CISC 322/326 - Software Architecture
October 3rd, 2025
Assignment 1: Report
Conceptual Architecture of Void

Group 10: "welcome to my homepage"

Alec Nakhnikian – 22pht3@queensu.ca

Ori Dembo – 22gwhj@queensu.ca

Tomer Lapid - 23rtn2@queensu.ca

Mathieu Chretien – 22bgq2@queensu.ca

Logan – 20lwkb@queensu.ca

Linus Li –21ll84@queensu.ca

Section 1.01 Abstract

Void is an open-source, privacy-first fork of Visual Studio Code that brings "Cursor-class" AI assistance, quick inline edits, autocomplete, contextual chat, and agentic workflows, all without routing developer code through a proprietary relay. It connects directly to cloud providers or local runtimes, letting teams choose models while retaining control of their data and keeping the familiar VS code ecosystems of themes, key bindings, and extensions.

The report presents the system's conceptual architecture and evaluates how that design supports evolvability, testability, and day-to-day performance. At a high level, Void layers a provider-agnostic AI runtime and "apply/checkpoint" pipeline onto VS code's electron-based multi-process foundation (main process, renderer/workbench, extension host, and language servers). The AI runtime handles capability detection and streams structured edit intents, which the apply pipelines then turn into search/replace blocks, visual diffs, and checkpointed history for safe and reviewable changes. Recent releases strengthened this path withy features such as visual diffs in the edit tool, checkpoints for jumping between edits, and ecosystem integrations like MCP and expanded provider support.

Our Findings are threefold. First, the architecture's clear process and service boundaries enable parallelism while keeping the UI responsive, which is essential for developer trust and adoption. Second, the provider-adapter facade and capability gating isolate fast-moving model ecosystems from product UX, improving testability and reducing cross-team breakage. Third, these seams naturally define divisions of responsibilities: platform and upstream sync, AI runtime and provider integrations, workbench/UX, language platform and extensions, private/security review, release engineering, and QA/performance. We illustrate these conclusions with two essential use cases: Quick/Inline edit and Agent-driven multi-file refactoring. Collectively, the architecture positions Void to rapidly evolve with the AI tooling landscape while preserving transparency, auditability, and user control.

Section 1.02 Introduction & Overview

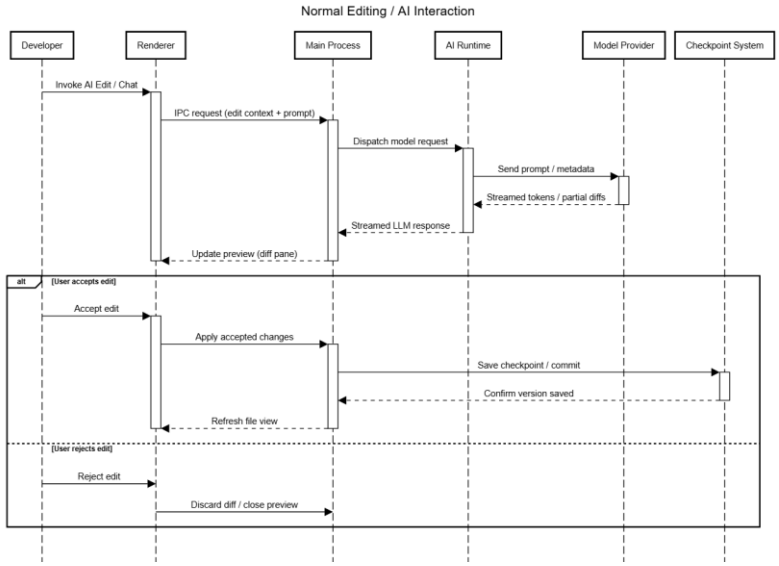
Software development tools are shifting from "editors with AI plug-ins" to AI-native environments where intelligent assistance shapes everyday workflows. Void occupies the latter category. As it is a Fork of Visual Studio Code, it retains the familiar workbench, extensions, and developer ergonomics while elevating AI to a first-class runtime. In practice, this means inline/quick edits, FIM-based autocomplete, contextual chat, and agentic operations that can read/write files, search the codebase, and invoke controlled tools under explicit user approval. Throughout, Void emphasizes privacy and choice: developers can connect directly to cloud providers or run models locally, and edits always flow through a human-in-the-loop review with rollback via checkpoints.

This report examines Void from a conceptual architecture perspective, answering two core questions: *what the system does* and *how it is broken into interacting parts*. We model the system as five cooperating subsystems with clear responsibilities and trust boundaries: the Core Editor for UI, workspace orchestration, and extension hosting the AI Agent Layer implementing Chat, Quick/Inline Edit, Gather, and Agent a Model/Provider Layer that adapts to local runtimes

and direct cloud APIs with capability detection an Apply/Checkpoint Pipeline that converts model outputs into deterministic diffs, presents visual review, applies changes atomically, and records provenance and Distribution & Community services that deliver updates, onboarding, and contributor support. The architecture blends layered, plug-in, and pipes-and-filters styles to balance extensibility with operational safety.

This report follows course expectations and industry practice. Conceptual Architecture outlines the five subsystems, their responsibilities, and interactions, with cleaned architectural/dependency diagrams. External Interfaces cover exchanges with provider APIs, local runtimes, system services, and language tooling. Use Cases trace two end-to-end flows via sequence diagrams. A Data Dictionary standardizes core artifacts, and the Naming Conventions document figures abbreviations.

