Proposing an Architectural Enhancement to the Void IDE

Project: Void IDE - AI-Assisted IDE built on VS Code and Electron
Deliverable: A3 - Architectural Enhancement
Course: CISC 322
Group Members: Colin McLaughlin (Group Lead), Adrian Yanovich-Ghitis (Presenter), Thomas Schrappe (Presenter), Jaiman Sharma, Mantaj Toor, Lingwei Huang
Date: December 1st, 2025

Abstract

This report proposes an extension to the existing command line interface (CLI) included with Void. Although Void incorporates a CLI for standard editor operations like file management and extension handling, none of its AI or LLM capabilities are available outside the GUI and workbench. Extending the CLI to support AI-driven workflows would allow developers to carry out code generation, refactoring, and contextual analysis directly from the terminal. This would enable headless execution, automation through scripts, and integration with continuous integration (CI) pipelines. The enhancement, as a result, fills a very clear gap in the current system and expands the environments/situations in which Void can actually be used.

The report outlines the architectural changes required to support AI-enabled CLI functionality and explains how the modification affects core subsystems like AI Services, Provider Integration, Platform Services, and the Build and Infrastructure layer. Two representative use cases are presented, and a sequence diagram illustrates the proposed flow of control and data. The analysis examines impacts on maintainability, evolvability, testability, and performance, and evaluates interactions with existing features, including chat-based code generation, inline suggestions, tool calls, and file-level edits. Key risks related to security, performance overhead, and architectural complexity are also discussed.

Two architectural alternatives for implementing the enhancement are evaluated using a SAAM-based approach. The first integrates AI operations directly into Void's existing Node and Rust-based CLI pipeline. The second introduces a separate Model Server that exposes AI functionality to the CLI while remaining independent of the workbench environment. As will be discussed later in the report, the recommended alternative is the Model Server based approach because it offers stronger portability, clearer separation of concerns, and better long-term maintainability. The report concludes with lessons learned and discusses the team's collaboration process with our AI teammate.

---

1. Introduction

1.1 Purpose of the Report

As mentioned in the Abstract, the purpose of this report is to analyze and propose an architectural enhancement for Void. A1 and A2 examined the system's conceptual and concrete architectures, which provided a clear understanding of its layered structure, its use of Electron and the Extension Host, and the responsibilities of major subsystems such as AI Services, Provider Integration, Platform Services, the Editor Core, and the Workbench Framework.

Building on that foundation we made, our A3 report introduces an extension to the system's existing CLI. The enhancement aims to make Void's language model capabilities available directly from the terminal, allowing developers to access AI powered functionality outside the graphical workbench.

1.2 Motivation for the Enhancement

Void currently exposes all AI related features through its GUI. Code generation, refactoring, contextual analysis, and other AI driven workflows depend on UI components, editor instances, and workbench contributions. Although Void does include a CLI, it supports only basic and more traditional editor operations like opening files, managing extensions, or running diagnostics, and it does not provide any access to AI capabilities. Extending the CLI to support AI related operations would allow developers to use these tools in scripts, CI pipelines, remote development environments, or anytime a GUI is not preferred/available. This enhancement fills a meaningful gap in the system and increases the flexibility and reach of Void.

1.3 Scope of the Enhancement

This report examines the architectural implications of adding AI functionality to the CLI. The analysis identifies which subsystems require modification, how the new interface interacts with existing services, and what types of control and data flow define the enhanced system. Specific attention is given to AI Services, Provider Integration, Platform Services, and the Build and Infrastructure layer, because these components form the foundation on which the extended CLI and supporting Model Server must operate. The report includes two use cases and respective sequence diagrams that help illustrate expected system behavior and help clarify how CLI commands travel through the modified architecture.

1.4 Overview of the Proposed Approaches

To determine the best way to support the enhancement, the report presents two architectural alternatives. The first involves integrating AI capabilities directly into Void's existing Node based and Rust based CLI pipeline. The second introduces a dedicated Model Server that reuses shared AI and provider logic while operating independently of the workbench environment. A SAAM based analysis then compares both alternatives from the perspective of stakeholders and non-functional requirements like performance, reliability, maintainability, and testability. The section concludes with a recommendation based on the outcomes of this analysis.

---

2. Proposed Enhancement

2.1 Enhancement Description

Our proposed enhancement is adding compatibility for CLIs. In essence what we intend to do is create a Vim plugin that would allow for Void to be used through Vim macros. The user would interact with it ideally with whatever macro the user would like to set it to but it would come with a default macro. The default macro could use Vims built in recording mode so the macro could pull from the recording register. Alternatively it could be as simple as :void/[prompt]. Void autocomplete could also be enabled/disabled with a similar macro, and could be set to on/off by default. This would not only attract more people to use Void but also increase Voids reach to every single coder that prefers to code in a CLI.

2.2 Scope of the Enhancement

The enhancement adds CLI compatibility by providing a Vim plugin that exposes Void's capabilities through macros or commands such as :void/[prompt]. Functionally, the plugin allows Void to read and modify buffer contents, generate edits, and perform autocomplete within Vim. Non-functional expectations include low latency due to running in a terminal environment, full configurability of macros and behavior to match a user's Vim setup, and reliable operation without interfering with Vim's editing state. Key constraints include provider limits on Void's request throughput, IPC restrictions on how the plugin communicates with the local Void process or API, and boundaries imposed by the workbench model that prevent arbitrary execution or privileged actions. The plugin must only write directly to the file buffer and must not issue Vim commands, shell commands, or any action outside controlled text insertion to avoid code execution or security violations.

