Queen's University - Fall 2025

CISC 322

Assignment 2

Concrete Architecture of Void

Group 7: The Trailblazers

Zijie Gan 21ZG6@queensu.ca

Nathan Daneliak 22TMX2@queensu.ca

Jinpeng Deng 22SS117@queensu.ca

Jeff Hong 22RQ20@queensu.ca

Emily Cheng 22TWS1@queensu.ca

Zipeng Chen 23QH10@queensu.ca

Abstract

This report aims to describe and analyze the concrete architecture of Void. Using Group 22's top-level conceptual architecture as the base, we identify congruences and discrepancies between its conceptual and concrete architectures. The new architecture is more process-oriented and component-based. Dependency graphs are generated using Understand to quantify call, reference, and map dependency relationships among subsystems. Visualization revealed that Void's architecture transitions from a simple layered model to a modular, cross-process structure involving both browser and main processes. We find that the AI service is a federation of cooperating services spanning the renderer and main processes. New key components including the Repository/Indexer and Validation/Feedback Handling have also been incorporated. Through the derivation and reflection analysis, we identify key dependencies, divergences, and architectural trade-offs that influence non-functional requirements like reliability, flexibility, and developer productivity. These findings provide a more accurate, procedure-oriented view of how Void integrates LLM-based functionality within the VS Code ecosystem to improve modularity, maintainability, and transparency.

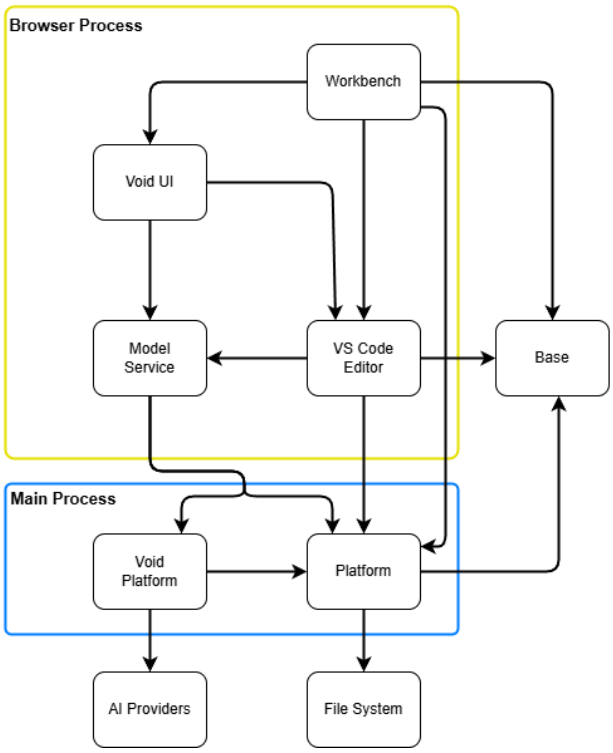
Introduction and Overview

The goal is to map conceptual components to their concrete code-level implementations in order to identify congruences and discrepancies between the theoretical and practical architectures. Void's design integrates LLMs for code editing, generation, and conversational assistance directly into the VS Code environment, which introduces challenges in cross-process communication, dependency management, and modular coordination. Understanding these architectures is critical for evaluating the system's maintainability, scalability, and performance. We explore concrete architecture, the derivation process to generate the concrete view, a description of the key components, subsystem architecture, reflection analysis, two typical use case diagrams and their dependencies, and a detailed analysis of one Model Service subsystem. Tools were used to create a dependency graph based on the top-level conceptual architecture that quantifies call/reference relationships between modules for static analysis. Architectural visualization and relationship mapping was conducted to capture control and data flow between the browser and main processes, which then revealed the internal structure of AI request handling and model coordination. Moreover, this document examines architectural qualities such as modularity, evolvability, and integration fidelity between Void and VS Code subsystems. To demonstrate how these architecture elements support real-world developer interactions, the report also analyzes two use cases Quick Edit & Apply, and normal Chat Mode. These scenarios illustrate how design components interact across subsystems to deliver AI-assisted editing and generation. After this introduction, we first revisit the updated conceptual architecture of Void, then outline the derivation process and present the concrete architecture. Subsequent sections analyze the Model Service subsystem, discuss representative use cases, share key findings on architectural qualities, reflect on lessons learnt, and conclude with an AI report.

Updated Conceptual Architecture

Whereas previous versions of this schematic were based on a simplified layered model, this updated figure emphasizes a realistic process-driven and component-based view where elements actually exchange data and control. The interface layer is formed by Void UI, VS Code Editor, and Workbench at the top, which are responsible for user interactions including typing,

chat inputs, and quick edits. Model Service receives user requests from components, creates prompts, maintains current conversation context, and decides which AI provider is suitable. It is a connection between the Browser and Main processes via the Void Platform to format messages, handle errors, and contact some external AI Providers. The Platform component in Void operates as an intermediary layer built on top of VS Code and Electron, providing configuration, extension management, and process communication services to other subsystems. The File System maintains access to persistent workspace data.



This updated conceptual view highlights the major control and data flows where secondary components are incorporated, but not explicitly shown. For example, Repository and Indexer manage the storage and retrieval of symbols, syntax trees, and embeddings to ensure AI responses are consistent with the most recent workspace state. Validation and Feedback Handling allow user responses to influence future suggestions while ensuring that the generated code meets privacy and quality standards.

Overall, the reviewed architecture can be seen as a progression from layered to modular design. Rather than a monolithic block, AI service is now a group of separate cooperating units communicating across processes. It offers a precise representation of how Void's components work and focuses on modularity, transparency, and tight integration with the VS Code platform.

