Conceptual Architecture of Void

October 10, 2025

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

Abstract

Void is an open-source, AI-assisted Integrated Development Environment (IDE) built as a fork of Visual Studio Code. It enables developers to interact with local or remote Large Language Models (LLMs) to accelerate code generation, modification, and understanding through natural language prompts and inline editing. Unlike closed-source tools such as Cursor AI or GitHub Copilot, Void removes dependency on proprietary servers, allowing direct model connections for greater transparency and data control.

This report presents the conceptual architecture of Void, focusing on its hybrid design that combines layered, client-server, implicit invocation, and modular architectural styles. The system is decomposed into three major layers: the Non AI Foundation, AI Integration, and Bridge Component Layer, each contributing to a balance between extensibility, privacy, and performance. Two representative use cases, Quick Edit and Agent Mode, demonstrate how these components interact to achieve low latency, privacy-preserving, and context-aware code editing.

Through its modular architecture and open-source philosophy, Void highlights how AI-assisted development can be achieved without compromising transparency or developer control. The report concludes by addressing the trade-offs and challenges of maintaining compatibility with Visual Studio Code while supporting a diverse and rapidly evolving ecosystem of LLMs.

Introduction and Overview

The integration of artificial intelligence into software development has transformed how developers write, debug, and refactor code. Commercial tools such as GitHub Copilot and Cursor AI provide real-time suggestions, automated documentation, and intelligent code generation directly within the editor. However, these solutions depend heavily on closed-source infrastructures that process developer code through third-party cloud servers. This dependency raises significant concerns around privacy, data ownership, and long-term accessibility, particularly for organizations handling proprietary or sensitive code.

Void was created to address these challenges by offering a transparent, flexible, and privacy-preserving alternative. Built as a fork of Visual Studio Code, Void integrates LLMs into the development workflow without relying on a private backend. Users can connect to local models such as Llama or DeepSeek or to remote providers like Gemini and Claude, ensuring

complete control over how and where data is processed. This open approach combines the familiar usability of Visual Studio Code with the intelligence of AI-driven development, making Void both developer-friendly and conscious about privacy.

System Overview
Conceptually, Void operates as a multi-layered system. The core layers are:

1. **Non-AI Foundation Layer** provides the base editor functionality inherited from Visual Studio Code, including User Interface Layer, Tools Layer and Data Layer.
2. **AI Integration Layer** (or **Model Connector Layer**) handles the orchestration of LLM connections, prompt construction, and streaming responses from either local or cloud models.
3. **Bridge Component Layer** connects user input and interface components to the AI services, coordinating interactions such as chat, quick edit, and agent mode.

These layers communicate through a hybrid of architectural styles, including layered, client-server, and implicit invocation models, to ensure modularity, low latency, and system stability. The design supports incremental evolution, allowing new models and tools to be integrated without altering existing components.

System Scope
The scope of Void focuses on extending Visual Studio Code with AI capabilities for code editing, refactoring, and automation. Its main concerns include the client-side interaction with LLMs, user interface design for AI features, and configuration management for privacy-preserving connections. Activities such as training LLMs, hosting cloud servers, or redesigning the Visual Studio Code core are outside Void's scope.

In Scope:
- AI-assisted code completion, inline editing, and chat
- Direct connectivity to local and cloud LLMs
- User interface for AI control and feedback

Out of Scope:
- Training or developing new AI models
- Redesigning or replacing the Visual Studio Code base
- Server-side data analytics or telemetry

Stakeholders
Void's architecture serves a variety of stakeholders, each benefiting differently from its open-source and modular design.

Developers: Access to fast, reliable AI-assisted coding without compromising privacy.
Open Source Contributors: Ability to extend or customize functionality via modular service

contracts.
Organizations and Academia: Control over data handling and model integration for compliance and confidentiality.
LLMProviders: Opportunity for broader adoption through standardized model connections.
Glass Devtools Inc. (Maintainers): Sustainable growth of a community-driven, privacy-first IDE ecosystem.

Key System Goals
Privacy: Void avoids proprietary backends and supports both local and direct model connections. All data flow remains user-controlled, with open-source transparency enabling independent verification of data handling.

Flexibility: The architecture supports multiple LLM providers and can easily integrate new models through dependency injection interfaces. Developers can switch between local and remote modes with minimal reconfiguration.

Developer Productivity: Intelligent autocompletion, Quick Edit, and conversational code manipulation streamline coding tasks without requiring users to leave the editor. Features such as checkpoints, lint integration, and streaming updates ensure smooth, low-latency workflows.

Together, these goals form the foundation of Void's conceptual architecture, driving design choices that emphasize extensibility, privacy, and a seamless developer experience.

