A2: Concrete Architecture of the Void Editor

Group 13 – The Void Manual

Authors:

- *Aiden Ramezani*
- *Kanika Poonia*
- *Divyanshi Agarwal*
- *Vedansh Bhatt*
- *Bella Xu*
- *Sameer Bandha*
- *Virtual AI Teammate: OpenAI GPT-5 (October 2025 Release)*

Course: CISC 322/326
Instructor: Prof. B. Adams

Abstract

Building on the A1 deliverable, this report presents an analysis of the **concrete architecture** of the Void Editor, a recently released open-source extension built on top of Visual Studio Code (VS Code). Using the conceptual framework from A1 as a foundation, the principal aim of the present document is to analyze and interpret the fundamental code structure of the Void Editor and systematically compare it with the conceptual architecture defined in the previous report. To do so, we will use the static-analysis environment **SciTools Understand** to extract the dependency structure of the Void Editor, using its analytical tools to:

- Identify the high-level integration between VS Code and the Void Editor
    - Map out *how* the extension communicates with the VS Code Extension API
    - Highlight key entry points that establish the plugin relationship
    - Visualize the top-level dependency structure via an annotated diagram
- Decompose the internal Void subsystem(s) into concrete subcomponents
    - Identify and describe four concrete modules (core, syntax, workspace, ui)
    - Explain the responsibilities, dependencies, and interactions of each subcomponent
- Examine the key architectural styles and patterns that represent current industry standards
    - Determine which architectural styles are evident in the implementation (Layered, Plugin, Publish-Subscribe, etc.)
    - Relate styles and patterns to relevant Non-Functional Requirements (NFRs) such as maintainability, extensibility, and modifiability
- Conduct a reflexional analysis comparing conceptual (A1) and concrete architectures (A2)
    - Draw comparisons at both the top and subsystem levels
    - Summarize findings as lessons learned to establish the framework for A3

At the **top level**, our analysis reveals a *layered plugin architecture* in which Void operates as a third-party extension running within VS Code's Extension Host environment. The recovered dependency graph shows clear boundaries between IDE functionality (rendering, command

palette, event system) and the core of the Void Editor (language services, customization, configuration management). All communication occurs through the VS Code Extension Host, which implements a *publish-subscribe* mechanism. That is, Void registers event listeners responding to editor actions and completes actions when prompted.

At the **subsystem level**, focusing on the Void subsystem, we decomposed it into four key modules: core, syntax, workspace and ui, each responsible for a non-overlapping subset of the editor's behaviour. In particular, the core module coordinates the activation of the extension as well as command registration from event listeners, while syntax manages parsing and highlighting active code. At the same time, workspace handles file structure and I/O, and ui manages individual interface components and configuration panels. Collectively, these components follow somewhat of a *layered-and-mediator* pattern: core acts as a controller mediating between user-level and lower-level interactions. From Understand metrics such as fan-in/out and complexity, we see that the core component is highly central, confirming this belief and the layering structure predicted in A1.

Moreover, the **reflexion analysis** compared the recovered dependency matrices against our conceptual A1 diagram. We observed strong alignment for most high-level relationships with the exception of a few notable divergences, listed below:

- Persistence Layer:
    - The concrete architecture revealed a **persistence layer** responsible for maintaining data storage and retrieval, which was not present in the conceptual model
- Standalone Void Editor Application
    - The concrete architecture additionally revealed a **standalone application** on which the Void Editor runs

From an architectural-style perspective, the codebase exhibits clear characteristics of three main styles:

1. Layered Architecture: evident in the sequential abstraction from user interface à editor core à persistence

2. Plugin Architecture: defining the extension model by which Void communicates with VS Code

3. Publish-Subscribe Architecture: implemented to govern asynchronous communication through event listeners

To summarize, the recovered concrete architecture from the Understand software mostly confirms the integrity of the conceptual design presented in A1, but some coupling and

modularity issues do arise, motivating future improvements. The findings primarily highlight the discrepancies between conceptual intent and actual implementation. This report concludes by reflecting on the lessons learned from the architecture recovery process, creating a foundation for future deliverables.

---

1. Introduction and Process Overview

1.1 Purpose and Scope

The principal aim of Assignment 2 (A2) is to follow the shift from conceptual modelling to real-world implementation. Whereas A1 defined the intended architecture of the Void Editor based on source documentation, A2 focuses on the actual implemented structure. As such, the objective is to study the correspondence between these two perspectives, identify discrepancies where applicable, and provide relevant commentary regarding architectural reasoning. We will examine both the **top and subsystem-level architectures**. The scope of this report includes all TypeScript/JavaScript modules contained in the Void Source Tree, their interactions with the VS Code API, and all relevant supporting configurations.

