CISC 322 - Conceptual Architecture

David Balan
Mark Nistor
Maxim Logunov
Ameek Bains
Luca Del Sorbo
Radmehr Vafadarfalavarjani

October 14, 2025

Contents

List of Figures

Abstract

Void is an AI integration tool built on top of VS code (forked from VS Code) allowing for the integration of LLMs to assist with coding tasks, whether that be editing, generation, chats or gathering information about your code. Its entire purpose is to use a conventional IDE with an AI experience while allowing privacy, security and transparency through open-sourcing. This security is done by sending requests to LLMs directly rather than using a third-party go-between. It keeps VS code's base architecture (its electron shell and the main VS code rendering processes) and extends it with new services oriented around LLM interactions.

There are many abilities it brings to VS code and this report will group them by functionality. The system routes user intent from UI and commands to a request pipeline that assembles context and prompts, invokes a provider adapter in the main process, streams model output back to the UI, and optionally applies diffs to code. The architecture preserves VS Code's two-process model while layering an AI integration surface composed of modular services: chat orchestration, edit application, autocomplete, tooling/actions, settings, and model capability management. The design emphasizes responsive streamed interaction, explicit diff-based editing, and capability-gated features to support transparency, privacy, and evolvability.

1 Introduction and Overview

Void is a fork of VS Code that augments the conventional IDE with AI capabilities, letting developers issue prompts, perform edits, and chat with models directly inside their coding environment. It preserves VS Code's underlying structure (Electron, renderer/main process separation, extensions) while layering in services to manage model communication, diff application, and controlled tooling. The architecture is organized as modular services—settings, messaging, editing, autocomplete, chat, and tooling—that interact through a central pipeline to preserve both flexibility and transparency.

Scope. We focus on what the system does, how it is broken into interacting parts, how parts interact, how the system evolves, global control/data flow, concurrency, and developer responsibility boundaries.

Salient points (verbatim excerpts from the Void Team).

- "Its entire purpose is to use a conventional IDE with an AI experience while allowing privacy, security and transparency through open-sourcing."
- "It keeps VS code's base architecture (its electron shell and the main VS code rendering processes) and extends it with new services oriented around LLM interactions."
- "The architecture is organized as modular services—settings, messaging, editing, autocomplete, chat, and tooling—that interact through a central pipeline to preserve both flexibility and transparency."

2 Architecture

2.1 Overall Structure

Conceptual Architecture Overview: Browser & Main Void inherits VS Code's Electron architecture.

The bprocess handles everything the user sees. It shows the UI (editor, chat) and gathers what the user wants to do. Main process handles everything that needs LLM access for the Browser and OS access.

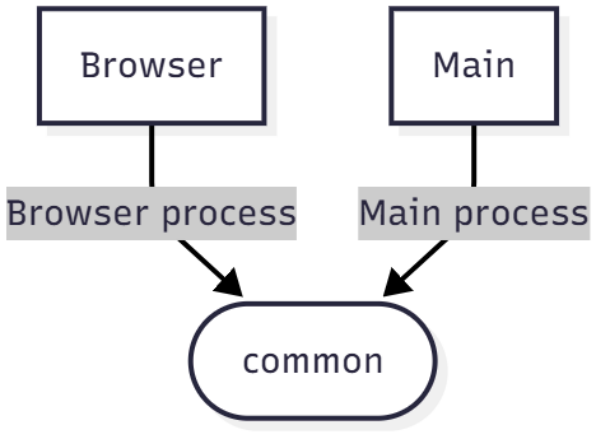

Figure 1. Conceptual Architecture Overview: Browser & Main

Core Editor / IDE Functionality (Inherited from VS Code)

- Editor Core – Base VS Code fork for text editing, file management, extensions, and UI.
- Editor Services – Standard VS Code features managing text models and editing operations.
- VoidModelService – Keeps text models alive and referenced, preventing premature disposal.
- UI Components – React-based components (e.g., command bars, chat windows) providing the interface for both normal IDE actions and AI features.

AI + LLM Integration Layer

- LLM Connection Layer – Manages communication with local or cloud-hosted LLMs.
- LLMMessageService – Central coordinator for all AI interactions; sends prompts, streams responses, routes results.
- EditCodeService – Takes LLM output and applies edits via diff zones; supports streaming integration.
- AutocompleteService – Provides inline completions, leveraging prefix/suffix context.
- ChatThreadService – Handles multi-turn chat with AI, storing conversations and applying changes.
- ToolsService – Exposes controlled actions like reading files or running commands (under user supervision).

Bridge Components (Glue Between IDE and AI)

- Context Extraction – Gathers the relevant code or project context to feed into prompts.
- Prompt Engineering System – Builds structured prompts tailored to user intent (edit, chat, autocomplete).
- Response Integration – Interprets and streams back AI outputs, embedding them seamlessly into the editor (diffs, inline text, chat snippets).

Each part extends the VS Code slice:

- Common services like VoidModelService preserve IDE stability.
- Renderer/browser layer hosts UI and services like EditCodeService, AutocompleteService, and ChatThreadService.
- Main/Electron layer bridges to model providers and performs provider communications.

In code organization, three top-level groups are used: `browser/` (renderer/UI services), `common/` (shared types and services), and `electron-main/` (main process services). In addition, much of the "Void-specific" code lives under `src/vs/workbench/contrib/void/` to isolate from VS Code's core.

2.2 Components and Interactions

Big picture components and their roles (verbatim selections):

- **SidebarChat UI**: component the user types into; render chat output and status.
- **CommandService / Actions**: the central dispatcher. UI invokes commands and the dispatcher routes it to the right service.
- **chatThreadsService (CTS)**: chat orchestrator. Reads settings, validates model, sends IPC requests, receives streamed chunks, and pushes them to UI.
- **voidSettingsService (VSS)**: stores all Void settings for model selection and configuration.
- **modelCapabilities (MC)**: guard for limits per model before sending prompt to LLM.
- **sendLLMMessageChannel (IPC→Main)**: Browser-to-Main bridge. From the Browser's POV, this is just a message bus; it hands off to Main and also delivers streamed responses back.
- **sendLLMMessage (Main)**: validates request; calls a provider adapter; streams chunks/errors back to Browser; supports cancellation.
- **Provider Adapters (Main)**: HTTP/stream clients for specific providers; this unifies streaming, error, and abort semantics.
- **External Providers**: OpenAI/Ollama/Anthropic endpoints returning streamed tokens or structured edits.

