CISC 322
Assignment #2 - Report

Concrete Architecture of Void

November 7, 2025

Team Rocket
Adam Abou Zaki (22dgl3@queensu.ca)
Mehr Chelani (22twb2@queensu.ca)
Miller Gao (22fy14@queensu.ca)
Sarah Mohammad (sarah.mohammad@queensu.ca)
Wangyi Lu (22xb30@queensu.ca)
Yubo Wu (22cm58@queensu.ca)

Concrete Architecture of Void

Abstract

This report covers the concrete architecture of Void, a VS Code fork, by adopting and slightly modifying a conceptual architecture that places four Void components on top of VS Code's Workbench, Editor, Platform, File System and Base to provide a clear, top-level frame. Using the extracted dependency file, VoidEditor, and the Understand software, the team iteratively grouped code files and folders into the components dissected in A1, produced a readable high-level diagram, and labelled cross-component edges with concrete connections to form a structure design at the "right level". The resulting concrete view reveals ten code-backed components with a process placement around Electron's browser and main processes made explicit for traceability. A subsystem deep dive is provided to decompose the Void subsystem within this architecture into sub-components backed by source code and explicit concrete dependencies. Reflexion analyses are provided for both the high-level architecture and the Void subsystem to document key divergences and discrepancies between the conceptual and concrete architecture based on source code. Two use cases, Quick Edit and Agent Mode, are presented with supplementary sequence diagrams whose method calls can be traced to the concrete dependencies.

Updated Conceptual Architecture

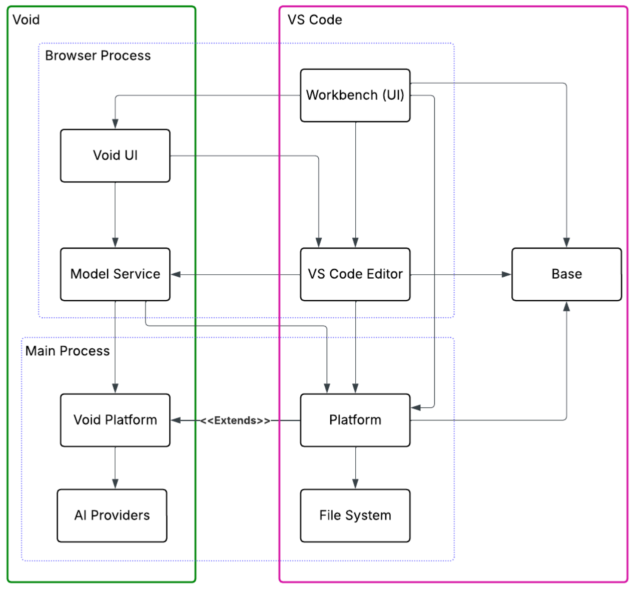

Figure 1. Conceptual Architecture
Legend:
→ Solid arrow - dependency (A → B = A dependent on B)

This is our updated conceptual architecture derived from Group 22's A1 deliverable. It consists of a higher-level abstraction that includes Visual Studio Code (VS Code) components and Void

Components. The VS Code components are Workbench (UI), VS Code Editor, Platform, File System, and Base. The Void components are Void UI, Model, Service, Void Platform, and AI Providers. These components are dependent on and interact with each other to provide the Void Editor that is built as a fork of VS Code. It also highlights components within the browser process and the main process of the whole system separately.

VS Code Components
VS Code is organized around the Workbench shell that hosts window layout, terminals, and view/command routing while coordinating the Monaco-based VS Code Editor, with both relying on Platform services for configuration, IPC, process startup, and file access that typically surface File System capabilities through Platform rather than direct calls. The Editor provides text models, language features, and editing commands that the Workbench instantiates and displays, while Base supplies shared primitives (events, disposables, async helpers) used pervasively by Workbench, Editor, Platform, and the Extension System that manages extension loading/activation and the API surface extensions use to contribute views, commands, and language providers.

Void Components
Void augments the Workbench with Void UI elements such as a prompt panel, model selection, and inline AI tools that gather editor/user context and submit requests to the Model Service. The Model Service bridges UI/editor context to the main-process path by shaping requests and forwarding them to Void Platform, then streaming responses back to Void UI for display. Void Platform extends the host Platform with AI-aware commands and IPC endpoints that route requests to pluggable AI Providers, leveraging Platform facilities and shared Base utilities while remaining provider-agnostic at the UI/editor boundary. AI Providers represent the actual back ends that receive well-formed requests and return completions or chat responses, allowing deployments to swap models without changing the higher-level editor and UI layers.

Rationale for Updating Conceptual Architecture
Adopting Group 22’s A1 for our conceptual architecture is justified because it presents a clear VS Code top-level with the addition of four well-scoped Void components. This aligns with the assigned task for this assignment and is something we failed to consider as part of our A1 conceptual architecture. Group 22’s architecture models Void as a VS Code fork on a higher level of abstraction than our conceptual architecture in A1, which focuses more solely on the architecture on the Void subsystem alone. We chose Group 22’s architecture specifically to adopt because their organization displays the layered client-server architectural styles present in Void and VS Code. The minor adjustments we made were to include boxes to separate the Void components and the VS Code components for a higher-level abstraction.