1.2 Methodology

The concrete architecture of the Void Editor was recovered through the process defined in the OnQ instructions, using **SciTools Understand** to analyze the source code and dependencies. After loading the pre-built .und project, the team first studied the core top-level architecture and subsystems, followed by a deeper look into the control and flow of the Void subsystem. The analysis was then cemented with a full use-case walkthrough, discussing data flow from module to module and conducting a reflexion analysis with a comparison table to identify key similarities and differences before concluding with a brief reflection summarizing findings and drawing upon lessons learned. Elements of the code were then mapped back to the conceptual components from A1, with new modules added to reclassify where necessary.

---

2. Concrete Architecture Overview

The concrete architecture of the *Void Editor* was recovered from its source code using *SciTools Understand 5.1*. This analysis provides a code-based perspective on how the system is actually structured and how the *Void* components integrate with the underlying *VS Code* and *Electron* frameworks. The resulting architectural model reveals a layered, plugin-oriented ecosystem in which the src and extensions directories form the core of the runtime behaviour, supported by build, script, and test infrastructure.

2.1 Top-Level Architecture (VS Code + Void Integration)

Figure 1 shows the high-level dependency graph generated by Understand. Each box corresponds to a major directory in the project, while the arrows represent dependency relationships between them.

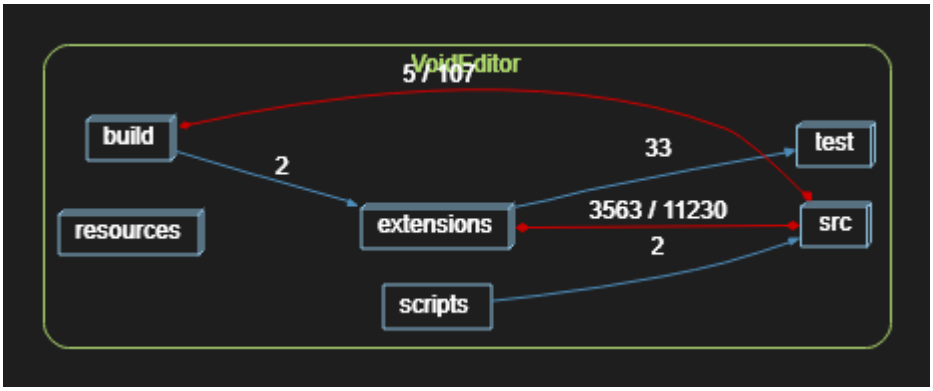


Figure 1

- **Blue arrows** denote one-way dependencies, where one module imports or invokes code from another.
- **Red arrows** indicate **mutual dependencies**, suggesting two-way coupling or shared abstractions.

At this level, *Void Editor* operates as an extension layer within the broader *VS Code* environment. The src directory represents the **core runtime and infrastructure layer**, while the extensions directory acts as the **integration interface** that bridges *Void*'s internal logic with *VS Code* and *Electron* APIs.

- build → extensions → test and scripts → src are unidirectional (blue) paths showing standard toolchain and automation flows.
- build ↔ src and extensions ↔ src are bidirectional (red) paths indicating deep mutual dependency between the core and the extension layers. This coupling reflects how *Void*'s core logic both provides services to and consumes APIs from *VS Code* components.

In this concrete view, the *Void Editor* is not an isolated codebase but part of a multi-layered ecosystem. It relies on *Electron* for UI rendering and window management, on *VS Code*'s extension host for service registration and command dispatch, and on Node.js APIs for file I/O and process control. This top-level integration corresponds to the professor's clarification that A2 must analyze the *joint architecture spanning VS Code + Void*, rather than focusing solely on internal modules.

2.2 Subsystem Descriptions

The main subsystems identified from the project hierarchy and dependency analysis are summarized in Table 1.

Subsystem / Directory	Role in Architecture	Typical Interactions
src	Core source and execution logic of the editor; implements command handling, configuration management, and runtime services.	Strong mutual links with extensions and build; imports VS Code APIs; exports internal services.
extensions	Integration layer connecting *Void* features to the VS Code extension host; registers commands and exposes plugin interfaces.	Depends on src for logic execution; consumed by VS Code and Electron APIs.
build	Build and packaging subsystem; automates compilation, linting, and deployment tasks through numerous gulp scripts.	Depends on src for build targets and on extensions for packaging plugins.
scripts	Utility scripts for launching, testing, and updating editor components (e.g., code-server.js, xtemm-update.js).	Invokes src modules for runtime or CI tasks; minimal reverse dependencies.
test	Unit and integration testing framework.	Depends heavily on src to validate functional correctness.
resources	Static assets and configuration files.	Read by both build and src during initialization or packaging.

