Introduction

In the past few years, the sudden and rapid improvements in LLM technology has heralded a boom of new software looking to make the professional lives of developers easier and streamlined. From plucky start ups to corporate titans, organisations like OpenAI and Anthropic are fighting for attention and customers, waging war on each other, and their userbases, by restricting their proprietary models and tools. Now that LLM chatbots like ChatGPT are so ubiquitous, the direction of progress has turned to integrating this technology directly into your workflow, through AI Agents, and being placed directly into the IDE.

Services like Co-Pilot and Claude seek to leverage the power of LLMs by giving them direct access to editing tools, allowing them to see the context of your code, and directly affect it. Given all these services are paid - usually an additional fee on top of accessing the model itself - conditions were ripe for an open-source alternative, enter VoidEditor. The software offers an alternative to the Cursor IDE, with full VSCode integration - allowing developers to seamlessly integrate the LLM of their choosing into their workflow, focused locally, and without a hefty subscription for the privilege.

Derivation Process

To understand VOID's conceptual architecture, our group began by exploring its publicly available documentation and GitHub. The primary sources for architectural analysis were the VOID GitHub repository, which contains the full source code, issue tracking, and changelogs, and the official VOID website, which details key features such as Agent Mode, Checkpoints, and model integration. From these, we constructed an understanding of how the system is structured conceptually, rather than at the implementation level.

VOID's documentation reveals a modular framework built around open-source IDE principles and VS Code's architectural inheritance. Since VOID is a fork of VS Code, it retains the multi-process model — separating the renderer (UI), extension host, and background processes for responsiveness and stability. On top of that base, VOID adds AI-specific components: the AI Bridge, Model Connector, Checkpoint Manager, and Security Sandbox, all communicating through an internal event bus.

To analyze how these components work together, we referenced VS Code's architecture overview and traced how VOID extends it through asynchronous communication and plugin-driven design. The GitHub changelogs also document progressive updates, such as the addition of Agent Mode and Gather Mode, which significantly affect the system's interaction model and data flow (VOID changelog).

This combination of open-source documentation, version history, and architecture diagrams provided the foundation for mapping VOID's conceptual structure. From this, the group derived a layered, event-driven conceptual architecture that explains how the editor's components interact to provide responsive, privacy-focused AI assistance.

Architectural Overview

VOID's conceptual architecture can be described as a hybrid of layered, plugin-based, and event-driven styles. Each style contributes specific strengths: the layered structure provides clarity and maintainability; the plugin system enables extensibility; and the event-driven bus allows asynchronous communication among modules.

At a high level, the system is divided into three conceptual layers:

1. Presentation Layer: The user interface and editor core handle input, display, and rendering.

2. Logic Layer: The AI Bridge, Model Connector, and Plugin Manager process events, generate prompts, and interpret results.

3. Data Layer: The Checkpoint Manager, Security Sandbox, and API Gateway manage persistent data and control external communication.

These layers are connected through an internal event bus, ensuring that data and control flow remain decoupled. This allows new features or model connectors to be added without restructuring existing code. The event-driven model also supports concurrency — multiple modules can perform independent tasks simultaneously (e.g., syntax checking, model querying, and rendering).

Architecturally, VOID's design emphasizes user control, data transparency, and modular evolution. Unlike commercial AI IDEs, it processes everything locally when possible, and model communication is explicitly managed through the AI Bridge. This balance of openness and structure makes it a representative modern example of event-driven IDE evolution.

Subcomponents

At its core the system we are looking at during this project is an AI code editor. As such when we are asked about what the system does, or its functionality, we answer that simply Void Editor is nothing more than an AI-powered code editor. This software combines the basic standard features of any normal IDE, such as VS Code, and

combines into that some AI assistance, from AI such as Gemini. Our basic IDE usually will do actions like text editing, syntax highlights, file navigation, Git integration, and finally an integrated terminal and now our basic AI assistance will also perform certain acts, like autocompletion, refactoring, summarization, and conversational code editing. As such the functionality of our Void editor can be summarized as an IDE with an extra tool, namely the AI assistance. Now then, we are going to go and look as to what the core parts of this software, Void editor, are. First, we have the IDE core, namely the editor core. Then, we have the AI engine, a key part of our AI assistance. Afterwards, we have the Language services part, which implements our code's "intelligence". Then, we have the Project/Workspace manager, which is again a piece of our IDE part of Void editor. Now, we have the user interface layer, this manages the menu and more. Finally, we have the Integration layer, which connects to Git, our terminal and more.