Derivation Process

We derived the concrete architecture by executing the following steps. We first reviewed and compared our own conceptual architecture to those of other groups and decided to adopt Group

22's. Then we used the course-provided Understand file to create an architecture beginning with the conceptual components. We continued by going through the files and folders in the VoidEditor repository, looking for key folder names and file processes to guide us toward a working organization of the components in our architecture. This iterative process included expanding folders, sifting through sub-folders, reading code and comments, and mapping files and folders to their respective components. We also made use of our AI teammate to help us decide where some of the files and folders should reside. During this process, we realized we needed to add a component, the Extension System, to handle the copious amount of extension services. Once our files were organized, we produced a dependency graph of our created architecture. This revealed the interactions and dependencies of our concrete components, which we then modelled in a more easily readable diagram. We kept the Void and VS Code component boxes separate on a higher level of abstraction and the browser and main process outlines for clarity.

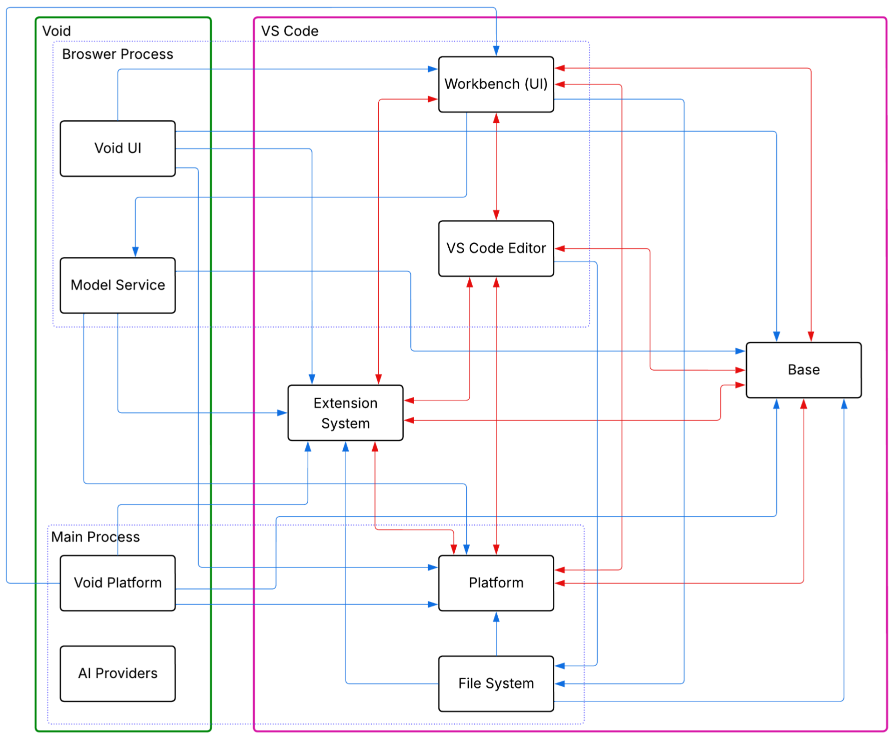

Figure 2. Concrete Architecture
Legend:
→ Blue Arrow - single-direction dependency
↔ Red Arrow - bidirectional dependency

High-Level Concrete Architecture

The concrete architecture of the Void Editor is manifested as a layered, Electron-based application. It extends the original VS Code structure with several AI-oriented subsystems. The code-level analysis performed in Understand reveals nine recurring subsystem blocks, with the addition of one: Workbench (UI), VS Code Editor, Platform, Base, File System, Void UI, Model

Service, Void Platform, AI Providers, with the addition of Extension System. Between them, these subsystems capture both the original VS Code responsibilities and the new Void features.

The figure shows the final, high-level, concrete architecture we reconstructed. It is based on the directory-level dependency graph extracted from the VoidEditor repository (src/vs/*, extensions/*, scripts/*) using Understand.

VS Code Components

Workbench (UI)

The Workbench is realized in src/vs/workbench/ and consists of the visible shell, command routing, and view contributions that host the Monaco editor and Void UI. This is confirmed by mapping the modules identified in the Understand dependency view to the conceptual Workbench component. It depends on Platform services for configuration, file dialogues, and IPC, exposed through the main process APIs, and it emits UI events and commands that the Editor and Void UI handle. Important outbound edges include: Workbench ↔ VS Code Editor for open/close/execute commands and for editor lifecycle, Workbench → Platform for startup, configuration, and IPC bootstrap events, and Workbench ↔ Base for shared utilities like event/disposable helpers used across layers [1,3].

VS Code Editor

The VS Code Editor component is realized in src/vs/editor with entry points editor.main.ts, editor.api.ts, editor.all.ts, and editor.worker.start.ts, and subfolders including standalone, which is used for Monaco host integration. Important dependencies include: VS Code Editor → Platform, where the editor consumes services exposed by the main process and relies on configuration and communications services, and VS Code Editor ↔ Base as the editor relies on shared utilities like events and helpers and exposes base-typed events used elsewhere [1,3].