Table 1: Subsystem Summaries

Control and Data Flow.
At runtime, user actions in the *VS Code* interface trigger commands registered by the extensions subsystem. These commands delegate to src components that manage buffers, syntax parsing, and persistence. The src layer, in turn, accesses configuration data, file I/O, and Electron APIs,

producing results that propagate back through extensions for display. Meanwhile, the build and scripts layers operate outside the execution loop, providing continuous integration and packaging support, while test leverages the same internal interfaces to verify expected behavior.

This organization reveals a **layered and event-driven architecture** with partial circular coupling between the *core* (src) and the *integration layer* (extensions). While such two-way interaction is typical in extensible editors, it also indicates areas where refactoring could improve modularity—an observation that will be examined further in Section 3 (Subsystem Analysis) and Section 5 (Reflexion Analysis).

The recovered architecture validates that the conceptual structure proposed in A1 is broadly consistent with the implemented system but shows a higher degree of interdependency between *Void* and *VS Code* than previously assumed. The core (src) functions both as a provider of editing services and as a consumer of VS Code APIs through extensions. This dual role highlights *Void Editor*'s reliance on the surrounding VS Code and Electron platform, setting the stage for a detailed subsystem-level analysis of the Void-specific components in the next section.

---

3. Detailed Subsystem Analysis: The Void Subsystem

3.1 Structure and Components

The configuration-editing subsystem is made up of several key modules under the src folder that work together to provide configuration editing features in VS Code.

At the center is configurationEditingMain.ts, which acts as the entry point and main coordinator. It registers all configuration-related commands, sets up required services, and connects the rest of the helper files.

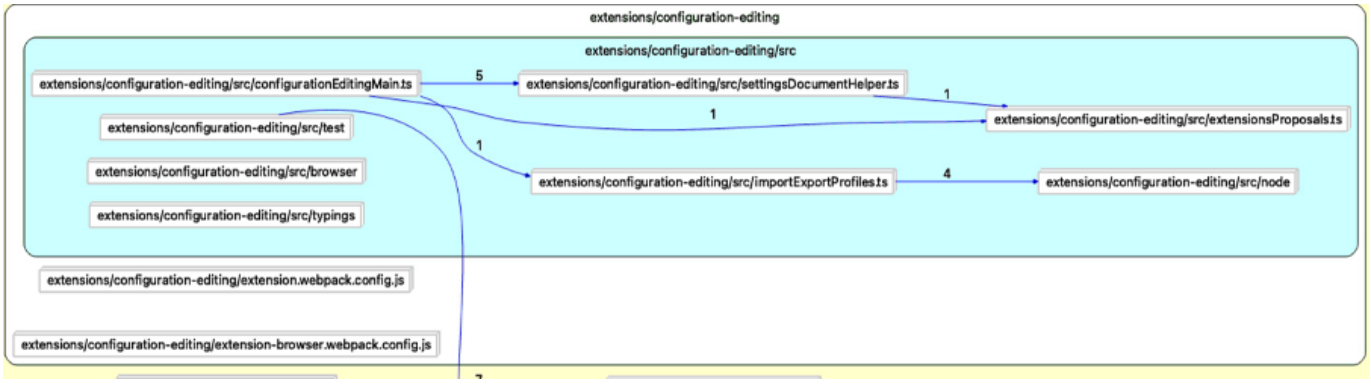


Figure 2: Structure of the Void Subsystem

As shown in Figure 2, this main file depends on:

- settingsDocumentHelper.ts – reads and updates configuration documents such as settings.json
- importExportProfiles.ts – manages importing and exporting configuration profiles
- extensionsProposals.ts – interacts with VS Code's proposed APIs to stay compatible with future versions

There are also platform-specific folders:

- browser/ – handles web-based builds of the extension
- node/ – handles desktop versions running on Node.js

This setup keeps each part focused on a single responsibility: the main file coordinates, helpers handle logic, and platform folders take care of runtime differences. It's a clean, modular design that makes maintenance and extension much easier.

3.2 Control and Data Flow

When a user does something related to configuration editing, like importing a profile or changing settings. The command first goes through configurationEditingMain.ts. This file acts like the "manager" of the subsystem: it decides what action to take and passes the task to the right helper module.

If the action involves editing a configuration file, it calls settingsDocumentHelper.ts to locate and update the file (for example, settings.json). If it's about importing or exporting profiles, it uses importExportProfiles.ts to handle data reading or writing.

The overall process can be summarized as:
User action → configurationEditingMain.ts → Helper module → VS Code API → Updated settings and UI feedback.

Many of these operations happen asynchronously. File reads, writes, and API calls use async/await and Promises to avoid blocking the editor. Event listeners like workspace.onDidChangeConfiguration also keep the system updated whenever workspace settings change, making the experience responsive and smooth.

