A1: Conceptual 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 (September 2025 Release)

Course: CISC 322/326

Instructor: Prof. B. Adams

1. Introduction

This report presents the conceptual architecture of the Void Editor, a newly released software tool designed to speed up development by connecting developers with the latest AI systems directly in a VS Code-like environment. The goal of this document is not to describe the implementation details of the system, but rather to underscore its high-level structure and major responsibilities. A focus on conceptual design over technical structure allows the system to be understood in terms of its usage rather than becoming bound by the specifics of current versions.

The Void Editor has been built with a set of interdependent subsystems, each of which works together to provide a high-quality development environment for games and software systems alike. Its architecture places particular emphasis on modularity, ensuring that teams can extend features on new branches without worrying about destabilizing the existing product. Take, for example, the development of a game. The Void Editor stores game-world information (such as game objects, lighting, and other assets) in data formats independent from the engine logic. As such, designers may modify game content without directly altering code. For instance, changes to a background scene can be saved as metadata that the engine loads and renders, eliminating the need to recompile software. This separation of data from logic reduces the likelihood of introducing errors during development.

Similarly, the editor's graphic interface, responsible mainly for buttons and panels, is considered a separate entity from the core engine functionality. Consequently, updating scenes, saving/loading data, rendering graphics, and applying physics are entirely separate from any user interaction. The GUI layer and engine layer communicate through well-defined event handlers or API calls, but they are not fundamentally linked. Consequently, improvements to the GUI, be it small UI tweaks or entirely new features, can occur without affecting the stability of the engine.

To effectively demonstrate real-world usage, this report will focus on two fundamental use cases that illustrate both standard editor functionality and AI-assisted development: **opening files and editing text,** as well as **AI-assisted syntax highlighting**. The former is one of the core mechanics of any code editor, enabling real-time development. Users open files and make changes, which are immediately stored and reflected in the UI, creating a responsive workspace. The latter is what sets apart the Void Editor from the conventional editing tools of years past: through AI-assisted highlighting, the developer can save valuable hours by catching subtle mistakes before they become bugs, receiving relevant and applicable patching suggestions at a rate inconceivable to even the most diligent manual reviewers. Rather than rely on static grammar rules, the editor collaborates with AI services to infer structure and predict language patterns, actively adapting highlighting to match the context of the situation. These use cases make up the fundamental part of the report's architectural description.

2. System - Functionality and Subsystems

Core Engine
The control center coordinates lifecycle, command routing, and the internal event bus. All upper layers depend on it, but it exposes stable, narrow interfaces to keep coupling low.
Depends on: — (base of the stack)

UI Layer
Renders views and handles user interactions (panels, dialogs, notifications). It forwards normalized user intents to the Core and reads preferences from Configuration to render consistently.
Depends on: Core Engine, Configuration System

Editor Component
Owns text buffers, cursor/selection, decorations, diff/preview, and edit application. It consumes syntax and language services but isolates them behind editor-facing APIs to keep the Core clean.
Depends on: UI Layer, Syntax Engine, Core Engine

Syntax Engine
Provides lexical/syntactic analysis for highlighting, folding, and basic structure awareness. It serves the Editor only (not the Core), which keeps language concerns out of the control plane.
Depends on: Editor Component

LSP Client
Communicates with external Language Servers to obtain completions, diagnostics, go-to-definition, etc. Results flow into the Editor; file access happens via the File Manager rather than the Core.
Depends on: Editor Component, File Manager

File Manager

Single entry point for reading/writing files, watching changes, and applying patches to the workspace. It hides persistence details and emits change events the Core can react to.
Depends on: Core Engine, Persistence Layer

Plugin System
Loads, activates, and isolates third-party extensions while exposing well-defined extension points. Security checks are enforced here before plugins can call sensitive capabilities.
Depends on: Core Engine, Security Layer