Platform

The Platform component resides in src/vs/platform under the Electron main process and exposes OS and host services to the browser-process components. It includes service packages such as configuration, files, IPC, commands, and extensionManagement that implement the boundary and host integration. Key dependencies include: Platform → Workbench (UI), where the platform notifies the workbench of lifecycle and settings changes and Platform ↔ Extension System, where the platform manages extension hosting/activation and exposes extension management services, while the Extension System uses Platform to register, activate, and communicate with extension processes [1,2,3].

File System

The File System component provides workspace I/O, watching, and path/URI services that other subsystems call for opening, saving, and monitoring resources. Some important dependencies are: File System → Platform to access main process adapters like disk I/O, dialogues, and handlers surface events. File System → Extension to expose workspace file operations used by extensions, and VS Code Editor → File System for buffer persistence, model, reloads, and file watching [1,2,3].

Base

The Base component is realized under src/vs/base and is a shared utility layer. Almost all other components depend on it because it provides foundational primitives such as event emitters/observables, disposables, async/cancellation tokens, logging, and UI parts. Some of the Base double-dependencies were discussed above, and the few additional key ones include: Base ↔ Platform, where services are implemented with primitives, and Base ↔ Extension System, as the extension system uses Base's event/disposable patterns for activation and exposes contracts to extensions [1,2,3].

Extension System

The extension system is VS Code's built-in framework for loading, activating, and hosting extensions for registering commands, views, language features, and contributions across the UI and editor. This component bridges between the browser process and the main process while relying on Base primitives. Almost all other components depend on the extension system because it provides these essential services [1,2,3].

Void Components

Void UI

Void UI runs in the Electron browser process and implements the chat/prompt panel, model selector, and inline AI tools as views within the Workbench shell using VS Code. At this level, important Void UI dependencies include Workbench (UI) for hosting/command routing and register views and Platform for configuration and IPC for main processes [1,3].

Model Service

The Model Service component runs in the browser process and orchestrates AI requests. It collects context from the Workbench UI and VS Code Editor components, prepares prompts, and forwards them to the main-process path via the Platform component, then returns the results to the UI. Key dependencies consist of: Workbench (UI) → Model Service, since the workbench triggers AI operations like inline completion or chat actions, and Model Service → Platform sends AI requests across the process boundary using IPC to reach the main process [1,3,6].

Void Platform

The Void Platform runs in the main process with the VoidEditor/scripts folder and acts as the AI bridge between the browser side (Workbench UI and Model Service) and privileged main services (commands and IPC endpoints) that route requests to configured providers. It contains bootstrapping and main (.ts) scripts showing its responsibility for hosting the AI runtime surface. Its important dependencies are: Void Platform → Platform since it uses the core Platform services to expose and service AI commands and host tasks, and Void Platform → Workbench (UI) to register and expose commands and panels for user interactions [1,3,6].

AI Providers
The AI Providers component is an external component with no static incoming or outgoing dependencies. This reflects that, rather than in-repo modules, provider SDKs/endpoints are outside of the codebase and are reached at runtime through API calls. Ollama, which is integrated by Void, receives network calls from the host, but the calls do not appear as code-level dependencies [1,3,6].

Void Subsystem Concrete Architecture

This section focuses on the Void-specific subsystem instead of the VS Code platform as a whole. We discovered after looking at the actual source tree that Void is actually implemented as a small pipeline of collaborating components that spans both the main process and the browser process. This pipeline's high-level components are Void UI → Model Service → Void Platform → AI Providers.

Conceptual structure of the Void subsystem

At the conceptual level, the Void part of the system is an AI request pipeline that sits on top of the regular VS Code browser/main-process split. User engages with AI features through the browser process's Void UI; the Model Service enhances the user's intent with context from the editor or workspace; the request is then routed to the main process, where the Void Platform carries it out; and finally, the AI Providers provide a response or completion that is sent back to the user interface. Each component in this idealized view has a single responsibility, and the dependencies are arranged in a clean linear chain: user interface → model logic → platform → provider. This conceptual framework assumes that the VS Code workbench is primarily a host, loading the Void UI and making editor state accessible. Similarly, the main-process extension ("Void Platform") is viewed in the conceptual diagram as a single block that merely "extends VS Code platform to talk to AI," with no indication of how this extension is actually integrated into the actual platform/bootstrap code. This is the version created in A1, and it corresponds with the Void paper's description.

Concrete decomposition in the source code

When we mapped the repository in Understand, we found that the four conceptual blocks are present, but each is realized by multiple code artifacts.

The Void UI is implemented inside the browser-side code that VS Code uses to start the workbench, specifically under src/vs/code/browser and the surrounding workbench view/contribution code. This indicates that instead of getting a stand-alone SPA, the UI is an addition to the current workbench. Specifically, it uses the browser process to read the editor state and calls workbench services to render views. Void UI → workbench/browser code is the first concrete dependency that this gives us that was not conceptually explicit.

The code for the conceptual Model Service is divided into a minimum of two services under the workbench services area: src/vs/workbench/services/aiRelatedInformation/ and src/vs/workbench/services/aiEmbeddingVector/.