First, let's look at the editor core which manages the core part of our IDE which is one of the two main pieces of the Void editor. This part of the system handles all the text rendering, document structure, cursor movement, undo/redo operations, and file I/O. Basically, it is the part of the system that we use when we want to write or edit into a file we have already opened. It's the central hub of our system which means if we were to look at its interactions with the other subsystems we would have all of them plug into this editor core.

Next, we will look at the subsystem we'll call the language services subsystem. This piece is meant to continuously feed diagnostics and type information to the editor core. Moreover, this piece is meant to look over the text written and analyze the syntax, showing any mistakes, and giving pop-up tooltips, in case any help is needed. This subsystem communicates with the editor core, but also with external language servers to not have to embed the logic for every language directly into the editor.

Now, we are going to observe another important subsystem, the Project manager. This subsystem gives to our system the project-wide context which consists of the file paths and the dependencies. While a workspace is opened our project manager goes and scans the structure of the directory, indexes the files and folders and recognizes the programming language. The project manager goes and gives to the user the global understanding of what they are working on by giving file paths, and the entire list of files in which the user is working on. This subsystem regularly interacts with the editor core and the AI to give to both the broader context of where the project is.

Then, we will look at the AI engine, a key part of the void editor. This key part consumes all the data from, mainly the editor core, and from the project manager and the language services. It goes and takes all this data and communicates it to LLMs, such as OpenAI, Gemini, and Anthropic. It also constructs contextual prompts based on

the data it has received, notably from the project manager subsystem, the language services subsystem, and the editor subsystem. Finally, it takes the answer from the LLM and processes it into actionable results within the editor subsystem.

Now, we are going to observe the UI layer subsystem. This layer simply gets the user's actions like typing, chat prompts, quick edits and displays their results. This layer gives a UI similar to that of Visual studio code for the user's comfort. This subsystem communicates with most other subsystems as it displays everything the backend manages.

Finally, the last layer is the Integration layer. This layer provides most of the external connections that both the user and the AI invoke. This subsystem mostly interacts with connections like Git, the terminal, and APIs then communicates the data it gets from there to the other subsystems.

The communication between the processes in the system is event-driven mostly and asynchronous.

Control Flow

The control flow in Void is event based, which means the system reacts to the user's actions as they happen. For example, if a user highlights a section of code and asks the AI to explain it, that action will then start a chain of events in the program.

1. User-Driven Events
   The process starts when the user activates a command which requires AI such as "Explain Code" or "Generate Function." This request is taken by the UI and is passed to the main editor.

2. Command Dispatch
   The Editor Core then sends this event to the AI Interaction Layer, along with information about the selected text and programming language.

3. Prompt Creation and Model Call
   The AI Interaction Layer takes that information and creates a detailed prompt for the AI. It might also include additional context from the Project Context Engine, like nearby code or related files. Once ready, it hands control to the Model Interface Layer, which sends the prompt to an AI model—either online or locally hosted.

4. Waiting for a Response
   Since AI responses take time, Void uses asynchronous (non-blocking)

communication. This means the editor stays responsive while waiting. Once the AI finishes, the response is sent back and the editor updates automatically.

5. User Feedback Loop
   The user has the ability to accept, reject, or edit the AI response. Each of these creates its own events, creating the interaction loop.

This flow allows Void to handle multiple operations at once without freezing up, since tasks that take longer, such as talking to an AI model, run in the background.

Data Flow

The data flow in Void editor focuses on how information moves throughout the system when the user uses AI tools including the user's code, prompts sent to the AI, and the responses that come back.

1. Source Data Input
   The Editor Core manages the code files being worked on. When an AI feature is triggered, it extracts the relevant part of the code (and sometimes related pieces) using the Project Context Engine.

2. Prompt Construction
   That code snippet and its context are then passed to the AI Interaction Layer, which turns it into a structured prompt for the model. Users can view or edit this prompt if they want to see exactly what's being sent.

3. Sending to the Model
   The Model Interface Layer sends the prompt to the model. If the model is local, everything stays on the user's machine. If it's remote, Void sends the request securely over the internet.

4. Processing the Response
   The AI's reply is then sent back to the AI Interaction Layer, which formats it to display to the user.