Derivation Process

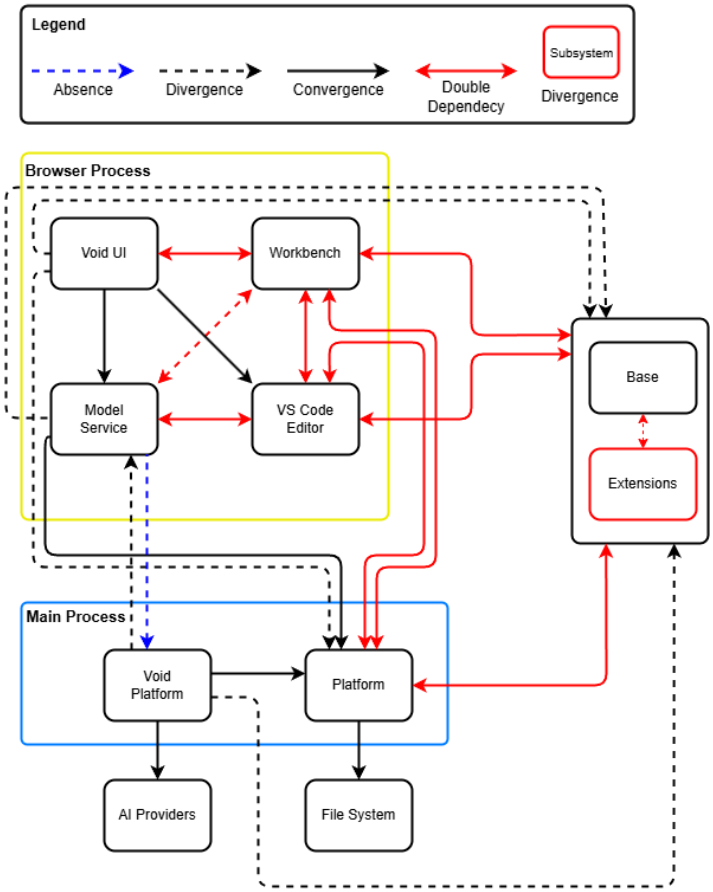
Our group concluded that assignment one was not sufficient as the base for our concrete architecture. Initially, we focused on Void specific components without accounting for how they interacted with VS Code's architecture. This time, we decided to use group 22's top-level conceptual architecture as the base of our concrete architecture.

To draw a diagram of Void's concrete architecture, we utilized the Understand software to visualize the dependencies between the subsystems in our conceptual architecture. We first split

each subsystem of the conceptual architecture into their respective architectural nodes to assign them code files and folders. Much of the main logic is located in the directory `/src/vs/`, where the four main VS Code components (base, editor, platform, and workbench) lie. Void-specific components had most of their code within `/src/vs/workbench/contrib/void/` with folders for UI, model service, and Void platform components. While assigning every file, we found a vital extensions folder that was not originally included in the conceptual architecture. As a result we created a new extensions subsystem that fits to the side of the layered architecture style, similar to the base subsystem.

After assigning every file, Understand created a dependency map for our concrete architecture. Using this simple diagram, we drew a clear picture of Void's dependencies. Comparing with the conceptual diagram, all absences, convergences, divergences, and double dependencies of the concrete architecture are outlined with different colours and line types.

Concrete Architecture



The concrete architecture still represents the **layered architectural style**. Though many bidirectional dependencies appear and make the layered style messier, we continue to see the general trend of higher level components depending on lower level components. This is especially clear in the Void-specific components where we see a clear dependency line from top to bottom. That trend was transferred from the conceptual architecture to the concrete architecture, while VS Code dependencies are much less layered and contain bidirectional dependencies between subsystems.

VS Code Specific Subsystems

1. **Workbench:** Handles user interface and all application logic. Also hosts the Monaco (VS Code's editor) core [9]. This subsystem is part of the browser process (HTML).
2. **VS Code Editor:** Contains the Monaco editor core for things like language features and syntax highlighting, and depends on the Model Service component to facilitate AI model use [9]. This subsystem is part of the browser process.
3. **Platform:** Provides base services for VS Code including settings and preferences, extension management, IPC, visual customization, usage analytics, and more found in the `/src/vs/platform/` directory. This subsystem is part of the internal main process.
4. **Base:** A side layer that provides general utilities to all other layers.
5. **Extensions:** A side layer that provides built-in extensions (ex. GitHub integration) to all other layers.
6. **File System (User):** The platform component depends on this to read or write from files, and carry out necessary OS tasks (IPC).

Void Specific Subsystems

1. **Void UI:** A browser process that handles the custom UI implemented by Void, including the model settings/selection, sidebar actions, quick edit, and autocomplete interfaces.
2. **Model Service:** A browser process that handles AI model management, configuration, usage analytics, MCP services, and message transformation, which is the process of converting a chat message into a specific format required by LLM providers.
3. **Void Platform:** Main process that interacts directly with the AI providers, extracting grammar and prompting AI's with the formatted message.
4. **AI Providers:** Models supported by Void (Anthropic, OpenAI, Gemini, etc) [7].

Dependencies

All subsystems in the concrete architecture depend on both the base and extension components because they provide general utilities required for most systems to function. We will focus on dependencies within the browser and main processes. Nearly every other component depends on the workbench, which contradicts the original conceptual architecture where it was intended to be independent. However, our analysis in Understand shows that while many components still rely on the workbench, its own outward dependencies have been significantly reduced. This shows that the conceptual architecture is generally correct, with just a few outliers that may have skipped the formal methods. Some notable dependencies within the workbench are on scripts for the terminal and debugging. A similar pattern appears with divergent dependencies where most other components have fewer than 50 incoming links, while their outgoing (convergent) dependencies are generally much higher.