The first file is responsible for creating or obtaining embedding vectors for any code or context that is included with the AI request. The second gathers or calculates "related information," which AI should also see. The Understand graph indicates that the data-preparation logic is shared and reusable because both of these services import helpers from a common location. Therefore, Model Service is not just "one service," but rather {embedding service + related-info service + common utilities} at the concrete level.

The Void Platform is realized by a set of main-process and bootstrap files rather than a single module. We noticed that the Void-specific service folders had dependencies on scripts/ and a collection of files in src/, including server-main.ts, bootstrap-cli.ts, and bootstrap-server.ts. This shows that the AI dispatch logic is integrated into the normal platform startup layer. The browser-side service does not talk to a “voidPlatform.ts”; instead, it talks over the platform’s IPC/command surface, and the code that actually forwards to providers is in the same location as the rest of VS Code's main-process orchestration. Thus, the concrete picture is Void Platform → scripts + platform bootstrap files.

In the concrete codebase we analyzed, we did not find a dedicated folder or module that implements AI providers inside the Void repository. This implies that providers are either external services or kept in a different repo. We still keep “AI Providers” as a box to preserve the end-to-end flow, but we mark it as a conceptual endpoint rather than a code-backed subsystem. Its presence is necessary to explain why the Void Platform exists, even though the actual provider code is not part of the analyzed source.

Interactions inside the Void subsystem

The first internal interaction is between the user interface (UI) and the model. When a user action initiates inline AI, the UI component forwards the request to the model layer within the same browser process. The model layer then decides which specific services to call. For example, if the request needs code-context embedding, it may call aiEmbeddingVector; if the feature needs additional metadata, it may call aiRelatedInformation. Both of these calls are handled by code under the src/vs/workbench/services/... directory, so this step is clearly tied to the workbench service infrastructure.

The second interaction is model to platform. The model layer must transfer a structured AI request to the main process once it has been assembled because it cannot finish it in the browser process. This is accomplished in code via the platform's IPC/command surfaces, which is why we observed dependencies between the platform bootstrap files, scripts/directory, and model service folders. Although it looks like actual imports to the main-process startup files in code, this is precisely the Electron/LSP-style split that was discussed in the conceptual diagram.

The third interaction is platform to providers. AI requests are initially received by the extended platform on the main-process side. After that, it selects the appropriate provider and sends the request to them. Because it might require credentials, network access, or file-system permissions that are only available through the Visual Studio Code platform, this logic must execute in the main process. Because of this, the Void Platform code is incorporated into scripts and platform bootstrap files, which are already running with the necessary permissions.

Finally, there is result propagation back to the UI. After the provider returns a response, the platform sends it back over the same IPC channel to the browser process. The model layer may again use the common utilities to post-process or format the response, and then the UI displays it in the chat/prompt panel or injects it as inline code. This completes the round trip and shows that the same components are used in both directions.

Reflexion Analysis

High-level

Extension System

Within the conceptual architecture, the Extension System does not stand out prominently and does not appear as a core component in the conceptual architecture diagram. However, in the concrete implementation, as evident from the concrete architecture diagram, the Extension System has become a critical node for the entire system. It not only establishes bidirectional dependencies with multiple modules within the Browser Process (Void UI, Workbench, VS Code Editor) but also maintains continuous communication with the Main Process (Platform, File System, AI Providers). This demonstrates the Extension System's evolving role as a “communication hub.” While this shift enhances system flexibility and increases coupling, but any failure within the Extension System could disrupt the entire system's data flow and impact fundamental services.

Changes in Dependency Direction Between Void UI and Workbench UI

In the conceptual architecture, the Workbench user interface is considered an outer container of the browser layer (whose subordinate functional module is Void UI). In this structure, the dependency is unidirectional: Workbench sends events or calls interfaces to Void UI, which only responds (Workbench → Void UI). However, in the actual implementation, this relationship has changed. Workbench is now responsible for the entire visualization shell, command routing, and view registration logic. The Void user interface registers functional views via the command system and view mechanism of the Workbench. The diagram has therefore been changed to reflect a bidirectional dependency (Workbench ↔ Void UI). This change reflects the integration of Void's capabilities into the Workbench framework. This has the advantage of making interactions smoother and allowing users to access AI capabilities through a unified interface. However, the disadvantage is that the boundaries between the modules become blurred, and maintenance and testing of the UI layer becomes more complex, as certain Workbench logic is now embedded in the Void function logic.

The centralized dependency of each module on the Base layer

In the conceptual architecture, the Base layer serves as a passive support layer, providing only generic utility functions and event mechanisms. However, in actual implementation, nearly all major modules directly or indirectly depend on Base. It contains foundational tools such as event triggers, asynchronous control, and UI components. Void UI, Workbench, Extension System, and Platform all utilize the foundational capabilities provided by Base. For example, Workbench employs Base's asynchronous and event functions to manage commands and views. Precisely because these modules rely on the same underlying interfaces, multiple sections in the diagram exhibit dependencies on Base. Simultaneously, some modules form bidirectional dependencies with Base. Base ↔ Platform and Base ↔ Extension System. The Extension System utilizes Base's event dispatch mechanism and registers callback functions back to the Base layer during plugin loading. This bidirectional communication enhances system flexibility but reduces layering clarity. On one hand, the Base layer, as a universal support layer, significantly reduces code duplication, unifies event handling and logging interfaces, and improves development efficiency. On the other hand, it also becomes the structural center of gravity for the system.
Once the underlying mechanisms of the Base module are modified, Platform, Workbench, and even Extension System may be affected. This high degree of centralization makes the system more efficient, yet also more fragile.