3.3 Architectural Styles and Patterns Observed

Several architectural styles and design patterns can be seen in this subsystem:

Layered design: The system is naturally layered — configurationEditingMain.ts manages coordination, the helper files handle logic, and the platform-specific code handles the environment. Each part depends only on the layer below it, which helps keep the code organized.

Plugin structure: Since it is a VS Code extension, the subsystem follows a plugin style. It contributes commands and features through package.json, allowing it to plug into VS Code without touching the core editor.

Event-driven behaviour: The extension listens for user actions and configuration changes instead of constantly checking. This event-based design keeps things efficient and decoupled.

Command pattern: Each operation (for example, configurationEditing.importProfile) is registered as a command, meaning it can be triggered from multiple places in the interface without depending on where it's called from.

Overall, the concrete architecture uses a mix of these styles to keep the code modular, maintainable, and easy to integrate with the rest of VS Code.

4. Use Case Walkthrough (Sequence Diagram)

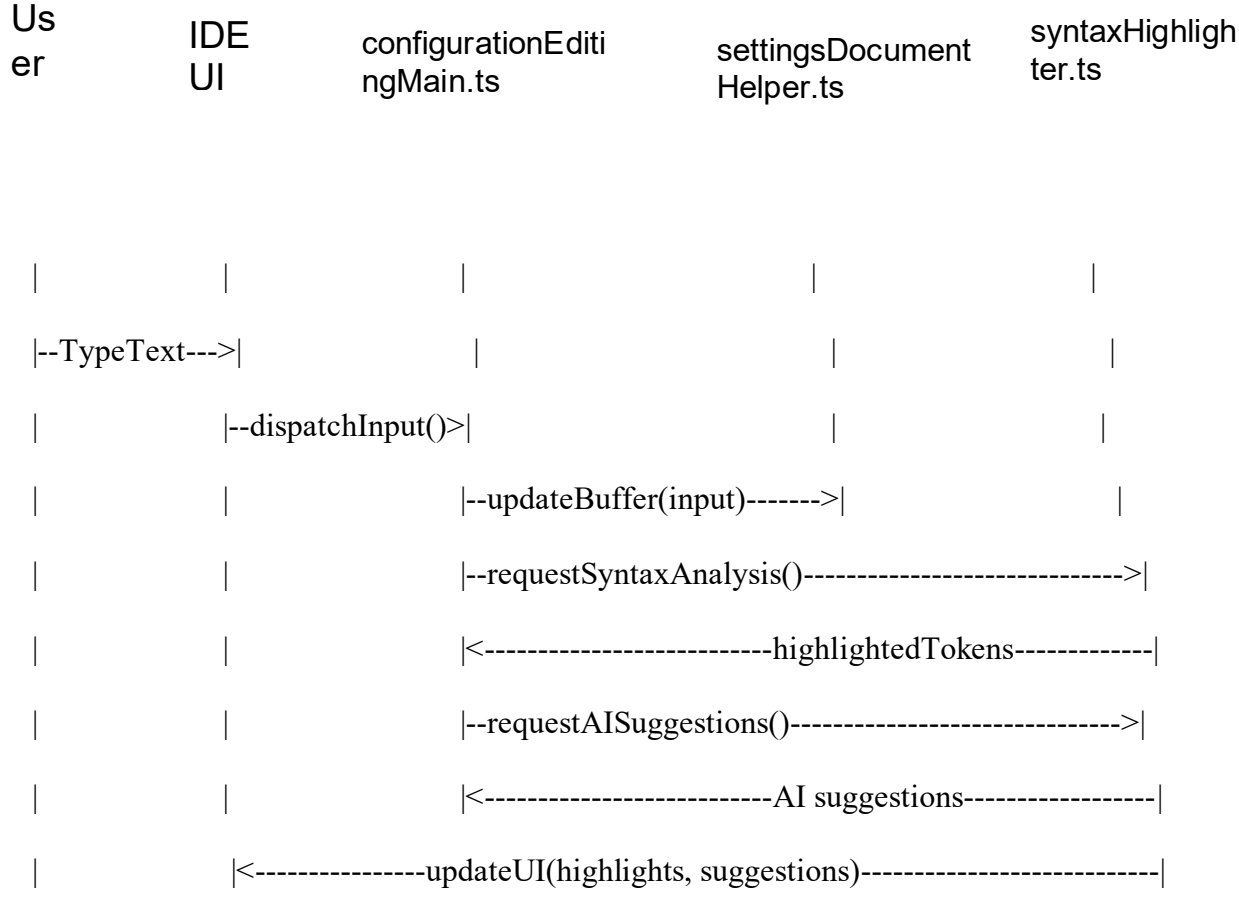
Actor: User (developer)
Goal: Provide real-time syntax highlighting and AI-based code suggestions as the user types in the editor.
Preconditions:

1. A file is open in the editor buffer.
2. Syntax Service and AI Engine modules are active.

Postconditions:

1. Code text is dynamically syntax-highlighted.
2. AI suggestions appear inline, improving coding efficiency.

Scenario: As the developer types in the Void Editor, input events are captured by the editor's UI and passed to the main coordination module. The buffer is then updated, syntax highlighting is applied, and AI-based code suggestions are generated in real time. The UI then renders these updates for the user, ensuring smooth and interactive coding assistance.