We add a brief evolution summary framed by “experiment -> stabilize -> refine,” plus control/data flow notes on concurrency and isolation. The implications for the Division of Responsibilities map seem to be clear to teams. In short, Void’s architecture is AI-native: it preserves VS Code familiarity, adds agentic depth, and protects developer autonomy through clear boundaries and audited apply paths.



The Developer invokes an AI action in the Renderer, which sends an IPC request to the Main Process containing the edit context and prompt. The Main Process forwards this to the AI Runtime, which communicates with the external Model Provider to generate a streamed response. As tokens or diffs stream back, the Renderer updates the preview in real time.
If the user accepts the edit, the Main Process applies the changes, saves a new checkpoint, and refreshes the file view. If the edit is rejected, the diff is simply discarded.

Section 1.03 Conceptual Architecture

Void is a VS Code-based open-source editor that integrates AI assistance without a proprietary relay. User connect directly to model providers (Claude/GPT/Gemini, or OpenAI-compatible endpoints) or to local runtimes like Ollama, keeping prompts and code under their control while

retaining the VS Code ecosystem of themes, key bindings, and extensions. These invariants, VS code compatibility and “no middleman” data flow, shape the system’s top-level design and the seams between its subsystems.

At the highest level, Void composes five cooperating subsystems on top of Electron/VS Code’s mutli-process foundation. (1) Workbench/UI (renderer) hosts the editor surface, chat panel, settings, and diff reviews, while orchestrating prompts and approvals. (2) AI Agent Layer exposes user-visible capabilities, Quick/Inline editing, Tab-autocomplete when a model supports FIM, contextual chat, “Agent” and “Gather” modes with controlled tool access. (3) Model/Provider Layer (main process) adapts cloud APIs and local runtimes, performs capability detection (FIM, tool-calling, “reasoning” modes), streams tokens, and enforces rate/timeout budgets. (4) Apply/Checkpoint Pipeline turns model outputs into structured search/replace blocks, renders visual diffs, and writes approved changes while recording checkpoints for rollback. (5) Platform and Integrations cover the Extension Host and Language Server Protocol (LSP) processes, the built-in terminal, auto-updates, packaging, and release plumbing. The changelog traces these seams explicity: Agent/Gather/Chat and capability auto-detection (v1.0.2), Fast Apply defaults, checkpoints (v1.2.1), Linux support and background terminals for Agent (v1.3.9), and MCP plus visual diffs in the Edit tool (v1.4.1).

Key Interactions follow the process boundaries that VS Code and Electron encourage. The renderer remains responsive while heavy work, such as the model I/O or file mutation, routes through the main process and out-of-process services. Electron’s IPC connects renderer to main, VS Code’s extension host isolates extension logic, and LSP servers run as separate processes and speak JSON-RPC. In Void, this yields a stable loop: the Workbench captures a prompt, the Agent Layer formulates a provider request, the Provider Layer streams results back, the Apply pipeline synthesizes diffs and awaits user approval, and the approved edits update the workspace and feed back into the context (symbol links, history, etc.) The project even codifies “model calls from Node via IPC” to avoid browser CORS/CSP pitfalls, reinforcing the renderer/main split.

Architectural Styles (Garlan and Shaw). The system is primarily layered (Workbench/UI as the core, with Agent Layer, Provider Layer, and the Apply Pipeline successively), with a strong microkernel/plug-in flavor via the Extension Host and LSP integrations. AI “tools” behave like plug-ins behind typed facades. Cross-component communication is implicit invocation (streams, IPC, JSON-RPC). The LSP and provider adapters add client-server interactions to external processes and APIs. Which leads us to a heterogenous composition, layered + microkernel + implicit invocation + client-server, which is textbook Garlan and Shaw practice for large, evolvable tools.

High-lever patterns and abstractions.

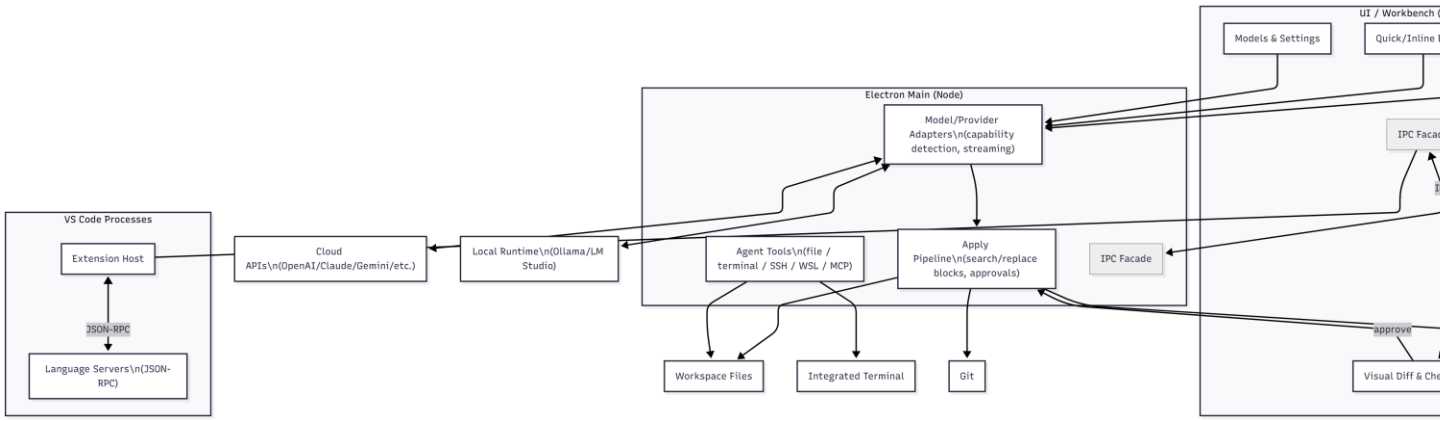
- Adapter/Facade in the model/provider layer decouples product UX from vendor-specific APIs and capabilities (FIM , tool-calling, and reasoning levels).
- Pipeline in apply/checkpoints turns unstructured model text into deterministic edit blocks, which then goes to diffs, then approved writes, and then checkpointed history.