2.3 Two Primary Use Cases

<u>Use Case 1: Text autocomplete in Vim</u>

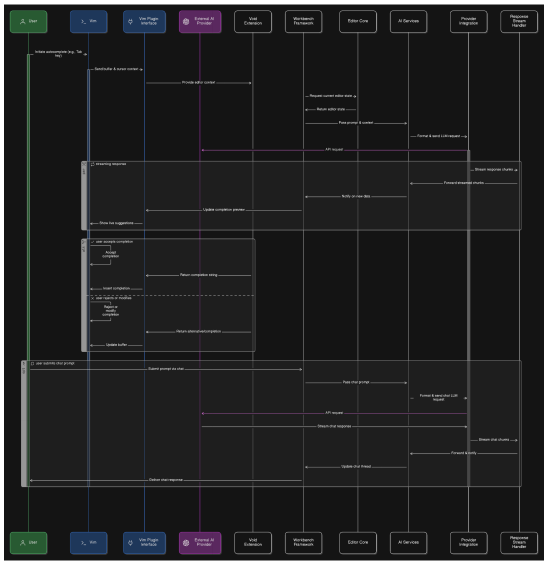
The user is editing a file in Vim and activates Void's autocomplete through a bound macro or command. Vim sends the current buffer context and cursor position to the Void plugin interface. Void processes the context and returns a completion string. The plugin inserts the completion directly into the active buffer without issuing additional Vim commands or modifying the editor state outside the text region. The user can accept, reject, or continue refining the completion using normal Vim operations. Autocomplete can remain active continuously or be triggered on demand, depending on the user's configuration.

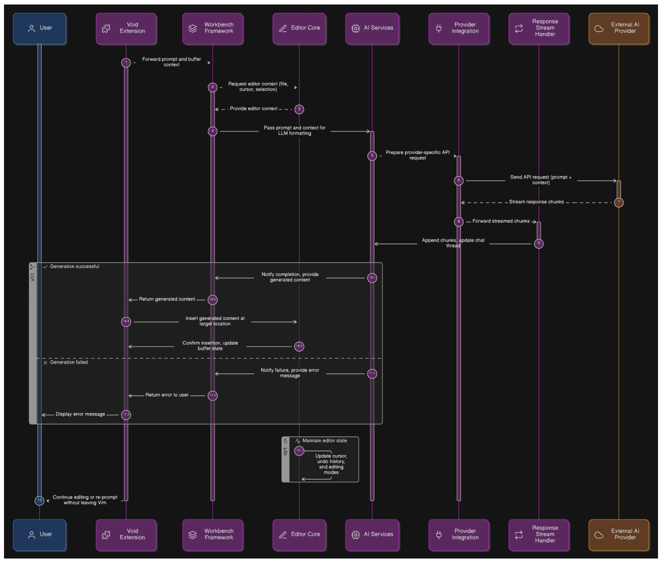
<u>Use Case 2: Prompting for code generation in Vim</u>

The user invokes a generation prompt using a macro or a command such as ':void/[prompt]'. Vim forwards the prompt and relevant buffer context to the Void backend. Void generates the requested code—such as a function template, refactoring result, or documentation block—and returns it to the plugin. The plugin inserts the generated content at the designated location in the buffer. Vim maintains full control of cursor position, undo history, and editing modes, ensuring the generated text behaves like normal typed input. The user can continue editing or re-prompt Void without leaving the Vim environment.

---

3. Behavioural View & Sequence Diagrams




User
Void Extension
Workbench Framework
Editor Core
AI Services
Provider Integration
Response Stream Handler
External AI Provider
Forward prompt and buffer context
Request editor context (file, cursor, selection)
Provide editor context
Pass prompt and context for LLM formatting
Prepare provider-specific API request
Send API request (prompt + context)
Stream response chunks
Forward streamed chunks
Append chunks, update chat thread
alt
Generation successful
Notify completion, provide generated content
Return generated content
Insert generated content at target location
Confirm insertion, update buffer state
Generation failed
Notify failure, provide error message
Return error to user
Display error message
opt
Maintain editor state
Update cursor, undo history, and editing modes
Continue editing or re-prompt without leaving Vim

4. Architectural Changes Required

This section of our report describes the modifications to the architecture that are required to introduce a Void AI-supported CLI and a Void Model Server that enables cross-editor AI usage (Vim/Neovim/Emacs/terminal workflows). These changes affect and change both the high-level conceptual architecture from A1 and the lower-level concrete architecture from A2.

4.1 High-Level Architectural Modifications

The new CLI feature impacts the following subsystems within the conceptual architecture that we previously identified in our A1 conceptual architecture report. The new subsystem introduced in our A2 concrete architecture, called "Build/Infrastructure," is also going to be affected, this is since the subsystem itself was used to represent a foundational library for all the shared helper functions, common data structures, development tooling, application assets, test suite, and runtime server code that are used by other subsystems. The addition of our new CLI feature it means that there would have to be an updated test suite, more application assets, new development tooling for the CLI, and possibly even more helper functions to support new functionality.

With the current state of Void, we previously discussed the 2 electron processes that make up our conceptual architecture, the main process and browser process; however, we propose a new third process that exists outside of both. This is required due to the requirement of external usage of AI tooling (within the CLI).

We needed a way to expose AI Services outside of the Electron framework; however, the 2 current existing processes are designed for VSCode, not an AI-supported CLI or plugin that requires API access to the Void system. In the case of Void and VSCode, the main process is responsible for window lifecycle, OS integration, and extension activation, and originally cannot do much without the browser process, but with a new 3rd dedicated process, it can support a much more lightweight setup of the new AI-powered CLI. The browser process hosts the workbench, editor core, and UI components, but cannot initialize AI services without loading the full VS Code environment. Because the browser process was reliant on the main proces,s and they usually needed to be launched together, the current architecture provided no way to expose AI tooling externally without launching the entire IDE. An example of this could be a developer that aims to launch solely the CLI or use VIM with a plugin, but the main process's lifecycle management is only designed for the management of the IDE structure (originally), meaning managing a CLI without having all of Void open and running would not be possible, leading to our solution of new independent CLI/plugin support process called the CLI lifecycle process, which is much more lightweight than the entire IDE, hence why running the browser process now becomes optional when using the AI powered CLI and the main process and CLI lifecycle process are now able to work together (low level changes discussed below)

4.2 New Subsystems

Void CLI

The Void CLI subsystem is the user-facing command line interface layer that exists to enable developers to interact with Void IDE's AI capabilities without opening the actual Void IDE environment and using Void as a plugin for other existing editors, such as VIM. This subsystem provides input parsing, user workflows, and Void Model Serve,r but it does not handle/perform the AI tasks itself.