2.3 Architectural Styles

Style 1: Layered

Why it fits *Void*.

- **Clear separation of tiers.** *Void* divides functionality into:
  - **Browser / Renderer (Front-End)** — UI, interaction, chat/edit orchestration.
  - **Main / Electron (Back-End)** — provider adapters, streaming, model invocation.
  - **Common (Shared)** — utilities, types, and cross-cutting helpers.
- **Unidirectional flow across layers.** Prompts flow *down* from UI → settings/message services → main/adapter → external model; responses stream *up* back to the UI. A dedicated "Apply" subsystem chooses between *Fast Apply* (diff/block substitution) and *Slow Apply* (full rewrite) while lower layers execute the concrete transformation.
- **Modularity & maintainability.** The renderer never embeds provider logic; provider adapters can evolve (or new models be added) without touching UI code. The repo layout (`browser/`, `electron-main/`, `common/`) reinforces these boundaries.

How This Is Presented in the Report. *Void* employs a layered architecture: the renderer/browser layer hosts UI, diff zones, and chat & edit orchestration; a separate main/electron layer houses model adapters, streaming, and provider communication; and a common layer shares utilities and types across both. This separation enables clean concerns, modular evolution (e.g., swapping adapters), and controlled data flow from UI through to LLM and back.

Layer mapping of key components (examples).

- **Browser/Renderer:** `SidebarChat UI`, `CommandService/Actions`, `chatThreadsService`, `editCodeService`, `AutocompleteService`, `DiffZones`.
- **Main/Electron:** `sendLLMMessage` handler, provider adapters (e.g., OpenAI/Ollama/Anthropic), streaming/cancel logic.

- **Common:** shared types, `modelCapabilities`, utilities used by both sides.

Advantages & Disadvantages (for *Void*).

- **Pros:** Clear separation of concerns yields easy maintenance and modular updates. Unidirectional data flow makes behaviour and debugging simple. Reusable shared layer (common/) makes development simple and doesn't require duplication of processes.
- **Cons:** Layering the IPC overhead yields slower performance during real-time streaming/editing. Strict boundaries make it difficult to implement cross-layer experimentation or hybrid features. Dependency management between layers is more difficult.

Style 2: Client–Server

Why it fits *Void*.

- **Structural pattern.** Client–Server organizes distributed data/processing across components; *clients* call services provided by *servers* over a network connector. In *Void*, the Browser (renderer/UI) acts as a client; the Main/Electron process acts as a server to the Browser and as a client to external LLM *providers* (servers) over HTTP streaming—forming a simple multi-tier client–server chain.
- **Connector.** Browser ↔ Main via IPC; Main ↔ Provider via network (HTTPS streaming). This directly matches the client–server connector definition (network).

How we present this in the report. *Void* adopts a Client–Server style: the Browser process is the *client* issuing requests (prompts, autocomplete, apply-edit commands) to the Main process *server*, which performs validation and dispatch via provider adapters; the Main process then acts as a *client* to external LLM *servers* and returns streamed results to the Browser. This aligns with the course definition (clients call services on servers; network as connector) and the presented client–server diagram of clients connecting to multiple server types (file/database/object).

Mapping to *Void* components.

- **Clients:** `SidebarChat UI`, `CommandService/Actions`, `chatThreadsService` (issue requests; render streamed replies).
- **Server (tier 1):** Main/Electron handler `sendLLMMessage`, capability checks, cancellation, error handling.
- **Servers (tier 2):** External LLM endpoints (OpenAI/Ollama/Anthropic) returning streamed tokens/edits (akin to "database/file/object servers" examples).
- **Connectors:** IPC (Browser↔Main), HTTPS streaming (Main↔Provider).

Advantages & Disadvantages (for *Void*).

- **Pros:** straightforward distribution of data; location transparency; heterogeneous mix-and-match; easy to add/upgrade servers.
- **Cons:** performance depends on network; design/implementation can be tricky; service discovery can be hard without a registry.

2.4 Performance-Critical Parts

- **Streamed rendering loop**: frequent UI updates from token streams; risk of jank without batching/back-pressure.
- **Diff application on large files**: fast path (targeted search/replace) vs slow path (rewrite); correctness and latency trade-off.

- **IPC throughput**: sustained chunking across Browser↔Main; cancellation must be prompt to avoid wasted work.
- **Autocomplete latency**: tight budget for keystroke-to-hint; requires lightweight context and early abort on typing.

2.5 Global Control and Data Flow

Use Case 1 — Ask Chat / Send LLM Message (End-to-End) (Condensed high-view end-to-end diagram first, then browser slice, then main slice.)