- Command/Intent objects represent edits and toll invocations, enabling preview/undo and auditability.
- Policy gates around Agent tools (terminal/SSH/WSL, MCP) enforce explicit user approval and scope. These choices appear in release notes as capability gating, Fast Apply blocks, background terminals, and MCP integration rather than ad-hoc shortcuts.

Goals and their architectural support.

- Evolvability. Provider adapters and capability detection let the team add models (Claude 4, OpenAI-compatible, Gemini, Grok), protocols (MCP), and settings (context length, “thinking” support) without destabilizing the Workbench. Regular patches adding providers and features illustrate the payoff of this separation.
- Testability. Narrow IPC facades and an approval-centric Apply pipeline make behaviors observable. Token streams, generated blocks, diffs, and user decisions are all separatable and testable.
- Performance/Responsiveness. Electron’s main/renderer split, VS Code’s Extension Host, and out-of-process LSP keep typing/scrolling smooth while AI calls, diagnostics, and background terminals run concurrently.
- Global control flow. Human-in-the-loop is enforced. No file writes occur outside the Apply pipeline. Agent tool use is mediated by explicit approval and scoped permissions. Recent visual diffs make this control visible to users.
- Concurrency. Streaming model responses, background terminals, and multiples LSPs operate in parallel across processes. IPC/JSON-RPC decouple producers and consumers, and checkpoints/approvals mitigate race conditions between “AI edits” and “human edits”.

As per external interfaces, on the model side, Void talks to hosted APIs (OpenAI compatible, Anthropic, Google, etc.) and to local runtimes (e.g. Ollama), typically on localhost or via SSH port-forwarding. On the developer side, it exchanges content with the filesystem (read/write/rename), integrates terminals (commands and output streams), and VCS (like AI-generated commit messages). Community guides and the official site corroborate “any LLM, anywhere” setups and local-model workflows.
The following is a Dependency Diagram modeling this:



This structure perfectly fits Void's mission. The UI/Agent/Provider/Apply layers keep responsibilities crisp: the Workbench focuses on interaction and stability, the Provider Layer absorbs model churn, the Apply pipeline guarantees reviewable, reversible changes, and Platform processes (Ext Host, LSP, terminals) stay fault-isolated. This matches the project's public stance on privacy and direct connections and is borne out release-by-release as features (Agent/Gather, Fast Apply, checkpoints, visual diffs, MCP, and expanded provider support) drop into existing seams instead of introducing brittle detours.

Finally, the architecture is intentionally operationally flexible. Developers can point at local or hosted models, switch providers per task, and still get uniform UX because adapters and the pipeline normalize capabilities. This "any LLM, anywhere" posture, validated by community how-tos and first-look reviews, is less a feature list than an architectural commitment to choice with guardrails.

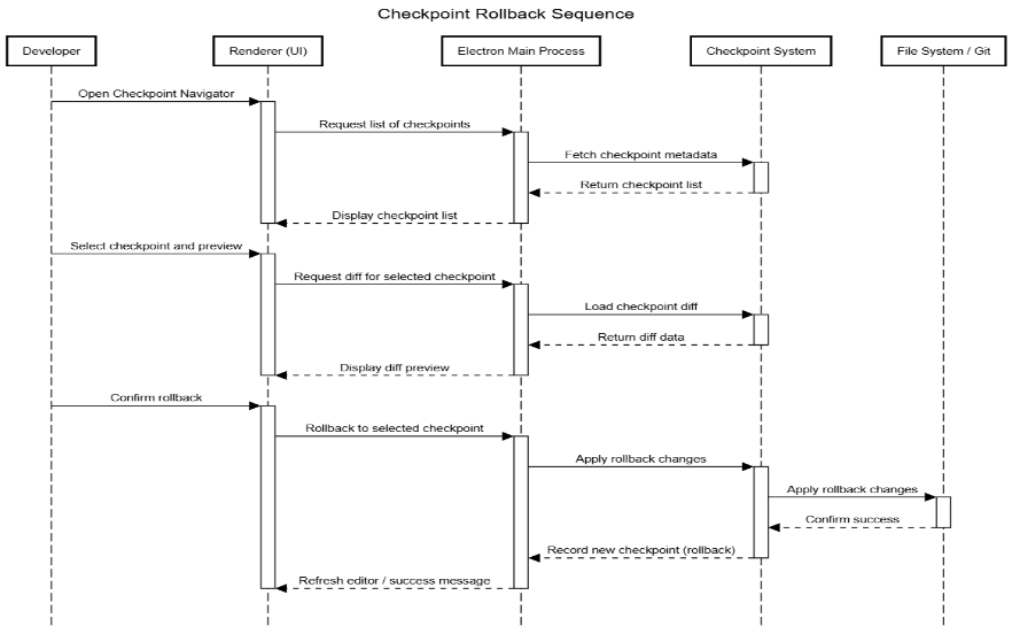
1.04 System Functionality

Void is an open-source, AI-powered code editor forked from VS Code that aims to deliver Cursor-class assistance while keeping developers' data under their control. At a feature level, it provides inline/quick edits, tab-autocomplete, contextual chat, and agentic workflows with file-system awareness and prompt transparency. Because it inherits VS Code's ecosystem, users retain familiar themes, key bindings, and extensions.