Architecture

High-Level Description

The conceptual architecture of Void IDE outlines the abstract structure and major design principles. It defines the system's organization and operation. At this level, we focus primarily on the important structural details of Void without delving into implementation-specific details. We concern ourselves with several key aspects of Void's conceptual architecture - including its architectural styles, subsystem decomposition, concurrency model, and primary use cases.

Architectural Style

Void's architecture is best described as a hybrid architecture. It combines a layered system structure, a client–server interaction model, an implicit-invocation runtime, and modular/DI extensibility on top of the VS Code platform.

1. System-Level - Layered Style
On top of the VS Code platform, Void follows a layered organization:

User Interface Layer – surface for developer interaction and AI output.
Bridge Component Layer – mediates between editor events and AI services.
Model Connector Layer – abstracts connections to external LLMs via MCP; enables adding or swapping models without core rewrites.
Tools Layer – plug-in style services (e.g., analyzers, linters, diagram generation) integrated through contracts.
Data Layer – privacy-preserving local storage and configuration, with encrypted channels for remote access.

The layers localize change makes AI features evolve mostly in the workbench scope without destabilizing the editor substrate or cross-cutting platform services.

2. Runtime Interaction: Client–Server Style (Renderer ↔ Main ↔ LLM)
Requests are initiated in the renderer (UI), flow through an IPC bridge into the main process, and out to providers in either cloud or local LLMs. The renderer does not make direct provider invocations; rather, it receives streamed tokens, final result sets and errors as request ID correlated events.

Calling LLM from main circumvents CSP/runtime limitations and centralizes capability checks, metrics, cancellation and isolates privileged dependencies. It also cleanly separates provider substitution as a backend concern, enhancing evolvability.

3. Runtime Style: Implicit-Invocation Style(Events, Contributions, Streaming Callbacks)
Void introduces an implicit-invocation runtime on the workbench (such as components subscribe to framework-level events, like editor lifecycle, configuration changes, context keys), and contributes handlers via extension/contribution points. AI flows also depend on asynchronous reverse calls – intermediate tokens, deltas and errors are pushed back to the UI rather than letting the UI pull them synchronously.

Coupling is lowered and response is enhanced through limiting or downstream implicit invocation. The UI is kept responsive when LLM operations take a long time to stream updates. The addition of new behaviours can be made by registering an additional subscriber, without having to modify existing callers, promoting incremental evolution and safer parallel work.

4. Structural Style: Modular / Dependency-Injection Extensibility
Void emphasizes modularity and service contracts based on DI. Core functionalities, such as settings, model routing, LLM messaging, and editing of all the model attributes, are hidden behind interfaces resolved at runtime. Providers like OpenAI, Anthropic or local engines can be integrated via adapter implementations of a common contract line. These will be discovered and picked by configuration and capability checks.

DI makes it possible to switch out implementations (e.g., change provider or model without changing the UI). It tests things apart from one another by using mock services, and divides work into small parts that can be delivered and destroyed when done. It confines variability to well-defined modules, hence improving maintainability and parallel development.