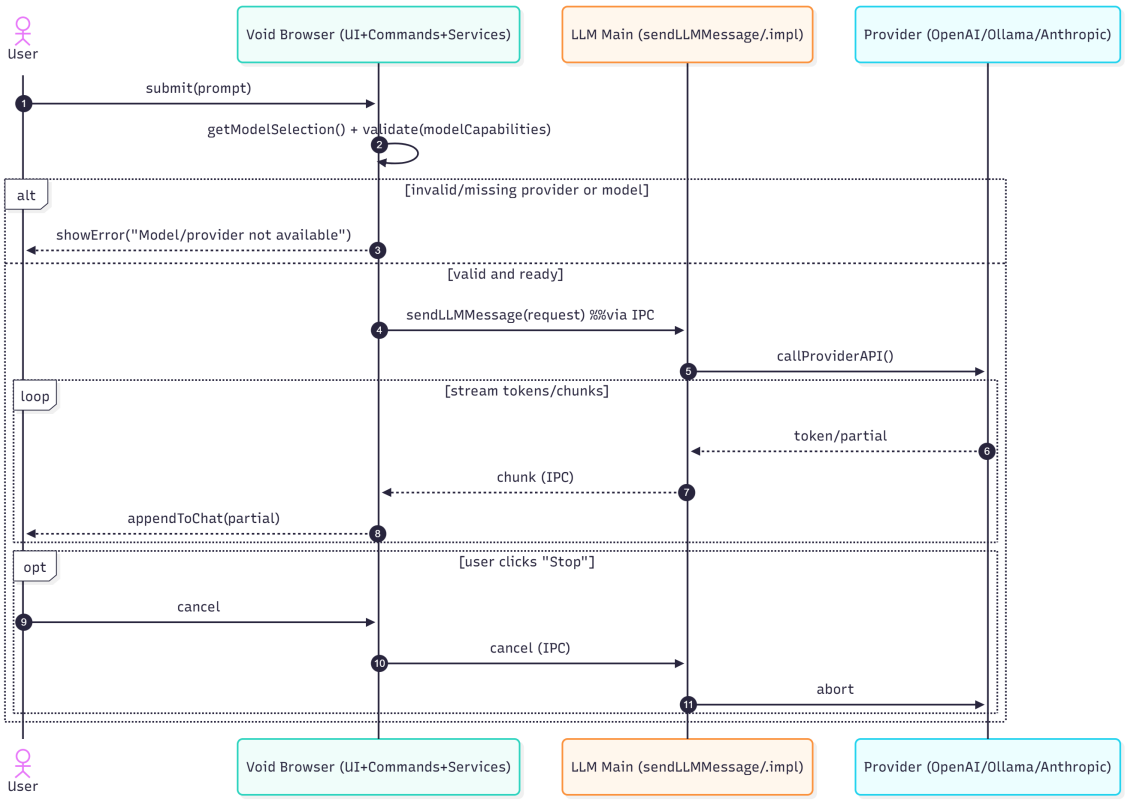

Figure 2. Use Case 1 — Ask Chat / Send LLM Message (End-to-End)

Flow (textual outline).

1. User → Browser: submit(prompt).
2. Browser: getModelSelection() + validate(modelCapabilities).
3. If invalid/missing provider or model: showError(`Model/provider not available`).
4. Else (valid): Browser → Main: sendLLMMessage (request) via IPC.
5. Main → Provider: callProviderAPI() (streaming enabled).
6. Provider → Main → Browser: loop tokens/chunks; Browser appends partials to Chat.
7. Optional cancel: User clicks Stop; Browser → Main: cancel; Main → Provider: abort.

Use Case 1a — Browser Process (Deeper Slice of Chat Flow) Objects: SidebarChat UI; CommandService/Actions; chatThreadsService (CTS); voidSettingsService (VSS); modelCapabilities (MC);

sendLLMMessageChannel (IPC→Main).

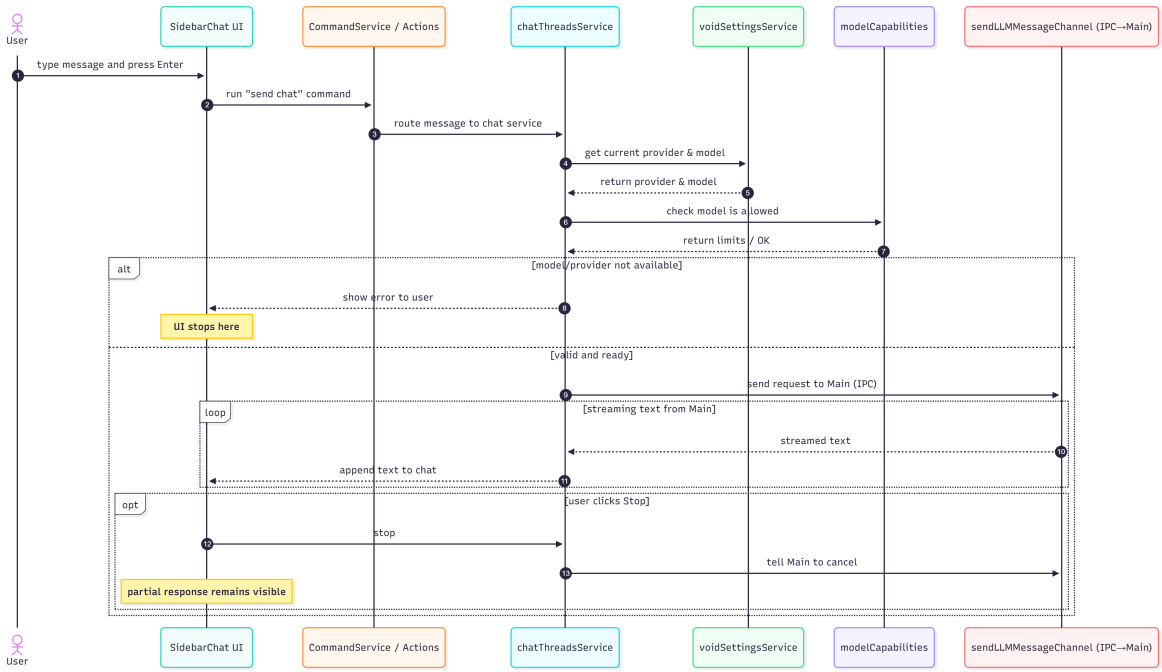

Figure 3. Use Case 1a — Browser Process (Deeper Slice)

Flow (Textual Outline).

1. User → UI: type message and press Enter.
2. UI → CommandService: run "send chat" command.
3. CommandService → CTS: route message to chat service.
4. CTS → VSS: get current provider & model; VSS → CTS: return provider & model.
5. CTS → MC: check model is allowed; MC → CTS: return limits / OK.
6. *alt model/provider not available*: CTS → UI: show error to user (UI stops here).
7. *else valid*: CTS → IPC: send request to Main (IPC).
8. *loop streaming text from Main*: IPC → CTS: streamed text; CTS → UI: append text to chat.
9. *opt user clicks Stop*: UI → CTS: stop; CTS → IPC: tell Main to cancel (partial response remains visible).

Use Case 1b — Main Process (Deeper Slice of Chat Flow) **Objects:** Browser; Main (LLM handler); Model info (capabilities); LLM Provider.