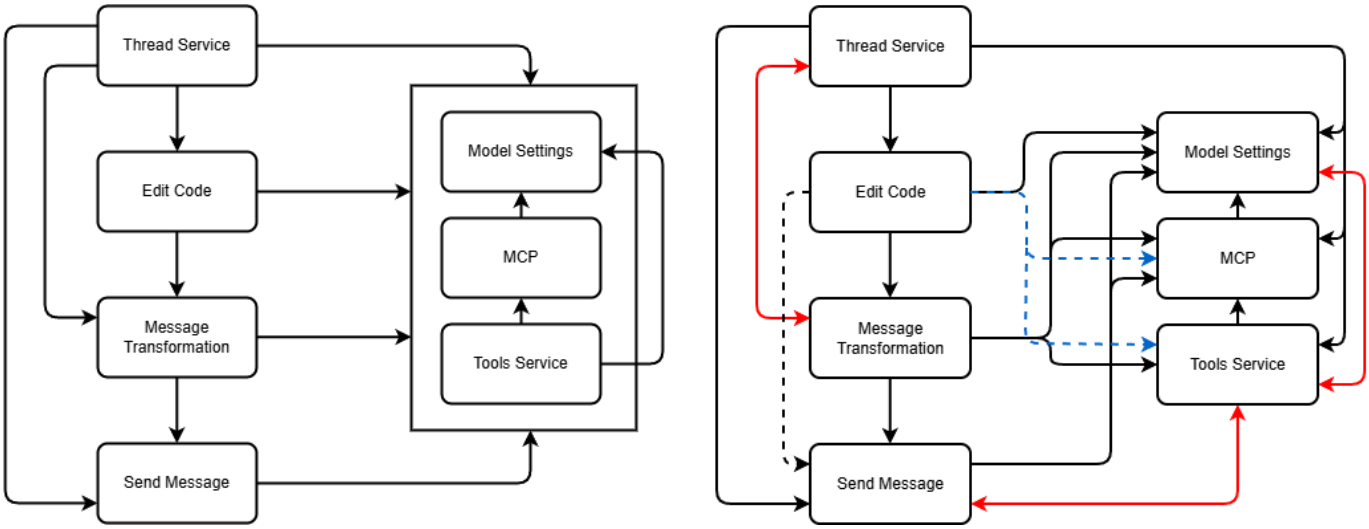
The platform component functions similarly to both the base and extension components. Positioned at the lowest layer of the architecture, it provides foundational services accessed by all modules in higher layers. All browser processes depend on the platform for its base services, so this role will not be discussed further for now. Void's platform depends on this platform and model service to obtain configuration and settings necessary to prompt the AI. It also communicates with AI providers to send and receive messages.

Next, the workbench depends most on the VS Code editor and Void UI components. This dependency on the VS Code editor is required since workbench hosts the Monaco editor, and

Void UI is required so users can access its UI components. The Void UI component then depends mostly on model service (UI configurations, transforming prompt messages), VS Code editor (Monaco base API, model/language services), and the workbench (general UI services). VS Code editor besides platform, depends on model service to facilitate AI Model use within, such as for automatic code completion. Finally, model services depend on the platform and VS Code editor component for basic language, model, and other services.

Subsystem Conceptual VS Concrete Architecture

The subsystem that we decided to focus on is the **Model Service** subsystem of Void.



- **Thread Service:** This component manages AI conversation context and runs Void's agent mode [8]. It must depend on Model Settings for mode type, as well as MCP and Tools Service for external tools and access to the editor. It also must depend on Edit Code, Message Transformation, and Send Messages for context and other AI tasks.
- **Edit Code:** This component manages creation, applying, and managing code changes through the AI [7], tracked in three regions: Ctrl + K, tracking, and diff zones. It depends on message transformation for LLM responses as well as for sending request edits to the AI. Finally, it depends on model settings to know whether fast apply is enabled.
- **Message Transformation:** Processes AI responses for code edits or other editor tasks, as well as transforming prompts into model-specific formats [7]. It depends on thread service for access to the chat, sending messages to communicate with AI, then model settings, MCP, and tools service for general preferences and tool access.
- **Send Message:** Serves as the central browser component for sending messages to the AI [7]. It depends on the tools service, MCP, model settings, and the preferences of tools.
- **Model Settings:** General user preferences and settings for chosen AI models.
- **Model Context Protocol (MCP):** Extends what built-in AI tools can do by allowing the use of external tools like web browsers, or access to the code editor itself [10]. MCP depends on model settings for access rights preferences.
- **Tools Service:** This component integrates different tools across different AI providers [7]. It depends on the model settings to check if tool lint errors are enabled.

Reflexion Analysis

High-Level Reflexion (VS Code ↔ Void)

At the high level, our reflection analysis compares what the conceptual architecture claimed—a single, browser-resident "AI/Model Service" block between the interface and providers, against what the codebase and dependency graph actually show. Using our git-informed mapping of directories (notably `/src/vs/workbench/*`, `/editor/*`, `/platform/*`, and the Void area under `/src/.../contrib/void/*`) with the Understand dependency map, we observe a consistent divergence that the AI path is not monolithic. Instead, it is a union of cooperating services that span the renderer and main processes. In practice, user interactions originating from the Workbench, Monaco editor, and Void UI, flow into a set of browser-side services responsible for conversation state, prompt shaping, and tool orchestration. Only then does it cross an explicit IPC boundary to the main process Void Platform that owns provider I/O, grammar extraction, and error isolation [7]. This cross-process split is visible in the traced code locations and call edges, challenging the earlier assumption that AI Providers were a single unit contained entirely within the renderer.

This modular, cross-process reality also surfaces additional elements that earlier concepts either omitted or underspecified. First, the Repository/Indexer now appears as a first-class participant in the model loop to supply symbols, syntax trees, and embeddings that keep prompts aligned with the most recent workspace state. Our inspection of references in Void's code and surrounding VS Code subsystems show concrete calls and event flows that were not captured in the original high-level picture [4]. Second, Validation and Feedback Handling behaves as runtime gates and learning loops on the path from provider responses to "Apply". This enforces privacy/quality constraints and captures user accept/reject signals, which is corroborated by concrete dependencies around edit/apply logic [7]. Third, while the conceptual model positions Workbench at the top with no inbound dependencies, the actual code reveals a handful of upward edges, such as the terminal and debugging helpers. Although small in number relative to Workbench's outbound dependencies, they still represent a directional discrepancy [4].