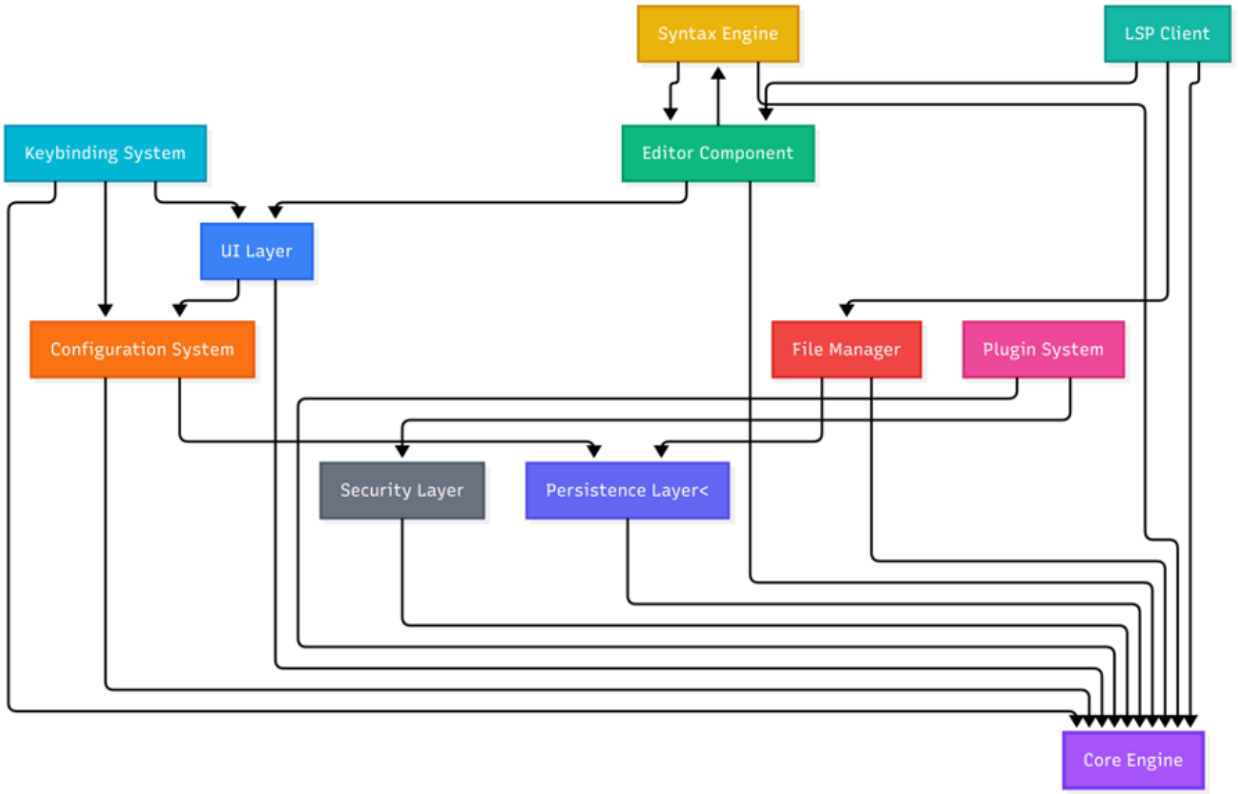
Configuration System
Centralizes user/workspace settings and policy toggles (keybindings, formatters, feature flags). It persists changes and broadcasts updates so UI and Editor can react without direct storage calls.
Depends on: Core Engine, Persistence Layer

Security Layer
Applies permission checks and guards sensitive actions (file writes, network access, plugin capabilities). By concentrating checks here, we keep the rest of the codebase simpler and safer.
Depends on: Core Engine (optionally Persistence Layer if you store policies/keys)

Persistence Layer
Provides local storage and caching for settings, indexes, and session state with clear invalidation rules. Other subsystems read/write through this layer rather than touching the OS directly.
Depends on: Core Engine

Key binding System

Listens to keyboard events and maps them to normalized commands. It reads bindings from Configuration and dispatches to UI/Editor pathways instead of talking to Core directly. Depends on: UI Layer, Configuration System



Upper-layer features (UI, Editor, Syntax, LSP, Keybindings) depend on platform services (File, Plugin, Config, Security, Persistence), which in turn depend on the Core Engine—yielding a one-way, acyclic structure.