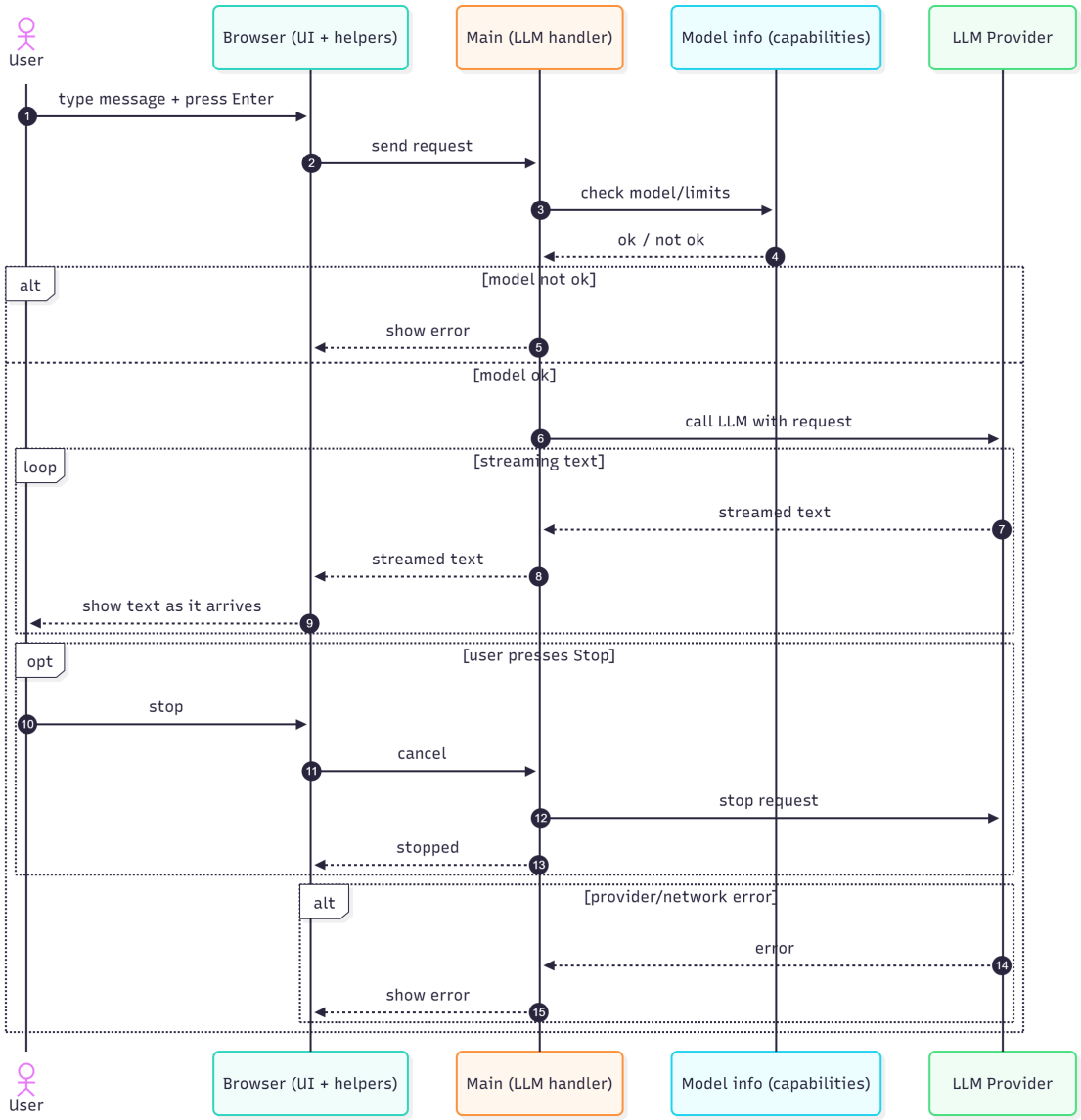

Figure 4. Use Case 1b — Main Process (Deeper Slice)

Flow (Textual Outline).

1. User → Browser: type message + press Enter.
2. Browser → Main: send request (over internal channel).
3. Main → Model info: check model/limits; Model info → Main: ok / not ok.
4. *alt model not ok*: Main → Browser: show error.
5. *else model ok*: Main → LLM: call LLM with request.
6. *loop streaming text*: LLM → Main: streamed text; Main → Browser: streamed text; Browser → User: show text as it arrives.
7. *opt user presses Stop*: User → Browser: stop; Browser → Main: cancel; Main → LLM: stop request; Main → Browser: stopped.
8. *alt provider/network error*: LLM → Main: error; Main → Browser: show error.

Common Initial Path (all paths share this).

1. The user submits a prompt in the UI (SidebarChat / Browser).
2. UI → chatThreadsService: run "send chat".
3. Model pick + validation (UI-only):
   - chatThreadsService → voidSettingsService: get current provider + model.
   - voidSettingsService → modelCapabilities: check allowed & within limits (context window, streaming support, etc.).
   - If OK, UI sends a sendLLMMessage(request) to Main over IPC. If not OK, it never calls Main.

Path A, Validation fails (no provider/model or capability/limit issue). Control flow:

1. modelCapabilities returns not ok (e.g., model disabled, provider missing, prompt too long for model).
2. chatThreadsService does not open IPC to Main.
3. UI triggers showError(`Model/provider not available` or limit message).

Data flow: Stays inside the UI process. No LLM requests or chunks.

Path B, Valid → Provider returns a complete streamed response. Control flow:

1. UI sends a request to Main via IPC.
2. Main calls provider with streaming enabled.
3. Provider streams tokens; Main forwards chunks to UI via IPC.
4. On finish, a terminal message with `finish_reason` is passed and the response is marked complete.

Data flow:

- **LLM request (UI → Main):** provider, model; messages and/or prompt; sampling (temperature, top_p, max_tokens); tools/tool_choice if supported; flags (stream, vision, json_mode); thread_id, message_id.
- **Streamed chunk (Main → UI):** delta (new text); message_id / chunk_index; `finish_reason` when done (e.g., stop, length, cancelled); optional tool_calls / logprobs if supported.

User sees: Text appears token-by-token. When finished, the message is marked as complete.