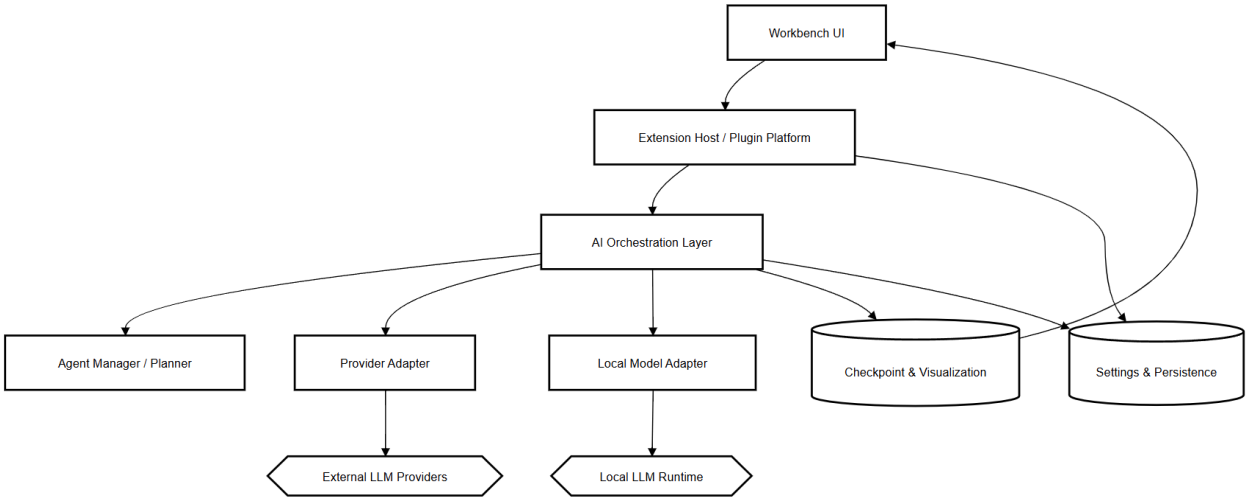

Figure 1, Dependency diagram

Concurrency

Concurrency is a foundational capability in Void that provides smooth interaction and safe application changes. On the editor side, it must remain responsive to user input as the editor operates in real-time, while on the AI side, it must account for unpredictable network latency. If the two were directly coupled, the UI would easily become blocked. Therefore, Void sorts concurrency out as a pipeline, achieving cancelability in the foreground, streaming in the background, and cross-process observability. This ensures that keystrokes and cursor movements are always treated as higher priority than anything the AI is doing—the GUI never waits for the model to finish responding. This mechanism is cooperative and preemptable. Every AI request is initiated with an epoch identifier and a cancel token. Whenever the user types or moves the cursor, the renderer can cancel or supersede the in-flight activity (e.g., autocomplete) and initiate a new epoch via IPC. The main process treats cancellation as fully preempting: if the provider supports termination, it sends a stop signal. Otherwise, any remaining data blocks from the old epoch are discarded to avoid corrupting the editor state. This design eliminates UI blocking and avoids priority inversion.

Furthermore, to preserve smoothness under variable network latency, the pipeline applies back-pressure and coalescing. Tokens are streamed, throttled and batched into renderable deltas, which update the UI in a predictable cadence. Changes are staged as diff zones instead of overwriting the entire document. This enables incremental refresh and safe undo.

Void also has a clear distinction between responsiveness and completeness in the foreground vs. the background. The actions in the foreground are always performed in sync mode on the editing thread. Background operations – for example, model calls or tool executions are usually completed asynchronously, though fast local operations like find-replace may be made synchronously if deemed safe. If a cancellation happens, it will trigger more substantial rewrites ("slow applications"), which happen out of the UI thread and are shown to users as previews that need to be accepted. In this process, observability data such as timestamps and request durations, termination reasons, and provider latency are returned via IPC in order to help the renderer determine whether to cancel, wait for more chunks, or roll back.

Subsystems

The subsystems of Void can be broken down into three layers, each with significant subsystems and interacting components.

1. Non-AI Foundation Layer

The Non-AI Foundation is the underlying conceptual layer of Void, built on top of VS Code, providing the editor shell, workspace, file services, extensions, and user interface into which AI features are implemented. It contains several subsystems and components which allow AI features to operate reliably without sacrificing stability, responsiveness, or extension compatibility.

Editor Core

The Editor Core can be thought of as a customized foundation built on Visual Studio Code, which provides the Monaco editor surfaces, extension system and runtime, interactive workbench framework, and service architecture while managing workspaces, files, and terminals. It coordinates the editor's lifecycle and state, directs user input to providers (inline completion, code actions), and reliably allows for edits without freezing the UI. Essentially, Void inherits VS Code's functionality – opening and saving files, tabs and split-screen editors, terminals, and support for a variety of languages – to preserve stability and familiarity with existing workflows.

Editor Services

Editor Services work behind the scenes to promote smooth code editing (undo, redo, cursor, etc.) and allow AI features to safely make code edits. They give access to the in-memory text models, handle apply/undo operations, manage selections, and offer the registration points that AI completion and edit features plug into. These services depend on the Editor Core for lifecycle, UI surfaces, and presenting contracts to the AI services. The Editor Services assess the context when a suggestion is needed to ensure the correctness of edits, safe concurrency during live updates, and stable interfaces that allow AI features to evolve without requiring changes to the Editor Core. A core component of this subsystem is the **VoidModelService,** which pins and

unpins models, provides stable handles during AI actions, and cleans up on acceptance or cancellation of edit suggestions. It uses the standard model APIs from Editor Services, is invoked by **AutocompleteService** and **EditCodeService** during longer interactions, and is designed to be extended without changing how it's called. These components work together to support the requirement that long-running operations can hold references to the underlying text models to prevent accidental edits.

UI components

The UI components are React panels that let developers view and control AI activity directly inside the editor. This includes a chat panel for conversations, command bars for quick actions, and a live diff view that allows accept/reject of suggestions. These components subscribe to the **LLMMessageService** for updates from the AI (so responses appear in real-time), coordinate with the Editor Services, specifically EditCodeService, to render and safely apply suggestions, and, when an operation requires changing files or running commands, display actions to support transparency under user control. This subsystem smoothly integrates the AI into the code editor to provide incremental results without freezing the UI, show proposed edits with accept/reject options, and ensure no commits until the developer explicitly approves them.

