# Concrete Architectural Analysis of Void

November 7th 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 analyzes the concrete architecture of the Void IDE, an open-source, AI-integrated development environment built on Microsoft's VS Code. The source code is examined and compared against the conceptual architecture outlined in this document, which has since been updated from deliverable A1.

# 2. Introduction and Overview

Void is an open-source, AI-powered code editor built on top of Visual Studio Code's Electron framework. It extends VS Code's architecture by integrating large language models (LLMs) for intelligent code completion, automated edits, and conversational programming assistance.

The purpose of this report is to perform a concrete architecture analysis of Void by examining its GitHub source code and using Understand for static analysis. This will allow us to conduct a reflection analysis comparing the conceptual and concrete architectures to identify key divergences in design and implementation. The analysis focuses on both the top-level structure of Void and the subsystem-level components that support AI integration, communication, and extensibility.

# 3. Derivation Process

There were four traceable stages: (1) anchor on the A1 scope, (2) instantiate Understand zones, (3) map directories to components, and (4) reconcile divergences so the high-level summary matches the detailed checklist.

1. Anchor on our updated conceptual architecture: Restating the Assignment 1 subsystems exactly as they appear in the new inventory: UI Components, chatThreadsService, editCodeService, sendLLMMessageService, voidModelService, VoidSettings Service, SendLLMMessageChannel, sendLLMMessage.impl, extractGrammar, and modelCapabilities. This kept the conceptual baseline synchronized with the service, channel, and implementation layers enumerated for the Workbench portion of the Void codebase.

2. Instantiate Understand zones with the mapped directories. Each subsystem received a zone whose members were drawn directly from the directory list. UI Components and chatThreadsService drew from `browser/parts/activitybar`, `browser/parts/sidebar`, `browser/actions`, and the associated `browser/*/media` folders, while services with shared logic (voidModelService, VoidSettings Service) referenced `model/common` and `common/editor`.

3. Iterative file-to-component assignment. For every zone, drilling into the concrete directories to confirm ownership of pipeline elements-e.g., SendLLMMessageChannel (Channel Layer), sendLLMMessage.impl with extractGrammar and modelCapabilities (Implementation Layer), and sendLLMMessageService/chatThreadsService/editCodeService (Service Layer). Cross-links were logged whenever a service imported helpers from `common/editor` or `browser/parts`, keeping the recorded data and control flows tied to the mapped directories.

4. Reconcile divergences. Whenever conceptual placement differed from concrete structure-such as UI Components reaching into `common/editor` helpers or sendLLMMessageService sharing `common` contracts that also serve backend logic-we documented the rationale with directory references.

# 4. Architecture

## 4.1 Updated Conceptual Architecture

Our group has decided to update our conceptual architecture since our previous analysis overemphasized Void's AI integration. For this analysis, we examined a document provided on Void's GitHub titled *VOID_CODEBASE_GUIDE.md*, which gives an overview of how VS Code operates as an Electron app (voideditor). From this, we determined that Void maintains the same **two-process model** as VS Code. By referencing *Void Core Module Structure*, we confirmed that Void consists of a **browser process** and a **main process**, supported by a shared **common codebase**. The browser process manages user interactions and the graphical interface, while the main process handles core operations such as file management, API calls, and communication with large language models (LLMs). The common code folder provides utilities accessible to both processes, ensuring reusability and consistency across the architecture (zread.ai).

*Void Core Module Structure* also shows that Void follows a **layered architecture**, organizing components into hierarchical layers: **Base**, **Platform**, **Editor**, and **Workbench**. The **Base Layer** initializes fundamental utilities and environment setup, while the **Platform Layer** manages configuration, IPC, extension systems, telemetry**,** and theming. The **Editor Layer** contains the Monaco editor core for code editing and syntax highlighting, and the **Workbench Layer** manages the main interface, user contributions, and AI-related extensions(zread.ai).