CLI AI Services

This new module is required for the AI services to function within the CLI without using the original AI Services from our old architecture. We cannot reuse the exact AI Services subsystem within our new feature since our problem is with the entire Void environment needing to be active for us to have access to AI Services; however, the entire point of the CLI is a lightweight editing and AI tooling without having to launch the entire environment. We proposed CLI AI services acting as a pure AI logic "Library" for existing AI access within our CLI (only the required tools) without having to load the browser process, essentially acting as an extension of AI Services.

Void Model Server

The Void Model Server is a new subsystem that is designed to introduce AI operation support in the CLI environment (such as VIM) outside of the browser process and main process. This subsystem acts as the functionality for these AI features that are supplied by the CLI AI Services, it acts as the core of the CLI environment when the Void IDE is not running and handles all AI related tasks, including preparing any requests, invoking the LLM providers through Provider Integration, applying edits through Database/Files, and returning results to the Void CLI. It reuses AI logic extracted from the existing AI Services subsystem, but it does not depend on the Browser Process or the IDE's UI subsystem. This differs from the CLI AI Services module because this subsystem acts as the core of the pure logic functions/features supplied by the CLI AI Services module, similar to the browser and main processes, since on its own, the new AI services module cannot do anything to manage the actual CLI environment.

4.3 Changes to Major Subsystems

Platform Services

With the addition of our new AI-powered CLI, the platform services is now responsible for also routing requests to the new process, the CLI lifecycle process, meaning we need new IPC functions between our new subsystems, such as Void Model Server and the core functionality of platform services, such as model provider settings, provider api keys, OS level services and more.

Low Level Changes: Update platform services to store configuration for the CLI and Void Model Server, such as sharing the same environment configuration as the IDE settings. Add functionality to detect if the CLI lifecycle process is currently active.

Editor Core

Editor Core mostly remains unchanged, the new AI powered CLI feature avoids adding new responsibilities to the Editor Core to avoid changing IDE behaviour that doesn't pertain to CLI functionality.

Low Level Changes: No changes required

Workbench Framework

The Workbench Framework is still only responsible only for IDE behaviour and UI editing, none of the workbench framework's responsibilities transfer to our new CLI behavior.

Low Level Changes: No changes required

UI Components

UI Components continue to maintain it's role in AI requests from IDE interactions but do not participate in any new CLI workflows.

Low Level Changes: No changes required

AI Services

AI Services remains a Browser Process subsystem but is extended to support the new AI based CLI feature, the IDE implementation stays unchanged, but the core logic is extended to the new CLI AI Services module to provide support while the browser process is offline, allowing reuse of some basic AI functionality.

Low Level Changes: Extending core AI logic features into new CLI AI Services module

Provider Integration

Provider integration is now responsible for servicing both the AI Services subsystem (IDE Features) and the Void Model Server module,

Low Level Changes: Add a secondary endpoint for Void Model Server requests and prepare for LLM responses for Void Model Server for CLI support

Extension System

The extension system is largely unchanged as the extension system does not pertain to the CLI support.

Low Level Changes: No changes required

Database / File

The subsystem's usage is changed to support access from the Void Model Server, which operates without Editor Core and only works with the Void CLI

Low Level Changes: Add support for all file operations within Void CLI, ensure no race conditions are possible by using semaphores to lock files to stop editing on both CLI and Void IDE at the same time.

4.4 Updated Conceptual Architecture Diagram

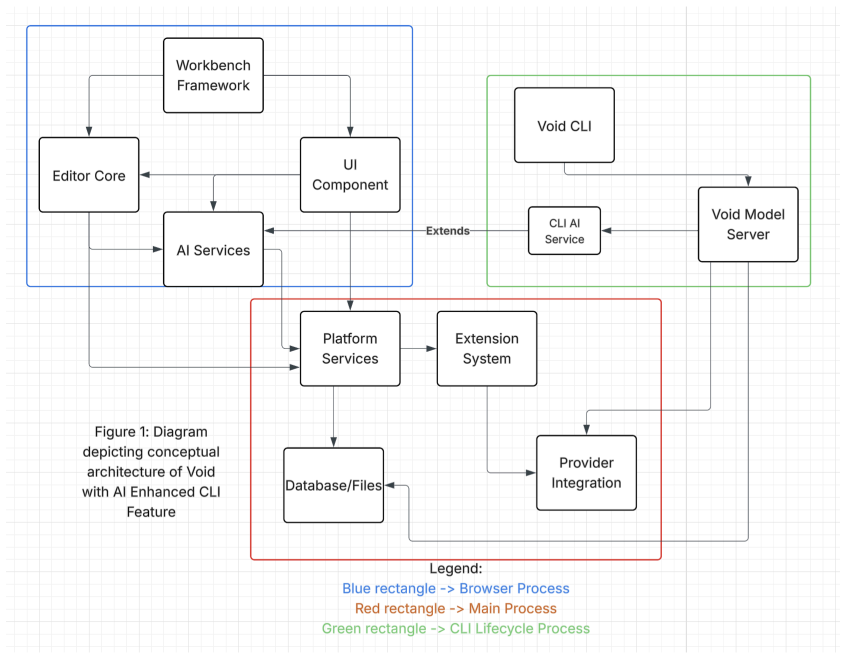


Figure 1: Diagram depicting conceptual architecture of Void with AI Enhanced CLI Feature

Our conceptual architecture is quite different from our first copy. This is due to the addition of our new process, meaning we needed to improvise a way to implement file I/O, provider integration, and some core AI logic, which led to us researching the structure and idea behind the electron framework, we learned that the main process is able to run without the browser process since the browser process largely provides user interface, meaning if our CLI or plugin is running, the main process must be running too since it holds core responsibilities like provider integration, database/files, and platform services, this meant our Void Model Server subsystem could rely on database/files for I/O and provider integration without issue, unlike AI services since we needed our plugin/CLI to be able to run without the browser process, but with the main process.