Dependencies pointing to Platform

In the conceptual architecture, the Platform is treated as an intermediary layer, responsible for relaying commands and data between the browser process and the main process. The Platform primarily receives requests from upper layers and invokes underlying system services or extension interfaces, with dependencies intended to be unidirectional: Browser → Platform. However, in concrete architecture, the Platform has become one of the most centralized dependency points in the system. As shown in the diagram, nearly all core modules(including Workbench, Void UI, Extension System, and Base) directly or indirectly depend on Platform. Consequently, Platform's dependency structure has evolved from unidirectional communication to bidirectional interaction. This shift has expanded Platform's role from a "communication bridge" to a "system hub." On the one hand, this structure enables rapid collaboration between different modules. For example, the Extension System can leverage the Platform to handle plugin activation and registration, thereby improving system responsiveness and integration efficiency. On the other hand, the centralization of the Platform introduces risks. Once the internal interfaces or event mechanisms of the Platform are adjusted, multiple modules at the upper layer will require simultaneous modification. This “multi-point dependency” makes the system more susceptible to cascading effects during updates and increases the complexity of debugging and testing. From an architectural perspective, the Platform's evolution reflects trade-offs made for efficiency during actual development. It enables more direct communication between modules but weakens layer independence.

Void Subsystem

Fragmented “Model Service” layer

At the conceptual level of the subsystem, Model Services should function as an independent service module dedicated to handling context modelling and request generation. However, at the concrete code level, this service module has been decomposed into multiple distinct sub-services, including aiEmbeddingVector, aiRelatedInformation, and several shared utility functions. As mentioned in the previous section, these services reside within the src/vs/workbench/services/... directory and exhibit mutual dependencies. Consequently, in practical implementation, the Model Service component functions more as a “service cluster” composed of multiple submodules rather than a single logical layer. While this approach achieves a degree of reusability, it also increases coupling and coordination complexity among the different sub-services.

Void Platform Coupling with VS Code Launchers

In the conceptual architecture, the Void Platform exists as an independent bridge module designed to connect the model layer and AI Provider. However, in the actual implementation(concrete architecture), there is no independent “voidPlatform.ts” file. Its related logic is actually embedded within VS Code's startup scripts and bootstrap files, such as server-main.ts, bootstrap-cli.ts, and bootstrap-server.ts. In other words, Void's main process logic is directly integrated into VS Code's startup layer. So this design facilitates platform-level integration, but from an architectural perspective, it means the boundary between Void Platform and the VS Code platform becomes blurred. This reduces the independence between subsystems, making subsequent testing and replacement more challenging.

Absence of the AI Provider module

As mentioned in the previous section, the AI Provider module is depicted within the conceptual architecture layer as a component that should be incorporated within the Void system. However, no corresponding directories or file implementations were identified within the actual code repository. Consequently, the AI Provider is retained at the conceptual level to maintain procedural integrity, though at the concrete level, there is no AI Provider that is supported by actual code.

Implicit Dependencies in Inter-Process Communication

At the conceptual design level, communication between the Model Service(browser process) and the Platform(main process) should strictly occur via IPC or through command channels. However, the dependency diagram reveals that certain Model Service modules directly reference scripts within the main process. This indicates the existence of implicit dependencies or shared content between them. Logically, this cross-process injection breaches the isolation between the browser process and the main process, complicating the system's layered architecture. This contravenes the fundamental principle of frontend-backend separation established at the conceptual level.