*VOID_CODEBASE_GUIDE.md* describes a **LLM Message Pipeline** which is in charge of the flow of AI related data. We obtained the structure of this pipeline from an article titled **LLM Message Pipeline Architecture** which describes the architecture of this subsystem as having 3 layers: **Service**, **Channel**, and **Implementation.** Based on both of the sources reference we derived Void's conceptual architecture. The Service layer is in charge of the **browser process**, user input is captured and processed through several UI-facing services. The **UI Components** handle interactions within the editor, sidebar, and command palette, while the

**ChatThreadsService** manages chat sessions, runs agent mode, and routes messages between the user and the AI. Supporting services such as the **EditCodeService** enable AI-assisted code editing, allowing the LLM to make inline modifications or suggestions, and the **VoidSettingsService** manages user configurations such as selected models, API providers, and general Void preferences. The **SendLLMMessageService** then packages chat or code-editing requests for transmission, routing them through the **SendLLMMessageChannel** in the Channel Layer, which acts as the inter-process communication (IPC) bridge between the browser and main processes. The Implementation Layer handles the main **process**. Messages are received and processed by **SendLLMMessage.impl**, which coordinates communication with external AI providers such as **Anthropic**, **OpenRouter**, and **Ollama**. It references **modelCapabilities** to determine provider specifications, such as context limits, supported tools, and data formats, while **extractGrammar** parses the structure and logic of AI-generated responses before returning processed results to the browser. The **VoidModelService**, while not explicitly placed in the documentation diagram, is likely part of the **browser process** within the **Service Layer**, as it manages text models and URIs for file editing. This placement aligns with its described function—handling in-memory file updates without manually loading or saving files, which is consistent with editor- and UI-level operations rather than backend communication.

Overall, these ten components, **UI Components**, **ChatThreadsService**, **EditCodeService**, **SendLLMMessageService**, **SendLLMMessageChannel**, **SendLLMMessage.impl**, **modelCapabilities**, **extractGrammar**, **VoidSettingsService**, and **VoidModelService**, work together to form a modular and layered system. This structure enables Void to extend VS Code's Electron foundation with integrated AI functionality while maintaining clear separation between **front-end interaction (browser process)**, **inter-process communication (IPC channel)**, and **back-end execution (main process)** (*VoidEditor*).

Additionally, we found that Void uses a **Publish–Subscribe (Pub/Sub)** style within the **Workbench Layer** to support its extension system. This allows different extensions to subscribe to editor events without tight coupling, promoting modularity and scalability. For example, chatThreadsService and editCodeService subscribe to events from sendLLMMessageService and LLMMessageChannel to update interface when new AI responses or code edit comes through. The loose coupling allows LLM Message Pipeline subsystem and editor services to communicate to each other asynchronously without being tightly dependent on each other.