---

5. Impact Analysis

5.1 Maintainability

Introducing the Vim plugin builds a clear separation between the plugin's editor-specific logic and the core Void engine. All procedures that the Vim is in charge of such as, handling macros, reading and writing buffer content, and communicating with the Void process, are isolated within the plugin module. Meanwhile, Void's core model invocation and diff-generation logic remain untouched. This creates a modular boundary that improves the maintainability of the system because updates to the plugin do not require for any of the core systems to be edited and vice versa. On the other hand, maintainability is affected negatively since we now need to support two editor environments. So any modification to Void's request or response formats now risks breaking the plugin, increasing integration upkeep. Overall, we can see that the new Vim plugin would increase maintainability since we strengthen modularity, but one issue would be that it demands more cross-environment coordination.

5.2 Evolvability

The new Vim plugin would enhance evolvability since it would make it easier to build new plugins for editors like emacs and NeoVim. This is possible due to how the plugin exposes Void functionality through text based commands, macros, and simple buffer edits, providing a blueprint for future extensions. Meanwhile, it does add some new challenges such as, new Void features that rely on complex GUI elements may not translate well into Vim's minimalist terminal. Because of this, when making future features, developers would have to take into consideration compatibility across graphical and non-graphical editors. Overall, we can see that the Vim plugin allows for easier implementation of other cli plugins but does make it a bit more difficult when wanting to edit or update UI-heavy features.

5.3 Testability

The new Vim plugin improves testability at the component level because the communication layer between Vim and the Void system can be mocked, allowing us to use unit testing to verify correct behavior. However, full system testing does become more complicated due to Vim's model interface and lack of native testing framework. This makes end-to-end testing more fragile, making it necessary for scripted editor sessions or specialized harnesses to produce realistic user actions. It can also cause regression testing to become more complex to implement, since the differences between Vim versions, keybindings, and terminal environments could impact behavior. Overall, we can see that testing becomes easier when the system is isolated but is more difficult to validate on the entire system.

5.4 Performance

The new feature adds extra inter-process communication and context handling, which slightly increases latency across the system. It also consumes additional CPU and memory resources, particularly when processing large inputs or running AI-related tasks. The added IPC overhead can impact performance under high concurrency, though the feature itself remains efficient for typical usage. With all of this being said, the new feature itself would perform faster than the GUI version as it would be using a much simpler system without the need to render a GUI. This change would not be quite large however, as the majority of the latency issues and delays would come from the LLM generating text.

---

6. Impacted Files and Directories

This section lists only the existing files that require modification to support the AI enabled CLI. The Workbench Framework, Editor Core, UI Components, and Database/File subsystems remain unchanged. They are not listed below because none of their concrete files require modification for the CLI feature. The files are grouped using the subsystem structure established in A1 and A2 to maintain a direct link to the conceptual and concrete architectures.

6.1 AI Services Layer

src/vs/workbench/contrib/void/electron-main/llmMessage/sendLLMMessage.ts
Must let LLM requests run without the GUI. The current version expects the renderer process.
src/vs/workbench/contrib/void/browser/toolsService.ts
Needs a headless access path so the CLI can run tools like file read, write, and search without browser components.
src/vs/workbench/contrib/void/browser/editCodeService.ts
Requires support for file edits and diff application without editor widgets or UI elements.

src/vs/workbench/contrib/void/common/sendLLMMessageTypes.ts
Must add new request types so the CLI can pass its own parameters to LLM calls.

6.2 Provider Integration and Model Management Layer
src/vs/workbench/contrib/void/common/voidSettingsService.ts
Must let the CLI access API keys and provider settings without relying on browser storage.

6.3 Platform Services Layer
src/vs/code/node/cli.ts
Must parse new CLI flags for chat, generation, code edits, diff application, and tool execution.
src/vs/code/node/cliProcessMain.ts
Must initialize the AI services in a headless mode instead of launching the full workbench.
src/vs/platform/environment/node/argv.ts
Requires new argument definitions for the added AI related commands.

6.4 CLI and Build System Layer
cli/src/commands/args.rs
Requires new entries in the Rust argument parser for the AI commands.
cli/src/bin/code/main.rs
Must route the new AI commands to Node based handlers.

---

7. Interaction With Existing Features

Bringing AI-enabled CLI functionality to Void affects several existing features in Void. Particularly, those affected are the ones who rely on shared AI Services and file-level operations. The most significant impact is the possibility of concurrent edits when a user interacts with the same file from both the IDE and the CLI. If a CLI-initiated edit and a workbench-initiated edit occur at the same time, this may create a race condition that can lead to overwritten changes. This risk is not unique to the enhancement, but the CLI introduces a second entry point for AI-based modification, which increases the chances of this conflict happening.

The enhancement may also influence chat-based code generation and inline suggestions. Both features depend on shared model providers and the same AI Services subsystem. Running the CLI and the IDE simultaneously could cause contention for provider requests or create an inconsistent state if a user expects both interfaces to share the same context. Although this does not directly cause errors, it may lead to unpredictable model output or duplicated tool calls if both agents act on the same project.

