Void AI — Concrete Architecture Report (CISC 322/326 A2)

Due: Nov 7th, 2025

Group 23: We'll Fix It in Post
Jaivik Joshi - jaivik.joshi@queensu.ca
Zack Wood - 22RQLJ@queensu.ca
Robin Houiler - 22BT35@queensu.ca
Charlie Kevill - 21cmk11@queensu.ca
Kai Maddocks - 25xx12@queensu.ca
Jared Simon - 22WGK@queensu.ca

Abstract

This report presents the concrete architecture of VOID, an AI-assisted IDE built on a VS Code fork. We derived the structure by clustering the VOID contribution under src/vs/workbench/contrib/void into subsystems and validating their interactions with command handlers, webview bridges, and dependency views. The high-level diagram (Figure 2) shows ten subsystems organized around a webview-to-workbench bridge, an orchestration core, a provider gateway/router, and a patch/diff pipeline. We provide a subsystem deep-dive of the Agent Orchestrator (Figure 4) and two end-to-end use cases that label concrete control and data edges. Our Reflexion analysis contrasts this concrete view with our A1 conceptual model, explaining splits (e.g., Provider Router vs Patch Engine), cross-cuts (Telemetry), and explicit framework boundaries to the VS Code host.

Introduction and Overview

VOID is an open-source, AI-augmented editor that retains VS Code's extension model while adding chat, agent/gather modes, and checkpoints. Its codebase guide and issues indicate most VOID-specific workbench features live under src/vs/workbench/contrib/void. We treat the VS Code workbench/extension host as an external black box and model VOID's contributions around it.

Constraints: VS Code/Electron internals are black-boxed; we only draw framework edges where VOID calls or is called via workbench services or the webview bridge. Provider backends (OpenAI/Anthropic/Ollama/etc.) are treated as external endpoints behind a gateway/router.

Recap of Conceptual Architecture

Our repaired A1 model organizes the system into four parts: Electron Main, the Renderer/Workbench, the Extension Host, and a Void subsystem that spans the last two. Control flows through events. The Renderer handles the UI; the Extension Host manages editor semantics; and Main performs privileged operations. Within this frame, the Void subsystem adds an AI chat surface, an orchestrator that translates user intent into actions, provider adapters for different models, a checkpoint trail that records edits, and a rules layer that gates agent behavior (Agent vs. Gather).

Derivation Process

We examined the VOID repo and contribution text to confirm where custom features live, discovering most of Void's code lives in the folder src/vs/workbench/contrib/void/; issues also reference .../browser/ and .../browser/react/ under that path (e.g., build.js). This guided grouping in our dependency views and informed where to search for webview plumbing and React UI components.

Mapping Folders → subsystems:

.../void/browser → UI/webview panels and diff/review surfaces; .../void/common → services (chat orchestration, context building, provider routing, patch engine, checkpoints, telemetry); .../void/node or .../electron-sandbox → network/filesystem/IPC glue if present.

Mapping Symbols/strings → edges:

We searched identifiers and message strings like sendLLMMessage (fig. 1), applyPatch, buildContext, postMessage, onDidReceiveMessage, workspaceEdit to label sequence edges with concrete names.

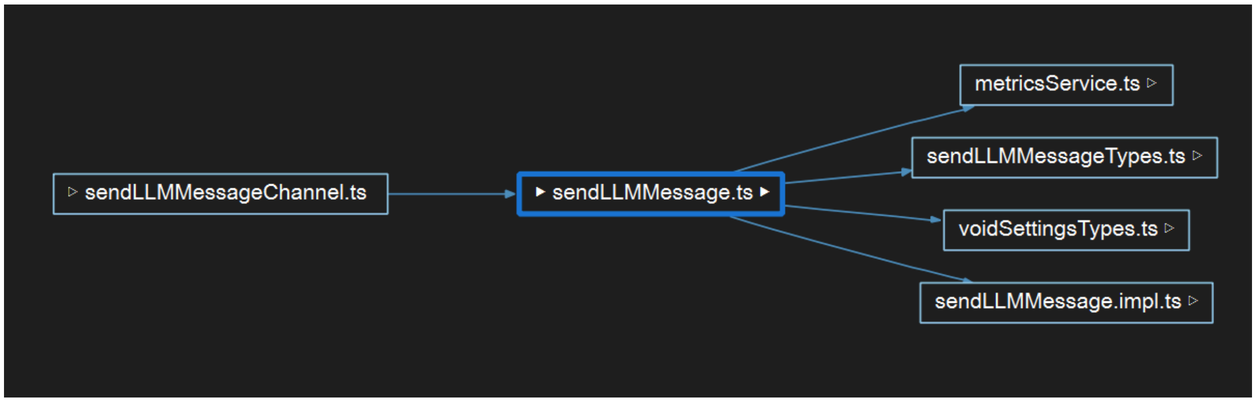

Figure 1: sendLLMMessage on Understand

Unexpected edges & validation:

Any edge not forecast in A1 was validated by checking call sites and command registrations in the workbench contribution files, then either kept as a true dependency or suppressed if it was purely incidental. Iterating from A1. Where A1 grouped things broadly (e.g., "AI Bridge / Model Connector" and "editCodeService"), A2 splits them into Provider Gateway/Router and a Patch/Code-Action Engine because those are distinct responsibilities in the code path.

Top-Level Concrete Architecture

VOID follows an evented workbench contribution style with a webview bridge for UI and a request/response flow to providers, plus a pipeline for patch/diff/apply. The VS Code host is external; our diagram shows clearly labeled framework edges.“VOID operates across three cooperating runtimes: Main (privileged OS gateway), Renderer (workbench UI and VOID webviews), and Extension Host (command handlers and editor APIs). A webview interaction becomes a command; the handler gathers context, invokes providers and tools, and applies edits directly or delegates long-running/privileged work to Main. Streaming keeps the UI responsive.”