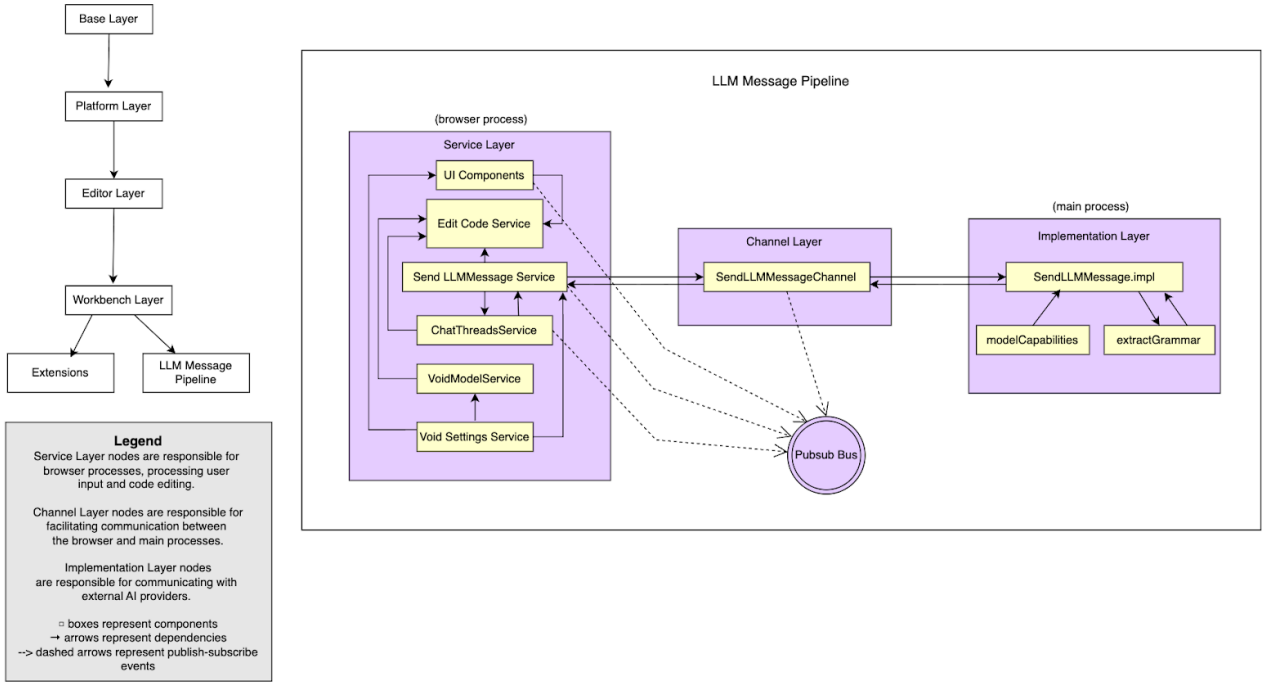



*Fig 1. Updated conceptual architectural diagram. Right: Overview of Void's architecture. Left: In depth look at Void's LLM Message Pipeline.*

## 4.2 Top-Level Architecture: VSCode and Void

Using Understand, we examined the Void codebase and found that the VS Code–inherited structure (Base → Platform → Editor → Workbench; two-process Electron model) aligns closely with our updated conceptual architecture. The extension subsystem within the Workbench Layer also remains consistent with both our conceptual model and VS Code's original design. The Publish–Subscribe (Pub/Sub) mechanism continues to serve as the foundation for the extension system, allowing extensions to subscribe to editor events or services without direct dependencies. This confirms that Void preserved VS Code's extensibility model even as it introduced AI-driven features and new service layers, demonstrating architectural continuity between the two systems.

Observed divergences (concrete vs. conceptual):
While VOID_CODEBASE_GUIDE.md conceptually defines the LLM Message Pipeline as being divided into three distinct layers (Service, Channel, and Implementation) our concrete analysis shows that this structure is not explicitly reflected in the actual code organization. Instead of separate directories corresponding to these conceptual layers, the LLM-related components are distributed across shared (common/) and process-specific (main) directories. This flatter organization indicates that Void's implementation prioritizes modularity and shared access over strict layering, while still maintaining the same overall data flow between the browser and main processes.

- Browser vs. common: Components we modeled as belonging to the browser process (ChatThreadsService, SendLLMMessageService, VoidSettingsService, and VoidModelService) are implemented under the common/ directory. This reflects a code-sharing strategy using isomorphic modules and shared contracts rather than a change in responsibility. Although their functionality remains browser-facing (e.g., managing chat input, user settings, and in-memory text models), their placement under common/ allows both processes to import and reuse shared interfaces and type definitions.
- Channel location: The SendLLMMessageChannel, which conceptually functions as an independent communication layer between the browser and main processes, is located within the main process in the actual implementation. This design choice ties the channel's registration and endpoint binding to the main process, effectively coupling its initialization with backend logic instead of treating it as a neutral bridge.
- Provider capabilities: Instead of maintaining a single centralized modelCapabilities component, the main process implements multiple provider-specific directories, each named using the convention <LLMName> + Tools (e.g., AnthropicTools, OllamaTools). Each directory defines parameters, tool definitions, and compatibility logic for its respective provider. This decentralized approach enhances extensibility but departs from the conceptual model's unified capability structure.

Implications:
These differences reflect organizational and implementation decisions, including the use of shared common/ modules, main-anchored channel registration, and per-provider capability directories, rather than fundamental architectural shifts. For our concrete model, we therefore (i) retain the conceptual responsibility mapping (those services remain browser-owned), while (ii) documenting their physical placement in common/ as an implementation tactic, (iii) noting the channel binding in the main process, and (iv) representing provider capabilities as modular directories rather than a single centralized component.

## 4.3 Detailed Explanation Of LLM Message Pipeline

The LLM Message Pipeline is a subsystem within Void's Workbench Layer. It manages all communication between the code editor(in this case, VSCode) and all the various AI model providers. It turns the internal chat object in the (browser) into requests for providers, and then sends them to the LLM model providers. It then streams back the responses, handles tool calls, surface errors, and records metrics while hiding provider differences. Internally, the LLM Message Pipeline also uses publish-subscribe style architecture for event handling. When LLMMessageChannel sends out partial text, final messages, or error events, these are "published" to subscribed services such as chatThreadsService and editCodeService. This design decouples message generation from message consumption, allowing UI updates and code editing

responses to occur asynchronously and in real time without direct coupling between components, creating less dependencies.