Finally, Extensions and several platform services act as side rails for both VS Code and Void code paths. Its presence was initially absent in the conceptual view but is now especially prominent from import sites in Git. Overall, these findings justify revising the top-level architecture into two cooperating clusters: browser-side model services and main-side platform/provider adapters, connected by explicit IPC channels. This structure is further supported by the clearly delineated collaborators Repository/Indexer and Validation/Feedback. The revision preserves Void's layered intent but replaces the misleading single-box "AI service" with a faithful modular depiction of how data and control actually move across processes.

2nd Level Reflexion (Focus: Model Service)

Zooming into the Void subsystem, the "Model Service" divergence actually becomes more pronounced when conceptual views are confronted with concrete implementation. Conceptually, we treated Model Service as a single component responsible for ingesting user requests, maintaining conversational context, assembling prompts, and selecting providers [2][3]. However, the Git-backed code structure and dependency relations surfaced by our analysis reveal a set of cooperating modules with explicit boundaries under `/src/.../void/*`. In particular, ThreadService manages conversation state and agent-style control flow; EditCode materializes

and applies edits under policy constraints; MessageTransformation performs provider-specific prompt/response shaping; SendMessage orchestrates outbound requests and token streams; ModelSettings centralizes policy and preference data; and MCP together with ToolsService; integrates external capabilities and tools [7]. Crucially, SendMessage does not communicate with providers directly. It instead traverses the renderer => main boundary to the Void Platform, which handles the networked provider interactions and returns streamed results over IPC. This design indicates an architectural separation not present in the usual monolithic Model Service box, as a consequence of Electron's process model and its security/runtime constraints [10].

This finer-grained decomposition also clarifies that a conversation "context" is continuously derived and refreshed beyond a chat service by querying Repository/Indexer interfaces for symbols, syntax trees, and embeddings, making those dependencies part of the primary model path rather than an incidental detail [7][9]. The flow from provider output to applied edits follows a gated process where Validation filters each result and emits signals to Feedback Handling which in turn, updates suggestions and policy toggles in ModelSettings. This arrangement reflects standard architecture mechanisms for quality, safety, and feedback control loops rather than an ad hoc pipeline [7]. Within the subsystem, we observe latent cycles where for example, ThreadService updates messages that are re-formatted by MessageTransformation and fed back for further orchestration, indicating that the single-box conceptualization hides coordination concerns better expressed through explicit module contracts. Finally, a small number of helper paths still depend upward on Workbench/editor facilities (e.g., language services and registrations), against the clean top-down boundary implied in the conceptual model. Such edges are consistent with the realities of VS Code extension and platform architecture but are nonetheless, candidates for refactoring behind platform abstractions.

In light of these findings, Model Service should be redrawn in the conceptual diagram as a constellation of named modules: stateful (ThreadService, EditCode), transformational (MessageTransformation), I/O-centric (SendMessage in the renderer paired with the main process), and policy/extensions (ModelSettings, MCP, ToolsService). Explicit arrows should connect these modules to the Repository/Indexer for context queries and eventing, then with explicit gates/loops to Validation and Feedback Handling around "Apply." Representing the renderer-main IPC as a first class edge resolves the principal divergence of aligning the model with Electron's two-process semantics, failure handling, and accurately locating latency [7]. More broadly, replacing the monolithic abstraction with a modular cross-process design aligns the conceptual artifact with the codebase's actual structure and accepted architectural guidance on decomposing monoliths into cooperating services under clear contracts [5].

Subsystem:

Initially, the Void subgroup was defined as loosely coupled layers of user interface, model service, platform, provider, and validation. In reality, these components interact more closely with one another through feedback loops and a constant flow of information. The UI is a direct interface and relies on the model service to coordinate requests and maintain conversation state. In turn, model service uses the platform to handle configuration management, cross-process communication and routing of model requests. This general flow is largely unchanged from the conceptual view, although we now have a return path for validation and feedback. Validation assesses the integrity and safety of results, before feedback catches user reactions and modifies system rules. Hence, the flow of information is no longer purely top-down but has become circular, with the system capable of self-correcting its outputs. Repository and indexer are also

more dynamic, maintaining up-to-date workspace information so that model service always operates on fresh, accurate context. This synchronization leads to higher accuracy and speed gap in the system response. Altogether, this subsystem level analysis demonstrates how Void extends its static layered architecture to also incorporate feedback and events. The design emphasizes responsiveness, reliability, and transparency while keeping the modules fairly isolated. These refinements make the system easier for integration and modifiability to accommodate new models or providers [4][6].

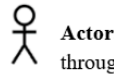
Use Cases

As an overview, the most typical use cases for a developer using Void IDE include:

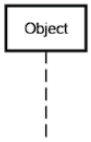
1. **3rd Party AI Integration:** Configure endpoints and enterprise models. Void Platform manages its authentication and direct connection without intermediate servers.
2. **Settings Interface:** Accessed through the IDE panel sidebar. Feature-specific parameters can be adjusted, global preferences modified, and providers configured. Void's main AI interactions interface lie in this sidebar [11].
3. **Tab Autocomplete (Code Suggestion):** Users type in the editor for Void UI to automatically request a suggestion from the AI service.
4. **Quick Edit:** Highlight code with Ctrl+K and request an edit with a specific prompt for the AI model to return its patch. If accepted, Fast Apply inserts the edits inline while Slow Apply rewrites the entire file [6].
5. **Chat Modes:** In Normal Chat, questions are asked and an LLM response appears in the UI panel, to be inserted into the editor. This use case branches into an automated Agent Mode that runs off of delegated tasks from the developer and Gather Mode which analyzes the project without modifying files (read-only) to return its findings [7].

Sequence Diagram Legend

Note: Asynchronous reply arrows are dashed asynchronous message arrows.

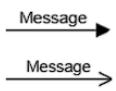


Actor: The user (developer) who interacts with the Void IDE system and thus its objects, through the interface.



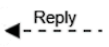
Object: Represents a component or class of Void's concrete architecture.