2. AI Integration Layer

The AI integration layer is everything in Void that depends on model inference. This includes connecting to local or cloud LLMs, constructing and streaming prompts and responses, editing suggestions and completions, managing chatbot conversations, and executing file/terminal tools under user control. This layer is intentionally separated from the editor to keep the IDE stable while AI performs tasks and the following are the main functions it contributes:

Model Connector Layer

The Model Connector Layer is a gateway that connects Void to both local AI runtimes like Ollama and cloud APIs like Claude or Gemini, handling credentials and endpoints, advertising each model's capabilities, and returning cancellable token streams to its callers for low-latency interaction. It selects the active model and normalizes requests across vendors, establishes streaming transports, exposes capabilities to higher services, and surfaces errors with options to retry and cancel, so features remain responsive even under failure. It consumes provider SDKs or HTTP endpoints, is called by the **LLMMessageService** component (the central message service that directs prompts and streaming), and relies on the non-AI editor side for configuration, storage, and interprocess communication, keeping the editor kernel separated from the model. Its goals are privacy and control via direct connections, interoperability, low-latency streaming to the UI, and graceful failover. It must also support both local and cloud models, honour provider differences, maintain cancellable streams, and publish capabilities. It emphasizes evolvability with its ability to add new providers without changing callers and extend capabilities for new features per model.

AI Interaction Orchestration

The AI interaction is centred around the main component **LLMMessageService**, which builds prompts from inputs, sends them to the selected model through the Model Connector Layer, manages streaming, delivery, and cancellation, and handles tool calls. It relies on and calls the Model Connector Layer, reads each model's capabilities, and publishes streams to UI components and feature services, such as EditCodeService and AutocompleteService, for inline editing and suggestions.

Editing and Completion Services

The Editing and Completion Services subsystem turns model output into reviewable changes and suggestions. Its main components are **EditCodeService** and **AutocompleteService**. EditCodeService is responsible for converting AI responses into diffs that stream into the UI for the user to read and choose whether to accept or reject, with the ability to handle each case gracefully. It does this by extracting a code slice, crafting an edit, then applying only the accepted changes using the editor's underlying apply services and rolling back with the undo services if needed. Its dependencies include LLMMessageService for prompt handling, the Editor Services for IDE edits and model handles, and UI components to render diffs and gather approvals/denials. It needs to ensure human control with minimal friction, accuracy, and no silent changes. AutocompleteService complements these tasks by delivering inline completions through the editor's base provider. It extracts context from around the cursor, chooses a model based on capabilities, requests a completion and surfaces the result in a non-intrusive suggestion that can be accepted with Tab or ignored. The Editor Core invokes it to request suggestions and render them inline, it reads cursor position, selection, and surrounding text via the Editor Services, and it calls LLMMessageService to route the request to the selected model through the Model Connector Layer. It is meant to be fast, accurate, and provide unintrusive completions.

Conversational Management

The Conversational Management subsystem is centred around the **ChatThreadService** component, which manages chat threads, attaches files and code snippets for context, and enforces mode switching (Normal, Gather and Agent). When a message is sent, it gathers the project context into a request, forwards it to the AI message layer, and then displays the reply in the editor's chat UI. It relies on the non-AI foundation for storage of chat threads and attachments.

Tooling/Agent Execution

The Tooling/Agent Execution subsystem is implemented by the **ToolsService** component. This component safely exposes file and terminal actions to the agent with strict guardrails, explicit confirmations, and an audit trail, while also integrating tools using the Model Context Protocol (MCP). It can read, write, create, delete, and search files, run commands, prompt for approval for

state-modifying tools, log every action, and support MCP tools. It is invoked by the LLMMessageService when models issue tool calls, coordinates approval prompts with the UI, and executes the requested operation through Editor Core APIs or MCP servers. Its goals are to deliver agent abilities without sacrificing safety or privacy. New tools can be added incrementally and discovered dynamically, ensuring evolvability.

3. Bridge Component Layer

The Bridge Component Layer supports the native feeling of the AI in the IDE. It picks the most relevant bits of project context for each action, staying within the model's limits, turns the context plus user intent into prompts matched to the selected model, then validates the model's output and displays it as inline suggestions or diffs. It keeps outputs relevant, formats predictable, and edits all while staying separate from the editor.

Context Extraction

Context Extraction assembles a compact, high‑signal context packet for the current AI action by selecting cursor‑adjacent code, directly related symbols/imports, and a few semantically related files within the active model's capabilities. It depends on Editor Core and Editor Services for open buffers, cursor/selection state, file tree, and symbol indexes and on UI Components for user‑attached files/snippets and mode constraints. Its goals are relevance‑first, budget‑aware, auditable inclusion, with pluggable strategies (locality‑only, semantic, hybrid) validated via golden‑context fixtures, large‑repo stress, and ablation tests.

Prompt Engineering System

The Prompt Engineering System converts user intent plus the context packet into prompts using consistent roles, templates, and stop conditions across chat, edit, and completion. It depends on Context Extraction for the context packet, and on the LLMMessageService for streaming request execution and cancellation. Its goals are predictable outputs, minimal response time, as well as clear, consistent prompts.