Use Cases & Diagrams

Use Case 1: Quick Edit

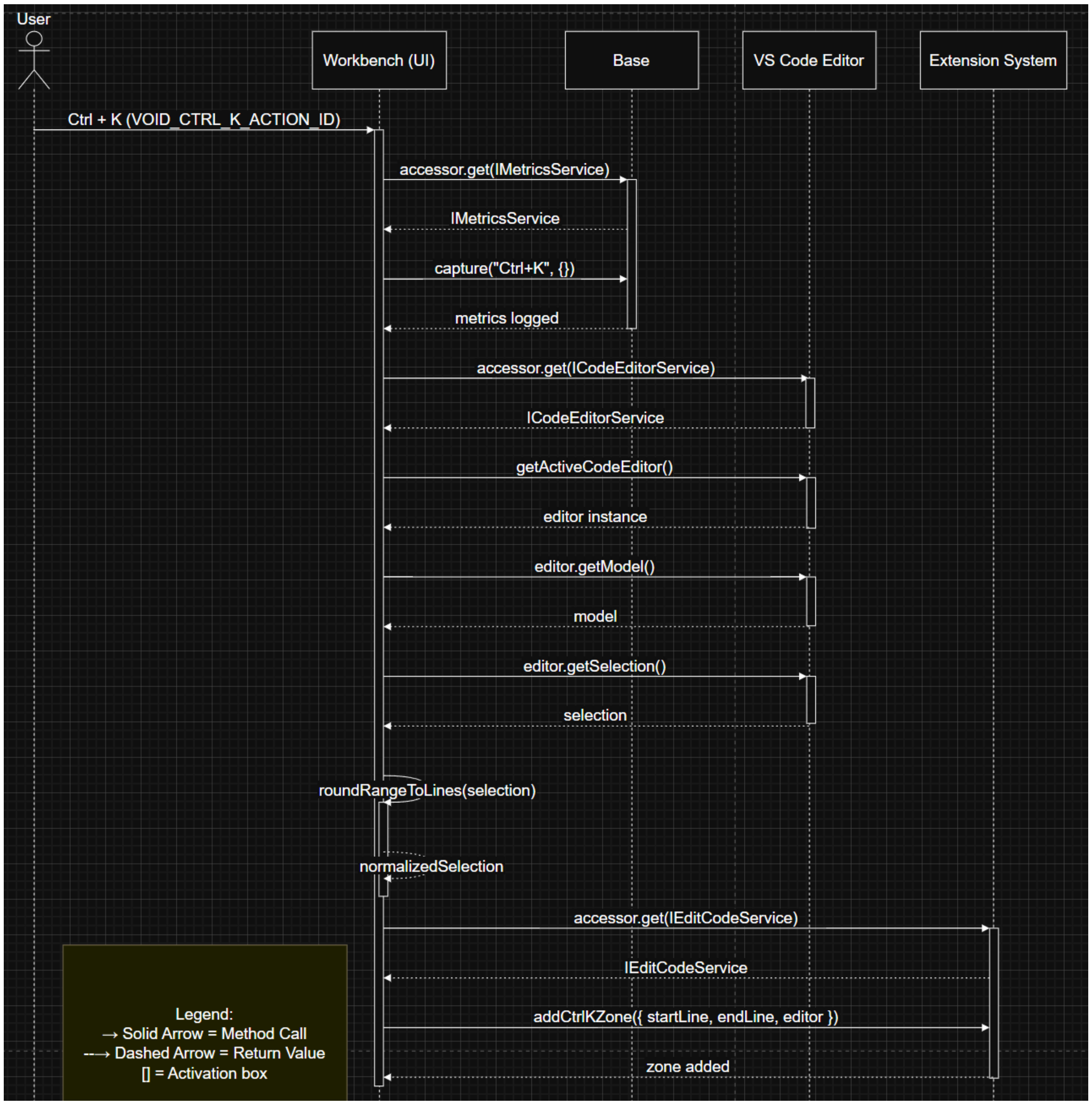

Figure 3. *Use case 1: Quick Edit*

This use case describes how the Void Platform handles the Quick Edit (Ctrl + K) command within the editor environment. When the user triggers the Ctrl + K shortcut, the Workbench (UI) receives the action through the registered VOID_CTRL_K_ACTION_ID event. The Workbench first accesses the Base component to log the event using the capture("Ctrl+K") method from the Metrics Service, ensuring that editor actions are tracked for analytics. It then requests an active editor instance from the VS Code Editor component through the getActiveCodeEditor() method. Once the active editor is obtained, the Workbench retrieves the text model and current selection region by invoking editor.getModel() and editor.getSelection(). The selection is then normalized internally using the helper function roundRangeToLines(), which aligns the selected lines for proper editing. After preparing the selection range, the Workbench calls the Extension System through the addCtrlKZone({startLine, endLine, editor}) method to create an editable zone in the document. The Extension System interacts with other subsystems to prepare the editing interface and returns confirmation to the Workbench. Finally, control is released, completing the Quick Edit activation process.