Lifeline: Vertical dashed line from each box, represents existence and activity over the passage of time as it goes downwards.



Message: Interaction or communication between objects, represents a method call, signal, data exchange between components.

- **Synchronous:** Normal method call where the sender waits for a response.
- **Asynchronous:** For background tasks or events, the sender doesn't wait.



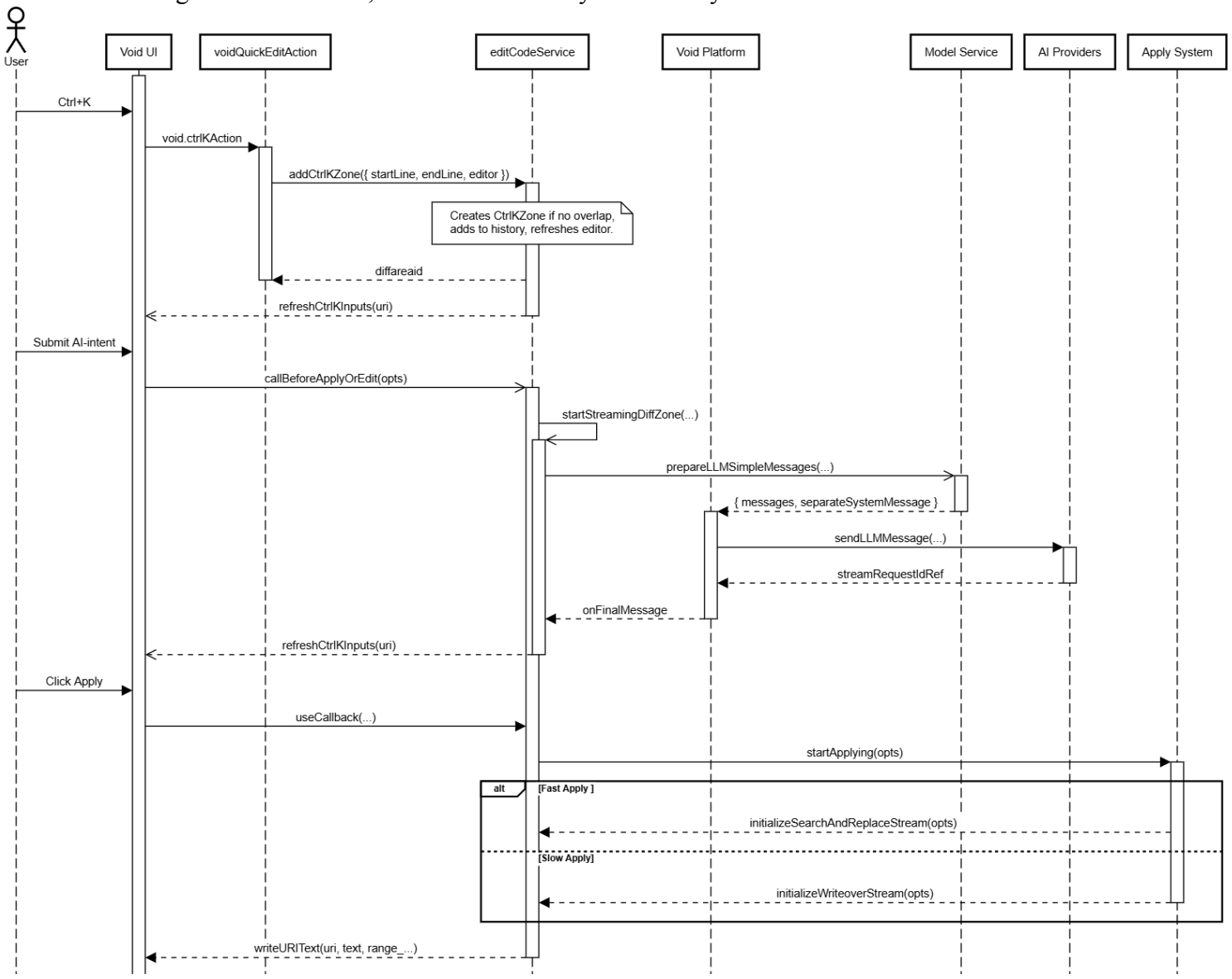
Reply: A return message, response sent from the receiver of a message when it has finished processing.

Activation Box: Rectangular box along an object's timeline that represents the time it takes for a task to be completed.

Essential Use Case 1: Quick Edit and Apply

Cross-referencing Void's concrete Github code with documentation, our first essential use case is applying an AI-assisted code edit. The primary actor is a developer user with a required precondition that an LLM provider is configured with code files that are editable. By the end, Void's editor should reflect the accepted change or remain the same if rejected [7].

Tracing code into `void/src/workbench/contrib/void`, the browser folder file `actionIDs.ts` lists constant command IDs like `'void.ctrlKAction'`. Firstly, users select code and activate the process of generating a quick AI inline edit with the keybinding Ctrl+K. Synchronously in `quickEditActions.ts`, class `voidQuickEditAction` is inherited by `registerAction2`. The editor state is preserved with `getActiveCodeEditor()` and the line numbers from `roundRangeToLines()` are passed as parameters into `addCtrlKZone()` from `editCodeService.ts`. This method creates an inline Quick-Edit UI, records an action in the edit history, and refreshes the editor view by calling `refreshCtrlKInputs()`. After returning the DiffZone ID, Void UI renders asynchronously.



When writing the AI-edit prompt, Void UI's React component in `QuickEditChat.tsx` automatically invokes `callBeforeApplyOrEdit()` to initialize and save the active file model. The method `startApplying()` is given a mode and initiates the corresponding DiffZone generation. Model Service begins immediately after through an asynchronous `prepareLLMSimpleMessages()` from `convertToLLMMessageServices.ts` with the appropriate parameters to return a transformed prompt message. Now, this can be processed by AI Providers through `sendLLMMessage()` to get the generated code patch edit. Upon receiving the hunks of suggested edits, Void UI will be refreshed with a preview for the user to accept. From React's `useCallBack()`, editCodeService requests changes for the Apply System to either edit with Fast Apply or Slow Apply [12]. Once final changes have been made to the native VS Code Editor, it is reflected on Void's UI with `writeURIText()`.