Response Integration

Response Integration consumes streamed model output, validates structure and intent, and turns it into safe editor suggestions (inline or readable diffs) with explicit accept/reject and no silent writes. It depends on the LLMMessageService for token streams and errors, on Editor Core and Editor Services for apply/undo and stable text model handles, on UI Components (chat and diff panels) to render changes and capture approvals, and on ToolsService for routing any model‑initiated actions under confirmations and mode boundaries. Its goals are responsive streaming and deterministic apply semantics with conflict detection, rebase or rejection when buffers change, and full undo/auditability.

Control and Data flow

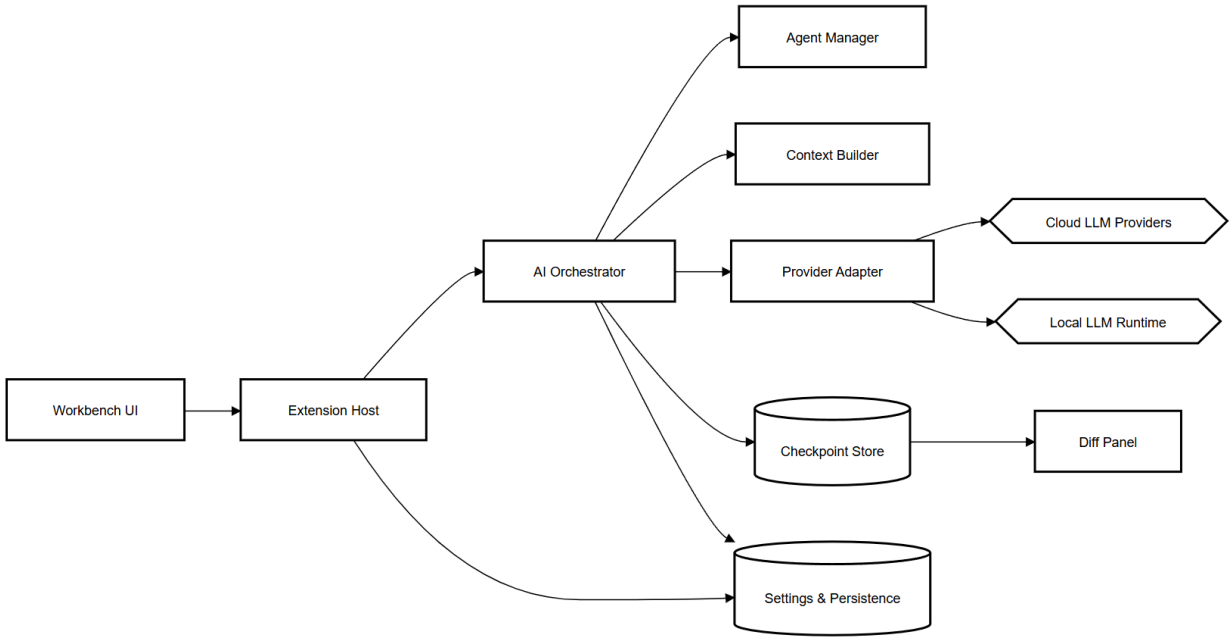

Figure 2, Data flow Diagram

In Void's architecture, control flow proceeds top-down, reflecting a layered and pluggable design. When users perform actions within the Workbench UI (such as triggering Quick Edit, sending chat prompts, or executing Agent commands) requests first reach the Extension Host, an intermediary layer between the interface and core logic. The Extension Host interprets commands, checks context, and then forwards the request to the AI Orchestration Layer. Within this layer, the AI Orchestrator manages several submodules, such as the Agent Manager and Context Builder. For simple editing tasks, the Orchestrator assembles relevant code snippets into a prompt. For multi-step tasks, it delegates planning and execution to the Agent Manager. Once the request is prepared, the Orchestrator selects an appropriate Provider Adapter (e.g., OpenAI, Gemini, DeepSeek, or the local Ollama), sends the request to the model via HTTP or WebSocket, and receives streaming data back—including text or modification suggestions. Orchestrator then aggregates this data and sends it back to the Extension Host for display in the interface. Users can preview, accept, or reject these changes. Upon acceptance, the Extension Host saves the current file to the Checkpoint Store and invokes the Diff and Visualization Panel to display pre- and post-modification differences, enabling rollback.

Data flow, similar to control flow, describes the information transmitted within a system. Between the user interface and the orchestration layer, data includes selected code regions, prompt text, and the models or patterns being used. The Context Builder generates a “Context Pack” containing code snippets, symbols, and reference information. The Orchestrator packages this data into a Prompt Envelope containing model parameters such as temperature, sampling constraints, and model name. After the model generates results, the Orchestrator sends a Patch Bundle back to the Extension Host. The Patch Bundle contains modified text, annotations, and

confidence scores. Checkpointing saves these snapshots and different information for subsequent display. API keys, routing configurations, and other settings are sourced from the Settings & Persistence Store, accessible to all modules.