5. Updating the Editor
   The Editor Core then shows this suggestion in the user's workspace—either by inserting it into the code or displaying it side by side for review.

Void keeps the user's file data separate from the AI communication data while all

of this happens. This helps to protect the user's privacy and makes it easier to track what information is being shared externally.

Implications for Design

The way Void handles control and data flow gives it many benefits to both the user and the system:

- The system stays fast because AI tasks don't block the main editor due to them being run in the background.

- Since the AI system is modular, users can swap in different AI models or prompt types without having to change the main editor.

- Users can see and control what data is sent to the AI which is very helpful to creating trust for the user and increasing security

Void's control and data flow are designed to balance ease of use, performance, and security. The way these parts communicate keeps the IDE running smoothly while still making AI processes possible.

Concurrency

Void is an AI-assisted IDE built on a fork of Visual Studio Code, and most of its concurrency comes from the architecture it inherits from VS Code plus the extra, non-blocking work needed to talk to AI models. In day-to-day use you can type, see the UI update instantly, get diagnostics from one or more language servers, and receive streaming tokens from an AI assistant, all at once. That experience is only possible because different parts of the system run independently and coordinate through asynchronous messaging rather than by sharing memory or blocking the editor's main thread.

At the base of everything is VS Code's multi-process model. The "renderer" process (called the workbench) owns the user interface and must remain responsive. Extensions do not run in that renderer. Instead, they execute in an isolated Extension Host process so that misbehaving or slow extensions can't freeze the UI. There's also a privileged background process (called the main or shared process) used for heavier tasks and brokering operations that the sandboxed renderer is not allowed to perform. These processes communicate via IPC; none should block the others. This separation is core to how VS Code stays responsive, and because Void is a fork, it follows the same pattern.

Language tooling adds another obvious source of concurrency: the Language Server Protocol (LSP). In VS Code's model, a language server is a separate process that talks to a lightweight "language client" running inside the Extension Host. The editor can actively speak to multiple servers at once (for example, TypeScript, ESLint, and a formatter), and the servers reply asynchronously over JSON-RPC. The protocol even builds in cancellation so the editor can abandon stale requests when the user keeps typing, avoiding "late" results that would clobber newer context. All of that work (the analysis, the completions, the diagnostics) happens concurrently with user input and UI rendering.

Void's AI features introduce their own concurrent pipeline. The project's documentation explains that LLM messages are sent from the main process (not the UI) to avoid Content Security Policy issues and to enable the use of Node modules. In practice, that means the network I/O and token streaming happen off the UI path, and results are pushed back incrementally for display. As tokens arrive, the chat panel or inline edit UI updates piece-by-piece while you continue editing code, which is another clear example of non-blocking concurrency layered on top of the editor's normal language features.

Webviews; stand-alone, sandboxed HTML/JS panes used for things like chat or agent panels, are also their own concurrent contexts. A webview runs with its own event loop and talks to its extension backend by posting messages through a tiny bridge API (acquireVsCodeApi / postMessage). This means a chat panel can render incoming tokens and accept prompts independently of the main editor surface, while the Extension Host and language servers do their own work in parallel. The coordination is message-based: the webview never directly touches the Extension Host's memory or the editor's DOM.

Terminals and other long-lived subprocesses are another concurrent strand. VS Code hosts pseudo-terminals (PTYs) in privileged processes, and their output arrives as asynchronous events. Void's "agent" features can open and keep background terminals alive while you edit and chat. Those processes continue running and emitting output without blocking keystrokes or UI updates; they're just more actors talking to the editor over event channels.

On top of local processes, there's also the possibility that parts of the system are remote or web-based. VS Code supports running the Extension Host remotely (SSH/containers) or inside a browser as a "web extension host." When you work that way, you add extra concurrency boundaries: local UI ↔ remote Extension Host ↔ language servers on the remote machine. Messages still flow asynchronously; they just cross machine or runtime boundaries. Void inherits this ability because it is a VS Code

fork, so the same separation applies.

What's important is how this concurrency is controlled. The architecture favors asynchrony and message passing over shared-state locking. In the renderer and webviews, you have the browser event loop; in the Extension Host and main/shared process, you have Node's event loop. Work is dispatched as promises or streams. For language features, the LSP client associates request IDs with each call and can cancel them; the server is required to respond even on cancel, which gives the editor a reliable way to drop stale results. For UI features like chat, streaming is used so partial results can be rendered without waiting for a full response. The result is a system that tolerates variance in computation and network latency without freezing the editor.