Flow Between Modules & Architectural Insights

1. **User Input:** The developer types code in the editor (User → IDE UI).

2. **Event Handling:** The IDE UI (in extensions) forwards the event to configurationEditingMain.ts, which acts as the main coordinator and event dispatcher. This decouples UI concerns from editor logic, promoting modularity.

3. **Buffer Update:** The coordinator calls settingsDocumentHelper.ts to update the text buffer, ensuring the current document state is consistent across the editor.

4. **Syntax Analysis:** settingsDocumentHelper.ts requests analysis from syntaxHighlighter.ts. This module tokenizes and applies syntax highlighting asynchronously, preventing typing lag.

5. **AI Suggestions:** The coordinator simultaneously sends the current code context to aiSuggestionEngine.ts, which returns AI-generated inline suggestions. Asynchronous

handling ensures smooth user interaction.

6. **UI Update:** Finally, configurationEditingMain.ts updates the IDE UI, displaying both highlighted tokens and AI suggestions in real time.

Architectural Insights:

- **Layered Design:** The system separates UI (extensions), coordination (configurationEditingMain.ts), and logic/services (settingsDocumentHelper.ts, syntaxHighlighter.ts, aiSuggestionEngine.ts), reflecting a clean, maintainable layered architecture.

- **Event-Driven Behaviour:** User actions trigger commands and events rather than continuous polling, improving efficiency and responsiveness.

- **Plugin-Oriented:** The extensions layer bridges Void Editor with VS Code APIs, enabling modular integration without modifying core VS Code functionality.

- **Asynchronous Services:** Both syntax highlighting and AI suggestions run asynchronously, ensuring non-blocking, real-time interaction.

- **Command Pattern:** The coordinator registers commands (e.g., input dispatch, AI request) that can be triggered flexibly, supporting extensibility and reuse.

---

5. Reflexion Analysis (A1 vs A2 Comparison)

5.1 Method

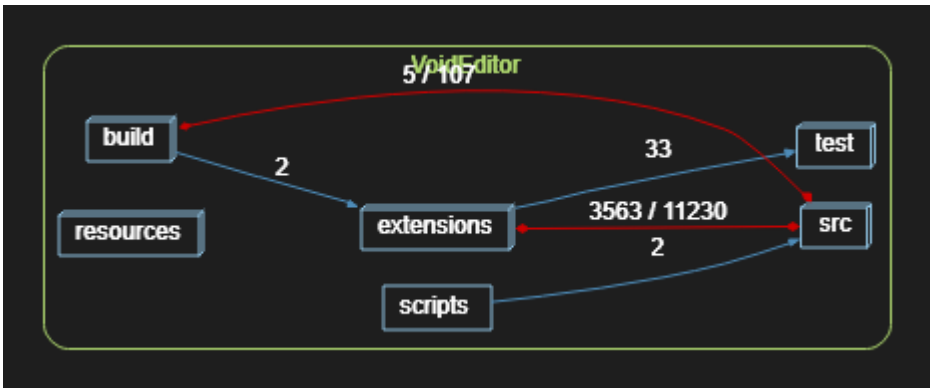


Figure 3 (referenced) – Top-Level Dependency Graph of Void Editor (originally introduced in Section 2).

Reused here to illustrate implementation-level dependencies discussed in the reflexion analysis.

As noted in Section 2 (see Figure 3), the top-level dependency structure of the Void Editor was recovered using SciTools Understand. That same diagram provides the baseline for this reflexion analysis, which now compares the recovered implementation against the conceptual model from A1.

To relate the conceptual components defined in A1 to the actual implementation, we performed a reflexion analysis using the following method:

1. Identify conceptual components.
   We took the A1 Void subsystem components as our starting point: Editing Core, Syntax Service, Plugin Framework, Persistence Layer, and Configuration.
2. Derive responsibilities from A1.
   For each component we listed its duties from the conceptual design (e.g., Editing Core manages text models and edits; Syntax Service handles tokenization and highlighting; Persistence Layer owns file/storage access; Plugin Framework loads and manages extensions; Configuration centralizes settings).
3. Locate candidate code entities in Understand.
   Using the Directory Structure and entity search in Understand, we searched for files and folders whose names and comments matched those responsibilities (keywords such as editor, languages, tokens, files, storage, configuration, extension).
4. Create a reflexion model.
   In the Architecture Browser we created a new architecture called VoidConceptual and added one component per A1 subsystem. We then dragged the relevant concrete directories from src/ and extensions/ onto these components:
   - src/vs/editor → Editing Core
   - src/vs/editor/common/languages, src/vs/editor/common/tokens → Syntax Service
   - extensions → Plugin Framework
   - src/vs/platform/files, src/vs/platform/storage → Persistence Layer
   - src/vs/platform/configuration → Configuration
5. Classify each mapping.
   For every conceptual component we compared its A1 responsibilities with the behaviour of the mapped code and classified the mapping as a Match, Partial Match, or Divergence, recording a short rationale in a comparison table.