User operations and model responses are asynchronous. Void supports concurrent processing: multiple chat or editing tasks can run simultaneously. Model outputs can be generated and displayed in real-time, while background index updates do not affect the interface. This event-driven design ensures fast system responsiveness without lag. The system uploads only the minimum necessary context information to protect privacy. Users can also run offline using a local model (Ollama). This approach guarantees both high efficiency and security.

Use Cases

Use Case 1 – Quick Edit

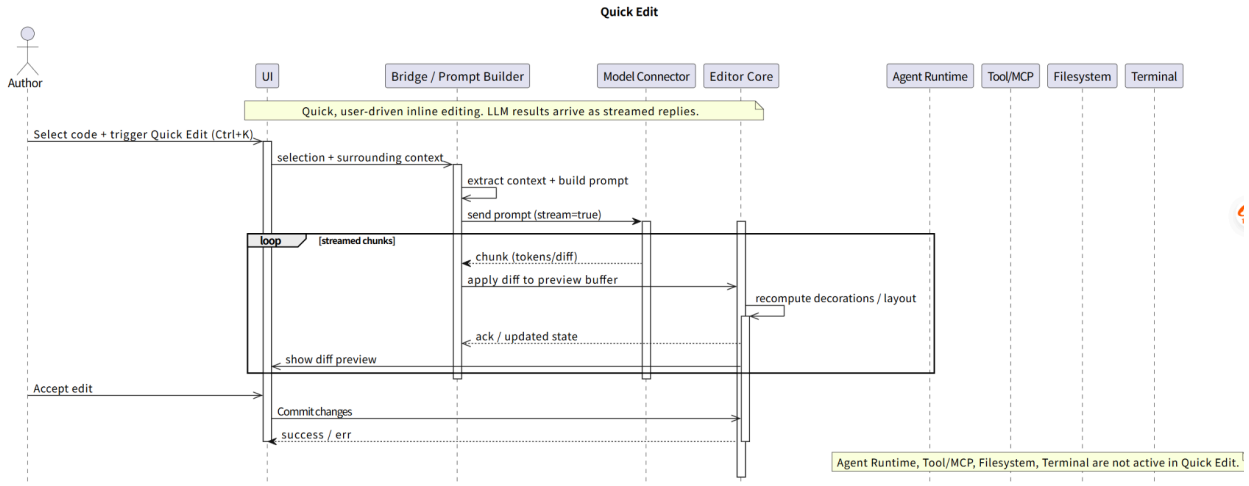

Figure 3, Quick Edit Sequence Diagram

The Quick Edit use case shows how Void's interactive pipeline links the major parts of the system to produce real-time, AI-assisted edits (see **Figure: Quick Edit Sequence Diagram**). After the author highlights a code block and invokes Quick Edit, the UI captures the intent and selected region, then passes both to the Bridge / Prompt Builder, which bridges the editor and the AI layer. The Bridge gathers nearby code for context, builds the prompt, and sends it through the Model Connector (LLMMessageService/EditCodeService) to the chosen model (either a cloud provider or a local Ollama instance) via the external interfaces. The model returns streamed chunks of output; these flow back asynchronously to the Bridge, which integrates the proposed diffs and calls the Editor Core to apply them to a preview buffer. The Editor Core then updates the UI with a live diff for review; if the author accepts, the UI asks the Editor Core to commit the changes. If accepted, the UI asks the Editor Core to commit the edits to the file.

This scenario demonstrates the main workflow (UI → Bridge → Model Connector → Editor Core) and shows how it meets the non-functional requirements (including low-latency, privacy-preserving editing) of Quick Edit by combining asynchronous model streams with synchronous, deterministic updates to the document.