A typical editing moment makes this concrete; you type a line, the renderer updates the viewport instantly. Simultaneously, the language client pushes a textDocument/didChange to one or more servers, completion and diagnostic requests go out, and the AI chat (if open) might send a prompt or context window to a provider through the main process. As answers come back, the editor threads them into the UI; diagnostics underline code, completions appear, tokens stream into the chat panel. If you type again quickly, the editor cancels the now-stale requests and ignores any late responses that still trickle in. None of this requires threads that share mutable state; it's all independent actors exchanging messages and events.

There are clear architectural benefits to doing concurrency this way. Responsiveness is protected because the UI is isolated from heavy work. Faults are contained because extensions and servers live out-of-process. Scalability comes "for free" as you add more language servers or swap AI providers; they're just more endpoints on the same non-blocking message fabric. And, because Void routes provider calls through the main process, it can support both remote APIs and local models without violating the renderer's sandboxing rules or locking up the interface during long requests.

It's also fair to point out the limitations and how the design addresses them. With many concurrent actors, you can get races (for example, an old completion arriving after the document changed). LSP's cancellation and request IDs exist precisely to tame that issue. Streaming AI responses can momentarily display suggestions that no longer match the current buffer, which is why the UI treats streamed content as provisional and updates or discards it if the context changes. Because terminals and AI calls operate in other processes, there's always the possibility of failure or delay, as the isolation prevents those problems from taking the editor down with them.

In conclusion, there's a lot of concurrency in Void. The editor UI (renderer), Extension Host, and main/shared process run independently, one or more language servers work in parallel as separate processes, webviews host their own event loops, terminals and agent tasks can run in the background, and AI requests stream through the main process. These pieces coordinate through asynchronous IPC and JSON-RPC, using request IDs, cancellation, and streaming to keep the interface responsive and the results coherent while you work. That is the conceptual concurrency picture of Void as an AI-augmented fork of VS Code.

Evolution

VOID's architecture has evolved steadily since its first release, with each stage focused on making the system more modular, responsive, and adaptable. Over time, it has shifted from a straightforward layered design into a hybrid, event-driven framework capable of handling asynchronous AI operations and agent-based automation.

In its earliest form, Version 1.0 relied on a simple layered structure adapted from Visual Studio Code. The editor core managed the interface and workspace, while the AI Bridge linked user prompts directly to language models for autocomplete and code generation. At this stage, plugins were static and tightly bound to the editor, and data moved in a single, top-to-bottom flow through the system.

With Version 1.5, VOID's design matured into an event-driven model. The addition of an internal event bus allowed modules to communicate asynchronously, which reduced direct dependencies and made the architecture far more flexible. Features like checkpoints and background model requests could now be introduced without touching core components, and multiple AI tasks could run simultaneously without slowing the editor.

Version 2.0 marked a major step forward with the arrival of Agent Mode. This version introduced a capability registry to organize the actions the agent could perform and a sandbox system to ensure that file edits and command executions were safe. The assistant could now act on the project autonomously—reading, modifying, or generating code—while maintaining strict permission checks.

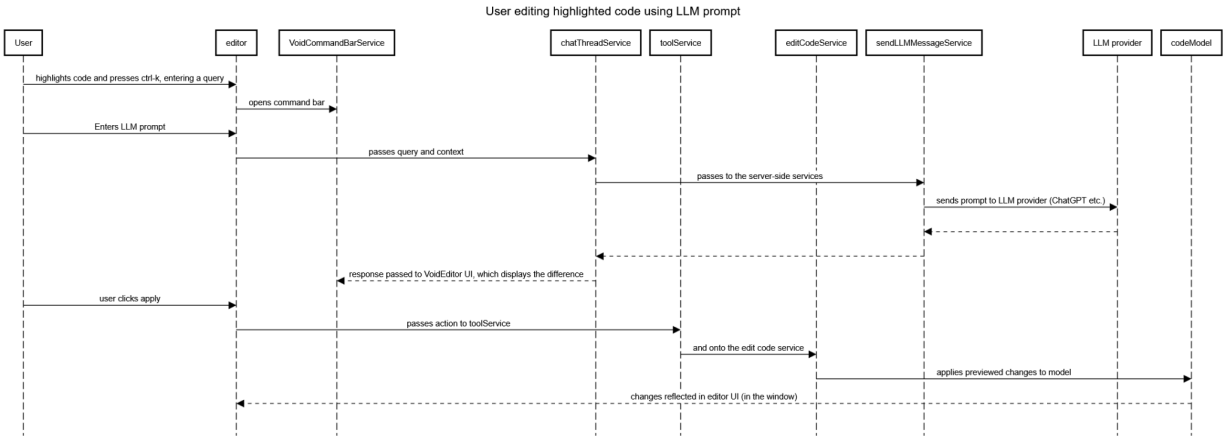
By Version 2.5, the focus turned to persistence and collaboration. Checkpoints were extended to sync between local and cloud storage, and a new API Gateway handled communication for shared sessions. These changes prepared VOID for distributed use while preserving its core principle of user-controlled privacy.