Use Case 2: Agent Mode – Multi-File Refactor and Test

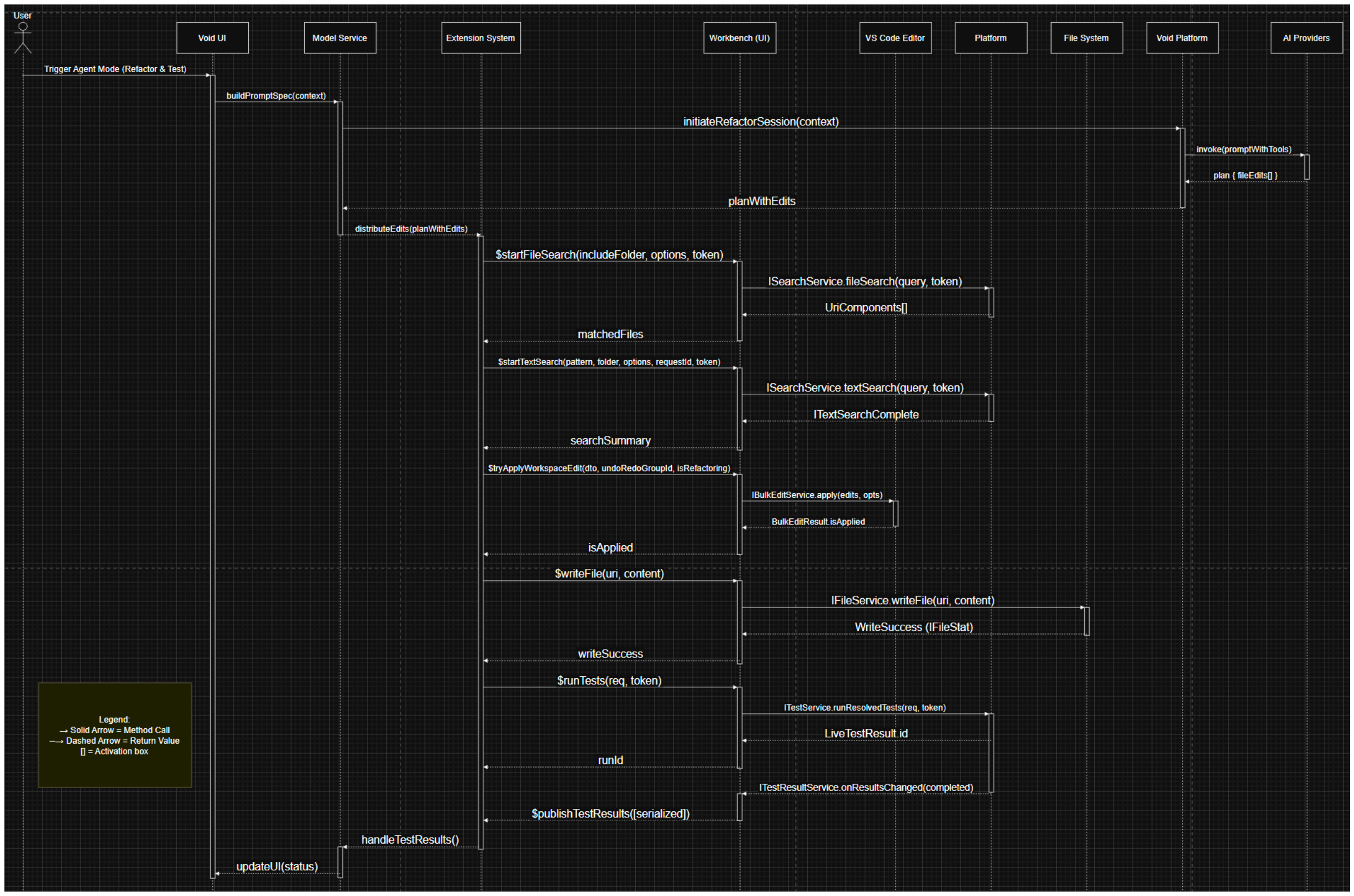
User
Void UI
Model Service
Extension System
Workbench (UI)
VS Code Editor
Platform
File System
Void Platform
AI Providers
Trigger Agent Mode (Refactor & Test)
buildPromptSpec(context)
initiateRefactorSession(context)
invoke(promptWithTools)
plan { fileEdits[] }
planWithEdits
distributeEdits(planWithEdits)
$startFileSearch(includeFolder, options, token)
ISearchService.fileSearch(query, token)
UriComponents[]
matchedFiles
$startTextSearch(pattern, folder, options, requestId, token)
ISearchService.textSearch(query, token)
ITextSearchComplete
searchSummary
$tryApplyWorkspaceEdit(dto, undoRedoGroupId, isRefactoring)
IBulkEditService.apply(edits, opts)
BulkEditResult.isApplied
isApplied
$writeFile(uri, content)
IFileService.writeFile(uri, content)
WriteSuccess (IFileStat)
writeSuccess
$runTests(req, token)
ITestService.runResolvedTests(req, token)
LiveTestResult.id
runId
ITestResultService.onResultsChanged(completed)
$publishTestResults([serialized])
handleTestResults()
updateUI(status)
Legend:
→ Solid Arrow = Method Call
-→ Dashed Arrow = Return Value
[] = Activation box

Figure 4. *Use case 2:* Agent Mode – Multi-File Refactor and Test

This use case describes how the Void Platform executes the Agent Mode (Multi-File Refactor and Test) process to perform autonomous, large-scale code modifications. When the user activates Agent Mode through the Void UI, the Model Service constructs a refactoring prompt based on the current workspace context and sends it to the Void Platform. The platform then delegates the request to connected AI Providers by invoking the invoke(promptWithTools) method, enabling the generation of a detailed refactor plan containing multiple file edits. Once the plan is received, the platform returns it to the Model Service, which distributes the structured edits to the Extension System through the distributeEdits(planWithEdits) call. The Extension System coordinates workspace analysis by invoking $startFileSearch() and $startTextSearch() on the Workbench (UI), which in turn queries the Platform layer through the ISearchService to locate affected files. After collecting file paths and code patterns, the Extension System applies the AI-generated edits by calling $tryApplyWorkspaceEdit(dto, undoRedoGroupId, isRefactoring) via the VS Code Editor's BulkEditService. Once the modifications are applied and saved through the File System, the Extension System triggers automated testing using $runTests(req, token) from the Platform's ITestService. Test outcomes are returned to the Workbench, serialized, and propagated through the Model Service, which updates the Void UI with the refactor and test results, completing the full Agent Mode workflow.