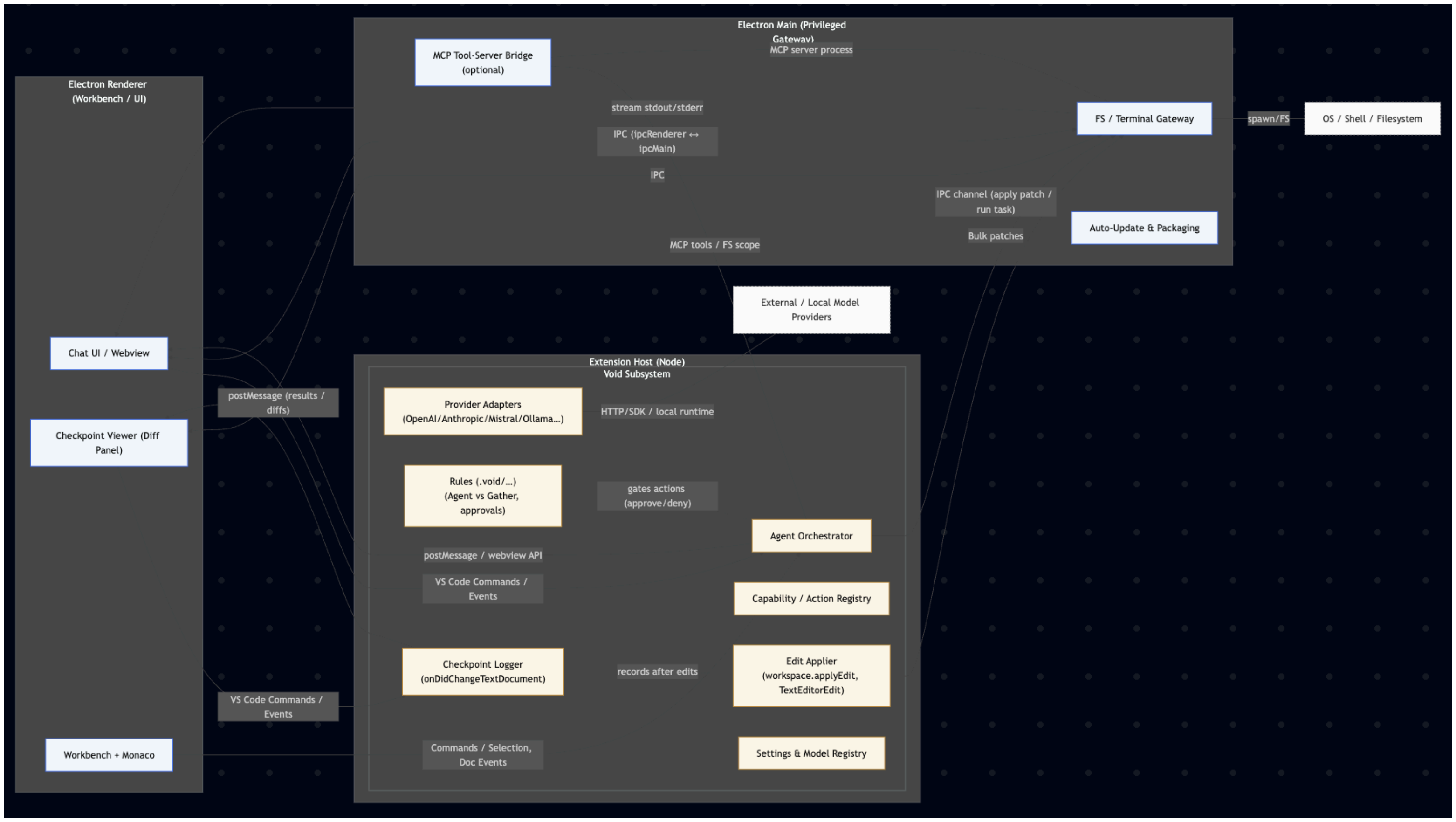
Electron Main (Privileged Gateway)
MCP Tool-Server Bridge (optional)
MCP server process
stream stdout/stderr
IPC (ipcRenderer ↔ ipcMain)
IPC
FS / Terminal Gateway
spawn/FS
OS / Shell / Filesystem
IPC channel (apply patch / run task)
Bulk patches
Auto-Update & Packaging
MCP tools / FS scope
Electron Renderer (Workbench / UI)
Chat UI / Webview
Checkpoint Viewer (Diff Panel)
Workbench + Monaco
postMessage (results / diffs)
VS Code Commands / Events
External / Local Model Providers
Extension Host (Node)
Void Subsystem
Provider Adapters (OpenAI/Anthropic/Mistral/Ollama...)
HTTP/SDK / local runtime
Rules (.void/...) (Agent vs Gather, approvals)
gates actions (approve/deny)
Agent Orchestrator
postMessage / webview API
VS Code Commands / Events
Capability / Action Registry
Checkpoint Logger (onDidChangeTextDocument)
records after edits
Edit Applier (workspace.applyEdit, TextEditorEdit)
Commands / Selection, Doc Events
Settings & Model Registry

Figure 2: Top-Level Concrete Architecture

Chat / Prompt UI

This subsystem provides the visible interface through which users interact with the AI assistant. It handles user input (text prompts or natural language commands), displays streaming responses, and presents inline code edits or summaries. The Chat UI focuses on interaction and clarity, ensuring smooth communication between the user and the underlying orchestration logic.

Chat Orchestrator

Acting as the central coordinator, this subsystem manages the full lifecycle of a chat session. It receives input from the UI, gathers relevant project context, forwards requests to AI providers, and manages the response flow back to the user. Its main responsibility is to sequence and synchronize the many moving parts involved in an AI-assisted edit—from request creation to result application.

