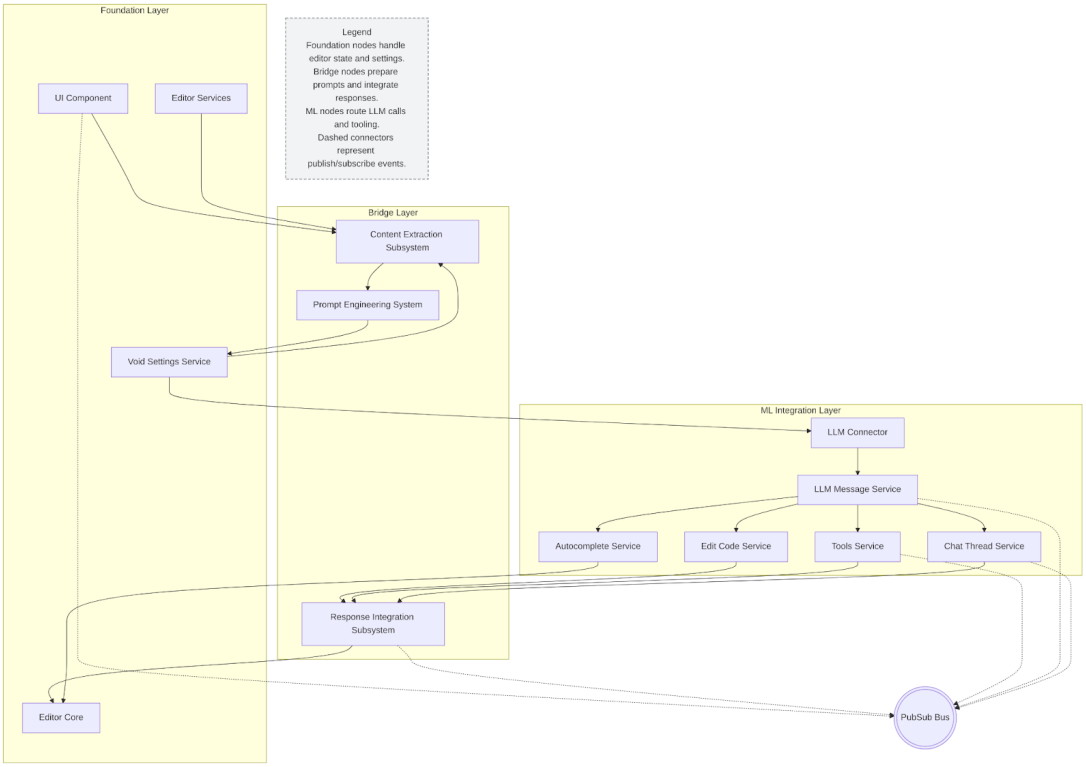
All other components discussed in the ML integration layer are complemented by the Tools Service, which gives AI agents an interface that allows for exposing system-level operations such as file manipulation, searching, and command execution with crucial security constraints. This suite of agentic tools allows for enhanced abilities of the AI agents with strict security, ensuring user trust and system stability.

5. Diagrams

5.1 Box and Arrow Diagram

The diagram presents the architecture of Void, showing major components which are organized into three layers. The Foundation Layer includes the UI Component and Editor Services, which handle user interactions, editor state, and settings, along with the Editor Core responsible for basic editing operations and the Void Settings Service, which manages configuration and credentials. The Bridge Layer features the Content Extraction Subsystem and Prompt Engineering System, which process code context and build prompts for AI requests, as well as the Response Integration Subsystem, which validates and merges AI-generated changes. The ML

Integration Layer consists of the LLM Connector and LLM Message Service, which facilitate communication with external AI models, and services such as Autocomplete, Edit Code, Tools, and Chat Thread Service, each delivering specialized AI-powered features. The diagram uses lines and arrows to show the flow of data and control between components, with dashed connectors representing event-driven communication via the PubSub Bus, enabling asynchronous updates and loose coupling throughout the system. This layout highlights the separation of responsibilities, modular design, and event-driven architecture central to Void.



6. External Interfaces

Void relies on a small set of external channels. The table below summarises each interface and the components that drive it.