The system is organized into five cooperating subsystems: the Core Editor (Workbench), which renders the UI and manages edits; the AI Agent Layer, exposing Chat, Quick/Inline Edit, Agent Mode, and Gather Mode; a Model/Provider Layer that connects directly to cloud APIs or local runtimes like Ollama, with capability detection; the Apply/Checkpoint Pipeline, which converts AI outputs into search-and-replace blocks, displays visual diffs, and records Checkpoints for rollback; and Distribution & Community services. These boundaries reflect a privacy-first approach and support everyday workflows.

Key interactions proceed as follows: the Workbench routes user prompts to the Agent Layer, which selects a provider via the Model/Provider Layer, streams results back, and emits structured edit intents. The Apply/Checkpoint Pipeline transforms these intents into diffs, presents them for human approval, and saves a checkpointed history. Accepted changes update the workspace and inform subsequent context. Recent beta updates formalize these steps with Fast Apply, visual diffs in the Edit tool, reasoning-effort controls, and expanded provider compatibility, strengthening the feedback loop between agent outputs and safe file mutation.

Externally visible behavior aligns with this architecture: InfoQ and its FOSS emphasize inline editing, contextual chat, code generation, file-system-aware context, and privacy/transparency as first-class properties. Community walkthroughs showcase "any LLM, anywhere" setups that integrate local Ollama with cloud models while maintaining local data. Together, these confirm both the feature surface and the privacy-centric integration strategy.



The Developer opens the Checkpoint Navigator in the Renderer, which requests checkpoint data from the Main Process. The Main Process queries the Checkpoint System and File System to fetch and return the list and diff data for preview.
After the user confirms, the Main Process instructs the Checkpoint System to apply the rollback and record a new checkpoint. The Renderer then refreshes the editor with the restored state. This asynchronous flow keeps the UI responsive while heavy file operations run in separate processes.

Section 1.05 System Evolution

This section covers the evolution of the Void Editor since its initial launch on October 1st, 2024. As of the most recent release on June 5th, 2025, there have been eight versions ranging from the initial early launch to v1.4.1. Along with covering the added features, the section will showcase how the editor moved from just simple inline edits to full agentic workflows. [2]

1.05.1 Early Launch (Oct 1, 2024)

This initial launch saw Void's website and repo be initialized. At this point, early builds already possessed basic features like editor-embedded LLM streaming, custom history, a custom editor, and it allowed users to build an early prototype version of a slow apply system, setting the stage for the later 1.0 beta. [2]

1.05.2 Void 1.0.0 Beta Release (Jan 19, 2025)

The first official release saw the introduction of quick edits (FIM-prompting, output parsing, and history management) and autocomplete. The editor is now shipped with native migration into the VS Code codebase. It also added a streaming-diff UI, one-click settings, provider selection, Ollama auto-detection, IPC-based local model routing, native Accept/Reject controls, and chat-driven file suggestions. There was also a license switch to Apache-2.0, further reinforcing the open-source idea that the Void Editor holds to this day. [2]

1.05.3 Void 1.0.1 Beta Patch #1 (Jan 23, 2025)

This patch brought a few new features to the editor. A new default theme was added, chat performance improvements were implemented, and autocomplete was temporarily disabled to re-implement it in a future release. Diff streaming and FIM parsing algorithms also saw refinements, prompt fixes were applied to Gemini and OpenAI o1, and DeepSeek support was added. This version saw early stability features being implemented and increased provider coverage. [2]

1.05.4 Void 1.0.2 Beta Patch #2 (Mar 22, 2025)

This version introduced separate editor modes; Chat mode acts as just a chatbot, Gather mode allows for reading and searching through the code without edits, and Agent mode was the original standard mode, being able to read/write files, search the codebase, and control the terminal. Void also auto-detects model capabilities like support tools, FIM, and thinking/reasoning, with a slider being added to modify reasoning level performance. It also re-enabled autocomplete, introduced Fast Apply, links code symbols in chat mode, rebased to VS Code 1.99.0, and expands model support to include Claude 3.7, Deepseek V3, Gemini 2.0, and QwQ. [2]

Section 1.05.5 Void 1.0.3 - Beta Patch #3 (Apr 7, 2025)

This experimental build was pushed to Discord for early testing of the next patch and acted as the initial version of the next patch. [2]

Section 1.05.6 Void 1.2.1 - Beta Patch #4 (Apr 14, 2025)

This change in version number from v1.0 to v1.2 added checkpoints to jump between LLM edits, extended the Agent mode to fix lint errors, upgraded tool-calling so any model can run as an agent, dynamic context squashing, and SSH + WSL support. Following previous trends, it also further enhanced model support, allowing for the usage of Gemini 2.5 Pro, GPT-4.1 (Quasar Alpha), OpenHands LM, DeepSeek V3, and Phi-4. Void Editor also implemented auto-update support, leading to a shift towards “smaller and more frequent changes”. [2]

Section 1.05.7 Void 1.2.4 - Beta Patch #5 (Apr 16, 2025)

This small update added first-launch model onboarding with OpenAI o3 / o4-mini support. [2]

Section 1.05.8 Void 1.3.9 - Beta Patch #6 (May 14, 2025)

Jumping up in version number to v1.3, this version further enhances the editor’s capabilities by adding configurable per-model parameters for context length, thinking support, tool formatting and more, new @file and @folder abilities, persistent background terminals support for Agent mode, faster Fast-Apply, improved context-window truncation and a redesigned chat history and settings pane. In a big change for this version, native Linux support was added, and there was also improved onboarding for Gemini and OpenRouter. [2]

Section 1.05.9 Void 1.4.1 - Beta Patch #7 (Jun 5, 2025)

The latest version of the Void Editor added MCP support enabling internet access, AI-generated commit messages, visual diffs replacing plaintext in the Edit tool, and increased support for Claude 4, Azure, and Ollama.