Tool execution is another area that can be indirectly affected. Tools will now be callable from both the CLI and the IDE. If both interfaces attempt to modify or query the same resources, the system may experience short periods of inconsistent state unless guardrails are implemented. The enhancement does not break these features, but it does increase the importance of clear separation between CLI state and workbench state.

Overall, the impacts of the enhancement are manageable and do not interfere with the correctness of existing features. The main considerations relate to shared AI resources and the coordination of file-level operations across the two independent entry points for AI-assisted workflows, being the GUI and the CLI.

---

8. Risks & Mitigation Strategies

8.1 Security Risks
The proposed enhancement does pose some potential security risks. such as, having to give the plugin access to the user's file buffer which could lead to the risk of unintended writes. Another risk would be the API calls to Void. These could expose sensitive code if not correctly secured. Another risk is possibility for race conditions where async responses write to the wrong buffer.

8.2 Performance Risks
The proposed enhancement does pose some potential performance risks such as, possible slow provider/API latency, large files also must be serialized and sent to Void which could cause lag, and blocking calls could freeze Vim temporarily.

8.3 Maintainability Risks
The proposed enhancement does pose some potential maintainability risks such as, having to support both VS code and Vim which could introduce divergent code paths. We could also risk future Void API changes which could break the plugin. Also there can be some edge cases in the Vim modes which can complicate the logic.

8.4 Test Coverage Risks
The proposed enhancement does pose some potential test coverage risks such as, async responses may arrive late or interleave which could make tests inconsistent, streaming outputs can be difficult to fully simulate in tests, and Vim's modal behavior makes automated coverage tricky.

8.5 Mitigation Strategies
There are many different strategies we can use to mitigate some of these risks. We can enforce strict text-only write boundaries, which would ensure that the risk for unintended writes would be mitigated. Another strategy we can use is to use non blocking async calls with clear timeouts which would help performance. For maintainability issues we can define a stable, versioned API contract between the Void system and the plugin. For testability we can create mockable IPC layers so that most plugin behaviors can be properly tested without having to run Vim directly. With these strategies we are able to mitigate some of the risks posed by the newly proposed enhancement.

---

9. Testing Plan (1 page)

9.1 Unit Testing
Mock AI provider, mock editor, DI boundaries.
Test command parsing for correct handling of flags, missing arguments, malformed input, and help messages.
Verify serialization/deserialization of prompt requests sent through the IPC interface or daemon channel.

Check error-handling paths, including invalid model selection, provider unavailability, and unreachable services.
Validate streaming output handlers to ensure partial tokens display correctly without blocking terminal output.

9.2 Integration Tests
Simulate use cases inthe Extension Host sandbox.
Test the sequence: **CLI → Command Parser → IPC/Daemon → LLMMessageService → Provider → Output**, confirming data flows match the designed sequence diagrams.
Verify that edits generated through the CLI correctly propagate to the workspace and appear in the user's file system.
Ensure model selection, context gathering, and settings retrieval behave identically to GUI-based workflows.
Run integration scenarios involving multi-file prompts and confirm that the system handles them consistently across both interfaces.

9.3 UI/UX Tests
Check panel rendering, loads, and sequence flows.
Check that CLI-generated code edits appear correctly in the GUI as diff zones, undoable edits, or updated file content. Ensure the chat panel, conversation history, and file decorators remain visually stable when actions originate from the CLI.
Validate that no unexpected notifications, error pop-ups, or misaligned UI components appear due to new IPC channels. Test synchronization: opening Void after running CLI commands should reflect updated files, generated outputs, or AI responses.
Confirm that UI responsiveness (rendering, input handling) is not negatively affected when heavy prompts are executed via CLI.

9.4 Regression Testing
Ensure existing features (chat, autocomplete) still function.
Re-run scenarios previously tested in A2 to confirm that new CLI pathways did not introduce architectural regressions.
Validate fallback behavior when providers fail so that GUI workflows continue unaffected.
Test workspace operations—file edits, diffs, formatting—to ensure CLI actions do not override or bypass existing UI logic.
Confirm settings and configuration modules behave consistently across releases, with no cross-contamination between CLI and GUI settings.

9.5 Performance Testing
Latency benchmarks, provider round-trip timings.
Measure CLI cold-start latency when invoking the extension host versus warm-start times during repeated prompts.
Benchmark round-trip times for small, medium, and large prompts to evaluate responsiveness under different workloads.
Analyze streaming performance: token arrival rates, chunk ordering, and response flush behavior in the terminal.

Conduct stress tests: execute rapid sequences of CLI commands to detect memory leaks, queue buildup, or IPC congestion.
Compare performance between Alternative 1 (direct Extension Host invocation) and Alternative 2 (standalone daemon) to validate SAAM assumptions.

---

10. Alternatives & SAAM Analysis

10.1 Two Architectural Alternatives

Alternative A - Extend Void's Existing CLI Pipeline

Add the AI functionality directly to Void's current CLI architecture. The existing Rust command parser would be expanded with new AI commands, which forward execution into the Node-based CLI runtime (cliProcessMain.ts). This runtime would load a minimal subset of Void's AI Services and Provider Integration logic. All AI operations would therefore run inside the same internal CLI execution path that already supports extension management, diagnostics, and workspace operations.

High-level idea:

Reuse the CLI that Void already has and extend it so that you can access the AI chat through the CLI without the GUI.

Alternative B - Standalone Node-Based "Void Model Server"

This approach creates a separate lightweight runtime dedicated to AI operations. The CLI becomes a thin client that sends requests (chat, generate, edit, diff, tool calls) to the Model Server via a local API or IPC channel. The Model Server hosts AI Services independently of the workbench and does not rely on the graphical environment. Both the GUI and CLI would call into this shared backend.