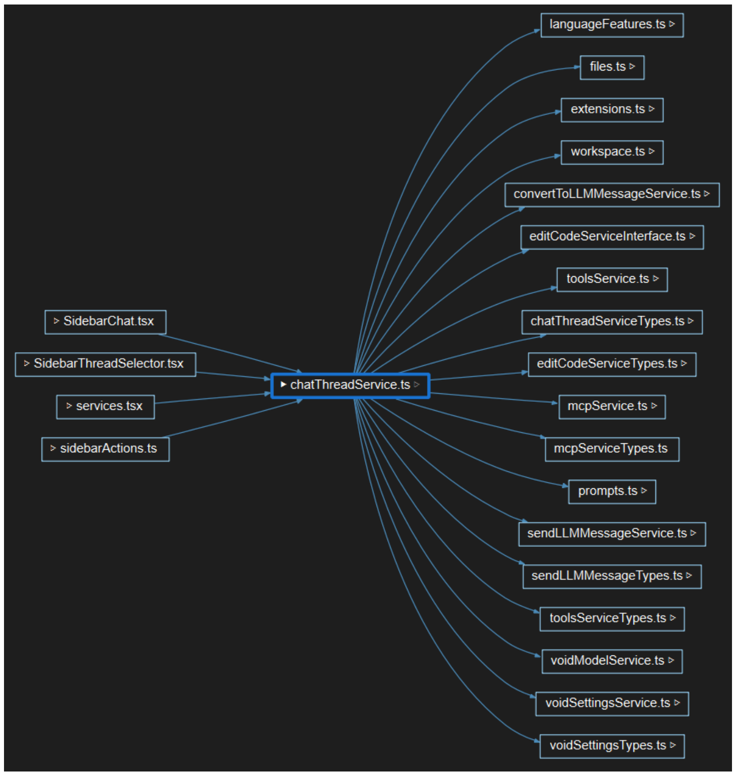

Figure 3: chatThreadService on Understand (Showcases the two chat subsystems above)

Context and Selection Builder

This subsystem is responsible for understanding the user's working context within the editor. When a user highlights code or positions the cursor, it gathers nearby code, metadata, and file information to form a meaningful “context package.” This context allows the AI to reason about structure, dependencies, and intent—producing more accurate suggestions.

LLM Gateway / Provider Router

This subsystem manages communication with external AI models. It abstracts the details of connecting to one or more large language models, routing requests to the appropriate provider, and handling streamed or batched responses. It ensures that the orchestrator can interact with models in a uniform way, even if the underlying APIs differ.

Patch and Code-Action Engine

Once the AI produces a suggested edit or transformation, this subsystem interprets that output into actionable code modifications. It creates patches, verifies syntax or formatting, and applies them safely to the project. Its main responsibility is transforming model output into concrete, reversible code changes, supporting features like preview, undo, and fine-grained acceptance.

Diff and Review UI

The Diff and Review subsystem presents proposed changes to the user for inspection before they are finalized. It highlights additions and deletions side by side, allowing developers to decide which changes to apply. This subsystem emphasizes human-in-the-loop control, bridging automation and manual oversight.

Checkpoints and Persistence

This subsystem handles saving, restoring, and versioning chat sessions, context data, and file states. It ensures that users can roll back or revisit previous interactions with the AI, supporting VOID’s focus on reliability and reproducibility. By storing incremental states, it also enables features like “checkpointing” before code is modified.

Agent and Tool Service

This layer is responsible for tool invocation and agent behavior—tasks like running tests, searching files, or gathering system information when requested by the AI. It provides controlled access to development tools, ensuring that automated tasks occur safely and within the project’s environment.

Telemetry and Monitoring

Telemetry collects performance and usage data across VOID’s components. Its responsibility is to record prompt latency, model response quality, error frequency, and general activity metrics.

This enables developers to evaluate the effectiveness of AI features and refine their operation over time.

Editor Integration (Workbench Bridge)

The final layer connects VOID's custom logic with the underlying VS Code framework. It registers commands, applies workspace edits, opens files, and handles all user-facing editor operations. This bridge ensures that VOID's AI-driven features behave as native VS Code actions, blending seamlessly into the editor environment.

Subsystem Analysis: Agent Orchestrator

The Agent Orchestrator mediates between conversation and concrete actions. It plans steps, checks capabilities and policy, executes tools, and commits edits with checkpoints and user approvals. In the concrete view it stretches across six parts. The Chat/Thread UI (webview) manages the conversation and asks for approval when a tool wants to act. The Orchestrator (Extension Host) turns requests into steps and chooses a strategy (autocomplete, single edit, or multi‑step agent). Provider adapters talk to hosted or local models. File and terminal tools perform edits and subprocess work either directly via VS Code APIs or through the Main process via IPC. Checkpoints log changes as they happen and present them back to the user. Rules enforce modes (Gather vs Agent) and other policies.