Layering and Design Intent

We model the Void Editor as three layers: Features → Platform → Core, with dependencies flowing strictly top-down and remaining acyclic. This structure minimizes coupling, stabilizes

interfaces, and localizes the impact of changes. The Core Engine is the orchestration point (lifecycle, routing, event bus) and exposes a narrow set of stable entry points consumed by upper layers or via platform services.

Subsystem Roles (Overview)

The Editor Component manages text buffers, selections, decorations, diff/preview, and edit application. It renders through the UI Layer and consumes language capabilities from the Syntax Engine (lexical/syntactic structure for highlighting and folding) and the LSP Client (completions, diagnostics, navigation from language servers).

Within the Platform layer, the File Manager acts as the single entry point for file I/O and patch application, leveraging the Persistence Layer for caching and durability. The Configuration System centralizes user/workspace settings and broadcasts changes to dependents. The Plugin System provides well-defined extension points with isolation, while the Security Layer enforces permission checks and guards sensitive operations.

Dependency Rationale

The Syntax Engine and LSP Client do not depend on the Core; their outputs are consumed by the Editor. This keeps language concerns out of the control plane, reduces coupling, and preserves a clear boundary around Core responsibilities.

File access and workspace mutations are consolidated through the File Manager, which can apply policies, emit change events, and coordinate Persistence for caching/invalidation—avoiding scattered storage logic.

Plugin capabilities are routed Plugin → Security → Core, concentrating risk and auditability at a single enforcement point. Modules that require preferences read from Configuration, rather than accessing storage directly.

Alignment with Quality Attributes

Extensibility： Acyclic layering and the Plugin System enable new features to be added in the Features layer without destabilizing Core; extension points remain tightly scoped.

Maintainability： Clear separation of concerns (Editor vs. language services vs. file/config services) confines changes to well-defined modules and their immediate dependencies.

Security and Reliability： Security checks sit at the platform boundary; Persistence provides controlled caching and recovery, reducing inconsistent state after failures.

Performance： Centralized I/O and caching in File Manager + Persistence supports coherent invalidation strategies; excluding language services from Core reduces contention in the control path.

Trade-offs and Mitigations

Indirection introduces an additional hop (e.g., LSP results via the Editor; file updates via the File Manager), but it yields lower coupling, easier substitution, and improved testability. Caches and indexes may drift; we mitigate this with file-event–driven invalidation and explicit rebuild hooks in Persistence. Plugin extensibility increases attack surface; gating sensitive capabilities through Security and maintaining minimal Core interfaces constrains risk.

3. Control Flow, Concurrency, and Data Flow

Global Control Flow

The Void Editor operates through an event-driven control model coordinated by the Core Engine. User input — keyboard, mouse, or command-palette — is captured by the UI Layer, normalized into high-level commands, and dispatched through the Core's event bus.

The Editor Component applies editing operations to the active buffer (text, cursor, undo/redo). It then notifies dependent subsystems:

- UI Layer → requests repaint to reflect buffer updates.
- Syntax Engine / LSP Client → receive buffer-change events to perform incremental tokenization and diagnostics.
- File Manager / Persistence Layer → handle explicit or autosave requests asynchronously.
- Plugin System → invokes registered callbacks via the extension API.

Control flow is strictly unidirectional:
User Input → Core Engine → Editor Component → Dependent Subsystems → UI Update.
Only the Editor Component mutates shared state; all others observe or derive from it.

Concurrency Model

Concurrency preserves responsiveness while minimizing synchronization complexity.