Path C, Valid → User cancels mid-stream ("Stop"). Control flow:

1. streaming underway.
2. user clicks Stop.
3. UI sends `cancel()` to chatThreadsService.
4. chatThreadsService sends `cancel(request)` to Main via IPC.
5. Main aborts provider call; sends terminal cancel to UI; UI marks stopped.

Data flow (until cancelled):

- LLM request (UI → Main): provider, model; messages/prompt; sampling params; flags; thread_id, message_id.
- Streamed chunk (Main → UI) (until cancelled): delta; message_id / chunk_index; eventual `finish_reason=cancelled`; optional tool_calls/logprobs.

User sees: Whatever text streamed so far remains visible ("partial response remains visible"). Message state becomes stopped (not an error).

Path D, Valid → Provider/network error during streaming. Control flow:

1. streaming underway.
2. Provider or network error occurs.
3. Main catches it and sends an error over IPC.
4. UI surfaces an error state but keeps the partial text already appended.

Data flow:

- LLM request (UI → Main): provider, model; messages and/or prompt; sampling; tools/tool_choice; flags; thread_id, message_id.
- Streamed chunk (Main → UI) (until error): delta; message_id / chunk_index; terminal error instead of normal `finish_reason`; optional tool_calls/logprobs.

User sees: partial response + error marker.

2.6 Concurrency (effective behavior)

At the heart of the architecture there exists a handler that initially processes the request and focuses on a single unit of work.

This handler seems to manage all incoming operations through a controlled sequence, making sure that only one task is truly active at a time while the rest are deferred or segmented into smaller mini tasks. Theoretically, each operation would yield at certain checkpoints, allowing the system to remain responsive giving the impression that multiple tasks are running in parallel when they are actually being changed constantly. It is likely that Void relies on an asynchronous loop or internal queue that schedules these tasks in rapid succession, creating a fluid and non-blocking experience. This approach would allow Void to mimic concurrency without the complexity or overhead of true parallel execution, maintaining consistency across its internal state while still appearing dynamic to the user.

When multiple commands are triggered in quick succession, Void appears to queue and resolve them in a way that maintains responsiveness while ensuring operations complete in a predictable order. Traces of sequential handling become evident when observing timing discrepancies or state updates that occur one after another rather than concurrently. This suggests the presence of an event driven architecture where tasks are orchestrated by a central loop rather than distributed across threads.

2.7 Evolvability

Void's Evolution Plan: As an open-source project, Void invites community updates, bug fixes, and improvements. Its GitHub provides contribution guidelines, a roadmap, Discord for suggestions, and issue tracking. Contributors clone the repo, install dependencies, run Void in developer mode, and live-reload to test changes. To submit work, contributors open a pull request (PR) describing changes and rationale; after review, the team may merge, test, and include it in the official distribution.

LLM Addition Evolvability.

- Centralized Capabilities Table: Traits (autocomplete, agent/gather, token limits, reasoning) live in a single registry per provider/model. Adding or updating models becomes a localized capability edit instead of feature rewrites; native Void capability updates follow the provider's distribution.
- LLM Configuration: A singleton settings service stores {provider, model}, API keys, and endpoints for direct provider access. Adding/maintaining models typically means updating this singleton, not creating new services.

- Stable IPC Pipeline: Void sends requests to providers and streams tokens back via a consistent IPC path handled by the singleton services—no per-provider plugin architecture required.
- Built-In Editor Configuration: Diff handling and agent/gather modes are centralized singletons gated by capabilities, not bespoke code. New providers mainly require configuration + capability flags; Void handles the rest.

2.8 Implications for Division of Responsibilities

Big picture. Void's VS Code fork runs with a strict Browser (renderer), and Main (Electron) split and a small set of services (chat/edit pipeline, settings, model capabilities). Responsibilities should follow those seams so the UI stays responsive and provider work remains safe.

Who Owns What?

- **Browser/UI (renderer):** This is the home for the SidebarChat, inline diffs, accept/stop buttons, and keybindings (via CommandService/Actions). It renders streamed text and triggers commands, but never talks to providers directly.
- **Chat/Edit Services (renderer):** These services run a chat or edit "turn": read voidSettingsService, check modelCapabilities, send the request over IPC, handle streaming and cancellation, and call Apply. The contract here is simple: keep streaming/cancel behavior predictable and consistent.
- **Apply Engine (renderer services):** Apply is where suggestions become edits. Fast Apply uses search/replace blocks; Slow Apply rewrites regions. Both flow through DiffZones so users see changes as they stream in. The priority is smooth refreshes and clean cancellation.
- **Providers (Main):** adapter wiring, retries/back-pressure/cancel, update modelCapabilities.
- **Config/Docs:** add settings fields once in voidSettingsService; explain capability-driven toggles.