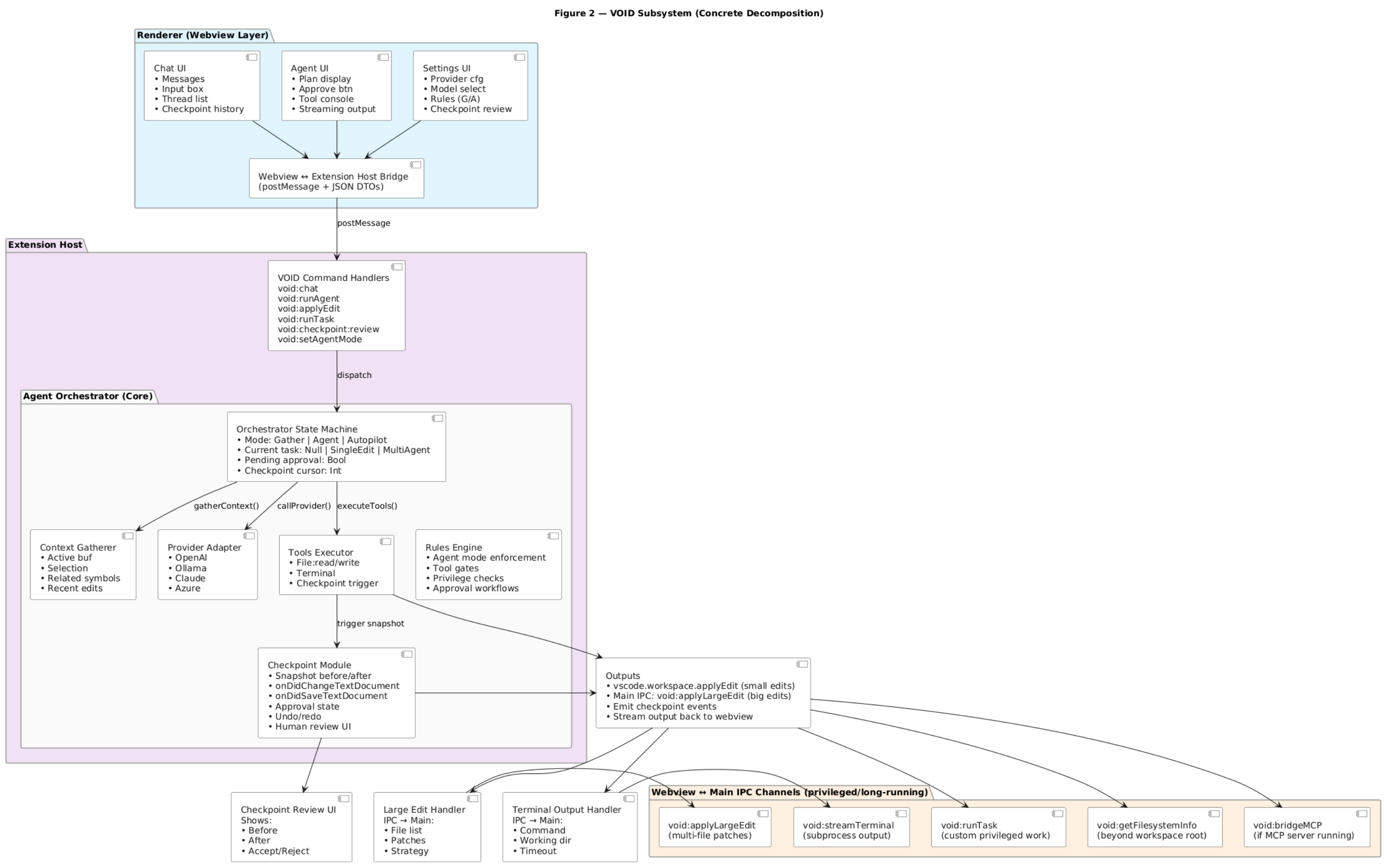
Figure 2 — VOID Subsystem (Concrete Decomposition)
Renderer (Webview Layer)
Chat UI • Messages • Input box • Thread list • Checkpoint history
Agent UI • Plan display • Approve btn • Tool console • Streaming output
Settings UI • Provider cfg • Model select • Rules (G/A) • Checkpoint review
Webview ↔ Extension Host Bridge (postMessage + JSON DTOs)
postMessage
Extension Host
VOID Command Handlers void:chat void:runAgent void:applyEdit void:runTask void:checkpoint:review void:setAgentMode
dispatch
Agent Orchestrator (Core)
Orchestrator State Machine • Mode: Gather | Agent | Autopilot • Current task: Null | SingleEdit | MultiAgent • Pending approval: Bool • Checkpoint cursor: Int
gatherContext()
callProvider()
executeTools()
Context Gatherer • Active buf • Selection • Related symbols • Recent edits
Provider Adapter • OpenAI • Ollama • Claude • Azure
Tools Executor • File:read/write • Terminal • Checkpoint trigger
Rules Engine • Agent mode enforcement • Tool gates • Privilege checks • Approval workflows
trigger snapshot
Checkpoint Module • Snapshot before/after • onDidChangeTextDocument • onDidSaveTextDocument • Approval state • Undo/redo • Human review UI
Outputs • vscode.workspace.applyEdit (small edits) • Main IPC: void:applyLargeEdit (big edits) • Emit checkpoint events • Stream output back to webview
Checkpoint Review UI Shows: • Before • After • Accept/Reject
Large Edit Handler IPC → Main: • File list • Patches • Strategy
Terminal Output Handler IPC → Main: • Command • Working dir • Timeout
Webview ↔ Main IPC Channels (privileged/long-running)
void:applyLargeEdit (multi-file patches)
void:streamTerminal (subprocess output)
void:runTask (custom privileged work)
void:getFilesystemInfo (beyond workspace root)
void:bridgeMCP (if MCP server running)

Figure 4: VOID Subsystem Concrete Decomposition

Orchestrator State Machine

Modes: Gather, Agent, Autopilot, and MultiAgent.

Responsibilities: owns the active run; advances steps; pauses for approval; cancels on user request; and maintains the "current checkpoint."
Key actions: gatherContext(), applyPatch(), executeTool().

Context Gatherer

Builds task-specific context: active selection, relevant files, recent edits, and symbol hits gathered earlier.

Outputs compact payloads that downstream steps can use (for the provider, tools, or patching).

Provider Adapter

Turns orchestrator requests into model prompts and reads structured tool-call suggestions back.

Abstracts provider differences (e.g., how streaming is delivered, token limits).

Tools Executor

Runs the concrete actions: read/write files, execute terminal commands, or trigger checkpoints.

Streams terminal and task output incrementally so the user and model can adapt mid-run.

Rules Engine

Safety net for every step: policy enforcement (read-only paths, denied tools), throttling, privilege checks, and approval workflows.
Short-circuits any step that violates policy and sends a human-readable refusal upstream.

Checkpoint Module

On risky or multi-file steps, takes a snapshot before and after.

Keeps a list of checkpoints, provides quick revert, and links to the diff view so the user can inspect what changed.

Some of our A1 boundaries hold: UI stays in the webview; edits and orchestration stay in the Extension Host; provider specifics remain behind adapters. The concrete differences are pragmatic. Some provider calls start on the webview side for latency reasons. Long-running tasks consistently route through Main, which is not just an optimization but a stability concern.

Reflexion Analysis at a High-level (System)

Conceptual Edge	Type	Rationale
Renderer → Extension Host	Convergent	UI uses commands to reach EH; no direct cross-process function calls.
Renderer → Main	Divergent	Renderer uses IPC to Main for privileged/long-running work to keep UI responsive.
Extension Host ← Editor changes	Convergent	Edits and checkpoints use workspace APIs as assumed.