Full diagram: https://postimg.cc/xJBz9YBB.

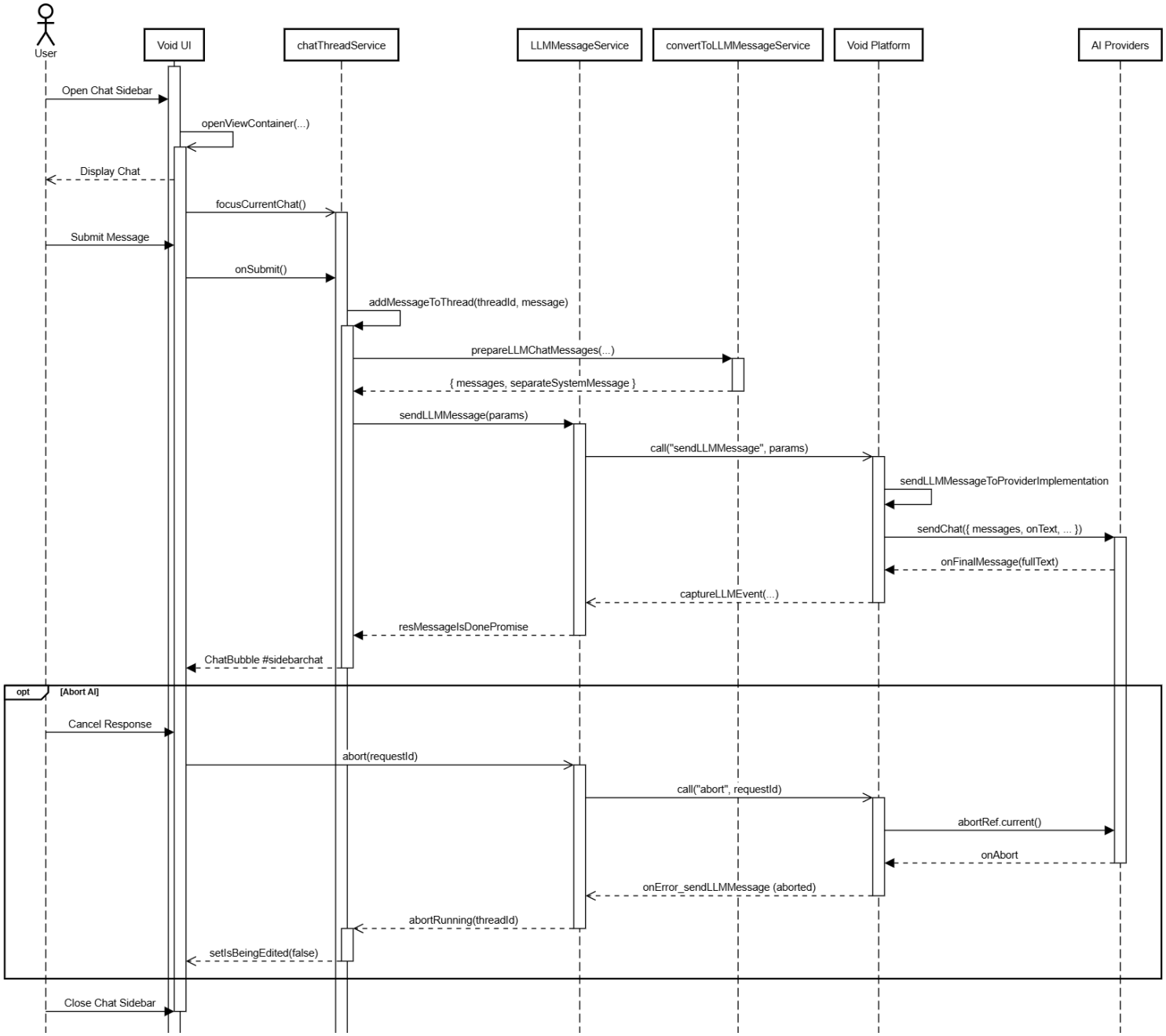
Essential Use Case 2: LLM Messaging (Normal Chat Mode)

The user communicates with Void's chosen LLM service through a chat interface for the connected APIs to request an intelligent explanation. This use case focuses on Normal Chat Mode (Ask operational state), without autonomous tool execution [11]. The user begins by opening Void UI's sidebar on top of the existing editor view. In the `../void/browser/` folder, `sidebarActions.ts` contains `openViewContainer()` to synchronously open the panel. The chat becomes visible with an asynchronous instance of `focusCurrentChat()` from the file `IChatThreadService` before the threads and input interface are ready on the backend.