- UI Thread – runs event loop and rendering; must stay non-blocking.
- Background Syntax / LSP Tasks – perform tokenization, linting, completions asynchronously, then post results to UI.

- Autosave Worker – monitors dirty buffers and writes snapshots to disk.
- File Manager Tasks – manage open/save operations and file watching.
- Plugin Tasks – execute third-party logic in isolated contexts.
- Security Checks – run in lightweight async tasks when sensitive actions occur.

The Editor Component and its buffer remain confined to a single thread. Workers access only immutable snapshots or diffs, ensuring deterministic concurrency.

Data Flow Between Subsystems

Path	Flow Summary
Typing Path	UI Input → Core Engine dispatch → Editor Component updates Buffer → Syntax Engine / LSP analyze → UI renders highlights & diagnostics
Open File Path	UI Command → File Manager loads file → Editor Component initializes Buffer → UI renders content → Syntax Engine warm-starts analysis
Save Path	UI or Autosave trigger → File Manager / Persistence serialize Buffer snapshot → Core Engine updates status → UI confirmation
Settings Path	Configuration System publishes updates → Core Engine / UI / Editor consume them dynamically
Plugin Events	Plugin System listens for buffer or command events → invokes registered handlers asynchronously

Solid arrows in diagrams indicate control passing (commands); dashed arrows indicate data passing (buffers, tokens, snapshots).

Responsibilities and Boundaries

Subsystem	**Primary Responsibility**
UI Layer	Capture input, dispatch commands, render state
Core Engine	Route commands, manage lifecycle, coordinate subsystems
Editor Component	Own buffer, cursor, undo/redo logic
Syntax Engine / LSP Client	Analyze code and provide diagnostics/completions

File Manager / Persistence Layer	Handle durable I/O operations
Plugin System	Load and sandbox extensions
Configuration System	Manage and broadcast settings
Security Layer	Enforce permissions on sensitive actions

This structure enforces single ownership of mutable state and clean separation of concerns.

Extensibility and Integration

The control/concurrency/data models integrate directly with the two primary use cases (Section 4):

1. *Open File & Edit Text* — shows file I/O and buffer update propagation.

2. *Syntax Highlighting While Typing* — demonstrates concurrent syntax analysis and UI refresh.

These will be depicted through sequence diagrams in the next Section.

4. Use Cases

Use Case 1: Open File and Edit Text

Actor: User (developer)

Goal: Open a source code file, load it into the editor, and make real-time edits that update both the interface and internal buffer.

Preconditions:

1. Void Editor is running.
2. The file system is accessible.

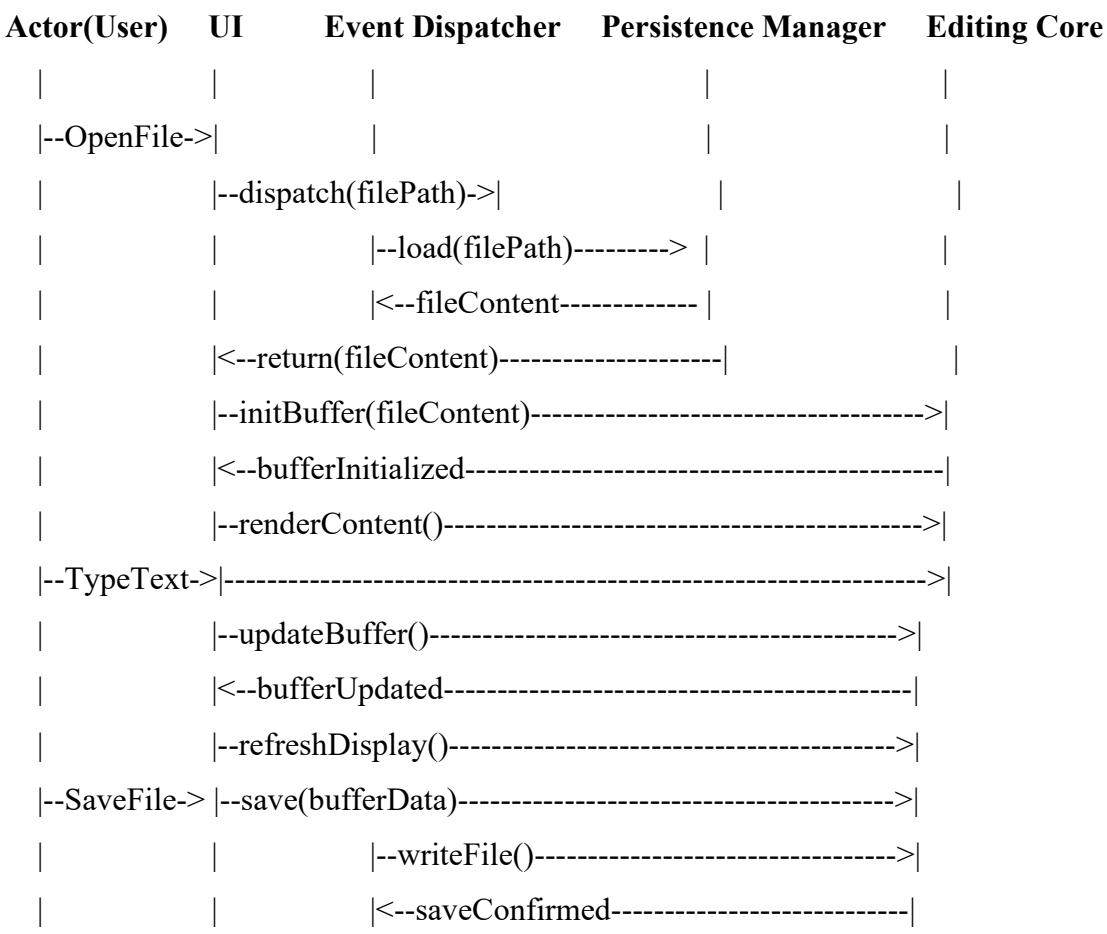
Postconditions:

1. The selected file's content is loaded into the editor's buffer.

2. The user's edits are reflected and maintained in memory.

Description	Interacting Components
1. The user selects a file via the "Open File" dialog.	User -> UI
2. The UI sends a load request through the Event Dispatcher to the Persistence Manager.	UI -> Event Dispatcher -> Persistence Manager
3. The Persistence Manager loads the file from the disk and returns its contents.	Persistence Manager -> Event Dispatcher -> Editing Core
4. The Editing Core initializes a buffer and stores the file content.	
5. The UI displays the loaded file	Editing Core -> UI
6. The user types and the UI sends keystrokes to Editing Core.	User -> UI -> Editing Core
7. Editing core updates buffer and refreshes UI display	Editing Core -> UI
8. When updated, saved content is sent back to Persistence Manager	Editing Core -> Persistence Manager

Sequence Diagram for Open File and Edit Text



This use case primarily activates the UI Layer, Core Engine, File Manager, and Editor Component subsystems.

Use Case 2: AI-Assisted Syntax Highlighting While Typing

Actor: User (developer)

Goal: Provide real-time syntax highlighting and AI-based code suggestions as the user types.

Preconditions:

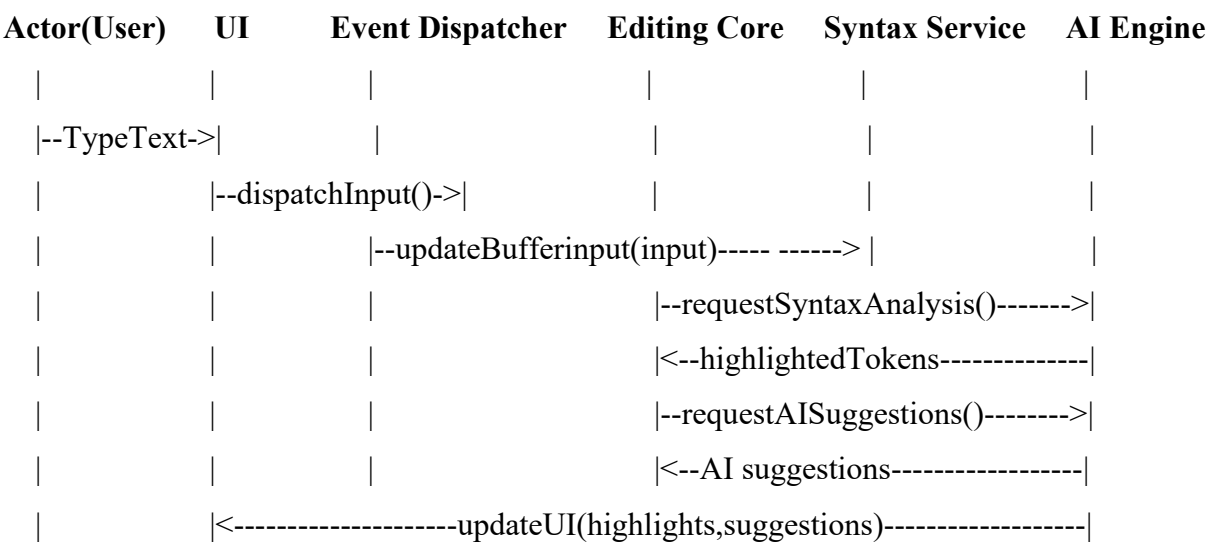
1. File is open in the editor buffer.
2. Syntax Service and AI Engine are active.

Postconditions:

1. Code text is syntax-highlighted dynamically.
2. AI suggestions appear inline.

Description	Interacting Components
1. User Types in the editor	User -> UI
2. UI dispatches input to Editing Core	UI -> Event Dispatcher -> Editing Core
3. Editing Core updates text buffer	Editing Core
4. Editing Core requests syntax analysis	Editing Core -> Syntax Service
5. Syntax Service returns highlighted tokens	Syntax Service -> Editing Core
6. Editing Core sends current context to AI Assistance Engine	Editing Core -> AI Engine
7. AI Engine returns suggestion data	AI Engine -> Editing Core
8. Editing Core updates UI with highlights and suggestions	Editing Core -> UI

Sequence Diagram for AI-Assisted Syntax Highlighting While Typing



This use case primarily activates the UI Layer, Core Engine, Syntax Engine, and AI Engine (LSP Client) subsystems.

5. External Interfaces

Graphical User Interface (GUI)

The GUI is based on the familiar VS Code layout (editor panels, sidebars, menus) since Void is a fork of VS Code. Key windows and panels:

- Main Window: Displays the active workspace, including open files and panels.
- Editor Pane: Core text-editing region connected to the syntax engine subsystem.
- Menu Bar: Provides access to file operations, settings, and plugin management.
- Panels: Optional UI components such as file explorer, terminal, or search view.
- Command Palette: Text-based interface for executing system commands. Mirrors VS Code design for user familiarity
- The GUI communicates with the core engine through an event-driven architecture, translating user interactions (such as clicks, commands, and shortcuts) into internal actions. It supports importing existing VS Code themes, key bindings, and settings.

a) **Keyboard/Mouse Input**

- Tab is used for accepting autocomplete suggestions.
- Ctrl+K triggers Quick Edit functionality for inline editing.
- Ctrl+L opens the Chat interface
- Mouse actions in GUI: selecting code blocks, invoking context menus, using the sidebar, etc.

b) **File System Interface**

The File System Interface allows Void to interact with local storage and external drives for reading and writing files. Conceptually, it connects the Persistence Layer with the File Manager Subsystem.

- **Read Operations:** Open files into editor buffers, preserving encoding and syntax mode.
- **Write Operations:** Save buffers to disk with file-locking and backup options.
- **Recent Files Management:** Stores access metadata in local configuration (YAML/JSON).

c) **Plugin API** – interface for extensions.