Looking ahead, Version 3.0 and beyond aim to unify these advances under a service-oriented architecture, enabling both local and remote agents to coordinate through a distributed message gateway. This evolution supports collaborative, multi-agent workflows and more adaptive orchestration across environments.

Taken together, VOID's development reflects intentional architectural refinement rather than simple feature growth. Each phase has improved modularity, concurrency, and security while keeping the system conceptually consistent. The shift from linear control flow to asynchronous event routing allowed it to scale smoothly, and the addition of sandboxed agents balanced automation with user trust. Collectively, these changes trace a clear path toward an adaptive, extensible, and privacy-focused AI development platform.

Use Cases/UML Diagram

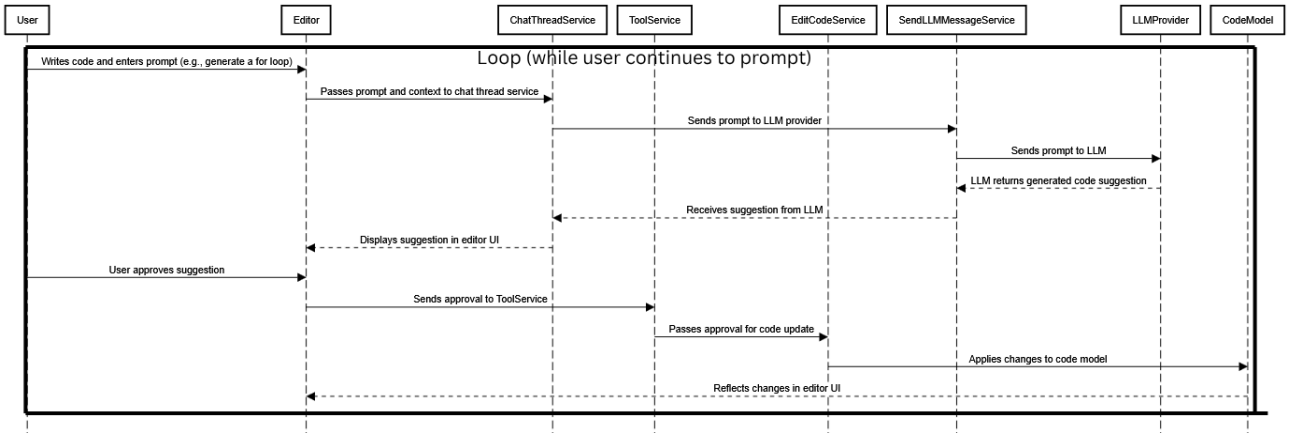
Use case 1: Code highlight and edit prompt.



The most common use case for VoidEditor would be a basic request to the LLM to refactor/edit a highlighted section of code. This begins with the user working in their editor of choice (such as VSCode) and highlighting the section of code they wish to affect. Then, they enter the prompt they will be sending to the LLM in the VoidEditor UI element, and submit. This is passed to the ChatThreadService, which manages chats with the LLM. This prompt, along with the context (highlighted code) is passed to the sendLLMMessage service, which – via the sendLLMMessageChannel (not depicted) – passes this to the server side components, into the sendLLMMessage component. From here, the prompt actually reaches the LLM provider (ChatGPT, Claude, etc.) which responds. This response works its way back up the chain to the ChatThreadService, and then to the Editor's UI which is updated to show the suggested changes and differences. Should the user then click apply, a signal is passed to the

clientThreadService again, which now knows to try and resolve the response into the code, communicating with the toolService, and then eventually the editCodeService, which then makes the actual changes to the code model file.