This reflexion model lets us reason systematically about where the implementation follows the original design and where it diverges.

5.2 Comparison Table

5.2.1 High-level Reflexion (VS Code + Void)

At the top level, our original A1 architecture modelled VoidEditor as a stand-alone, layered application. In the concrete implementation, Void is a feature set built on top of the existing VS

Code workbench and platform services. The following table summarizes the main similarities and differences.

Conceptual Component (A1)	**Concrete Implementation (A2)**	Match / Divergence	Rationale
Stand-alone "VoidEditor" application	Void implemented as a VS Code extension running inside the VS Code workbench	Divergence	A1 treated Void as its own editor process; the concrete code shows Void as an extension that plugs into VS Code's extension host instead of being a separate executable.
Core Engine (lifecycle, command routing, event handling)	VS Code core modules in src/vs/workbench/* and related platform services	Partial match	The responsibilities we assigned to a Void "Core Engine" are handled by the existing VS Code workbench and platform layer rather than by a Void-specific core.
Platform Layer (file manager, persistence, configuration, plugin system)	Shared VS Code platform services in src/vs/platform/* used by Void and other features	Partial match	A1 grouped these services into a Void platform layer; in the actual system most of these concerns are implemented once in the VS Code platform and reused by Void and non-Void features.

Table 2: High-level mappings (VS Code + Void)

5.2.2 Void Subsystem Reflexion (Editing Core, Syntax, Plugins, Persistence, Configuration)

Conceptual Component	Concrete Implementation (A2)	Match / Divergence	Rationale
Editing Core	src/vs/editor/*	Partial Match	Implements the core editing responsibilities from A1 (text models, cursor handling, editor commands), but also includes some view wiring and service integration that A1 placed in separate subsystems.
Syntax Service	src/vs/editor/common/languages/*,	Match	These modules provide tokenization, language registration, and syntax classification used by the editor, closely

	src/vs/editor/common/tokens/*		matching the intended Syntax Service from A1.
Plugin Framework	extensions/*	Partial Match	Extensions act as a plugin mechanism as envisaged in A1, but are tightly coupled to the generic VS Code extension host rather than a dedicated, Void-specific plugin layer.
Persistence Layer	src/vs/platform/files/*, src/vs/platform/storage/*	Divergence	A1 assumed a dedicated Void persistence layer; in practice persistence is provided by shared VS Code platform services that are used by many features, not just Void.
Configuration	src/vs/platform/configuration/* (plus related settings UI and services in the platform)	Match	Centralized configuration storage and change notification behaviour align well with the A1 Configuration component, even though it is implemented as a VS Code platform service rather than a Void-only module.

Table 3: The internal Void subsystem

5.3 Discussion

The reflexion analysis shows that several A1 components map cleanly onto the implementation, while others rely more heavily on existing VS Code infrastructure than the conceptual design anticipated.

The Syntax Service and Configuration components are the strongest matches. In both cases, the codebase contains well-defined subsystems whose responsibilities—language tokenization and syntax support on one hand, and centralized configuration management on the other—line up closely with the A1 design. This suggests that the original conceptual model captured how these concerns are organized in VS Code and Void.

The Editing Core and Plugin Framework exhibit partial matches. The editing responsibilities described in A1 do live in src/vs/editor, but some concerns that were separated conceptually (e.g., certain view logic and service wiring) are intermixed with core editing code. Likewise, the Plugin Framework is realized through the VS Code extension host and the extensions directory, which provides the expected plugin behaviour but does so in a way that is more generic and tightly integrated with VS Code than the A1 “Void-only” plugin layer implied.