3 Diagrams

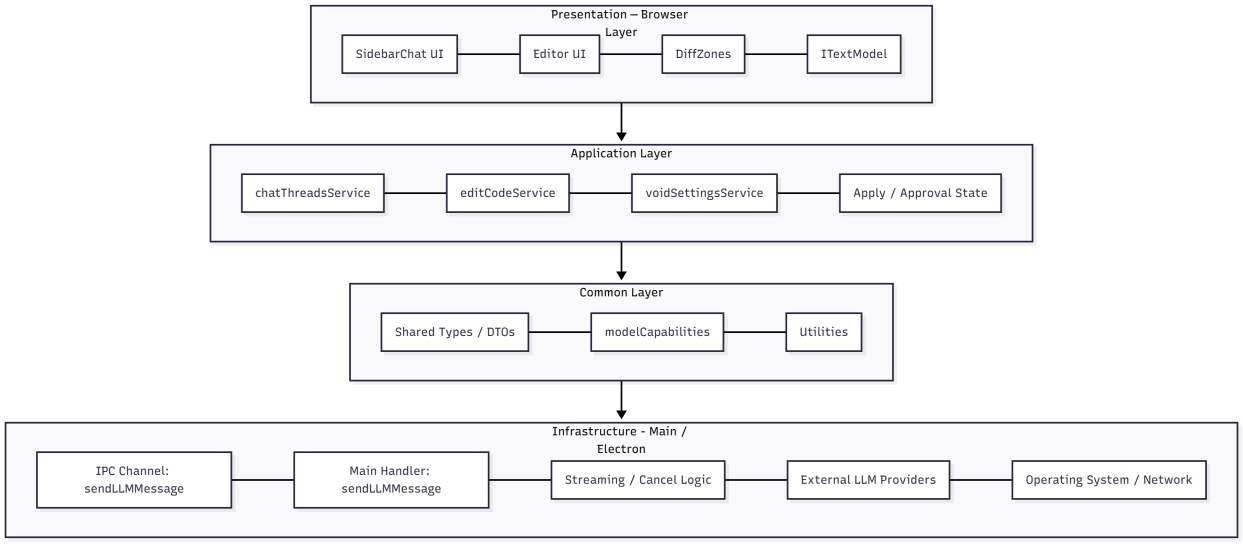

Figure 5. Layered Architecture

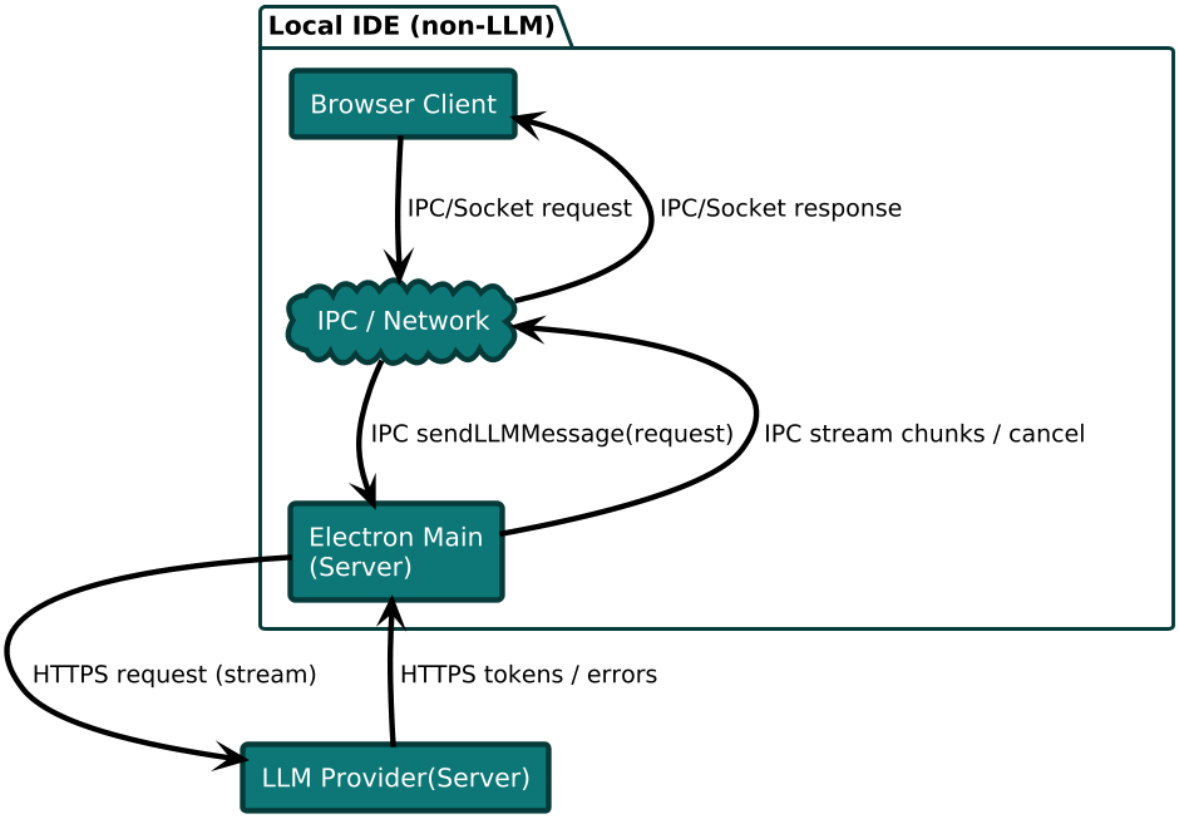

Figure 6. Client-Server Architecture

4 External Interfaces

Process responsibilities: Browser process handles everything the user sees. It shows the UI (editor, chat) and gathers what the user wants to do. Main process handles everything that needs LLM access for the Browser and OS access.

GUI Inputs/Outputs (Renderer): prompt text, stop/cancel, accept/reject edits, keybindings; outputs are streamed assistant text and diff previews.

IPC (Renderer ↔ Main):

- **LLM request (UI → Main):** "provider, model; messages (system, prior assistant/user turns) and/or prompt; sampling: temperature, top_p, max_tokens; tools / tool_choice if supported; flags: stream, vision, json_mode, etc.; thread_id, message_id for state reconciliation."
- **Streamed chunk (Main → UI):** "delta (new text); message_id / chunk_index; finish_reason when done (e.g., stop, length, cancelled); optional tool_calls / logprobs if supported."

Provider APIs (Main ↔ External): HTTPS endpoints with streaming token chunks; abort via request cancellation.

File I/O (Internal): voidModelService resolves pathname to TextModel; editCodeService writes accepted edits.

5 Use Cases

Use Case 1: Chat with Streaming and Cancel

User submits a prompt in the chat UI. chatThreadsService resolves provider/model via settings and checks capabilities. Request is sent to Main over IPC; Main calls the provider with streaming enabled. Provider streams tokens; Main forwards chunks over IPC; UI appends incrementally. If the user presses Stop, cancel propagates to Main; stream aborts promptly; UI marks the response as stopped.

See Figure 2 for the end-to-end sequence.

Use Case 2: Apply Model-Proposed Edits to File

"User reviews proposed edits and clicks Apply. CommandService calls editCodeService, which opens a DiffZone and resolves the file to a TextModel. For each edit, a fast apply (targeted search/replace) or slow apply (rewrite) is chosen. DiffZone updates to reflect success/conflict; user can accept/undo. On accept, the zone closes and the TextModel persists the change."