### 4.3.1 Subsystem: Browser Process

This process is the browser-side API surface and lifecycle manager that UI components call.This layer is the LLMMessageService. UI components such as chat box, conversation view, and cancel buttons would request from LLMMessage, and LLMMessageChannel would pass serialized requests and subscribe to streaming events.

### 4.3.2 Subsystem: Common (Shared Modules)

This subcomponent is the IPS boundary and relay between renderer (browser) and the main/background process. This layer is LLMMessageChannel. The layer serializes the requests from the renderer and then sends it to the main process, and then it streams onText, onFinalMessage, onError, and onToolCall events back to the renderer as they arrive. It also handles flow control and basic validation at the IPC. The LLMMessageService calls this layer and ConvertToLLMMessageService requests this layer.

### 4.3.3 Subsystem: Main Process

This subcomponent is the set of  provider implementations that knows how to talk to each LLM provider. The set of providers is sendLLMMessage.impl.ts. This layer translates normalized LLM requests into provider-specific API calls, handles streaming responses, manages tool calls and capabilities, and normalizes error and results. sendLLMMessage would call this layer and external provider endpoints would request from this sublayer.

### 4.3.4 Subcomponent of the Processes of LLM Message Pipeline

#### 4.3.4.1 Subcomponent: SendLLMMessageChannel

The LLMMessageChannel is Void’s message bus between the frontend and backend. It routes user commands, streams live LLM responses, handles cancellations, and connects the UI to the provider implementations. The difference between concrete and conceptual is that in the conceptual it was in the channel layer (common), while in the concrete it was moved to the implementation layer (electron-main). It interacts with the channel layer.

#### 4.3.4.2 Subcomponent: LLMMessage

The sendLLMMessage manages how Void sends, tracks, and handles AI requests. It logs metrics, streams partial and final responses, catches errors, and routes each message to the correct LLM provider implementation. It was not in the conceptual and was created in the concrete to hold extractGrammar and sendLLMessage.impl. Originally in the conceptual sendMessage,impl held extraGrammar and modelCapabilities. It interacts with the implementation layer (electron-main), extractGrammar, and sendLLMMessage.impl.

#### 4.3.4.3 Subcomponent: UI Components

The UI component is a series of components that creates the user interface. It was conceptually created in the service layer (browser), and it was kept there in the concrete version. It interacts with the service layer, quick-edit-tsx, sidebar-tsx, void-editor-widgets-tsx, void-onboarding, void-settings-tsx, and void-tooltip.

#### 4.3.4.4 Subcomponent: chatThreadsService

This file implements the ChatThreadService, the core system that manages all chat sessions (threads) between the user and the LLM inside Void. It handles creating, switching, deleting, and persisting chat threads, tracks messages, checkpoints, and file edits, and coordinates streaming AI responses or tool calls. It was in the service layer (browser) in the conceptual architecture and was kept in the service layer for the concrete. It interacts with the service layer.

#### 4.3.4.5 Subcomponent: editCodeService

This massive file implements the EditCodeService, which powers Void's real-time AI code editing system — the feature that lets users apply, preview, accept, or reject AI-generated code changes directly in the editor. It was in the service layer (browser) for both conceptual and concrete architecture. It interacts with the service layer.

#### 4.3.4.6 Subcomponent: sendLLMMessageService

This file defines the LLMMessageChannel, which acts as the bridge between the Electron main process and the renderer (UI) for all AI-related requests in Void. It handles IPC (inter-process communication) commands like sending LLM messages, aborting requests, and listing available AI models (from providers like Ollama or OpenAI). The channel receives commands from the frontend, calls the backend sendLLMMessage() functions, and uses emitters to stream live updates (text, errors, final responses) back to the UI. It was in the service layer (browser) during conceptual architecture but was moved to the channel layer (common) in concrete architecture. It interacts with limMessage.

#### 4.3.4.7 Subcomponent: voidModelService

This file defines the VoidModelService, which manages all text/code models opened in Void — essentially acting as the bridge between the editor and the file system. It loads files into memory (initializeModel), keeps persistent references so they don't get auto-disposed, retrieves models safely (getModel, getModelSafe), and saves changes (saveModel) without triggering extra editor events. It was in the service layer(browser) in conceptual architecture but was moved to channel layer (common) in concrete architecture. It interacts with channel layer

#### 4.3.4.8 Subcomponent: voidSettings Service

This file defines the VoidSettingsService, the central configuration manager for all AI, model, and feature settings in Void. It handles storing and encrypting user preferences (like selected models, providers, global flags, and MCP server states), syncing feature models (e.g., Chat → Apply/SCM), validating configurations, and updating settings reactively for the UI. It also logs every change through the MetricsService. It was in the service layer (browser) in conceptual architecture but was later moved to channel layer (common) in concrete architecture. It interacts with the channel layer.