Void supports modular extension through a Plugin API, allowing developers to integrate new syntax definitions, themes, or utilities. Conceptual design:

- **Interface Layer:** Defines a standard API for extensions to register commands, UI elements, and syntax rules.
- **Isolation Principle:** Each plugin runs in its own context to maintain editor stability.
- **Lifecycle:** The plugin system in Void enables extensions to add features in a controlled way, managing their loading, execution, and removal without affecting the core editor.

d) **Config Files**

Void centralizes all editor, theme, and AI configuration data into JSON/YAML files stored in the user's workspace. The Configuration System broadcasts updates dynamically so that other subsystems (UI, Editor, Plugins) can react without restarting.

6. Data Dictionary

Glossary of key terms:

Term	Definition
Buffer	In-memory representation of a text file.
Syntax Engine	The smallest unit of syntax analysis.
Plugin	Independent module extending functionality.
Event	Action triggered by user input (e.g., keypress, click) or system change.
Configuration File	JSON/YAML document defining user preferences.
Persistence Layer	Handles reading/writing data to disk.
Workspace	The set of currently opened projects and files.
Command Palette	UI component enabling quick access to actions via search.

Conceptual Naming Rules:

- **Modules and Packages:**
  - Use lowercase names separated by underscores (snake_case)
- **Classes:**

    - PascalCase (e.g., EditorWindow, FileHandler).
- **Functions and Variables:**
    - lowerCamelCase (e.g., saveFile(), loadConfig()).
- **Constants:**
    - ALL_CAPS (e.g., MAX_BUFFER_SIZE).
- **Plugins:**
    - Follow <editor>-<plugin> format (e.g., void-markdown, void-python).
- **Configuration Files:**
    - Stored under ".void/config/", named according to their domain (editor.json, themes.yaml).

7. Conclusion

The Void Editor's conceptual architecture cleanly separates features (UI, Editor, Syntax/LSP, Keybindings) from platform services (File Manager, Plugin, Config, Security, Persistence) atop a small, stable Core Engine. Control is event-driven and unidirectional (Input → Core → Editor → Subsystems → UI), while concurrency keeps the UI responsive by isolating background work to workers that consume immutable snapshots.
Strengths.

- Modularity: Acyclic, top-down layering enables local change without ripple effects.
- Extensibility: A scoped Plugin System with isolation points lets teams add features safely.
- Separation of concerns: Editor owns mutable buffer; language services and plugins observe/annotate.
- Security & reliability: Security Layer gates sensitive capabilities; Persistence supports recovery.
- Performance: Centralized I/O + caching in File Manager/Persistence reduces contention in the control path.

Potential limitations

- Plugin management complexity: Versioning, capability scoping, and sandboxing policies require governance.
- Concurrency pitfalls: Snapshot/diff discipline must be enforced to avoid stale data and race-adjacent bugs.
- Indirection overhead: Extra hops (Editor-mediated results, centralized I/O) add latency if not tuned.

A small core, strong boundaries, and disciplined flows make Void easy to grow—so long as plugins and concurrency are treated as first-class citizens, not afterthoughts.

Lessons Learned

- Architecture without code is still actionable. By focusing on responsibilities, invariants, and interfaces, we clarified how the system should behave before committing to implementation details.
- Boundaries are the real design. Letting the Editor be the sole owner of mutable state—and routing language services, plugins, and I/O around it—reduced coupling and made failure modes easier to reason about.
- Diagrams > paragraphs for flow. Sequence and dependency diagrams turned abstract rules (e.g., "unidirectional control") into checkable constraints and onboarding aids.
- Concurrency needs policy, not vibes. Immutable snapshots/diffs, UI thread non-blocking rules, and event-driven invalidation gave us a shared playbook to keep performance and determinism.
- Security belongs at the edge. "Plugin → Security → Core" concentrates risk and auditing, simplifying threat modeling.
- Centralization pays off. File Manager + Persistence as the single I/O lane improved caching, invalidation, and testability.
- Future work to watch. Formal plugin capability taxonomy, performance budgets for background workers, and golden-path diagrams for the top use cases (Open/Edit, Highlight-while-Typing).