Section 1.06 Control and Data Flow

When a user begins an action, such as requesting an AI-generated code suggestion, the event first occurs in the Renderer process, which handles the user interface. The Renderer captures the user's input, such as the location of a cursor in a source file, and packages it into a structured request. This request is then sent via Electron's Inter-Process Communication (IPC) system to the Main process, which has access to system-level operations such as reading project files, managing extensions, or invoking AI model inference. By routing this task to the Main process, heavy computation is offloaded from the UI, ensuring that the editor remains responsive while the request is processed.

Once the request reaches the Main process, it is directed to the appropriate service responsible for that operation. In this example, the editCodeService receives the AI prompt and coordinates with the LLM Connection Layer to generate suggested code edits. Parallel services, such as the tools system and approval state system, ensure that AI operations have controlled access to workspace resources and that suggested edits undergo user approval before being applied. Each service communicates asynchronously, enabling other operations, like editing files or searching the project, to continue without interruption [3].

During this process, configuration and state management are centralized using the Singleton pattern. The voidSettingsService ensures that all services, including the AI Service, editCodeService, and Language Server Service, access a consistent source of settings, API keys, and model capabilities [1]. This prevents conflicting behavior and ensures predictable results even when multiple AI computations or code operations occur concurrently [4].

After the AI service generates suggested edits, results are streamed incrementally back to the Renderer process. The React-based UI receives these updates and selectively renders affected components, such as the code editor, minimap, outline, or search panel. Because React updates only the necessary parts of the interface via the virtual DOM, the user sees partial results immediately, and the editor avoids freezing or full-page refreshes [5].

Finally, once the AI suggestions are approved by the user, the editCodeService applies the changes to the workspace. These updates are recorded in the system's context, which maintains symbol links, history, and other metadata. This chronological flow—from user input in the Renderer, through IPC to the Main process, through services for computation and validation, and back to the Renderer for display—illustrates how Void IDE coordinates control and data across multiple processes while maintaining responsiveness, stability, and modularity.

Section 1.07 Concurrency

As a fork of Visual Studio Code, Void inherits its multi-process architecture through Electron. Electron applications are composed of a Main process responsible for the complete application lifecycle, window creation, and privileged APIs, and one or more Renderer processes that operate to provide the user interface. Communication between these processes is facilitated through inter-process communication (IPC) [1][2]. The separation enables the Void user interface to remain responsive, even during complex operations that are performed concurrently.

Beyond Electron's split, Visual Studio Code (and hence Void) also splits extension logic into a standalone Extension Host process. This is an isolated Node.js runtime, ensuring that improper or computationally intensive extensions cannot block the interface [3]. This architectural feature is demonstrated by the vast /extensions/ directory in the Void repository, which supports language features in Python, C++, and Git [7]. Void maintains fault isolation and responsiveness by performing these tasks in a separate process. The language intelligence is delivered via the Language Server Protocol (LSP), which spawns the server components as independent processes. These servers can communicate with the Extension Host with JSON-RPC for diagnostics, completions, and refactorings. Multiple language servers can also be run at once (e.g., Python and C++ simultaneously), each in its own process and activated on demand [5]. This further delineates the concurrent and distributed design of the system architecture.

Void extends this concurrency model to support AI chat and agent features through a dedicated API surface that spans process boundaries as well. By viewing the source tree, one can see that these APIs are explicitly split across process borders. For example, the Extension Host side of the Chat Agent API is defined in src/vs/workbench/api/common/extHostChatAgents2.ts, while src/vs/workbench/api/browser/mainThreadChatAgents2.ts provides the equivalent Renderer-side logic [4][7]. In the same folder, files such as extHostChatSessions.ts can be seen, demonstrating further the pattern of separation. We can see from these examples that chat and agent tasks are processed separately from the UI, with results being sent back asynchronously. The Void Codebase Guide also states that large language model (LLM) requests are purposely routed via the Main process in order to circumvent browser constraints and take advantage of Node.js modules [7], which supports this architectural decision to relieve the heavy workload from the UI.

This concurrency architecture offers three important advantages. First, it ensures responsiveness by preventing AI calls or extension tasks from interfering with user tasks, such as typing and scrolling. Second, it offers fault isolation, as a crashed extension or language server can be restarted without disabling the editor as a whole [3]. Lastly, it supports parallel execution, as several processes, including extension hosts, language servers, and AI providers, can use concurrent CPU cores.

Despite these advantages, concurrency also introduces potential risks that the Void architecture includes safeguards to address. Firstly, concurrent edits from the AI agent and the user can potentially race against each other and/or conflict. The architecture mitigates this problem with a diff/approval flow, where Void waits for user confirmation or rejection before implementing an AI-suggested code change. Another aspect is that streaming AI output could overload the UI, but

throttling and IPC buffering prevent this from happening. Finally, Void routes AI calls through the main process. This restricts the renderer from handling privileged Node.js modules directly, reducing security risks and ensuring the editor's stability [3][7].