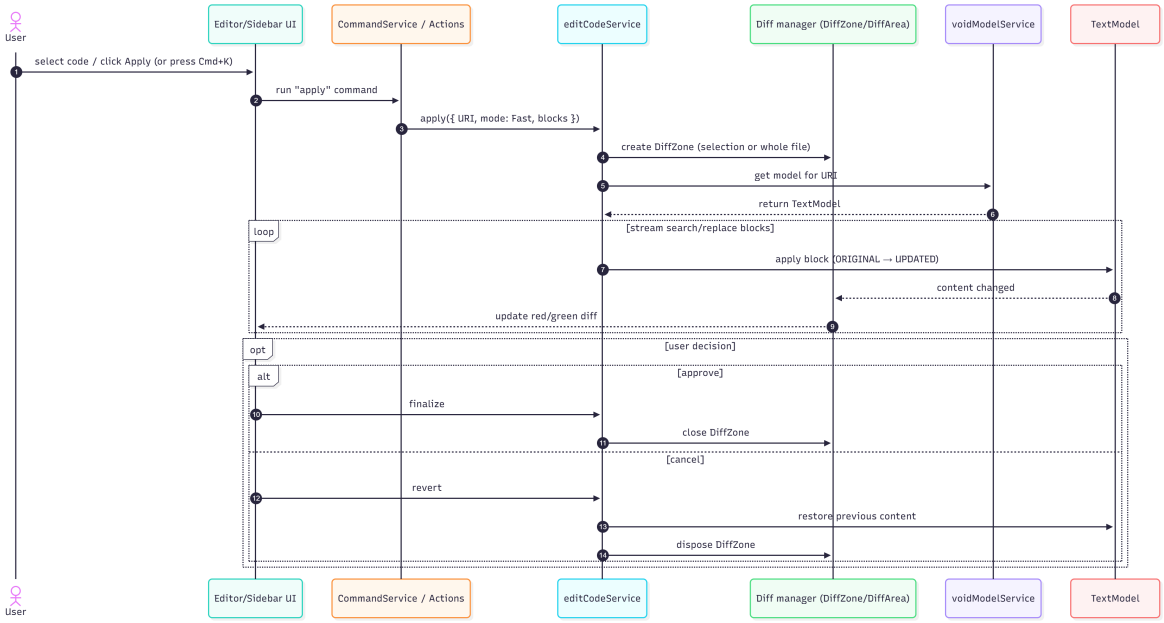

Figure 7. Use Case 2 — Fast Apply (Browser-Only)

6 Data Dictionary

Terminology:

Editor	Simply the thing you type code into, similar to Notepad. An editor can have multiple tabs with many different files open; however, it is still one editor.
Model	A representation of a file's contents within Void. Models are shared between editors, which allows for changes made across editors to sync. Each model represents a URL (path) to a file where the code is reflected.

Services This is how VS Code is organized. Services are Classes which are located and mounted a single time, also known as singletons.

Actions/Commands
These are functions registered in VSCode which implement a function within VSCode code and can be mapped to a hotkey for execution.

Inter Process Communication (IPC)
A pipeline to send information to interconnected processes.

Void Browser (UI+Commands+Services)
Everything on the browser side that prepares the request and renders the result.

LLM Main (sendLLMMessage)
Main-process handler that receives the request from Browser (via IPC), calls the external LLM with streaming enabled, and forwards streamed text/errors back to Browser.

Provider (OpenAI/Ollama/Anthropic)
External LLM endpoint that generates and streams responses.

SidebarChat UI
Chatbox object the user types into; shows the reply as it arrives.

CommandService / Actions
Central dispatcher; routes UI commands to services.

chatThreadsService (CTS)
Chat orchestrator: reads settings, validates model, sends IPC requests, receives streamed chunks, pushes them to UI.

voidSettingsService (VSS)
Stores user's model/provider selection and config.

modelCapabilities (MC)
Guard for limits/features per model before sending prompt to LLM.

sendLLMMessageChannel (IPC→Main)
Browser-to-Main bridge that carries requests and returns streamed responses.

Diff manager (DiffZone/DiffArea)
Shows red/green changes; streaming-friendly; used during Apply.

voidModelService
Resolves a file to an in-memory buffer for edits.

TextModel The in-memory representation that actually gets edited.

7 Naming Conventions

- Services use camelCase with `Service` suffix (e.g., `chatThreadsService`, `editCodeService`).
- Main-process handlers use explicit verbs (e.g., `sendLLMMessage`).
- Provider adapters named by provider (e.g., `AnthropicAdapter`).
- Registries/Singletons reflect authority (e.g., `voidSettingsService`, `modelCapabilities`).

8 Conclusions

Void maintains VS Code's two-process structure while layering an AI integration surface that is capability-driven, adapter-based, and explicitly streamed. The UI stays responsive while the main process handles provider I/O; edits are explicit and reversible via DiffZones; cancellation is first-class. Centralized capabilities and settings make provider/model changes low-friction. Main risks lie in streamed UI performance, large-file edit application, and cancellation latency, mitigated by batching, fast/slow apply paths, and abort semantics.

9 Lessons Learnt

Through this project we learned several practical lessons about working with software architecture and writing a group report.

One lesson was how difficult it can be to find clear and up-to-date documentation. We often had to piece things together from incomplete sources, which showed us the importance of careful research and double-checking before drawing conclusions. This made us value well-documented systems much more.

Another lesson was realizing that communication between components (threads, processes, and external services) is not always straightforward. Seeing how delays, errors, or cancellations behave differently across these boundaries helped us understand why architects make certain design choices.

We also learned how important group coordination is. At first, sections of our report did not match in style or detail, but regular check-ins and agreeing on common terms helped us produce a more consistent final document.

What We Would Do Differently

- Start by drafting a short outline of the architecture before splitting work.
- Keep a shared glossary so everyone uses the same terms.
- Review each other's sections earlier, instead of waiting until the end.

What We Wish We Knew Earlier

- How incomplete open-source documentation can be.
- That communication boundaries (threads vs. processes) strongly affect system behavior.
- How much effort it takes to keep group writing consistent.

10 References

References

[1] Void Editor. *voideditor/void (GitHub repository)*. https://github.com/voideditor/void/.

[2] Void Editor. *Architecture Overview*. https://zread.ai/voideditor/void/9-architecture-overview.