High-level idea:

Create a separate AI backend that the CLI communicates with, allowing headless operation and editor integrations without loading any VS Code components.

10.2 Stakeholder Identification

- CLI Users – Want low-latency, reliable access to AI tooling without a GUI for scripts.
- Vim/Neovim/Emacs Users – Want editor plugins that integrate cleanly with Void functionality for better use.
- Void Core Maintainers – Want architecture that is easy to evolve and debug, with similarities to the current architecture
- Extension Developers – Need static/predictable behavior to build on.
- CI/CD Engineers – Require capable software without a GUI for automated pipelines.
- AI Provider Integrators – Want consistent request/response behavior across all clients.
- Security Teams – Need isolation between AI tooling and user environment for privacy purposes.
- QA Engineers – Need static/predictable behavior and testable some sort of interface.
- System Operators – Need simple packaging for deployment.

10.3 NFR Identification per stakeholder

CLI Users - Performance

CLI work relies on fast, low-latency feedback during command loops and scripting, making speed a requirement for those using the CLI.

Vim / Neovim / Emacs Users - Reliability

Macro-driven editors require highly predictable behavior to prevent interrupting workflows.

Void Core Maintainers - Maintainability

Maintainability ensures long-term evolution of both the Model Server and CLI.

Extension Developers - Interoperability (API Stability)

Plugin authors need predictable blueprints to build/maintain editor integrations for future use.

CI/CD Engineers - Portability

Automated pipelines cannot depend on Electron or graphical system components; therefore, the system must be equivalent to the GUI version from the command line without sacrificing quality.

AI Provider Integrators - Accuracy

Proper formatting ensures the correct invocation of LLMs and prevents issues that the GUI already prevents in this new format.

Security Teams - Availability

Availability ensures predictable behavior during long-running automated or shared workflows.

QA / Test Engineers - Manageability

Manageability enables verification, regression testing, and root-cause analysis.

System Operators - Scalability

Multi-user environments or shared systems require concurrency support, as systems reliant on the software need the ability to change and grow without issues.

10.4 SAAM Tables & Comparisons

Performance (CLI Users)

Alternative A offers just average performance gains and yet overheads since it partially initializes VS Code subsystems. This renders the invocation of CLI slower than preferred when scripting workflows. Alternative B is much better since the Model Server is already warm and not dependent on Electron to run, which generates near-immediate answers to repeated requests.

Reliability (Vim/Neovim/Emacs Users)

Alternative A carries with it the assumptions of the editor-centric logic of VS Code, which will behave unpredictably when interacted with via modal editors such as Vim. Alternative B is more reliable as it reveals a dedicated API that is supposed to be used in deterministic interaction, where terminal-based editors can safely interact with the AI tools Void provides.

Maintainability (Void Core Maintainers)

Alternative A involves making the CLI and Void closely coupled, where the internal GUI pathways are more likely to regress due to an alteration of some parts of the workbench or provider integration. Alternative B is much more maintainable since there is a very clear

separation of concerns: the CLI, GUI, and Model Server can inter-communicate with a well-defined boundary, and architectural erosion will not occur over the long term.
Interoperability / API Stability (Extension Developers)
Alternative A exposes the frequent differences in request/response formats of GUI and CLI since they are based on dissimilar processes of execution. Alternative B ensures interoperability, since all of the clients, including GUI, CLI, and multi-editor plugins, share the same Model Server API and have the same contracts.
Portability (CI/CD Engineers)
Alternative A is limited by VS Code dependencies and cannot run in minimal Linux containers or CI environments without GUI support. Alternative B excels in portability. The Model Server can run in fully GUI-free environments, enabling easy deployment in scripts, CI pipelines, and SSH servers.
Accuracy (AI Provider Integrators)
Alternative A traverses logic that was mostly written to support GUI processes, which can ignore context or metadata anticipated by providers. Alternative B is very precise as all provider calls are routed by a central runtime that has consistent request formatting and provider metadata processing.
Availability (Security Teams)
Alternative A relies upon the lifecycle of the processes of VS Code, i.e., any instability in the systems impacts the uptime of the CLI. Alternative B will enhance accessibility by being a dedicated long-lived service that still operates even without the lifecycle of the IDE GUI.
Manageability (QA/Test Engineers)
Alternative A makes testing more difficult since CLI calls cross code paths with latent GUI assumptions, making behavior less repeatable. Alternative B supports manageability. The Model Server offers deterministic endpoints, which are useful in mocking, regression testing, and structured logging.
Scalability (System Operators)
Alternative A scales poorly because AI calls run inside a single CLI process with partial IDE initialization. Alternative B supports multi-request concurrency, enabling higher throughput and deployment in shared environments.

10.5 SAAM Interpretation

SAAM shows clear trade-offs. Alternative A ties AI to the existing CLI, limiting portability and slowing performance in scripts or large files. Alternative B separates AI into a standalone Model Server, improving latency, reliability, and multi-editor support, but adds a separate runtime to maintain. Alternative A really just lacks flexibility and would make the service harder to adapt and evolve. Overall, Alternative B offers better performance, maintainability, and flexibility, seeing as it works more as an extension of the Void service than a core component.

10.6 Selection of the Superior Alternative

Alternative B is clearly superior because it separates AI functionality into a standalone Model Server, avoiding dependencies on the graphical IDE and enabling true headless operation. This design reduces latency for CLI workflows, supports multi-editor integration, improves reliability and scalability, and simplifies testing and maintenance by clearly defining subsystem boundaries. The standalone approach also ensures consistent API behavior across GUI and CLI clients, making it more flexible for future extensions and better aligned with non-functional requirements such as performance, portability, and maintainability.