Use Case 2: LLM lead code generation



Most communication with the actual LLM providers is more or less the same, regardless of the case in which it is being used in the VoidEditor. The above case portrays the same basic pipeline to take the initial prompting in the editor, pass it to the LLM, and then return the response in a way the user can approve or deny. The distinction here is the overarching loop box, the user can continue to use the same chat with the AI (which is maintained and handled by the chatThreadService) in order to incrementally build a project. The user can start from scratch, provide a suggestion, and the AI will allow them to make their own choices from sets of options to gradually build out the solution they require. This is different from the original case, as it does not require the user to be so specific about where they wish for the AI to do it's work, instead, the whole base is accessed by the LLM, and worked on accordingly, with the use still the core arbiter of what does and doesn't get patched in.

Implications for Division of Responsibilities

Our system, VOID AI, is not just an AI-powered code editor, it is also a software ecosystem that combines modularity, decentralization, and open-source collaboration. This means when we look at how the responsibilities of the developers, we also have to look at how these roles connect together in a constantly evolving, and distributed architecture. The VOID Editor has no single owner, instead it is the product of many developers, each working on a specific layer or subsystem, while maintaining consistency with the overall architecture.

Now then, before we can properly analyze the implications for the developers, we need to understand the context. VOID AI follows a layered, event-driven, and plugin-based architecture. These characteristics make the system modular, but this also means that no single developer or team can control the entire system. Instead the responsibilities are divided across layers, the presentation layer, logic layer, and data layer. Each module is self-contained, but all modules must communicate through APIs or an internal event bus.

The first major group of developers are the frontend developers, who handle the Editor core, and user interface. Their job is to make sure that what the user sees and interacts with works currently with the backend logic. They take charge of everything that happens in the presentation layer, the text rendering, cursor movement, syntax highlighting, and user input. Because VOID is event based, the frontend developers cannot just directly call backend functions as in traditional applications. Instead, they must use asynchronous events dispatched through the event bus. The implication of this is that frontend developers have to understand concurrency and asynchronous programming, to make sure that the interface remains smooth even when the AI engine, or other subsystems are performing heavy background tasks.

The second major group of developers are the backend developers. They are in charge of the AI Bridge, the Model Connector, and the Checkpoint manager. This manages how prompts are created, how data is sent to large language models, and how the results are used in the editor. The implications for this are that backend developers have to design systems that can run securely on a user's local machine without leaking data, while still allowing integration with remote APIs. They also have to plan with the front end team to make sure both ends of the product are aligned, in terms of format and timing. Essentially, the backend developers connect the user's experience with the intelligence of the model, without blocking, crashing or sacrificing privacy.

The third major group are the plugin developers. This is one of the more unique roles, because of the Plugin Manager subsystem, that allows developers to create and integrate their own extensions without modifying the core system. The implication of this design is decentralization, so anyone can extend VOID's capabilities, whether that means adding a new AI tool, a syntax engine, or a debugging assistant. However, this also adds challenges. Since plugins are developed independently, developers must stick to the event protocols and API guidelines defined by the core team.

The fourth major group are the security and infrastructure engineers. These developers manage the Security Sandbox, the API gateway, and the Checkpoint storage system. These developers mostly focus on maintaining privacy, and safety.

Because the system aims to move toward distributed, multi-agent collaboration, these engineers have to design mechanisms that verify all actions taken by both the AI agent, and human users. The implication of their work extends across the entire ecosystem, they define what actions are allowed to be performed by an agent, which files they can modify, and make sure that the data stays encrypted both locally, and cloud storage. As the platform scales, these developers become more and more important.

The final group of developers are the testing and quality assurance developers. These are the developers who make sure that all the independent pieces, from the AI Bridge to the plugins, work together. Since VOID is open-source, testing has to be continuous, distributed, and automated. Each developer is responsible for validating their components before it is merged into the main repository. The implication is that testers have to adapt to event-driven systems, where behaviour cannot always be predicted linearly. For example, one event might trigger a chain of asynchronous updates that has to be verified. This means that the QA team has to design event simulators and parallel test environments to replicate real world situations.

Conclusion

VOID represents a modern rethinking of the integrated development environment — one that places AI not as a separate assistant, but as a deeply integrated, event-driven component within the editing workflow. Through its evolution from a layered fork of Visual Studio Code into a distributed, hybrid architecture, VOID demonstrates how modular design and asynchronous communication can enable complex AI functionality without sacrificing user control or performance.