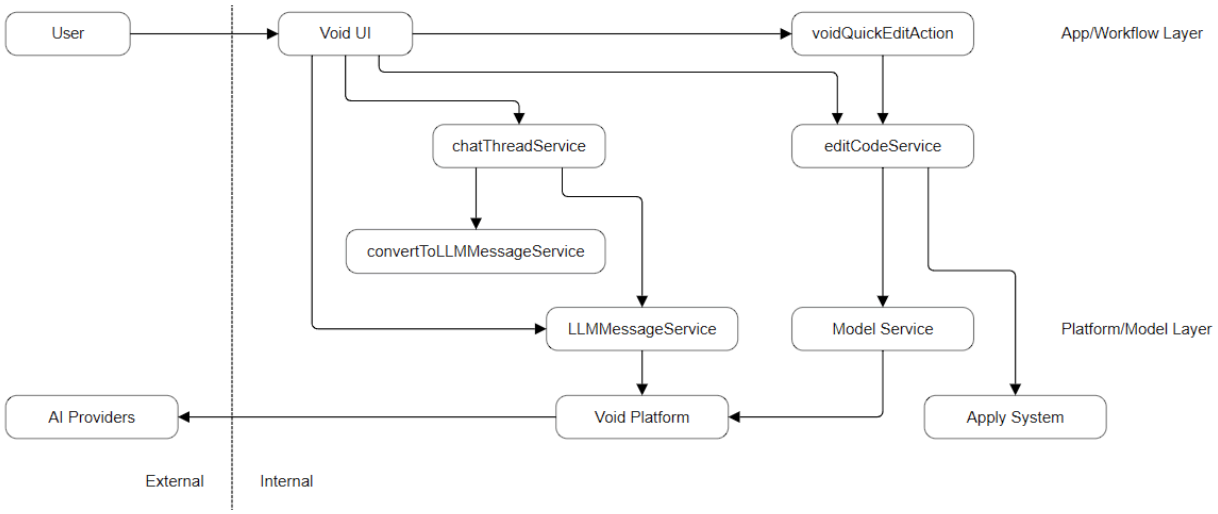
After typing and submitting a message, the `SidebarChat.tsx` method `onSubmit()` is called to update the state, process text, and update chat bubble content. This new message is added to the active thread containing sender role and context with `addMessageToThread()` local call in `chatThreadService.ts`. This raw conversation context is converted into an LLM readable request with `prepareLLMChatMessages()`, which returns the formatted message and additional instructions to `LLMMessageService` with `sendLLMMessage()`. The following LLM message method call in `sendLLMMessageChannels.ts` represents an asynchronous IPC request from the renderer/browser to the main process, where Void Platform logic executes and internally resolves provider information [13]. After, it calls `sendChat()` for the AI Provider to generate a token-by-token streaming response. Technically, incremental responses (onText events) are sent through Void Platform => … => ChatThreadService to progressively render text as it is generated. For clarity, this is collapsed into a single `onFinalMessage()` response, which travels back to the Void UI and appears as a chat bubble.

Optionally, if the user cancels an ongoing AI response through Void UI, that request is first asynchronously sent to `LLMMessageService` through `abort()`. It handles the IPC route with `call("abort", requestId)` to Void Platform, which contacts AI providers externally through synchronous `abortRef.current()` with an internal `onAbort` reply. Void Platform sends an error message, then `LLMMessageService` passes the threadId to `abortRunning()`, which lets `ChatThreadService` know that the AI generation has been cancelled. Void UI reflects the change in its synchronous React sidebar, where `setIsBeingEdited` is set to False.

Full diagram: https://postimg.cc/dZ12Mtpc.



The **box-and-line diagram** below summarizes dependencies among components participating in our two selected essential use cases. In addition to the external User and AI Providers, internal modules are organized by layer progressing from the App/Workflow layer at the top to the Platform/Model layer at the bottom. Arrows represent key control and data dependencies identified from the sequence diagram. This structure highlights how user requests flow through Void UI and its related services such as chatThreadService and editCodeService, before reaching the Model Service and Void Platform for model communication. The separation between external and internal also clarifies the boundary between the application-level logic and underlying AI integration framework.



Data Dictionary

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

Naming Conventions

- I/O: Input/Output.
- OS: Operating System.
- HTML: Hypertext Markup Language.
- APIs: Application Programming Interface.
- VS Code: Visual Studio Code.
- IPC: Inter-process Communication.
- IDE: Integrated Development Environment.
- LLM: Large Language Model.
- AI: Artificial Intelligence.
- MCP: Model Context Protocol.

Conclusion

Delving into the concrete architecture of Void helped us achieve a better, deeper understanding of the internal workings and structure of the architecture. The layered style of architecture stayed consistent with the findings in the conceptual architecture. The main difference was the separation of the layers, which were muddled with bidirectional dependencies between layers. This separation showed some inspiration and similarities to modular

architecture. The source code lets us confirm our original ideas of Void using external AI providers to create a helpful, responsive piece of software that makes the user feel like all their needs are being attended to with little delay or latency in the path between Void's IDE and the external AI providers.

Lessons Learned

When going into this second report detailing the concrete architecture of Void IDE, we made sure to keep the lessons we learnt from the first assignment in mind. Such as the architecture not being limited to one style, but instead being inspired by multiple styles. However, despite learning from the past assignment, there were still plenty of discoveries throughout this assignment.

For example, when going into the conceptual architecture, we thought that many of the dependencies in Void's layered style would be one-directional to keep the layers clear and consistent. Building on the idea that architectural styles would not be clear-cut, the layers of the void architecture are not clear-cut either. We found many bidirectional dependencies of components that made the borders of where layers start and end much messier.

Another key takeaway from this assignment was understanding the importance of visualization and communication when recovering architecture from a large database repository. While the technical analysis provided precise data on dependencies and module relationships, translating that information into clear sequence diagrams with written explanations proved to be equally challenging. We learned that an effective model must be both accurate, and interpretable to others. Simplifying complex dependency networks into meaningful structures required a lot of cross-referencing throughout multiple files of code which after careful judgement, helped us identify the most representative interactions to highlight in the final diagrams.

Finally, we found that Void displays the AI messages to its users in an incremental format, generating text and receiving tokens from the AI concurrently. This is much more efficient, allowing the user to receive partial responses as they are built, making the user feel that the software is always responding to any problems they may have.

AI Collaboration Report

Just like the first assignment, the LLM we chose to use was **OpenAI's ChatGPT-5** model, released in August 2025. The reason we did not change our model is that we were satisfied with ChatGPT's performance during our first report, and it continued to provide helpful assistance throughout the second report. In addition, OpenAI boasts that their GPT-5 model specializes in coding and tasks related to the domain of computing in general. Therefore, we briefly compared this model with alternatives such as Claude 3.5 and Gemini 1.5, but ChatGPT-5 proved to be more consistent overall in both its code-aware reasoning and better architectural vocabulary. As such, we believed that this specialized model would be the best way to analyze the architecture of a massive coding project like the Void IDE.