[3] Aditya Kumar. *Void IDE: The Comprehensive Guide to the Open-Source Cursor Alternative*. https://medium.com/@adityakumar2001/void-ide-the-comprehensive-guide-to-the-open-source-cursor-alternative-2a6b17cae235.

[4] DESOSA 2021. *Visual Studio Code — Architecture Essay*. https://2021.desosa.nl/projects/vscode/posts/essay2/.

[5] LogRocket Blog. *Advanced Electron.js Architecture*. https://blog.logrocket.com/advanced-electron-js-architecture/.

[6] Vishal Dwivedi. *Electron: Things to Watch Out For Before You Dive In*. https://medium.com/@vishaldwivedi13/electron-things-to-watch-out-for-before-you-dive-in-e1c23f77f38f.

11 AI Collaboration Report

Tool Selection: ChatGPT 5 Thinking (released August 2025), ChatGPT 5 Instant (released August 2025), DeepSeek-V3.2 (released September 2025).

Process for selecting a tool: When starting this project, we initially decided that we would purchase the premium version of an AI model if we found that it would give us a substantial benefit. After doing some research, we landed on ChatGPT. In doing some research and testing, we found ChatGPT would be a good fit because of its speed, price, reasoning capabilities and popularity, noting that Google Gemini was also a close contender. Another big draw of ChatGPT was its "deep research" capabilities, which were a huge help in getting our research off the ground (discussed later). Later on, when creating diagrams for our report, we found ChatGPT could be inaccurate and frustrating to work with in helping to learn and fix Mermaid code diagrams. After some research and testing, we found DeepSeek to be surprisingly fast, helpful and accurate in this regard; and as such, this became our resource for diagram assistance.

Tasks Delegated To AI Team Members:

- **Resource Aggregation:** By prompting ChatGPT with deep research mode to summarize a specific topic we were tasked to write about, or ask a semi-specific question about something we were going to write about and emphasizing sourcing for all of its claims, we were able to find incredibly valuable resources. While we quickly came to find out that we could not trust AI summaries or descriptions of topics, we did find it very effective at aggregating incredibly useful websites, which we could read and use for research. Moreover, for each claim ChatGPT made in its summary, we made it provide a source which we could read, allowing for incredibly efficient research for more specific topics using a more general query.

- **Document Navigation:** We found a couple of extensive pages published by Void discussing its architecture, which was a very comprehensive and useful read; however, it was quite long. After reading the whole document, we would have a topic we would want to write about for a certain section, and we could ask ChatGPT something like "in this document, it mentions 'voidSettignSerivces,' aggregate all sections in which this is discussed so I can easily navigate to them via the context in which it is used." A prompt like this allowed us to zero in on very specific things we knew we wanted to discuss, and then report on that topic using the various contexts in which it was mentioned.

- **Mermaid Code Assistant:** Mermaid Code is a useful tool for diagram creation; however, our members in charge of diagrams were not incredibly familiar with it. DeepSeek proved to be a very useful tool in guiding, fixing and adding to our Mermaid Code projects.

- **Topic Understanding:** In some of our research, we encountered ideas, phrases, or terms with which we were unfamiliar. ChatGPT was a huge aid in filling in these knowledge gaps, especially because we could ask something like 'what is a singleton, and how is it used in the context of this documentation?'

- **LaTeX Formatting:** ChatGPT was hugely helpful in report formatting.

- **Report Shortening:** Initially the report went over the 17-page limit, and AI was very useful as we could provide it with a paragraph or section and help us identify ways we could cut down on size; either by removing unnecessary verbiage, paraphrasing a couple sentences into one, or identifying other identifying parts of the report which included already explained information among other methods.

Interaction Protocol and Prompting Strategy: Our group designated a primary researcher responsible for managing AI-assisted research throughout Assignment 1. This member handled the broader interactions with ChatGPT (GPT-5) and used its deep research capabilities to find credible sources such as the Void

GitHub repository and the Zread.ai architecture overview. Other members contributed by suggesting keywords, refining queries, and diving into sources once they were collected. This structure gave us a consistent research direction while still keeping the process collaborative. One of the more complex prompts crafted by the primary researcher focused on how Void's architecture supports evolution. Early drafts were too broad, so the prompt was iteratively refined to include role guidance ("respond as a software architecture analyst"), formatting requirements ("organize the services into a chart and give direct sourcing for each"), and clarity constraints ("ignore incredibly specific parts of the codebase, keep the sourcing and information in the context of conceptual architecture"). Each iteration produced clearer, more structured insights that had useful information and, more importantly, direct sources that were used to validate them section by section.

Validation and Quality Control Procedures: In using AI for this Assignment, we quickly found that most output beyond a brief, specific question could not be trusted at face value. A prompt would yield a very confident and reasonable-sounding output, but after fact-checking, we would find that ChatGPT might give a mostly accurate answer with some details either contradicted by or not at all mentioned in any of the sources it used, meaning they couldn't be verified. After these findings, we decided as a team that all information provided by AI beyond quick questions or a few-paragraph summaries of user-provided text had to be carefully fact-checked. A great efficiency we found for this process was prompting ChatGPT to source each claim, allowing us to efficiently go directly to the source and either validate ChatGPT's answers or simply ignore it and use the provided source instead.

Quantitative Contribution to Final Deliverable: We believe that AI contribute to approximately 12% of our final deliverables, broken down as follows: 1% Mermaid Code, 1% LaTeX, 10% Research.

Reflection on Human-AI Team Dynamics: Overall, we found ChatGPT and DeepSeek incredibly useful for our group's research phase of the assignment. Their involvement was a big time saver when it came to finding useful resources and pinpointing where in those resources specific information could be found. When it came to decision-making, it did not have much involvement as our team developed a very functional and productive process. ChatGPT's involvement led to some disagreements in our group as it pertained to fact-checking. A group member may read over another's work and say 'Where did you get that claim?' and we had disagreements over whether "ChatGPT got it from here" was a good answer; eventually, we decided as a team it was not. Finally, as a group, we came to understand that AI was an incredibly useful resource for research and source gathering; however, specific claims and generalizations could seldom be blindly trusted.