Section 1.08 Implications for Division of Responsibilities Among Participating Developers

Void's technical foundation—built on Electron's multi-process model, the VS Code workbench, a separate Extension Host, multiple LSP servers, and its own AI layer—creates natural boundaries that define areas of responsibility. Dividing work along these seams minimizes cross-team coupling, maintains responsiveness, and enables rapid evolution of AI features without compromising platform stability.

The Platform and Upstream Synchronization team oversees Electron's main and renderer processes, IPC contracts, packaging, signing, and regular VS Code rebases. Their coordination ensures that architectural updates or security patches do not disrupt other layers. The AI Runtime and Provider Integrations team manages adapters for both cloud and local models, implements streaming pipelines, and maintains capability detection to support the project's "any LLM, anywhere" privacy-first philosophy. They are responsible for adding new protocols, such as MCP, and shielding other developers from provider-specific changes.

The Workbench and User Experience group focuses on maintaining the front-end renderer, including chat panels, settings, and diff views, while ensuring accessibility and consistent performance. In contrast, the AI Product Verticals team delivers user-visible capabilities like Quick Edit, Chat, Agent, Gather, and Checkpoints, treating the runtime as a black box while iterating rapidly on new features. The Language Platform and Extensions team maintains LSP clients and the Extension Host, preserving responsiveness and fault isolation during heavy AI operations.

Cross-cutting groups include Privacy and Security, which enforces explicit user approvals, hardens IPC boundaries, and reviews provider integrations to uphold the "no middleman" design, and Release Engineering and Developer Experience, responsible for reproducible builds, onboarding flows, and cross-platform compatibility. Finally, Quality Assurance and Performance maintains scenario testing, regression monitoring, and responsiveness benchmarks to guarantee smooth concurrent operations.

Collaboration follows practical contracts: all privileged operations pass through typed IPC interfaces; only the Apply Pipeline can modify workspace files; new tools require approval reviews; and upstream sync cycles are announced to keep teams aligned. This division of responsibilities ensures modular growth, stability, and transparent accountability as Void evolves.Conclusion

The approach behind the Void IDE is to extend an existing software ecosystem with AI capabilities while preserving performance and security. Void is built on top of VS Code's Electron-based multi-process architecture, and introduces new subsystems: An AI Agent Layer, Model/Provider Layer, and Apply/Checkpoint Pipeline. Each of these new subsystems is designed for isolation and accounting for future evolutions. From an architectural perspective,

Void combines elements of layered, service-oriented, and event-driven styles. The overall structure considers evolvability, so it is designed to allow AI features (inline edits, checkpoints, resonating sliders, etc.) to evolve independently of the core editor. This approach yields a flexible and testable environment that scales effectively across platforms.

Despite the gaps in documentation that needed to be filled by reviewing parent architectures and the challenges of tracing its evolution across versions, our research demonstrates that Void's design fundamentally relies on a modular and future-proof conceptual architecture. It establishes principles such as concurrency control, separation of concerns, and modular interfaces, all of which allow the system to be applied to real-world systems at scale. All in all, our research allowed us to understand how AI can extend an existing software with a conceptual architecture that ensures its stability and functionality.

Section 1.09 Lessons Learned

For this assignment, our primary objective was to analyze the conceptual architecture of Void, and in doing so, understand the fundamental design decisions that led to a large-scale, AI-integrated software being flexible, maintainable, and high-performing over time. Delving into the repositories allowed us to gain a deeper understanding of how an open-source project like Void can extend an existing system (Visual Studio Code) through the implementation of additional architectural layers, while maintaining user control and stability.

One of the main obstacles that our group faced was learning to navigate between the multiple layers of documentation. The Void repository provides a helpful overview of its functionality and the baseline of its extension through its codebase guide and changelog. Still, the project inherits much of its architectural structure from Visual Studio Code and Election. As a result, the group had to cross-reference these frameworks to determine how processes like the renderer and extension host interact with the added components introduced by Void. This added a layer of confusion, resulting in the group having to be much more careful when trying to discern which responsibilities belong to Void itself versus its parent frameworks.

Over time, however, as we continued to examine the Void repo, VS Code’s source organization, and Electron’s process model together, we began to better understand how the different repositories worked in tandem and became much more efficient and effective at comprehending the architectural structure as a whole. Connecting these layers proved to be a valuable lesson in open-source projects. It taught us the significance of tracing dependencies and understanding how existing architectures evolve as derivative systems extend them. This entire process also helped us develop a more systematic approach to architectural analysis, starting from high-level diagrams and progressively narrowing down into the minutiae of each process and data flow.

Aside from understanding architecture, this assignment also served as a lesson in communication, team coordination, and clear documentation practices. Our group initially sectioned off aspects for individual research and writing. After noticing that many of these sections interwove and that our edits, at times, were formatted entirely differently from one another with varying citation strategies, for example, we had to rework our approach. Instead, we

decided to meet, established a consistent layout, and actively collaborated so that uncertainty could be alleviated immediately, rather than postponed and dealt with later.