#### 4.3.4.9 : modelCapabilities

It lists the default settings, available models, and technical capabilities (like context size, reasoning support, system-message format, token costs, etc.) for every provider (OpenAI, Anthropic, Gemini, DeepSeek, Mistral, Ollama, etc.), along with logic to detect and fall back when encountering unrecognized or custom models. It also provides helper functions like getModelCapabilities() and getSendableReasoningInfo() to dynamically fetch a model's supported features (e.g., reasoning sliders, FIM, system message style) and prepare the correct payloads for API calls. In conceptual architecture this was under implementation in the subsystem sendLLMMessages.impl, but in concrete architecture it was moved to channel layer (common). It interacts with the channel layer in concrete architecture.

# 5. Diagrams

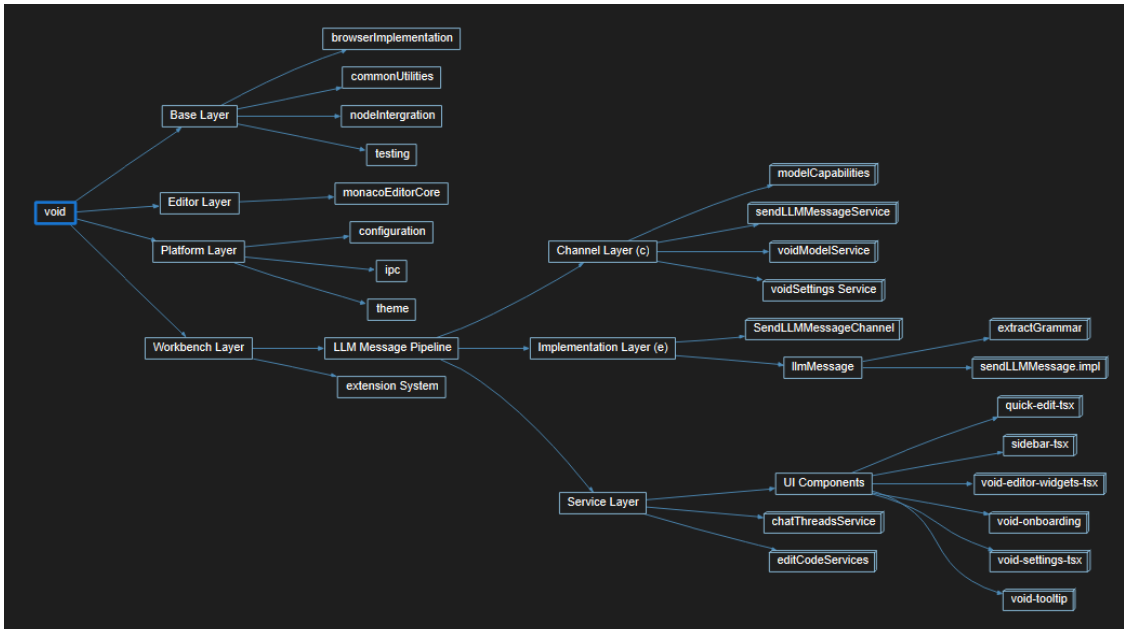
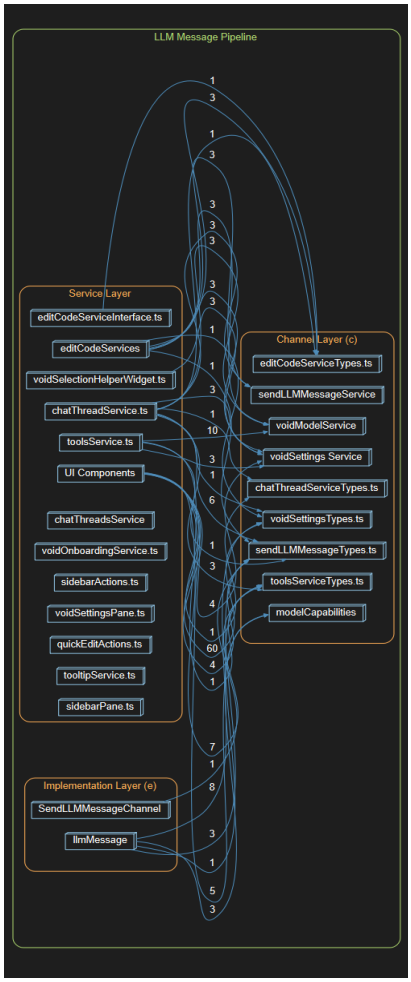
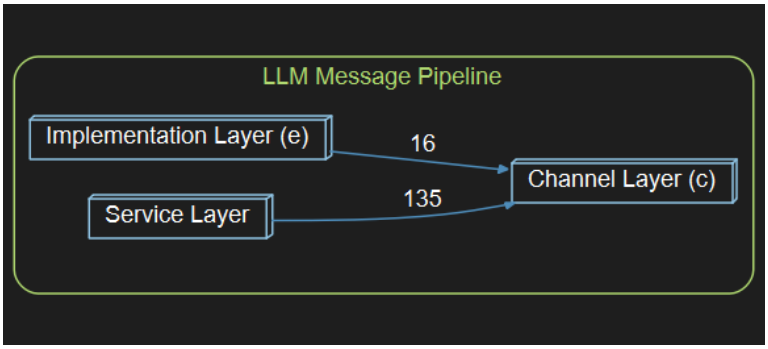