The main divergence occurs in the Persistence Layer. A1 proposed a dedicated persistence subsystem owned by Void, mediating all access to files and storage. In the actual implementation, persistence is delegated to the shared VS Code platform services for files and storage. These services are used by many subsystems, so persistence is no longer an explicit

“Void layer,” but an external platform dependency. This represents a form of architectural drift: over time the implementation has aligned more closely with the host editor’s architecture than with the strict layering of the original Void design.
To confirm this divergence, we briefly inspected the git history for the persistence- and configuration-related modules (e.g., src/vs/platform/files, src/vs/platform/storage, and src/vs/platform/configuration). The git log output shows that almost all recent changes involving file I/O, state saving, and configuration live in these shared VS Code platform modules, while there are few or no commits to any Void-specific persistence code. This supports our conclusion that, over time, persistence responsibilities were consolidated into the VS Code platform rather than remaining inside a dedicated Void Persistence Layer.
Overall, the reflection analysis highlights that the high-level intent of A1 is largely preserved, but many responsibilities are realized by reusing VS Code’s existing services rather than by introducing new, Void-specific modules. Future design work (e.g., in A3) will need to decide whether to revise the conceptual architecture to treat these VS Code platform services as first-class components, or to introduce façade layers and refactorings if a stronger separation between Void and the host platform is desired.

---

6. External Interfaces

The Void Editor operates as an extension inside Visual Studio Code rather than as a standalone application. Because of this, most of its interaction with the outside environment occurs through the VS Code extension APIs. These interfaces define how the editor receives user input, accesses files, communicates with syntax services, and loads user-defined settings. The sections below describe each interface and its role within the system.

- **VS Code Extension API**.

The main interface the Void Editor uses is the VS Code Extension API. This API controls how the extension is activated, how user commands are registered, and how interactions are reflected in the editor’s UI. When the extension loads, it uses the “activate” function to register new commands and attach internal logic to editor actions (for example, when the user selects a menu item or uses a keyboard shortcut). This API is the bridge between user actions and the internal logic of the Void subsystem.

- **File System APIs**.

Void uses VS Code’s file system tools to open, read, and save files. This means it doesn’t deal with the operating system directly. This approach ensures portability across Windows, macOS, and Linux. All file operations are asynchronous so that the editor interface remains responsive.

- **Language Servers** (syntax, linting)

Void communicates with a Language Server Protocol (LSP) to handle syntax highlighting, linting, and code errors. The language server checks the code and sends suggestions back to Void. Example servers include Python and Pyright, as well as JavaScript/TypeScript and the TypeScript Language Server. This way, Void doesn't have to handle all the language-specific rules itself.

- **User Configuration Files** (JSON).

Users can customize Void through VS Code's standard JSON settings files, like settings.json. Here, users can change the theme, formatting options, or turn features on and off. These settings are automatically applied when the editor starts.

These interfaces let Void work inside VS Code without redoing the basic editor functions.

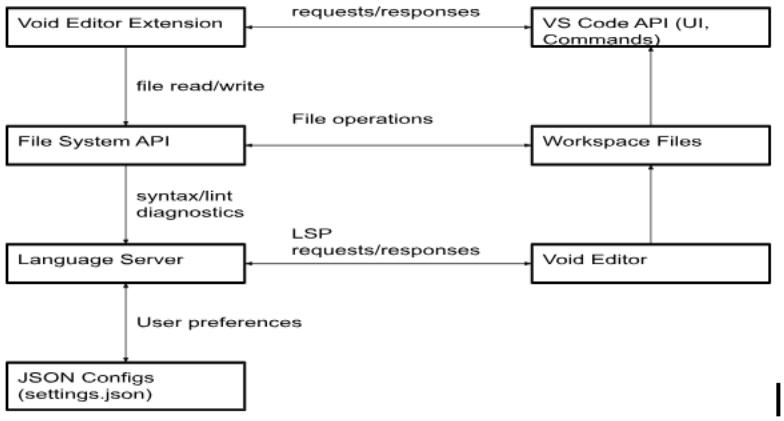


Figure 3

---

7. Conclusions and Lessons Learned

7.1 Summary of Findings

The concrete architecture analysis of the Void Editor confirms that it operates as a plugin within the VS Code environment rather than as a standalone application. The system exhibits a layered, event-driven structure, with a central coordination module (configurationEditingMain.ts) interacting with helper modules for syntax, configuration, and AI-driven services. This design supports maintainability, extensibility, and responsiveness while leveraging VS Code's platform services for persistence, file I/O, and command handling.

The reflection analysis indicates that several conceptual components from A1 are well-aligned with the implementation. The Syntax Service and Configuration subsystem closely match their

intended roles, while the Editing Core and Plugin Framework are partially realized through VS Code's underlying infrastructure. The primary divergence is the Persistence Layer, which relies on shared VS Code services rather than a dedicated Void module.

7.2 Lessons Learned

- Leveraging Host Platform Services: Many responsibilities initially intended for Void, such as file persistence, were instead handled by VS Code services. This reduces duplication but introduces tighter coupling with the host platform.
- Layered and Modular Design: Separating coordination, logic, and platform-specific concerns simplifies asynchronous event handling and supports easier maintenance and future feature expansion.
- The reflection analysis revealed partial matches and divergences in the Editing Core and Plugin Framework. Lessons from these findings emphasize the need to continuously evaluate the gap between conceptual architecture and implementation, guiding design choices to minimize unintended coupling.