Overall, not only did the assignment prove to be a valuable lesson in proper collaboration and communication practices, but our work on researching Void allowed us to bridge the gap between the higher-level architectural concepts learned in class and the realities of an active, open-source repository built on multiple interacting frameworks

Section 1.10 Data Dictionary

- **Workbench (Renderer):** UI surface that captures prompts, shows chat, diffs, settings, and orchestrates user approvals.
- **AI Agent Layer:** Exposes Chat, Quick/Inline Edit, Gather (read-only), and Agent (tool-using) capabilities; emits structured edit intents.
- **Model/Provider Layer (Main):** Adapters for cloud APIs and local runtimes; detects capabilities (FIM, tool-calling, "reasoning"), streams tokens, enforces budgets/timeouts.
- **Apply/Checkpoint Pipeline (Main):** Normalizes model output into deterministic blocks, renders **visual diffs**, applies approved changes, records **Checkpoints** for rollback.
- **Extension Host (Ext Host):** Isolated process that runs VS Code extensions.
- **Language Server Protocol (LSP):** Out-of-process language servers speaking JSON-RPC for diagnostics/completions.
- **EditIntent:** Canonical, structured proposal for a change (what/where/why).
- **DiffBlock:** Deterministic search/replace unit produced by the pipeline and shown in diff review.
- **Checkpoint / Checkpoint Navigator:** Versioned snapshots of accepted edits / UI to browse and restore them.
- **Agent Mode / Gather Mode:** Tool-using, read-write operations vs read-only navigation/search over the workspace.
- **Fast Apply:** Optimized path that turns validated model output directly into DiffBlocks.
- **Capability Detection:** Runtime feature discovery for a selected model (e.g., FIM, tool-calling, reasoning slider).
- **Background Terminal:** Long-lived terminal sessions usable by Agent under explicit approval.
- **MCP (Model Context Protocol):** Integration point for approved external tools/services accessed by Agent with user gating.

- **Approval Gate:** Explicit user confirmation step before any workspace mutation or tool execution.
- **IPC Channels:** Electron message paths between Renderer and Main that carry prompts, streams, and apply/rollback commands.

Section 1.11 Naming Conventions

- **Processes & subsystems:** Main, Renderer, ExtHost, LSP-<lang>, AgentLayer, ProviderLayer, ApplyPipeline, Workbench.
- **Artifacts (types/classes):** PascalCase → EditIntent, DiffBlock, Checkpoint, CapabilityProfile.
- **Variables & functions:** camelCase → reasoningLevel, buildDiffBlocks(), applyApprovedChanges().
- **Constants & flags:** UPPER_SNAKE_CASE → MAX_CONTEXT_TOKENS, ENABLE_FAST_APPLY.
- **Files & folders (code):** kebab-case → apply-pipeline/, agent-layer/, checkpoint-service.ts.
- **IPC channels:** dot-scoped, verb-last → void.agent.request, void.provider.stream, void.apply.commit, void.checkpoint.restore.
- **Provider IDs:** lower-kebab, vendor-first → anthropic-claude-4, openai-gpt-4o, google-gemini-2-5.
- **Runtime targets:** prefix for locality → cloud://openai, local://ollama, ssh://host.
- **Checkpoints:** time-sortable slug → cp-YYYYMMDD-HHMM-<file|feature>, e.g., cp-20251003-2142-refactor-parser.
- **Diagram labels (abbreviations):** UI/WB (Workbench), Agent, Provider, Apply, ExtHost, LSP, MCP, IPC.
- **Commit prefixes (optional for logs):** feat:, fix:, perf:, refactor:, docs:, aligned to changes in Agent/Apply/Provider/Workbench.

Section 1.12 References

Void. *Void Editor – Build with AI.* Void Software, 2025, https://voideditor.com

Void. Void Editor – Build with AI. Void Software, 2025. https://www.voideditor.com.

“Changelog.” Void Editor, Void Software, 2025. https://www.voideditor.com/changelog.

“Void IDE Architecture Overview.” Zread, Accessed 9 Oct. 2025. https://www.zread.ai/voideditor/void/9-architecture-overview.

"Process Model." Electron Documentation, Electron, Accessed 9 Oct. 2025. https://www.electronjs.org/docs/latest/tutorial/process-model.

"Inter-Process Communication." Electron Documentation, Electron, Accessed 9 Oct. 2025. https://www.electronjs.org/docs/latest/tutorial/ipc.

"What Is SOA? Service-Oriented Architecture Explained." Amazon Web Services, Amazon, Accessed 9 Oct. 2025. https://www.aws.amazon.com/what-is/service-oriented-architecture/.

"Singleton Method Design Pattern." GeeksforGeeks, 26 Sept. 2025. https://www.geeksforgeeks.org/system-design/singleton-design-pattern/.

"React Component Based Architecture." GeeksforGeeks, 23 July 2025. https://www.geeksforgeeks.org/reactjs/react-component-based-architecture.

"Extension Host." Visual Studio Code Extension API Documentation, Microsoft, Accessed 9 Oct. 2025. https://www.code.visualstudio.com/api/advanced-topics/extension-host.

"Source Code Organization." Visual Studio Code Wiki, GitHub, edited 2 Oct. 2025. https://www.github.com/microsoft/vscode/wiki/Source-Code-Organization.

"Language Server Protocol." Microsoft Open Source, Microsoft, 2025. https://www.microsoft.github.io/language-server-protocol/.

"Chat Participants and Agents." Visual Studio Code Extension Guide, Microsoft, Accessed 9 Oct. 2025. https://www.code.visualstudio.com/api/extension-guides/chat.

"VOID_CODEBASE_GUIDE.md." Void Editor Repository, GitHub, Accessed 9 Oct. 2025. https://www.github.com/voideditor/void/blob/main/VOID_CODEBASE_GUIDE.md.

Couriol, Bruno. "The Void IDE, Open-Source Alternative to Cursor, Released in Beta." InfoQ, 21 June 2025. https://www.infoq.com/news/2025/06/void-ide-beta-release/.

Rudra, Sourav. "Void Editor Is Shaping Up Well: Is It Ready to Take on Cursor and Copilot?" It's FOSS News, 25 June 2025. https://www.news.itsfoss.com/void-editor/.