Checkpoint approval workflow (A1 vague)	Divergent	Two steps: snapshot → user approval → commit.
Tools via Main process (absent in A1)	Divergent	Subprocess and filesystem work run in Main for privilege and stability.
Provider call latency optimization	Divergent	Some calls start from Renderer for lower latency; not only via EH.
Checkpoint UI location	Convergent	UI in Renderer; control/state handled in EH.
Workspace edits (small vs large)	Convergent	Small edits via workspace API; multi-file edits via privileged channels.
EH as orchestrator	Convergent	EH owns orchestration; Renderer initiates but does not coordinate runs.
"AI Bridge" (single) → Provider Gateway + Router	Split	Provider selection and invocation/streaming are separate concerns.
EditCodeService (single) → Patch Engine + Diff/Review + Apply	Split	Reviewability and rollback require distinct parts.
Telemetry/Monitoring edges (not explicit in A1)	Cross-cut	Timing, errors, and usage recorded across subsystems.
Policy/approval enforcement (implicit in A1)	Cross-cut	Rules and approvals applied around tool execution and patching.
Webview bridge boundary	Convergent	Webview ↔ EH boundary is explicit and consistent with A1 intent.

Reflexion Analysis at the Second-level (Agent Orchestrator)

A1 Element	A2 Concrete	Type	Rationale
"toolService" (single)	Capability Resolver + Execution Engine	Split	Capability checks and execution are separated.

“gather/agent mode”	State Machine (Gather/Agent/Autopilot/MultiAgent)	Divergent	Explicit states enable approval, retry, and pause/resume.
Tool output → model (opaque)	Observation Adapter	Divergent	Normalizes logs/snippets for model and UI.
Direct apply of agent edits	Checkpoint before + patch pipeline	Convergent	Snapshot, then apply; supports rollback.
File/terminal via EH	Privileged IPC to Main	Divergent	Long-running or privileged tasks go through Main.
Model chooses tools freely	Rules Engine	Cross-cut	Policy, throttling, and approvals enforced.
One-step agent call	Planned multi-step run	Divergent	Steps have preconditions, timeouts, and rollbacks.
Renderer orchestrates	EH orchestrates; Renderer provides UI & approvals	Convergent	Clear split between control and presentation.

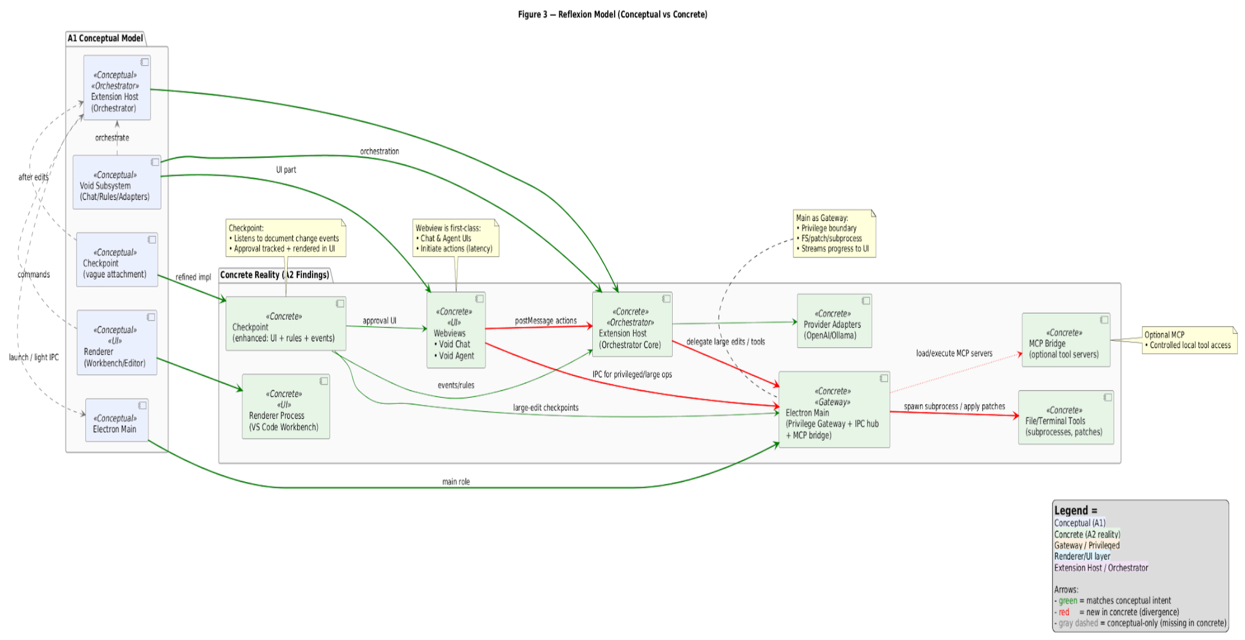
Figure 3 — Reflexion Model (Conceptual vs Concrete)
A1 Conceptual Model
«Conceptual» «Orchestrator» Extension Host (Orchestrator)
orchestrate
«Conceptual» Void Subsystem (Chat/Rules/Adapters)
«Conceptual» Checkpoint (vague attachment)
«Conceptual» «UI» Renderer (Workbench/Editor)
«Conceptual» Electron Main
after edits
commands
launch / light IPC
orchestration
UI part
refined impl
Checkpoint:
• Listens to document change events
• Approval tracked + rendered in UI
Webview is first-class:
• Chat & Agent UIs
• Initiate actions (latency)
Concrete Reality (A2 Findings)
«Concrete» Checkpoint (enhanced: UI + rules + events)
approval UI
«Concrete» «UI» Webviews • Void Chat • Void Agent
postMessage actions
«Concrete» «Orchestrator» Extension Host (Orchestrator Core)
«Concrete» «UI» Renderer Process (VS Code Workbench)
events/rules
large-edit checkpoints
IPC for privileged/large ops
delegate large edits / tools
Main as Gateway:
• Privilege boundary
• FS/patch/subprocess
• Streams progress to UI
«Concrete» Provider Adapters (OpenAI/Ollama)
«Concrete» «Gateway» Electron Main (Privilege Gateway + IPC hub + MCP bridge)
load/execute MCP servers
spawn subprocess / apply patches
«Concrete» MCP Bridge (optional tool servers)
Optional MCP
• Controlled local tool access
«Concrete» File/Terminal Tools (subprocesses, patches)
main role
Legend =
Conceptual (A1)
Concrete (A2 reality)
Gateway / Privileged
Renderer/UI layer
Extension Host / Orchestrator
Arrows:
- green = matches conceptual intent
- red = new in concrete (divergence)
- gray dashed = conceptual-only (missing in concrete)