We had ChatGPT provide input on any grammatical and structural errors within our written text (Question and Answer):

The reason we believed this to be a suitable task for the AI is that it is a difficult and time-consuming task to edit deeper and more complex grammatical errors outside of simple spelling and punctuation. While we had a human proofread the entire report for grammatical errors, we also had complex issues in writing, such as the flow of the writing or finding more

suitable synonyms to better express our points. Using AI for these higher level edits improved clarity and consistency across sections while still maintaining our original tone and intent.

We had ChatGPT help by giving pointers on organizing and drawing diagrams (Question and Answer):

We handed this task off to the AI as oftentimes when tackling such a large and extensive code database to analyze, it is hard to decide where to start. Although we did not have the AI actually look at the code, we found it helpful to get some guidance before starting the task and spending too much time getting lost in the various directories. Even without looking directly at the code base and repository, an AI model that claims to be specialized for coding tasks was trained extensively by its developers and given ample data of what the structure of the various repositories should look like. Still in several cases, ChatGPT produced confident but inaccurate statements (hallucinations) about class dependencies or architectural patterns that didn't actually exist in the code. Instead of assuming accuracy, it was crucial to identify and correct these responses before drawing the final relationship diagrams. Similarly, this is a good task for ChatGPT-5 to give our group members directions and save us time.

Looking at concrete architecture is much a more nuanced and detailed task compared to analyzing conceptual architecture off of documentation, so we used ChatGPT to help clarify questions (Question and Answer):

Since this is our first time analyzing the concrete architecture of a software, there are many questions that pop up as we go deeper into the source code and see the structure of Void. The AI teammate would be an easy way to get a quick answer to a question without having to research to break up the workflow and slow down the writing process. Once again, since the AI specializes in coding, we believe asking it questions regarding architecture is a good use.

Interaction Protocol and Prompting Strategy:

We did not believe it would be optimal for us to have solely one prompt engineer when communicating with ChatGPT. Everyone was working on a separate part and needed the AI to answer questions immediately, so having them send their requests through a dedicated prompt engineer would slow down the workflow process and go against our idea of using AI to save time when working on the project. Instead, any time someone would send a prompt to ChatGPT, they would send a link of their chat to a shared group chat with every group member in it. Later when people had finished their portion, we took a second look at the conversations with ChatGPT and how it affected the parts. The example from our second task is an example of a critical prompt where we refined and iterated our prompts to improve the response. We provided context on the type of software that Void is to get ChatGPT to respond for the specific case we needed it to. Furthermore, when it asked follow-up questions, we responded and returned with our own questions to get further details and information.

Validation and Quality Control Procedures:

As stated above, we kept a log of any chats with the AI to review later once the first draft of the report is done. After that, we fact-checked by checking against primary sources such as the source code of Void, online forums, and articles related to the questions we had, and course materials such as slides and readings. We frequently found that ChatGPT's summaries and explanations omitted smaller implementation details, which was corrected after manually

reviewing the source code. In addition, when it came to using AI for grammar checks and looking for synonyms, we had an initial review of the grammar before asking the AI, and then we had a secondary look after the AI made suggestions to ensure no mistakes were made. All of these careful quality control procedures reinforce the importance of human oversight when utilizing artificial intelligence like ChatGPT.

Quantitative Contribution to Final Deliverable

Given that we did not use any of the AI-generated text directly, only taking inspiration from its grammatical suggestions and learning from the facts it presented, the AI contribution is not very noticeable when looking at the text. However, the time that we saved using ChatGPT to answer questions and speed up our workflow must be taken into account. So, since ChatGPT helped to plan the outline of most parts in the report, had some grammar checks that changed the way we thought about sentences, and managed to save us a decent amount of time when fact-checking during the report, the overall percentage contribution is around 8%. More helpful than the last report, but like the last report, it was not a vital component.

Reflection on Human-AI Team Dynamic:

When it came to planning and actually writing the report, the AI helped us save a considerable amount of time by creating a plan for us and letting us circumvent research at the time. However, when it later came to double-checking the AI conversations and facts provided for accuracy, it created a considerable amount of work. Fortunately, this amount of work did not outweigh the time saved. It influenced our brainstorming process by streamlining the planning process during group meetings. In addition, it affected our decisions when it came to grammar checking with its own suggestions, which were deeply considered. It did not lead to any disagreements or challenges within our group. Going forward, we shall keep the idea that effective AI collaboration means using it to optimize the initial work process and save time before double-checking. Furthermore, having each person prompt the AI and logging chats is much more efficient than having one person prompt. Overall, the interactions with the AI helped us reduce the time spent overall, especially on planning and initial research.

References

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

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

[3] Electron, “Electron Documentation,” *Electron*. Available: https://www.electronjs.org/docs/latest/.

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

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

[6] Void Editor, “Void Official Website,” *Void Editor*. Available: voideditor.com.

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

[8] voideditor, “void/VOID_CODEBASE_GUIDE.md at main · voideditor/void,” *GitHub*, 2024. https://github.com/voideditor/void/blob/main/VOID_CODEBASE_GUIDE.md

[9] Microsoft, “Source Code Organization,” *GitHub*, Oct. 11, 2025. https://github.com/microsoft/vscode/wiki/Source-Code-Organization.

[10] “Introduction - Model Context Protocol,” *Modelcontextprotocol.io*, 2025. https://modelcontextprotocol.io/docs/getting-started/intro

[11] Zread.ai, “Quick Start | Voideditor/Void,” *Zread*, Jun. 25, 2025. Available: https://zread.ai/voideditor/void/2-quick-start.

[12] Zread.ai, “Apply System (Fast vs Slow) | Voideditor/Void,” *Zread*, Jun. 25, 2025. Available: https://zread.ai/voideditor/void/13-apply-system-fast-vs-slow.

[13] Zread.ai, “LLM Message Pipeline Architecture | Voideditor/Void,” *Zread*, Jun. 25, 2025. Available: https://zread.ai/voideditor/void/10-llm-message-pipeline-architecture .