Kumar, Aditya. "Void IDE: The Comprehensive Guide to the Open-Source Cursor Alternative." Medium, 24 Mar. 2025. https://www.medium.com/@adityakumar2001/void-ide-the-comprehensive-guide-to-the-open-source-cursor-alternative-2a6b17cae235.

Kumar, Ayush. "Void + Ollama + LLMs: How I Turned My Code Editor into a Full-Blown AI Workbench." DEV Community, 26 May 2025. https://www.dev.to/nodeshiftcloud/void-ollama-llms-how-i-turned-my-code-editor-into-a-full-blown-ai-workbench-eop.

Section 1.13 AI Reflection

Our team selected OpenAI GPT-5 as our AI team member. This version was chosen for its advanced language reasoning, synthesis capabilities, and accessibility, which aligned closely with the technical and analytical requirements of our project. Our process for selecting GPT-5

involved evaluating the specific needs of our assignment, which focused on analyzing and explaining Void IDE's architecture in detail. We required an AI capable of providing structured technical explanations, accurate summaries, and clear, well-organized academic writing.

Claude is known for its thoughtful reasoning and safety-oriented responses, making it strong at summarization and conversational clarity, but its output style sometimes favors cautious language that can be less concise or direct for technical writing. Gemini excels at creative problem-solving and integrating multi-step reasoning, but it is less widely familiar and may produce outputs in formats that require additional adaptation for structured academic reports.

DeepSeek has gained recognition for its accuracy in data interpretation and domain-specific analysis, yet its user interface and accessibility may pose barriers for students less familiar with its ecosystem. Image generators such as Midjourney were also unsuitable for our needs, as they are incapable of producing precise technical diagrams or structured text explanations, both of which were essential for applying the specific knowledge and concepts we learned in class.

GPT-5, by contrast, offered a balance of technical reasoning, structured output, and ease of access, with a familiar interface that our team could interact with efficiently. Its combination of clear explanations, reliability, and accessibility made it the ideal "hire" for producing a detailed and academically rigorous report.

The AI's responsibilities in our project included gathering source material, summarizing complex documentation, formatting initial drafts, analyzing our report against the marking rubric, and refining grammar and style. We primarily used GPT-5 as a research and organizational assistant during the early stages of the project. It helped us collect and condense technical materials, particularly documentation about Electron's process model, Visual Studio Code's internal architecture, and service-oriented design patterns into concise and accessible summaries.

These summaries allowed our team to quickly understand the foundational technologies behind Void, the open source IDE we were analyzing, and to start designing our conceptual architecture more efficiently. As part of this collaboration, we explicitly asked GPT-5 to generate a concise Data Dictionary and Naming Conventions section to standardize terminology across the report covering subsystem names, core artifacts, and common abbreviations. Beyond research support, GPT-5 played a key role in ensuring consistency across sections written by different team members. It helped standardize paragraph structure, enforce uniform academic formatting, and harmonize tone and word choice. This was especially helpful because group writing often results in noticeable differences in phrasing and technical depth. GPT-5 assisted in merging our individual contributions into a cohesive, professional document that flowed smoothly from one section to the next.

We also used it as a sort of "virtual teaching assistant." After drafting our report, we provided GPT-5 with the official CISC 322 assignment rubric and asked it to identify any missing or weak points. It pointed out several areas where our report lacked clarity or needed more explicit references to diagrams and naming conventions. This feedback saved us time and helped us prioritize our revisions. Prompt creation was a team effort rather than the work of a single

“prompt engineer.” Each member contributed ideas on phrasing and refining queries based on their understanding of the material.

Early prompts were broad, such as “summarize the Electron architecture” or “explain the VS Code extension host.” However, we quickly saw that more detailed prompts led to better results. So, we started adding contextual details about the course, the assignment goals, and the desired level of technical accuracy. This allowed GPT-5 to generate explanations that were both structured and detailed, helping us clarify complex topics such as process separation, service-oriented architecture, and inter-process communication.

By comparing GPT-5’s responses with what we learned in class, we were able to cross-reference concepts, verify our understanding, and draw connections between theoretical principles and their practical implementation in Void IDE. GPT-5 acted as a guide to deepen our comprehension; all critical analysis, synthesis, and the final report writing were conducted by our team. This iterative process not only helped us understand technical material more thoroughly but also reinforced our ability to translate classroom knowledge into applied analysis.

Validation and quality control were essential throughout every stage of the process. We treated GPT-5 as a contributor whose work always required human oversight. All AI-generated content was cross-checked against primary documentation sources, such as the official Electron or VS Code repositories and technical documentation sites. When GPT provided summaries or explanations, we manually confirmed their accuracy before including them in our report. After integrating AI-generated content, another team member would review each section for coherence, tone, and logical flow, ensuring that the writing reflected our group’s voice and met academic expectations.

When GPT-5 was used as a rubric checker, its feedback served as a helpful checklist rather than an authoritative evaluation. We continuously verified its claims manually, which prevented us from accepting errors or overgeneralizations. For grammar and style improvements, we used AI to suggest revisions, but the final editing decisions were made collectively to preserve meaning and technical correctness. Quantitatively, GPT-5 contributed 20 percent of the final deliverable. Although that number might seem small, its impact was significant in shaping our workflow and enhancing the clarity of our report. The AI provided a solid foundation for research and organization, but it never replaced our critical judgment or creativity. We found that using AI effectively required as much effort as working without it, just in different ways. The time saved in information gathering was often offset by the time spent verifying, editing, and refining our work.