Figure 5: Reflexion Model (Conceptual vs Concrete)

Interactions of the Agent Orchestrator (Use Cases –figs below)

Apply AI Edit with Checkpoint. The user asks the agent to improve a function. The webview submits a command to the Extension Host; the Extension Host handler gathers the current buffer and selection, calls a provider adapter, and receives a suggested patch. If it is small, the handler constructs a WorkspaceEdit and calls vscode.workspace.applyEdit; the onDidChangeTextDocument event fires and the checkpoint module records the snapshot. If the change is large or spans many files, the orchestrator dispatches via the privileged IPC channel to Main—via an IPC channel dedicated to patches—to apply it and to stream progress back for the chat.

Run Terminal Tool with Approval. The agent proposes to run a formatter. The chat panel shows the plan and the user approves. The webview invokes a void:runTask channel; Main spawns the subprocess and streams its output back. The Renderer forwards the stream into the chat via a VS Code command so the Extension Host can fold the transcript into the conversation state. If the tool created changes, the agent summarizes and offers to apply them, which leads back to the first flow and a new checkpoint.

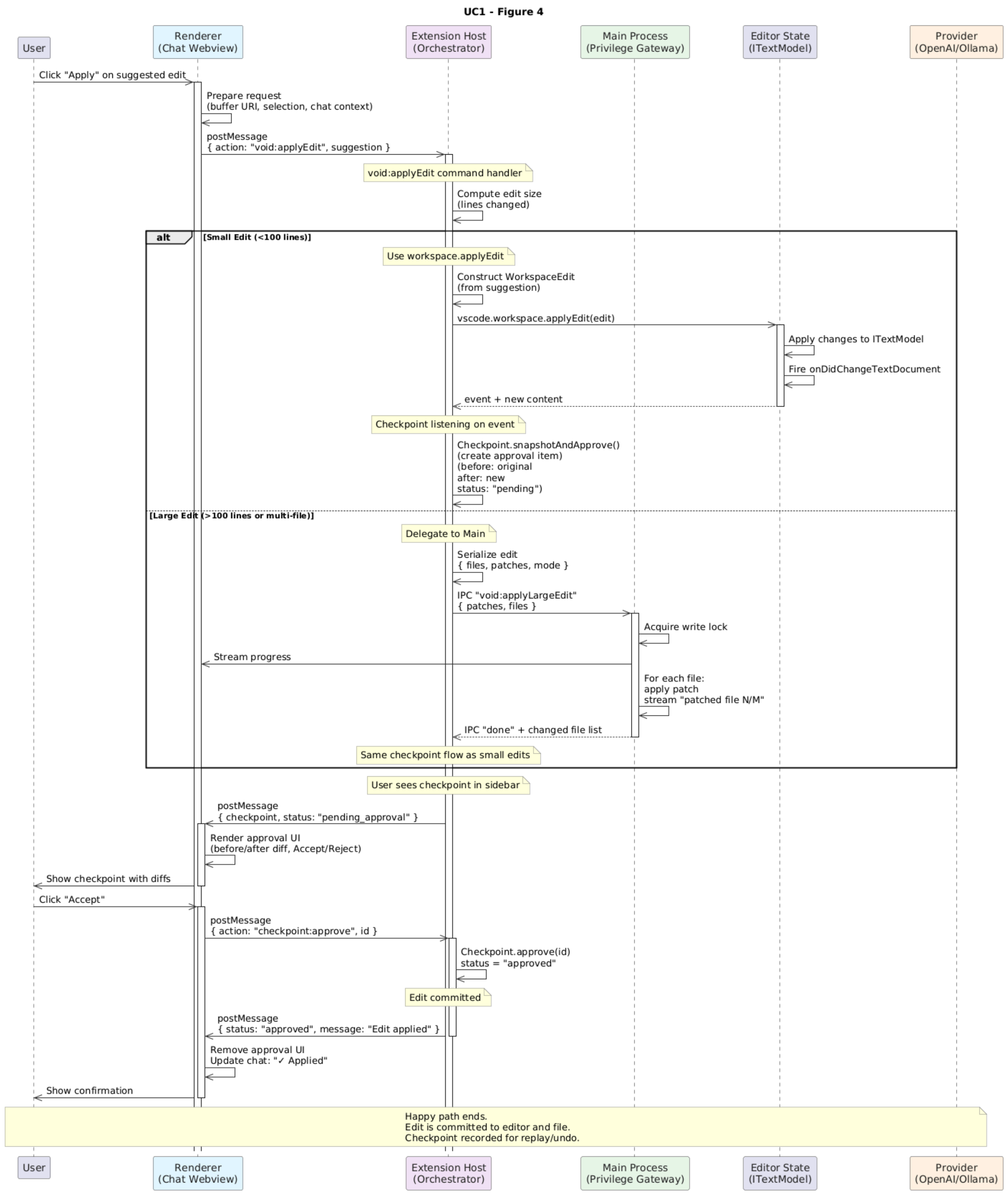
UC1 - Figure 4
User
Renderer (Chat Webview)
Extension Host (Orchestrator)
Main Process (Privilege Gateway)
Editor State (ITextModel)
Provider (OpenAI/Ollama)
Click "Apply" on suggested edit
Prepare request (buffer URI, selection, chat context)
postMessage { action: "void:applyEdit", suggestion }
void:applyEdit command handler
Compute edit size (lines changed)
alt
[Small Edit (<100 lines)]
Use workspace.applyEdit
Construct WorkspaceEdit (from suggestion)
vscode.workspace.applyEdit(edit)
Apply changes to ITextModel
Fire onDidChangeTextDocument
event + new content
Checkpoint listening on event
Checkpoint.snapshotAndApprove() (create approval item) (before: original after: new status: "pending")
[Large Edit (>100 lines or multi-file)]
Delegate to Main
Serialize edit { files, patches, mode }
IPC "void:applyLargeEdit" { patches, files }
Acquire write lock
Stream progress
For each file: apply patch stream "patched file N/M"
IPC "done" + changed file list
Same checkpoint flow as small edits
User sees checkpoint in sidebar
postMessage { checkpoint, status: "pending_approval" }
Render approval UI (before/after diff, Accept/Reject)
Show checkpoint with diffs
Click "Accept"
postMessage { action: "checkpoint:approve", id }
Checkpoint.approve(id) status = "approved"
Edit committed
postMessage { status: "approved", message: "Edit applied" }
Remove approval UI Update chat: "✓ Applied"
Show confirmation
Happy path ends. Edit is committed to editor and file. Checkpoint recorded for replay/undo.