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

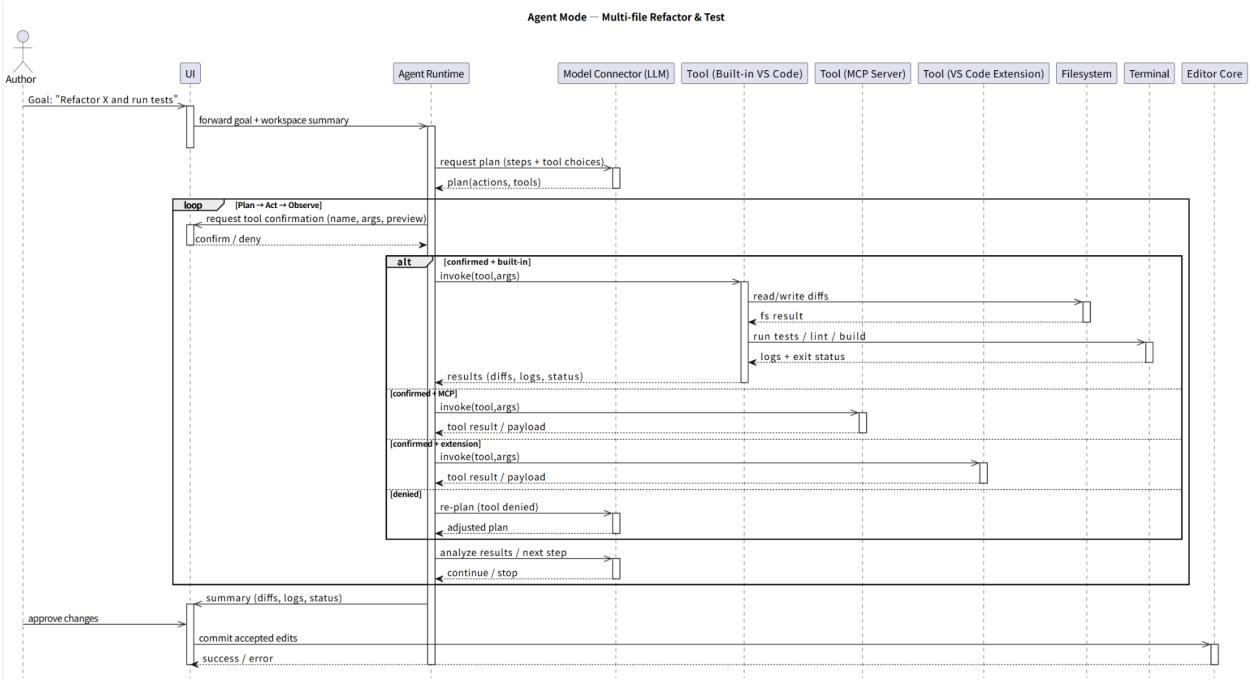

Figure 4, Agent Mode Sequence Diagram

The Agent Mode use case shows how Void's agentic pipeline links the major parts of the system to execute multi-file refactors and tests (see **Figure: Agent Mode Sequence Diagram**). After the author states a goal (e.g., *"Refactor modules and run tests"*), the UI captures the intent and workspace summary, then passes both to the Agent Runtime, which bridges the editor and the AI planning layer. The Agent queries the Model Connector (LLM/Planner) to produce a step-by-step plan, then requests tool confirmation from the UI. Once approved, the Agent invokes the chosen tool source via the external interfaces: built-in VS Code tools (search/edit file, run in terminal), or contributed tools from MCP servers or VS Code extensions. Built-in tools read/write diffs through the Filesystem and run builds/tests in the Terminal; MCP/extension tools return structured payloads (patches, analysis, results). Outputs stream back to the Agent, which iterates a plan → act → observe loop—analyzing logs and statuses with the Model Connector and re-planning as needed—while the Bridge maintains relevant context for subsequent steps. Finally, when the result is satisfactory, the Agent summarizes diffs and test outcomes in the UI, and after approval, the UI asks the Editor Core to commit the accepted edits.

This scenario demonstrates the main workflow (UI → Agent Runtime → Model Connector → Tools → Filesystem/Terminal → Editor Core → UI) and shows how it meets key NFRs for Agent Mode, including traceability, low-latency local execution, privacy-preserving model use, and reliability.

Data Dictionary

Agent Mode — it is a runtime mode which orchestrates planning and tool execution to make multi-file edits with human approval.
Gather Mode — it is a runtime mode which collects and summarizes the workspace context to prepare for later editing or planning steps.
MCP (Model Context Protocol) — it is a protocol which acts as a standard bridge for external tool servers to expose typed actions and data to the agent.
Checkpoint — it is a versioning artifact which represents a snapshot or patch set used for preview, rollback, and reproducible re-runs.
Quick Edit — it is an interactive function which performs inline, AI-assisted edits into a preview buffer for user review before commit.
Fast Apply — it is a performance optimization function which applies large or repetitive diffs efficiently with minimal recomputation.
Lint Integration — it is a quality gate which runs linters, formatters, and tests and reports diagnostics to the UI or agent; it can block commits on failure.
Model Connector — it is an adapter which provides a unified API for local and cloud LLMs, handling authentication, capability detection, routing, and streaming.
SDK – a set of tools for third-party developers to use in producing applications using a particular framework or platform

Naming Conventions

Components & Services

- **PascalCase** for major subsystems/services
- **Title Case** for modes/features in prose and headings

Files, Modules & Identifiers

- **File/module names:** kebab-case or lowerCamelCase
- **Tool/command IDs (UI/agent):** snake_case
- **Configuration keys & payload fields:** snake_case

Messages & API Surface

- **Operation names:** verb-first, concise, lowerCamelCase
- **Data objects (domain):** PascalCase

Diagram Conventions (Sequence & Components)

- **Lifelines:** single label at top only

- **Arrows:**
    - Synchronous call (solid line, filled triangle).
    - Asynchronous call (solid line, open arrowhead).
    - Reply message (dotted line, triangle) back to the original caller.
- **Activation bars:** show the time interval during which a lifeline is actively executing behaviour in a sequence diagram.
- **Frames:** loop for streaming; alt for branching

Abbreviations Used

- **UI** — User Interface (panels, command bars, diff views).
- **LLM** — Large Language Model.
- **MCP** — Model Context Protocol (tool/server integration).
- **DI** — Dependency Injection.
- **IPC** — Inter-Process Communication (renderer ↔ main).
- **SDK** — Software Development Kit.