The system’s conceptual design achieves a careful balance between autonomy and safety. The introduction of Agent Mode allows the AI to act independently on a user’s project, while the Security Sandbox enforces permission boundaries. Meanwhile, Checkpoints provide persistent recovery and collaboration tools that align with the system’s distributed vision. These architectural choices reflect an intentional evolution — each feature addressing a clear need in modern AI-assisted development rather than being added reactively.

The architectural analysis also highlights the value of open-source collaboration in managing such a complex system. Responsibility is divided cleanly between frontend developers, AI integrators, plugin authors, and infrastructure engineers, each working within an event-driven framework that promotes independence and stability.

Ultimately, VOID exemplifies the principles of sustainable software evolution: modularity, concurrency, and openness. It shows that a system can grow in capability and complexity while preserving clarity in its conceptual architecture. The result is a

privacy-focused, extensible AI development platform that remains transparent, adaptable, and resilient — characteristics that will be increasingly vital as AI systems become standard within professional software workflows.

Lessons Learned

From analyzing VOID, we learned that continuous evolution and modular design are key to sustaining modern software. VOID's transition from a basic layered IDE to an event-driven, agent-based system shows how iterative development enables new functionality without breaking the foundation.

We also learned that user trust and control are critical in AI-assisted tools. VOID achieves this by using local processing, transparent prompts, and sandboxed agent actions—ensuring safety while preserving autonomy.

Another major insight is the importance of collaboration and shared standards in open-source projects. VOID's success depends on contributors maintaining consistency across independent modules through clear architectural guidelines.

Overall, VOID taught us that sustainable innovation in AI systems requires adaptability, transparency, and collective effort—qualities that will define how future intelligent software is built and improved.

AI member profile and why we selected it

For A1 we worked with ChatGPT (GPT-5 Thinking, October 2025) as a virtual teammate. We tried short trials with Gemini and Claude for long-context synthesis, but ChatGPT gave us the best mix of fast turn-around, stable tone control, and a strong grasp of software-architecture vocabulary. It also handled long, constraint-heavy prompts without drifting off topic. Because this deliverable depended on understanding Void's public materials and then shaping a clean conceptual view, we needed an assistant that could (a) scan and summarize disparate sources, and (b) help us converge on a consistent component vocabulary. ChatGPT met those needs reliably. We did not need image/video tools for A1, and we chose not to use local LLMs because setup overhead would have slowed us down for this milestone.

What we delegated to the AI (and why)

We used the AI in two places: researching information about Void and figuring out the basic framework at the very beginning.

First, on research: we asked the AI to generate a research map of Void (what official pages and code areas are most relevant to conceptual architecture), which names Void uses for processes/components, and how those relate to vanilla VS Code. The reason this is a good fit for AI is that the task is breadth-first and clerical: gather, cluster, and summarize public information so humans can decide what matters. We explicitly told the model to surface uncertainties and to flag anything that looked like a guess so we could check it later.

Second, on the framework: we asked the AI to propose initial skeletons for the conceptual picture (candidate components, responsibilities, and interactions) and to suggest a few paragraph orders that would read naturally to a newcomer. This is suitable for AI because it accelerates the "blank page" phase; we still expect to change names and cut pieces once we verify them. We did not ask the AI to make authoritative claims or to produce the final prose. We owned the verification and the final writing.

How we interacted with the AI (workflow and prompting):
We designated one teammate as the prompt lead to keep a single coherent thread. Everyone else contributed prompt text and questions in a shared doc; the lead consolidated that into a single message so the constraints didn't contradict each other. Our workflow had two loops:

- Research loop. Prompt for a map of sources and terms → receive a structured summary → ask follow-ups to separate Void-specific facts from inherited VS Code behavior → request a compact "fact vs. inference" table so we knew what to verify.
- Framework loop. Provide our course constraints and audience → ask for 2–3 candidate conceptual skeletons (components, responsibilities, interactions) → pick one, then iterate on naming and scope until it matched how we wanted to teach the material.

A representative complex prompt from the research loop (excerpt):
"You are assisting a student team preparing a conceptual architecture of the Void editor. Your tasks: (1) list the most relevant public, official sources we should read; (2) summarize, in neutral language, what Void adds on top of VS Code (process boundaries, webviews, extension host interactions, AI provider path); (3) separate 'likely inherited from VS Code' vs. 'Void-specific' in a two-column table; (4) highlight any claims you are not certain about. Do not speculate. If you are unsure, say 'unknown.'"