Interface	Direction	How it is used within the architecture
Provider HTTP APIs (OpenAI, Ollama, Azure, Vertex, Bedrock, etc.)	Outbound	The LLM Connector and provider adapters in the ML integration layer construct requests and stream responses

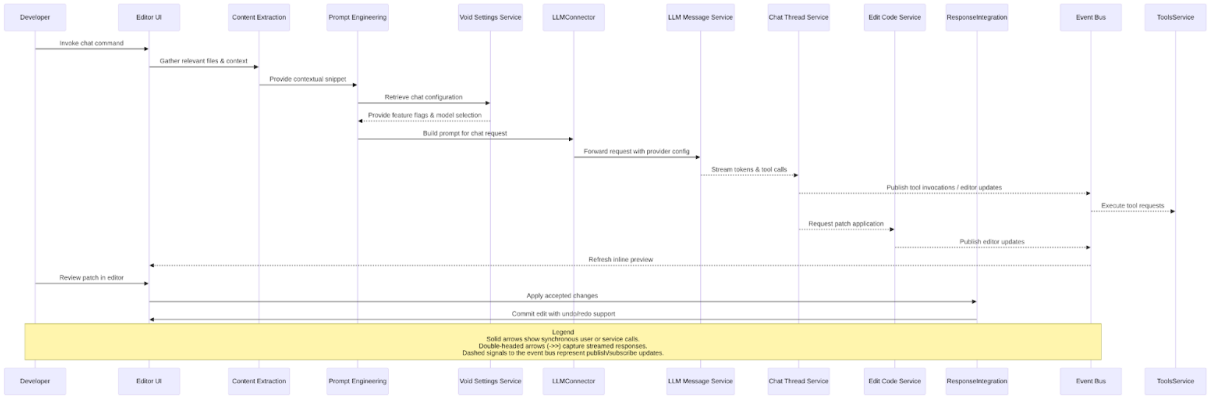
		via the LLM Message Service.
MCP servers	Bidirectional	The Tools Service and Chat Thread Service subscribe to MCP events, exposing tool metadata to prompts and returning results over the event bus.
VS Code workbench services	Bidirectional	Foundation-layer editor core and services consume VS Code singletons for storage, terminal, metrics, and undo/redo, publishing updates for the bridge and ML components.
Local filesystem	Bidirectional	The Void Settings Service and Tools Service validate URIs, read/write project files, and run workspace-scoped commands with appropriate guards.
GitHub release feeds	Outbound	The Void Update Service checks release metadata when automatic updates are unavailable and notifies the UI through publish-subscribe events.

7. Use Cases

The following scenarios illustrate how the layered architecture and implicit invocation provide end-user workflows:

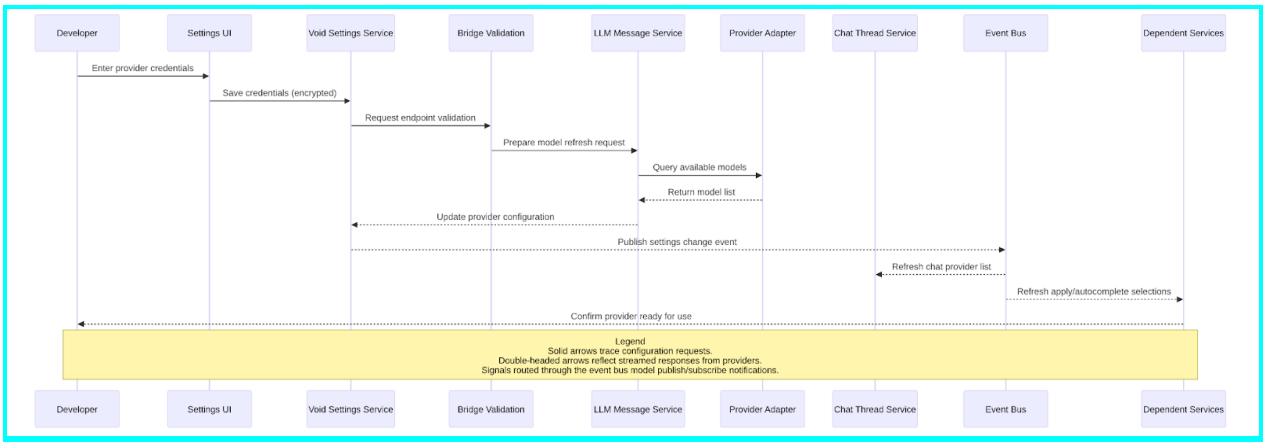
7.1 Ctrl+K Quick Fix Workflow

Quick Fix begins at the editor core when a developer presses Ctrl+K. Bridge services gather the relevant code, build a prompt, and hand it to the LLM Connector in the ML integration layer. The Edit Code Service streams candidate patches and raises diff events; frontend listeners subscribe to those events to render inline previews. Accepting a change routes the edit back through the editor core, showing collaboration between the layered components and the event bus.



7.2 Provider Onboarding & Model Refresh

Provider setup starts with the Void Settings Service storing credentials in the foundation layer. Bridge utilities validate endpoints and pass requests to provider adapters in the ML integration layer. The LLM Message Service publishes updated model lists, and settings events notify dependent services-chat, apply, and autocomplete so they refresh automatically. The flow shows how configuration propagates across the layers while publish/subscribe keeps components in sync.