Figure 6: Sequence Diagram A — Apply AI Edit with Checkpoint —

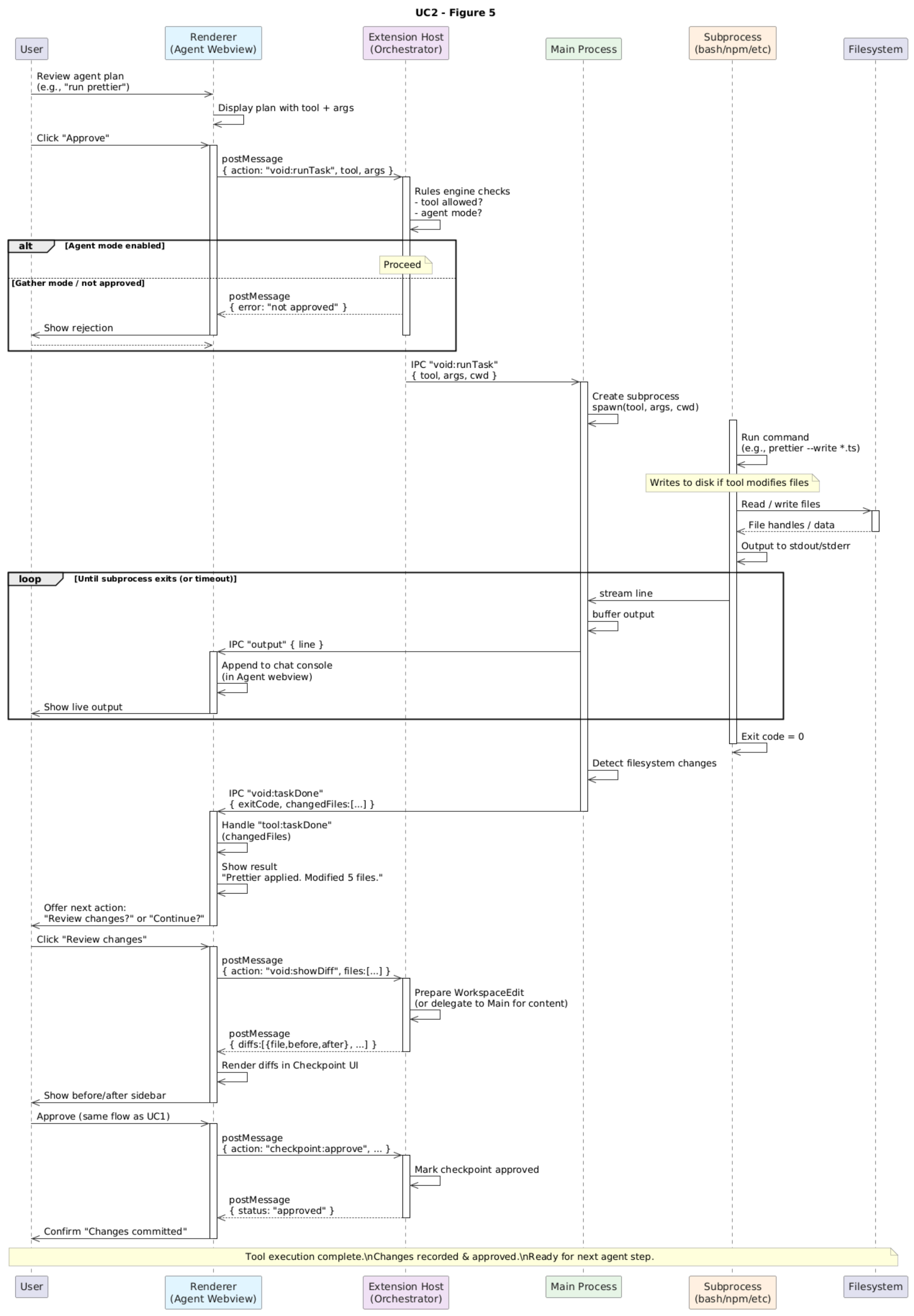
UC2 - Figure 5
User
Renderer (Agent Webview)
Extension Host (Orchestrator)
Main Process
Subprocess (bash/npm/etc)
Filesystem
Review agent plan (e.g., "run prettier")
Display plan with tool + args
Click "Approve"
postMessage { action: "void:runTask", tool, args }
Rules engine checks - tool allowed? - agent mode?
alt [Agent mode enabled]
Proceed
[Gather mode / not approved]
postMessage { error: "not approved" }
Show rejection
IPC "void:runTask" { tool, args, cwd }
Create subprocess spawn(tool, args, cwd)
Run command (e.g., prettier --write *.ts)
Writes to disk if tool modifies files
Read / write files
File handles / data
Output to stdout/stderr
loop [Until subprocess exits (or timeout)]
stream line
buffer output
IPC "output" { line }
Append to chat console (in Agent webview)
Show live output
Exit code = 0
Detect filesystem changes
IPC "void:taskDone" { exitCode, changedFiles:[...] }
Handle "tool:taskDone" (changedFiles)
Show result "Prettier applied. Modified 5 files."
Offer next action: "Review changes?" or "Continue?"
Click "Review changes"
postMessage { action: "void:showDiff", files:[...] }
Prepare WorkspaceEdit (or delegate to Main for content)
postMessage { diffs:[{file,before,after}, ...] }
Render diffs in Checkpoint UI
Show before/after sidebar
Approve (same flow as UC1)
postMessage { action: "checkpoint:approve", ... }
Mark checkpoint approved
postMessage { status: "approved" }
Confirm "Changes committed"
Tool execution complete.\nChanges recorded & approved.\nReady for next agent step.