*Fig 2. High level concrete architecture constructed on Understand (the layers are not in the right order because the software organizes subsystems on the same level alphabetically)*

*Fig 3 and 4. Dependency graphs between the components of concrete architecture constructed on Understand (Left: interactions between components; Right: interaction between layers)*



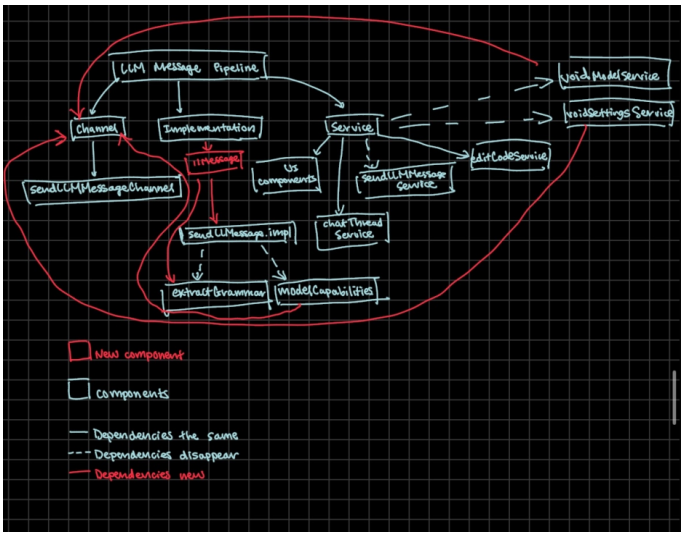




*Figure 5: Reflexion Diagram (bottom right)*

# 6. Use Cases

We present two use cases that show how the Browser/Workbench layer hands control to the Common channel and Electron-main services while the Event Bus propagates updates across each tier:

## 6.1 Ctrl+K Quick Fix Workflow

The developer presses Ctrl+K, the Workbench chain (Editor UI, Context Extraction, Prompt Engineering) gathers code, consults `voidSettingsService`, and sends the prompt via

`sendLLMMessageService` into `LLMMessageService`, which streams tokens before chatThreadsService, Task Service, and `ebsCodeService.ts` execute the patch over the Event Bus while Response Integration and telemetry push editor updates (Figure).