8. Conclusion

From the research conducted on Void, it becomes apparent that it uses a layered architectural style complemented by implicit invocation communication. The layered structure brings the system into separate layers with clearly defined responsibilities. Combined with the anonymous messaging strength of the pub/sub model, this design promotes scalability and concurrency across different system components

The implementation of AI functionality within an existing IDE framework demonstrates how modern software can embed ML systems without compromising user control or system stability. By retaining both extensibility and familiarity that users have with VS Code, Void successfully merges traditional software engineering paradigms with emerging AI driven development.

The architectural analysis of Void provides many valuable insights; however, there are still limitations in the reported findings. Since the application is still new and developing, there is insufficient documentation on the mechanisms and system interactions. This limits the report's ability to draw definite conclusions as much of the analysis was inferred by studying code. When evidence was scarce, the group primarily focused on the source code, though conceptual architecture often differs from the concrete architecture. Additionally, with the reliance on open-source contributors, there are inconsistencies with implementation styles across modules, making it challenging to accurately grasp the overall architecture. To improve the quality of future analyses, it will be beneficial to obtain input from the core developers and contributors or trace the system's event flow with external monitoring tools to gain a clearer understanding of the system's architecture and behaviour.

Although still a new platform, Void's community based development model allows it to be a significant step forward in AI assisted programming. As the technology continues to progress, Void will play an influential role in shaping how developers and AI systems work together to develop future software.

9. Lessons Learned

Through analysis of Void's conceptual architecture, our team gained many valuable insights into how modern software systems are evolving to accommodate artificial intelligence systems whilst maintaining end user control and system stability. Void represents a new class of development tools that aim to cooperate with AI, rather than replacing the programmer.

Void demonstrates the importance of modularity in its conceptual architecture; the software was built from the ground up to be continuously improved and extended. By expanding upon the established foundation of VS Code, Void's developers were able to integrate AI driven components without disrupting the decades old, traditional coding workflow. This approach highlights how leveraging existing frameworks can allow for ideally increased productivity while preserving the user's trust and the system's stability.

One significant factor contributing to Void's success as a system is the emphasis on architectural modularity. From using a layered architecture with implicit invocation elements, each layer gained specific and distinct responsibilities: the foundation brings deterministic, human driven IDE services whereas the ML integration layer brings in probabilistic elements from AI services and the bridge merges these two systems together in a way that is efficient and seamless for the

end user. The pub/sub model ensures these components remain loosely coupled, allowing new tools and AI services to be added or updated without requiring a redesign of the entire system. The design mirrors a growing trend in software architecture towards concurrent systems, which promote flexibility and scalability, as well as ease of maintenance in the long term.

Void's employment of open source development allowed for an international team of developers to communally add and expand the project. This collaborative structure encourages diversity in expertise and ideas, ultimately leading to a more robust platform. It also sets a precedent for modern software systems to be developed transparently and with heavy involvement of the end users.

Appendix A: Data Dictionary

AI Agent: an autonomous entity that perceives environment and takes actions to achieve goals
Autocomplete Service: provides real-time code suggestions to the user

Approval UI: Interface for reviewing and approving code changes.

Bridge Component Layer: mediator between non-ML foundation layer and ML integration layer

Chat Thread Service: manages the conversation between user and AI agent

Content Extraction Subsystem: collects the most relevant portions of the codebase to allow the

AI to generate accurate and meaningful outputs

Diff Zone: UI area showing code changes and raising events for approval.

Edit Code Service: extends the Editor Core with additional functions such as copy-paste, search, and syntax awareness. These support both manual editing and later ML-driven operations.

Editor Core: Encompass most of the IDE's basic non-ML functionalities

Event Bus: Centralized system for broadcasting and subscribing to events, enabling loose coupling.

Implication Invocation Style (Pub-Sub): an architecture style where the components (subscribers) interact through calls; the message is broadcasted through a central system and the subscribers only receive the events they are interested in

Integrated Development Environment (IDE): application that combines essential tools for software development

Layered Architecture Style: an architecture style where layers provide services for the layer above, and act as clients to the one below, creating a hierarchy.

LLM Connector: acts like a gateway and manager for all interactions with AI services

LLM Message Service: acts as the messaging and orchestration layer that handles request and responses between internal IDE services and the external AI backend

Machine Learning Integration Layer: third layer of Void's architecture; enables AI-powered functionality to be embedded directly into the developer workflow

Non-ML Foundation Layer: lowest layer of Void's architecture; contains traditional IDE infrastructure from VS Code with an added component that allows ML models to access user codebase

Prompt Engineering System: prepares and optimizes requests sent to the AI system

Response Integration Subsystem: component within the bridge layer of Void editor; how AI-generated responses are validated, processed, and incorporated back into the editor

Setting Service: Centralised configuration manager that securely stores and broadcasts user preferences and model settings across all system layers.