8. References

1. IEEE 830-1998: *Recommended Practice for Software Requirements Specifications*
2. ISO/IEC/IEEE 29148-2018: *Systems and Software Engineering – Life Cycle Processes – Requirements Engineering*
3. IEEE 1471-2000: *Architectural Description of Software-Intensive Systems*
4. Void Linux Documentation – Contributing & Naming Conventions
5. Void Packages Repository (GitHub)
6. Y Combinator – Void Company Profile

Appendix A — AI Collaboration Report

AI Member Profile and Selection Process

Our group's virtual AI teammate for this deliverable was **OpenAI GPT-5 (September 2025 release)**, accessed through ChatGPT Plus.
We chose GPT-5 because of its ability to generate clear, structured text and assist with general organization of large documents.
We compared it briefly with other tools (Claude 3.5, Gemini 1.5) and found GPT-5 more consistent when re-phrasing technical explanations and providing writing suggestions without altering the original meaning.

We treated the AI as a **writing and brainstorming assistant**. All architectural reasoning, subsystem mapping, and diagram design were completed by human members.

Tasks Assigned to the AI Teammate

The AI was used selectively for supportive, time-saving activities:

Task	Purpose
Brainstorming and clarifying **section outlines**	Helped structure the flow of the report before writing began.
Improving **grammar and clarity** in long paragraphs	Ensured writing consistency across sections written by different members.
Providing **slide summary phrasing**	Helped simplify technical sentences for presentation slides.
Suggesting **transitions** between sections	Made the report flow more naturally.
Generating **initial glossary wording**	Provided first drafts of short definitions for technical terms (later reviewed by team).

Interaction Protocol and Prompting Strategy

One group member (team lead) interacted directly with GPT-5 on behalf of the team.
The typical workflow consisted of:

1. **Preparing context:** sharing a rough outline or paragraph needing refinement.

2. **Prompting for support:** asking GPT-5 for feedback such as “make this paragraph more concise” or “suggest a clear way to describe the flow of control.”

3. **Reviewing outputs:** each suggestion was verified and, if useful, lightly edited before integration.

Example prompt:

> “Can you make this explanation of event-driven control easier to understand for a reader who isn’t familiar with concurrency models?”

This approach helped maintain human ownership while still benefiting from AI feedback.

Validation and Quality Control

To ensure accuracy and originality:

- All technical explanations were **cross-checked against course materials** and documentation before inclusion.
- AI-edited text was **reviewed and rewritten** by human members when needed to preserve the team’s own voice.
- No AI-generated diagrams, figures, or citations were used.
- Grammarly and manual proofreading were used to double-check the final version for tone and grammar.

The group agreed that the AI’s suggestions were useful primarily for writing flow and polish, not for content generation.

Quantitative Contribution Estimate

Category	Human %	AI %
Research & Analysis	95	5
Drafting & Editing	85	15
Slides & Formatting	90	10
Diagrams & Technical Content	100	0
Overall estimated AI contribution: ≈ **10–15 %** of total effort.		

Reflection on Human–AI Team Dynamics

Working with an AI teammate felt similar to having a writing assistant on call.
It accelerated minor editing tasks and helped unify the tone across sections written by different people. However, the group still spent time verifying every suggestion, since some edits slightly changed the technical meaning.

Key takeaways:

- GPT-5 is helpful for phrasing and structure, but should never replace domain reasoning.
- Small, focused prompts produce the most reliable results.
- Reviewing together as a group kept accountability and consistency high.

Overall, the collaboration was efficient and educational. We learned that using AI thoughtfully can improve clarity and save time, but the strongest results still come from collective human judgment and peer review.