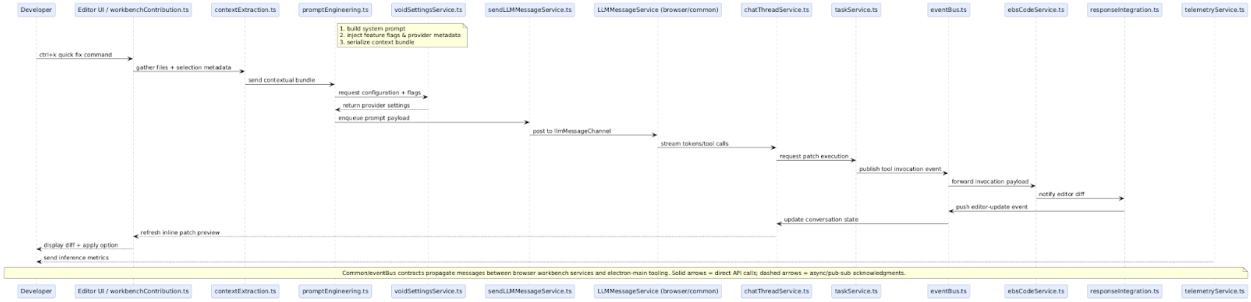



*Figure. Sequence Diagram for Ctrl K*

## 6.2 Provider Onboarding & Model Refresh

Settings UI stores credentials with `voidSettingsService`, `bridgeValidation.ts` and `LLMMessageService`/provider adapters stream model lists, and the Event Bus notifies chatThreadsService plus dependent apply/autocomplete services before Void Settings Service confirms readiness to the developer (Figure).

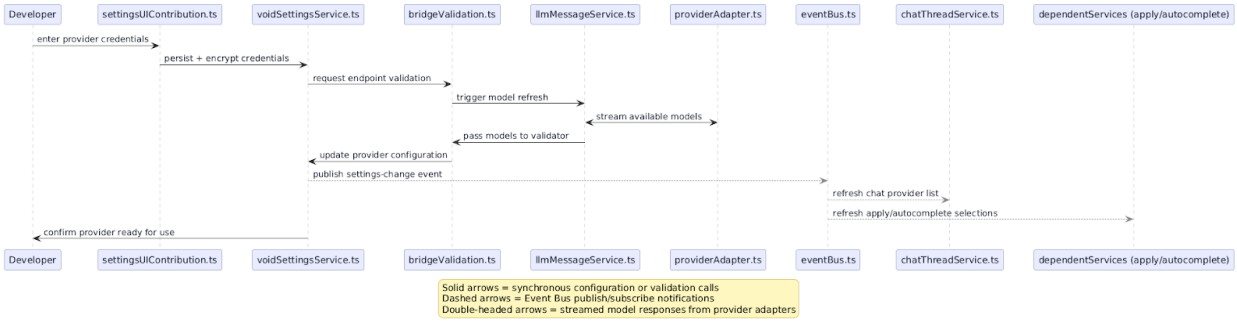



*Figure. Sequence Diagram for provider onboarding & model refresh*

# 7. Lessons Learned

Based on the feedback provided for A1, the team dove deeper into the research and documentation of Void to reconstruct the conceptual architecture and study the concrete architecture. The team revisited the documentation found in Void's Github and the supplementary materials provided on OnQ, finding several resources and article subcategories that had been overlooked. This allowed a more accurate reconstruction of the conceptual architecture.

Building on the conceptual, the concrete design was developed and the team learned to use Understand to create the diagrams. The team learned to organize source code files in the right place to construct the dependency graph and how to interpret it. During the process of file sorting and placing them in their corresponding layers from the source, key differences between the conceptual and concrete designs were identified. Specifically, many of the subcomponents from the service layer in the conceptual diagrams were discovered in the common (channel) layer of the actual implementation.

The team also discovered how Void uses VS Code's pub sub style, enabling asynchronous communication between components. This increased understanding on how the services like chatThreadsService and LLMMessageChannel communicate through events rather than direct calls, improving modularity and extensibility.

Finally, the differences observed between the conceptual and concrete architectures emphasized the complexity of Void's design and how multiple interdependent components and layers contribute to its overall functionality.

# 8. Conclusion

From using the source code of Void and putting it into Understand to help us map out which part to which subsystem and subcomponent, we created the concrete architecture that focused mainly on the LLM Message Pipeline system that was unique to Void.

Comparing the conceptual and the concrete diagram, many components that were in the server layer moved to the channel layered in the concrete diagram when we analyzed the VoidEditor folder code, in src, vs, workbench, and void folder, we found the three layer of LLM Message Pipeline which are browser (server layer), electron-main (implementation layer), and common (channel layer). This supports the style of layering in void.