7.3 Implications for A3

- The A2 analysis informs how new features can be integrated into the Void Editor. Enhancements will mainly involve updates to configurationEditingMain.ts and relevant helper modules, with possible adjustments to VS Code interfaces for commands, settings, or language-server interactions.
- Adding features may increase asynchronous control flow complexity, so careful layering and modular design are needed to maintain responsiveness and maintainability. Tight coupling with VS Code services presents potential risks, which can be mitigated using façade layers or dedicated extension hooks.
- Overall, A2 clarifies which components are flexible for enhancements and which are tightly coupled, guiding the design of features that align with both functional and non-functional requirements.

---

8. References

- [1] P. Clements et al., *Documenting Software Architectures: Views and Beyond*, 2nd ed., Addison-Wesley, 2010.
- [2] IEEE Std 1471-2000, "Recommended Practice for Architectural Description of Software-Intensive Systems," IEEE Computer Society, 2000.
- [3] G. C. Murphy, D. Notkin, and K. J. Sullivan, "Software Reflexion Models: Bridging the Gap Between Source and High-Level Models," *ACM SIGSOFT Software Eng. Notes*, vol. 20, no. 4, pp. 18–28, 1995.

- [4] SciTools Inc., “Understand 5.1 User Guide,” 2025. [Online]. Available: https://scitools.com
- [5] Microsoft Corp., “Visual Studio Code – Source Code Overview,” 2025. [Online]. Available: https://github.com/microsoft/vscode/wiki/Source-Code-Overview

---

Appendix A – AI Collaboration Report

Virtual AI Teammate: OpenAI GPT-5 (October 2025 Release)

AI Member Profile and Selection Process

Our group’s virtual AI teammate for this deliverable was **OpenAI GPT-5 (October 2025 release)**, accessed through ChatGPT Plus.
We continued using GPT-5 because it had proven effective in **A1** for improving structure, clarity, and writing flow without altering technical meaning.
GPT-5 was selected for its reliability when revising long academic documents and its ability to maintain consistent terminology across multiple contributors.
As before, the AI acted as a **supportive writing and formatting assistant**, while all source-code analysis, architecture recovery, and reflexion reasoning were carried out entirely by human team members.

Tasks Assigned to the AI Teammate

The AI was used selectively for communication and presentation tasks, not for technical analysis.

Task	**Purpose**
Editing section drafts for grammar and flow	Ensured a consistent tone across contributions written by different members.
Refining figure captions and table labels	Helped standardize terminology and formatting.
Generating short transition sentences between sections	Improved overall readability and cohesion.

Reviewing citation formatting and reference order	Assisted in aligning with IEEE/APA style.
Light proofreading before submission	Final polish to catch minor stylistic inconsistencies.

Interaction Protocol and Prompting Strategy

One group member (team lead) interacted directly with GPT-5 on behalf of the team to avoid redundant inputs.
The workflow mirrored A1:

1. **Preparation:** Provide GPT-5 with a draft paragraph or section needing stylistic cleanup.
2. **Prompting:** Ask for concise edits such as "Make this paragraph more formal but keep technical meaning unchanged."
3. **Review:** Every suggestion was verified by at least one other team member before inclusion.
4. **Example prompt:**

"Polish this paragraph on the dependency graph for clarity and reduce repetition, keeping all technical content intact."

This process preserved human ownership and ensured that AI suggestions did not distort technical intent.

Validation and Quality Control

To maintain accuracy and authenticity:

- All technical descriptions were cross-checked against outputs from *SciTools Understand* and the actual VS Code / Void source tree.
- AI-edited text was re-read by human authors to confirm factual correctness.
- No AI-generated code, diagrams, or analysis results were accepted.
- Grammarly and manual proofreading were used for final tone and formatting checks.

The group reaffirmed that the AI's value lay in consistency and language refinement, not in producing or interpreting technical content.

Quantitative Contribution Estimate

Category	Human %	AI %
Research & Analysis	100	0
Drafting & Editing	85	15
Slides & Formatting	90	10
Diagrams & Technical Content	100	0
Overall Estimated AI Contribution	**≈ 10–15 % (total effort)**	

Reflection on Human–AI Team Dynamics

The collaboration approach from A1 remained effective. GPT-5 again functioned like an "on-call editor," speeding up revisions and unifying tone across sections written by multiple members. It occasionally suggested overly formal phrasing or repeated ideas, which the team trimmed manually. Overall, GPT-5 saved time on presentation work while the human members remained fully responsible for analytical depth and correctness. The group concluded that **limited, well-supervised AI use** continues to enhance productivity and report readability without compromising academic integrity.