Data Dictionary

URI - Uniform Resource Identifier: A unique sequence of characters that identifies an abstract or physical resource

SDK - Software Development Kit: A package of development tools (including libraries, APIs, documentation, compilers, code samples, and utilities).

Conclusions

In conclusion, compared to the initial conceptual design, the concrete model exposes more detailed communication paths, particularly across the Workbench, Platform, and Extension System, where asynchronous interactions and service calls form the backbone of functionality. Moreover, the analysis revealed that real-world implementation requires balancing modularity with interdependence. The concrete architecture focuses on how user actions, such as editing and testing, are processed across multiple subsystems. Overall, this study deepens the understanding of how theoretical architecture adapts under practical constraints. By observing dependencies and workflows within Void, we gained insight into the flexibility and maintainability required in large-scale, extensible software architecture.

Lessons Learned

Constructing the concrete architecture of Void revealed that translating a conceptual design into an implemented system exposes many hidden dependencies and integration points. The Model Service, initially viewed as a single unit, emerged as several smaller modules that improved flexibility but introduced tighter coupling.

We also discovered the importance of accurately tracing inter-process communication among the Workbench, Platform, and Extension layers. Message directions and data flow were often unclear from code alone, requiring careful validation through dependency and sequence diagrams.
From a collaborative perspective, maintaining consistency between diagrams, source code, and documentation proved crucial. While AI tools accelerated drafting and visualization, human reasoning remained essential for interpreting architectural intent and validating design logic. Overall, this project deepened our understanding of how theoretical architecture evolves into complex, maintainable, and layered real-world systems.

References

1. Dobaj, M., Thomas, S., Batala, B., Mirchandani, E., Mourad, S., & Kodukula, N. (2025). *A1: Conceptual Architecture of Void CISC 322 Authors*. https://babinson2005.github.io/CISC322/assets/deliverables/A1-Conceptual%20Architecture%20Report-Group%2022.pdf

2. *Extension API*. (n.d.). Code.visualstudio.com. https://code.visualstudio.com/api

3. microsoft. (2025, October 11). *Source Code Organization*. GitHub. https://github.com/microsoft/vscode/wiki/Source-Code-Organization

4. *Process Model | Electron*. (n.d.). Electronjs.org. https://www.electronjs.org/docs/latest/tutorial/process-model

5. *VoidEditor/Void*. (n.d.). Zread. https://zread.ai/voideditor/void/1-overview

6. (2025). Github.com. https://github.com/voideditor/void

AI Collaboration Report

AI Member Profile and Selection Process

- GPT-5 (OpenAI) – a large language model used for text generation, refinement, and architectural reasoning. GPT-5 was chosen for its advanced reasoning, technical writing, and context-aware architecture analysis, outperforming other models (e.g., Gemini 2.0, Claude 3) in producing coherent, structured documentation.

- Mermaid Diagram AI tool – used to automatically generate sequence diagrams from the textual descriptions we provided.

Tasks Assigned to the AI

Sequence Diagram Generation – GPT-5 drafted Mermaid code from the provided descriptions, including real message calls, of sequence diagrams that were converted by Mermaid AI into finalized diagrams.

Proofreading and Polishing – GPT-5 performed grammar and clarity checks, ensuring consistent terminology.

AI Report Drafting – GPT-5 generated the first outline of this collaboration report

Understand file dragging and organization - GPT-5 helped decide where certain files belong within our architecture in Understand

Interaction Protocol and Prompting Strategy
Our group maintained a shared prompt document to coordinate all AI interactions, allowing every member to contribute edits, add context, and refine instructions collaboratively.

Prompts were developed through a structured three-step process: Firstly, clearly defining the course, deliverable, and technical objectives; Secondly, specifying the required output format; Lastly, reviewing AI outputs, identifying inaccuracies, and adjusting prompts to improve clarity and technical precision.

Example prompt: "Review the following use case description for clarity and coherence. Suggest improvements in wording, structure, and technical accuracy while keeping the tone formal and consistent."

Validation and Quality Control Procedures
To ensure the accuracy and reliability of AI-generated content, our team implemented a multi-step validation and review process:

- All statements produced by GPT-5 concerning design principles, architectural patterns, and technical terminology were cross-verified with lecture materials, course readings, and textbook sources to confirm factual correctness.

- Each Mermaid-generated diagram was manually compared with the team's finalized use cases and implemented logic to ensure consistency between visual representations and actual system behaviour.

- Human team members reviewed all GPT-5 outputs for clarity, conciseness, tone consistency, and compliance with assignment requirements.

Quantitative Contribution to Final Deliverable
GPT-5: 25% (drafting, editing, formatting, and consistency)
Mermaid AI: 5% (diagram generation)
Human team: 70% (concept development, validation, integration, analysis)

Reflection on Human-AI Team Dynamics
Overall, integrating AI collaborators improved efficiency and workflow coherence. GPT-5 streamlined brainstorming and editing, while Mermaid AI eliminated manual diagram formatting. The AI tools encouraged more structured communication within the group, as members needed to articulate precise instructions and evaluate outcomes critically. Minor disagreements arose when interpreting AI suggestions, but these discussions deepened our understanding of architectural reasoning.