Conclusions

In conclusion, Void's architecture combines the flexibility of open-source development with AI-assisted editing capabilities. As a fork of VS Code, it retains a familiar interface and robust extension ecosystem, and adds AI capabilities such as tab autocomplete, inline editing, an AI chat function and flexible LLM connections. Its major strengths lie in its modularity, extensibility and privacy. Each feature, from the Model Connector to the Bridge and Editor Core, functions as an independent module, enabling upgrades or replacements without affecting the entire system. The open-source model ensures transparency and developer control, addressing the privacy concerns commonly associated with closed-source tools. Furthermore, performance is enhanced through efficient streaming and prompt management, ensuring low latency and responsive interaction.

However, there are some limitations as well. Firstly, reliance on a VS Code fork introduces the long-term risk of upstream drift, whereby changes to VS Code could necessitate significant maintenance to ensure compatibility. Additionally, integrating multiple models and tools increases system complexity and may result in inconsistencies across LLM behaviours. While Void's privacy-first approach offers clear benefits, it also shifts responsibility for configuration and maintenance to end users, which could impact usability.

Despite these challenges, Void's design demonstrates strong evolvability. The Model Connector enables the seamless adoption of new or specialized LLMs, ensuring future compatibility with emerging AI models. The MCP protocol supports external tool contributions without requiring rewrites of the core, and modular services such as the Bridge and Tools layers make the

architecture adaptable to new workflows and integrations. In essence, Void's architecture strikes a balance between innovation and stability; it remains modular and privacy-preserving, and hence it is ready to evolve with emerging AI tooling.

Lessons Learnt

We learned that a successful software architecture must be modular, adaptable, and open to future change. Void's design fulfills stakeholder needs by balancing innovation with stability, using open standards such as MCP to ensure flexibility and long-term evolution. Designing with change as a premise enables the system to grow alongside future AI advancements while maintaining reliability and trust. We also learned the practicality of a layered architecture system. In discovering how Void works as a fork built on top of the Visual Studio Code IDE, we saw the real-world usage of how the layers interact to swiftly integrate more software on top of pre-existing software.

References

1. Gerred. (n.d.). *System architecture diagram: Making interactions responsive while handling complex operations*. In *Building an agentic system*. GitHub Pages. https://gerred.github.io/building-an-agentic-system/system-architecture-diagram.html
2. Heinig, I. (2025, February 4). *AI orchestration: A beginner's guide for 2025*. Sendbird Blog. https://sendbird.com/blog/ai-orchestration
3. Kumar, A. (2025, March 24). *Void IDE: The comprehensive guide to the open-source Cursor alternative*. *Medium.* https://medium.com/@adityakumar2001/void-ide-the-comprehensive-guide-to-the-open-source-cursor-alternative-2a6b17cae235
4. Microsoft. (2025, April 7). *Agent mode: Available to all users and supports MCP*. *Visual Studio Code Blog.* https://code.visualstudio.com/blogs/2025/04/07/agentMode
5. Microsoft. (2025). *Extension API architecture*. *Visual Studio Code Documentation.* https://code.visualstudio.com/docs/extensionAPI/architecture
6. InfoQ. (2024, July 15). *Void AI: An open-source Cursor alternative*. *InfoQ News.* https://www.infoq.com/news/2024/07/void-ai-cursor-alternative/
7. KDnuggets. (2024, December 3). *Enter Void: Open-source AI coding IDE*. *KDnuggets.* https://www.kdnuggets.com/enter-void-open-source-ai-coding-ide
8. Zread. (n.d.). *Void core module structure*. *Zread AI Documentation.* https://zread.ai/voideditor/void/9-void-core-module-structure

AI Collaboration Report

AI Member Profile and Selection Process

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

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

Tasks Assigned to the AI

Use Case Descriptions and Refinement – GPT-5 structured, edited, and aligned the wording across all use-case scenarios.

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

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

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

Interaction Protocol and Prompting Strategy

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

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

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

Validation and Quality Control Procedures

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

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

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

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

Quantitative Contribution to Final Deliverable

GPT-5: 25% (drafting, editing, formatting, and consistency)

Mermaid AI: 5% (diagram generation)

Human team: 70% (concept development, validation, integration, analysis)

Reflection on Human-AI Team Dynamics

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