As well with the reflection analysis between conceptual and concrete, it can be observed that many of the components that were originally in the Service layer (browser) of the conceptual model were moved to under the Channel Layer (common) in the concrete architecture. This shift shows how Void centralized shared interfaces, types, and events handlers in a common folder so that both browser and electron main processes can access them. This architecture relies on pub sub communication style, particularly Workbench and LLM Message Pipeline. The concrete system shows event callers to allow components like LLMMessageChannel, chatThreadService, and editCodeService to publish updates asynchronously to subscribed UI elements and services. Enforcing the pub sub style and responsiveness as well as modularity while keeping the loose coupling across layer for less dependencies. We can also observe in the concrete dependencies

graph, that the three layers of LLM Message Pipeline didn't have too many dependencies on each other, as in the other two layers have one way dependencies only on the Service layer.

From the reflection diagram for LLM Message Pipeline subsystem, the implementation layer and the server layer are both dependent on the channel layer that was derived from the concrete architecture diagram. This is mainly because of the many UI files, interface, and types files components that were moved from the server layer to the channel layer.

# Appendix A: Data Dictionary

**Electron Framework:** Open-source software framework designed to facilitate desktop app creation using web technology

**Large Language Models (LLMs):** AI Models that understands and generate human language

**Browser Process:** Void's process that is related to user interfaces and app functionality

**Main Process:** Void's process that handles core operations such as file management, API calls, and communication with large language models (LLMs)

**Monaco Editor:** Code editor shared by VS code and Void, does code validation and syntax colourization

**Application Programming Interface (API):** Communication protocol between software components

**Integrated development environment (IDE):** A software that allows programmers to develop code through an interface

**Interprocess Communication (IPC):** Shared data between running processes

**User Interface (UI):** Point of interaction between user and a digital product

# Appendix B: Naming Convention

The naming convention of our components reflects real module and service names found in Void's codebase and official documentation. We also used camel case when naming components since that is the naming convention used by void.

# 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 for most research-based topics due to the huge amounts of inaccuracies and false information in AI-generated text. Therefore, AI usage was generally 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.

The amount of files in Void's source code made it overwhelming and time consuming to analyze, so AI was used to summarize and interpret it. It also aided the selection of the files in Void's source folders to better understand subcomponent interactions and dependencies. Previously, it was recognized that Void also uses the publish-subscribe style, so the team looked into how it was implemented. The AI teammate pointed out that this design allows Void to support asynchronous communication between components such as chatThreadsService and LLMMessageChannel. Additionally, using the AI-teammate helped us see the discrepancies between the conceptual and concrete throughout the process of framing the diagrams in Understand. For example, the AI was prompted to help figure out what is considered UI files and what subsystems are unique to Void, such as LLMMessage. The team saw that a new subcomponent called LLMMessage was created from the source code and was able to confirm with ChatGBT.

## 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. When a team member was prompting the AI model, they also made sure to include background information to ensure accuracy and consistency amongst the results.

## Validation and Quality Control Procedures

Whenever AI was used to proofread a section, team members would prompt an AI model to suggest improvements or considerations, 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.

For prompts relating to file examining, our team members made sure to cross reference the recommended files to ensure their relevancy. This way, it minimizes the risk of inaccurate or hallucinated outputs.

## 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 15% in terms of word count, despite how definitive concepts in the text would most likely be identical, albeit with worse structures. Most of the research and derivation processes were completed by the human team members, with some help from the AI team member to make the process a little smoother. Additionally, all diagrams were produced by a human team as AI diagrams are often inaccurate and missing elements.

## 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. While the team tried to strive for consistency by providing similar background information where relevant, it is inevitable that using the AI teammate created disparities in style, detail, and emphasis throughout the report.

Compared to the A1 report, AI was used more frequently to aid the research process and analyze large quantities of code and files. While this teammate occasionally provided useful suggestions, it was often misinterpreted so the group needed to check the source code to ensure accuracy and re-prompt the AI model with more detail. While the support from the AI teammate allowed the research to progress more smoothly when the team was stuck in a particular phase, the team members still refrained from relying heavily on it.

# Works Cited

voideditor. “Void/VOID_CODEBASE_GUIDE.md at Main · Voideditor/Void.” *GitHub*, 2024, github.com/voideditor/void/blob/main/VOID_CODEBASE_GUIDE.md.

zread.ai. “LLM Message Pipeline Architecture.” *Zread.ai*, 9 Oct. 2025, zread.ai/voideditor/void/10-llm-message-pipeline-architecture.

zread.ai.“Void Core Module Structure.” *Zread.ai*, 9 Oct. 2025, zread.ai/voideditor/void/9-void-core-module-structure.