---

11. Conclusion & Lessons Learned

Our enhancement extends Void's AI capabilities past the GUI by introducing a CLI workflow that is supported by a standalone Model Server. Our proposed design creates a lightweight and headless execution path that allows AI features to run without the renderer or workbench. It also enables flexible use in terminal based editors such as Vim and Neovim while preserving the stability of the existing Void IDE.

Our architectural analysis showed that this separation is really necessary because the current main and browser processes are tightly coupled and cannot expose AI functionality externally without actually launching the full environment. Our SAAM evaluation confirmed that introducing a dedicated Model Server is the preferred option. It offers better maintainability, clearer subsystem boundaries, and long term extensibility compared to embedding AI functionality directly into the existing CLI pipeline.

A key lesson for us that we learned from this project was the importance of understanding where responsibilities belong in a layered architecture. An assumption we mistakenly made early on was that existing AI Services could be reused directly in our new design. Exploring this further highlighted the very tight coupling within the browser process, which led to us creating a separate CLI side module. Revisiting the conceptual diagrams from A1 and the concrete mappings from A2 helped us maintain a clear mental model of the system and also avoid any accidental dependencies as we introduced new components.

Overall, our proposed enhancement would strengthen Void's flexibility while remaining consistent with its established architecture. The project provided valuable experience applying architectural reasoning, evaluating alternatives, and extending a real software system in a structured and maintainable way.

---

12. AI Collaboration Report

Our group used OpenAI GPT 5.1, Cursor, Anthropic Claude Sonnet (both independently and within Cursor), and eraser.io throughout the development of this assignment. GPT 5.1 helped us with the production of drafting/report structure, architectural justification, and also made sure our work was aligned with the rubric. Sonnet was mainly used inside Cursor to help us properly navigate the Void repository, summarize subsystem relationships, and identify the files actually affected by our proposed CLI and Model Server enhancement. Cursor was also valuable for contextual code navigation and multi file reasoning when exploring the architecture. Furthermore, we used eraser.io for the sequence diagrams, and it helped with layout, organization of arrows and dependencies, and the overall visual structure. Compared to A2, AI tools were used at a similar level, although we relied more on AI for conceptual guidance and outlining/validating ideas rather than detailed analytical interpretation of the codebase and its structure. We experimented less with alternative models because A1 and A2 clarified which tools worked best for our workflow and the type of reasoning required.

12.1 Tasks Assigned to the AI Teammate

Task: Drafting and refining report sections
Reason: GPT 5.1 helped maintain clarity and consistency across the abstract, introduction, architectural modifications, and SAAM analysis.
Task: Searching and organizing repository files (Cursor + Sonnet)
Reason: Cursor's integrated environment allowed Sonnet to process the folder hierarchy, identify key subsystems, and determine which files required modification for the CLI enhancement.
Task: Evaluating architectural alternatives
Reason: GPT 5.1 helped compare the two proposed designs during SAAM evaluation and supported the articulation of tradeoffs.
Task: Drafting sequence diagram content (Eraser.io)
Reason: Eraser.io helped structure the diagram before final manual adjustments.

12.2 Interaction Protocol & Prompting Strategy

As in A1 and A2, our work was completed through collaborative prompting in both Cursor and Google Docs. Prompts typically followed three steps:

1. Provide architectural or technical background, such as a description of the CLI workflow or chosen enhancement.
2. Specify the required output, for example a list of affected files, a rationale for a change, or a reformulated section of the report.
3. Iterate with follow up prompts to refine accuracy, correctness, and alignment with course requirements.

Example Prompt:
"Identify which existing Void files must change for a headless AI enabled CLI. Focus on subsystems used in our A1 and A2 reports and summarize each file's role."

12.3 Validation & Quality Control

To ensure correctness and originality, the team followed a consistent review process:

1. Cross checked all AI summaries with the actual Void repository in Cursor.

2. Verified file paths, subsystem placement, and dependencies through manual inspection.
3. Ensured consistency with our A1 conceptual architecture and A2 concrete architecture.
4. Rewrote or corrected AI-generated text whenever it conflicted with repository evidence or architectural diagrams.

12.4 Quantitative Contribution

Category	AI Contribution	Human Contribution
Drafting and refinement	~35%	65%
Repository analysis (Cursor + Sonnet)	~25%	75%
Sequence diagram preparation	~20%	80%
Editing & integration	~10%	90%
Overall Estimated Impact	**~ 35%**	**~65%**

12.5 Reflection on Human/AI Team Dynamics
AI tools helped us organize arguments, identify affected files, and improve the clarity of architectural explanations. Human judgment was essential for us when validating architectural reasoning, interpreting subsystem boundaries, and confirming that conclusions matched the actual implementation of Void. Minor disagreement between us occurred when AI models suggested different architectural alternatives or proposed conflicting ways of implementing the CLI and Model Server. These differences were resolved by directly consulting the Void repository, rechecking subsystem responsibilities in our A1 and A2 reports, and verifying each option against the existing documentation and code. The collaboration improved our workflow and efficiency, but the final architectural decisions and interpretations were driven by human analysis.