Source Code Editor: an interface where code is written

Tool Service: Provides AI Agents with secure interface to perform system-level operations

User Interface (UI): Components that make up the space for user to interact with program

Workspace: User's current project files and resources.

Appendix B: Naming Convention

AI: artificial intelligence

API: Application Programming Interface

IDE: Integrated development environment

LLM: Large Language Model

ML: machine learning

UI: user interface

VS Code: Visual Studio Code

Appendix C: AI Usage Report

AI Member Profile and Selection Process

Models were selected based on accessibility (in terms of price per credit) and ease of use. Based on these criteria, two models were shortlisted due to: GPT-4 (May 2024), and Deepseek 8 (July

2024). Both models are free at the time of writing and each provide simple, easy to use chat based interfaces. This approach of chat based interaction allowed for quick responses that come naturally to human authors, and allowed us to interact with an AI teammate as if it were someone in a messaging service.

We ultimately chose GPT-4 as the model is more familiar to most members of our team.

Tasks Assigned to the AI Teammate

AI was seen as too probabilistic to use for anything research-based due to the huge amounts of inaccuracies and false information in AI-generated text. Therefore, AI usage was strictly limited to proofreading and as an advanced thesaurus, finding ways to rephrase and reorganise information in order to more effectively portray a point or concept.

Interaction Protocol and Prompting Strategy

When editing the project documents collaboratively, each team member had a copy of GPT-4 open in ChatGPT to reference from. This meant that if a team member needed to ask a question or was unsure of anything, the AI model could quickly respond. This allowed for each member to have a customized instance where they could ask for information specific to what they were currently working on.

Validation and Quality Control Procedures

Whenever AI was used to proofread a section, team members would prompt an AI model to suggest improvements, rather than produce a corrected model. This allowed team members to review the parity of these potential changes compared to the point already being made, to ensure that they do not erase any information or significantly modify tone and context. This process ensured no one was directly submitting text directly from the AI model, and using it to bolster our human expertise with its knowledge of grammar and vocabulary.

Quantitative Contribution to Final Deliverable

Due to our denial of using directly AI generated text into the report, giving a quantitative contribution is difficult to estimate. Giving a value for the text generated by AI could come out to lower than 10% in terms of word count, despite how definitive concepts in the text would most likely be identical, albeit with worse structures. All of the research and derivation process was completed by the human team members. Similarly, all diagrams were produced by human team

Reflection on Human-AI Team Dynamics

Throughout the research and report processes, the use of AI introduced many inconsistencies. While the team did not put any AI-generated content directly into the paragraphs, some sections remained more heavily influenced by AI suggestions while others remained purely human-written. Additionally, since each team member was interacting with their own version of GPT-4 and copying different sections of the report, their model had uneven knowledge on the textual content of the essay so far. This created disparities in style, detail, and emphasis of specific points.

The AI teammate occasionally provided useful suggestions or directions, but when the information was cross-referenced, it was often inaccurate or misinterpreted. As a result, the team refrained from relying heavily on AI, as it caused more confusion and time than it was worth.

In the end, an AI teammate proved to be more of a hindrance than an actual productive team member.

Works Cited

Ahwan, Abdul Aziz. "Void Editor: The Open-Source AI Code Editor Revolutionizing Developer Workflows." *Abdul Aziz Ahwan*, 23 June 2025, www.abdulazizahwan.com/2025/06/void-editor-the-open-source-ai-code-editor-revolutionizing-developer-workflow.html. Accessed 10 Oct. 2025.

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

"About Contributors | Voideditor/Void | Zread." *Zread*, 2025, https://zread.ai/voideditor/void/7-about-contributors.

"Overview | Voideditor/Void | Zread." *Zread*, 2025, zread.ai/voideditor/void/9-architecture-overview.

"Void/VOID_CODEBASE_GUIDE.md at Main · Voideditor/Void." *GitHub*, 2024, github.com/voideditor/void/blob/main/VOID_CODEBASE_GUIDE.md. Accessed 10 Oct. 2025.

"VoidEditor." *Github.com*, 2025, github.com/voideditor/void. Accessed 10 Oct. 2025.

Pareles, Andrew. "Void: The open source Cursor alternative" *Y combinator*, 16 Sep 2024,

https://www.ycombinator.com/launches/Lrh-void-the-open-source-cursor-alternative.