Figure 7: Sequence Diagram B — Run Terminal Tool with Approval —

Alternative Architecture

We considered two alternatives. A pure extension‑only approach would avoid Renderer↔Main IPC for most actions and push everything through the Extension Host. That would simplify some reasoning but would give up rich chat UIs and make terminal/FS interactions awkward or blocked entirely. At the other end, an all‑in‑Main custom desktop app without the Extension Host would give us full control but at the cost of the VS Code ecosystem, safety isolation, and established APIs. The current hybrid, where Void is an opinionated extension suite plus rich webviews with carefully bounded Main gateways, remains the most sensible choice.

AI Collaboration Report

We brought in GPT-5 only after the report and figures were done, using it as a careful proofreader—not a co-author. It's good at following tight instructions, handling long docs without losing context, and keeping tone consistent. With narrow prompts, it did small, mechanical fixes—grammar, tense, concise phrasing, figure numbering/legends, and reference cleanup—without inventing content or changing our claims. Example prompts: "Fix grammar and punctuation; do not change meaning. Keep terminology: VOID, Reflexion, Agent Orchestrator." "Standardize casing and terms; list conflicts found." "Check figure numbering and cross-references; propose minimal edits." "Tighten these paragraphs without dropping information." "Convert this list to a 3-column table without adding claims." In this manner it worked beneficially, as what we presumed to be well written or formatted had been replaced by a more academic, and report fitting writing style.

Every suggestion went through a diff review. We rejected anything that shifted meaning and re-read lines tied to figures or tables. A short terminology list (Agent Orchestrator, Provider Router, Patch Engine, Reflexion) kept names aligned. The pass caught minor language issues, a couple of figure renumberings, a missing reference index, and a few term mismatches—saving time on proofreading—while leaving the architecture, diagrams, and conclusions fully ours. It did really want to work on more than it should have, looking into our details and suggesting changes in there, yet problems resided as proof checks would show how GPT-5 made mistakes and assumed details where it shouldn't have.

We also used GPT-5 to tidy up the reflexion analysis so it reads cleanly and matches the figures. It didn't invent edges or conclusions, specifically it suggested clearer axis/tick labels and shorter node names so the graphs fit on the page without overlap; grouped edges by Convergent / Divergent / Split / Cross-cut, then ordered rows for scanability (system-level first, Agent Orchestrator next).

GPT-5 turned long paragraphs into two compact pieces: (1) a high-level table for system reflexion, and (2) a second-level table for the Agent Orchestrator. It kept our wording but trimmed it, added consistent column headers (*Conceptual*, *Concrete*, *Type*, *Rationale*), and removed duplicates. We reviewed all changes in diff view and rejected anything that altered meaning. The result is the same analysis, just easier to read over, maintaining meaning as well as the style of writing that a report should encompass.

Conclusion

The concrete architecture shows that the Main, Renderer, and Extension Host boundaries define how features are built. Once we mapped VOID across those three parts, the differences we first saw looked intentional, they follow how the platform is meant to be used. The Agent Orchestrator fits cleanly into this picture: it's a small set of pieces connected by commands, events, and IPC. A simple gateway centralizes file/terminal work and checkpoint logging, so the flow is easier to follow and safer to change over time.

Lessons Learned and Team Notes

We learned that where things run matters as much as what they do. Treating the webview as a real component and Main as a gateway made several design choices fall into place. We also stopped drawing edges that jump over the platform: if we can't explain a flow with registerCommand, workspace.applyEdit, onDidChangeTextDocument, and an IPC channel, we don't understand it yet. Finally, keeping the UI responsive isn't just a performance tweak—it's an architectural rule. Long-running work belongs in the right process, with explicit streams sending progress back to the user.

Limitations & Threats to Validity

- **Snapshot & version drift.** Findings reflect one repo snapshot; later commits can change boundaries and edges.

- **Static-analysis bias.** Understand + our filters can over/under-include framework glue; some weak but real edges may be hidden.

- **Event-driven/IPC gaps.** Several flows are implicit (events/streams) and need runtime confirmation beyond code inspection.

- **Provider black-box.** Model/vendor behavior is opaque and can change, affecting edge semantics and latency.

- **Human mapping error.** Folder→subsystem assignments involve judgment and may misclassify borderline files.

- **No performance profiling.** “Responsiveness” is argued architecturally; we didn’t measure latency or throughput.

References

[1] VoidEditor, “void,” GitHub repository. Accessed: Nov. 7, 2025. [Online]. Available: https://github.com/voideditor/void/
[2] Zread.ai, “Architecture Overview — Void Editor.” Accessed: Nov. 7, 2025. [Online]. Available: https://zread.ai/voideditor/void/9-architecture-overview
[4] Microsoft, “Visual Studio Code Wiki,” GitHub. Accessed: Nov. 7, 2025. [Online]. Available: https://github.com/microsoft/vscode/wiki/
[5] DESOSA, “Visual Studio Code—Architecture” (2021). Accessed: Nov. 7, 2025. [Online]. Available: https://2021.desosa.nl/projects/vscode/posts/essay2/
[6] Microsoft, “VS Code Extension API,” code.visualstudio.com. Accessed: Nov. 7, 2025. [Online]. Available: https://code.visualstudio.com/api