13. References

Dwivedi, V. (2018, April 25). *Electron: 4 Things to watch out for before you dive in*. Medium. https://medium.com/@vishaldwivedi13/electron-things-to-watch-out-for-before-you-dive-in-e1c23f77f38f

electron. (2025, February 13). *GitHub - electron/electron: :electron: Build cross-platform desktop apps with JavaScript, HTML, and CSS*. GitHub. https://github.com/electron/electron

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

Hijdra, R., Geffen, H. van , Petrescu, S., & Yarally, T. (2021). *VSCode - From Vision to Architecture - DESOSA*. Desosa.nl. https://2021.desosa.nl/projects/vscode/posts/essay2/

Inter-process Communication (IPC). (2025). Chromium.org. https://www.chromium.org/developers/design-documents/inter-process-communication/

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

microsoft. (2019, November 22). *Home*. GitHub. https://github.com/microsoft/vscode/wiki

Pareles, A., & Pareles, M. (2025). *Overview | voideditor/void | Zread*. Zread. https://zread.ai/voideditor/void/9-architecture-overview

Perkaz, A. (2021, July 14). *Advanced Electron.js architecture*. LogRocket Blog. https://blog.logrocket.com/advanced-electron-js-architecture/

Rascia, T., & Nolan, T. (2021, September 27). *Understanding the Event Loop, Callbacks, Promises, and Async/Await in JavaScript | DigitalOcean*. Www.digitalocean.com. https://www.digitalocean.com/community/tutorials/understanding-the-event-loop-callbacks-promises-and-async-await-in-javascript

voideditor. (2024). *GitHub - voideditor/void*. GitHub. https://github.com/voideditor/void/

Appendix A. Data Dictionary / Glossary

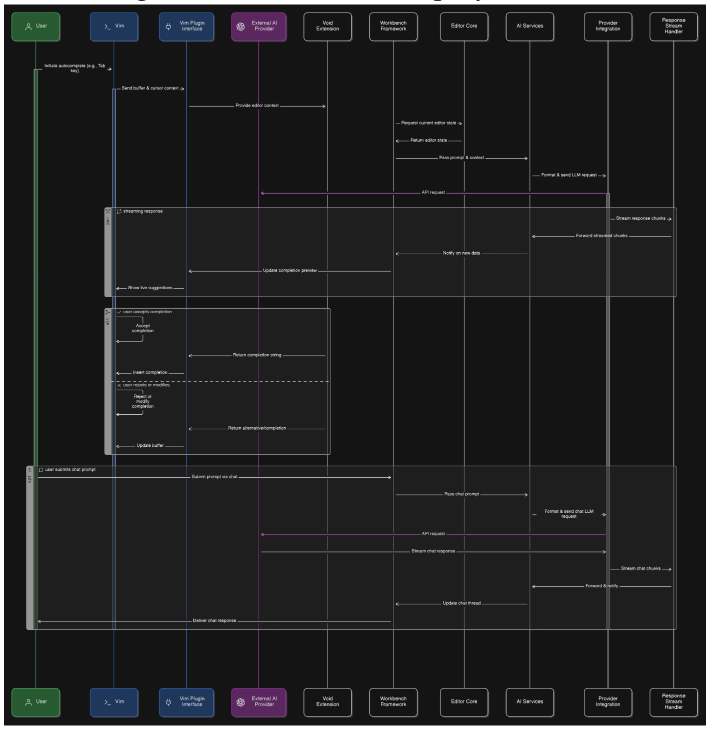
- LLM: Large Language Model
- LSP: Language Server Protocol
- Extension Host: Process running extensions in isolation
- Renderer: UI process handling user input/output
- Main Process: Central Electron process managing windows and IPC
- Void UI: Frontend interface of Void (SidebarChat.tsx)

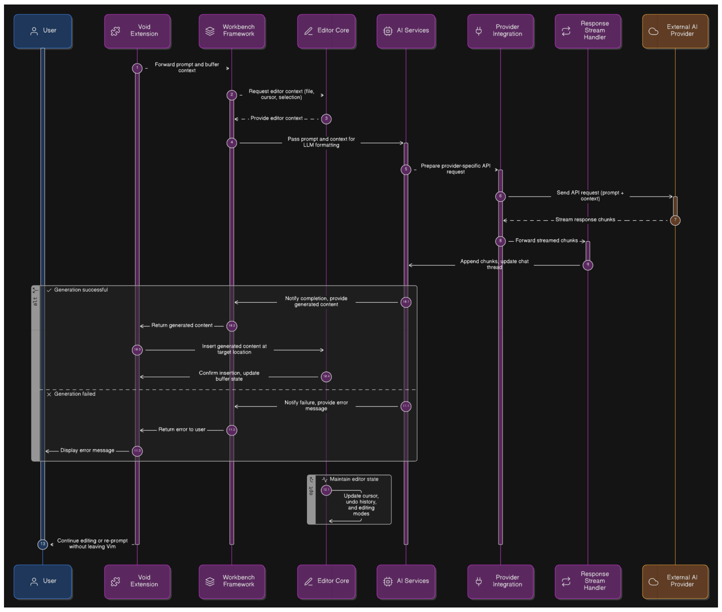
Appendix B. Naming Conventions

- camelCase
- UpperCamelCase
- snake_case
- UPPER_SNAKE_CASE

Appendix C. Abbreviations

- API: Application Programming Interface
- CLI: Command Line Interface
- SAAM: Software Architecture Analysis Method
- UI: User Interface
- AI: Artificial Intelligence
- IDE: Integrated Development Environment
- VS Code: Visual Studio Code
- IPC: Inter-Process Communication
- GUI: Graphical User Interface
- CI/CD: Continuous Integration, Continuous Deployment




User
Void Extension
Workbench Framework
Editor Core
AI Services
Provider Integration
Response Stream Handler
External AI Provider
1 Forward prompt and buffer context
2 Request editor context (file, cursor, selection)
3 Provide editor context
4 Pass prompt and context for LLM formatting
5 Prepare provider-specific API request
6 Send API request (prompt + context)
7 Stream response chunks
8 Forward streamed chunks
9 Append chunks, update chat thread
alt
Generation successful
10.1 Notify completion, provide generated content
10.2 Return generated content
10.3 Insert generated content at target location
10.4 Confirm insertion, update buffer state
Generation failed
11.1 Notify failure, provide error message
11.2 Return error to user
11.3 Display error message
opt
Maintain editor state
12.1 Update cursor, undo history, and editing modes
13 Continue editing or re-prompt without leaving Vim