We refined prompts as we learned. For example, after an early run mixed documentation with blog assumptions, we added: "Prefer official docs and repo documentation; avoid secondary blogs unless they paraphrase official text." For the

framework loop we used instructions like: “Propose three alternative component breakdowns (5–8 components each). For each, write a one-sentence responsibility and one sentence describing how it talks to its neighbors. Use the same names consistently across the proposal.”

Validation and quality control

Because we used the AI for research and early framing, our validation focused on source accuracy and vocabulary discipline:

1. Primary-source cross-checks. For each claim in the AI’s research map and skeleton, a team member checked it against official sources (Void repo/docs and VS Code/LSP documentation). If we could not tie a sentence to a primary source, we either rewrote it more cautiously or dropped it. We avoided “probably” and “likely” phrasing.
2. Fact/inference labeling. In the shared draft, we annotated sentences as F (fact, verified), I (inference, reasonable), or ? (unverified). Only F sentences survived into the final report. If an inference was useful pedagogically (e.g., “this is message-based, not shared-memory”), we converted it to a factual, sourced statement or moved it to discussion as an observation.
3. Terminology pass. We aligned names across the report (renderer, extension host, webview, main/shared process, language servers, agent/terminal). The AI’s skeleton was helpful here but occasionally mixed terms; we normalized to one vocabulary and deleted variants.
4. Recency and scope checks. Before finalizing, we spot-checked that nothing we kept depended on a minor or deprecated feature. If the AI mentioned something that existed only in preview branches or blog posts, we removed it unless we could find it in current docs.
5. Data-handling rules. We never pasted private code or non-public notes into the AI chat. All inputs were either our own outline or public documentation. When we extracted snippets for discussion, we paraphrased rather than pasting large chunks.

This process took real time, but it paid off: by the time we wrote the final text, the structure was settled and the claims were already vetted.

Quantitative contribution

We estimate the AI contributed ~20% to this deliverable, focused entirely on research organization and initial framing:

- Research mapping and clustering public info: ~12%
- Initial component skeletons and paragraph ordering options: ~6%
- Minor wording suggestions inside the outline (not final prose): ~2%

All verification, diagram choices, sequence elaboration, and final writing/editing were human.

Reflections on team dynamics and what we learned:

The biggest win was speeding up the blank-page phase. Having the AI produce two or three plausible skeletons forced us to articulate why we preferred one structure over another, earlier than we would have on our own. It also helped us converge on a single vocabulary and stick to it, which made later sections (like the sequence descriptions) easier to write.

The main trade-off was new work in validation. The AI's summaries were fluent, but they sometimes blended Void-specific details with general VS Code patterns; it's just what happens when multiple sources overlap, but it meant we had to police the boundary carefully. Our "fact/inference" labels and primary-source checks caught a handful of subtle errors (for example, wording that implied direct DOM access from a webview, which is not how that isolation works). We also learned to give negative instructions in prompts ("don't speculate," "avoid secondary blogs unless they quote official docs"), which measurably improved later outputs.

Another lesson was role clarity. Appointing a prompt lead kept the conversation thread consistent and prevented us from sending mixed signals. It also reduced rework: instead of five people trial-and-error prompting, we pooled constraints first, then sent one high-quality message. For future milestones we'll keep this role, but we'll rotate it so more teammates practice the skill.

Finally, we think the division of labor we chose, AI for breadth and scaffolding, humans for depth and judgment, fits the spirit of the assignment and how real teams work with AI tools. The AI accelerated the parts that are mostly clerical (collect, cluster, sketch), while we retained accountability for accuracy, emphasis, and pedagogy. Going into A2/A3, we plan to reuse the same pattern: start with an AI-generated research map and a few alternative skeletons, validate against primary sources, then commit to one conceptual cut and do the concrete verification and design work ourselves.

References

VOID Editor. *VOID: Open Source AI Coding Editor.* VOID, 2024. https://voideditor.com/

VOID Editor GitHub Repository. *voideditor/void (Main Repository).* GitHub, 2024. https://github.com/voideditor/void

VOID Documentation. *Features Overview and Agent Mode.* VOID Docs, 2024. https://docs.voideditor.com/

Microsoft Documentation. *Visual Studio Code Architecture Overview.* Microsoft Docs, 2023. https://code.visualstudio.com/api/extension-guides/overview