Void IDE - Concrete Architecture (A2)
Group 2: PHIL

Video URL (YouTube):

https://youtu.be/arYa7KZjomA

Agenda

- Abstract & Objectives
- Derivation process (A1 → A2)
- High-level concrete architecture (components & dependencies)
- Low-Level architecture deep-dive
- Two use cases (sequence diagrams with concrete methods)
- Reflexion analysis (high-level & 2nd-level)
- Conceptual architecture updates vs A1
- Lessons learned, limitations, references

Abstract

- We reverse-engineered VOID IDE from public source and docs using SciTools Understand.
- Produced a concrete, component-level architecture aligned with the onQ reference architecture.
- Validated with dependency graphs, imports, and selective code reads.
- Captured divergences between conceptual (A1) and concrete (A2) views; explained causes & impact.
- Demonstrated behavior with two sequence-diagrammed use cases tied to concrete methods.

Derivation Process (A1 ➜ A2)

We followed a clear, step-by-step process

1. Looked through sources (A1 conceptual model, codebase, docs, Understand graphs)

2. **Pass 1:** Folder & naming patterns ➜ tentative component bins

3. **Pass 2:** Imports, service wiring, extension points, runtime code paths

4. **Scoping:** Excluded build tooling/tests unrelated at runtime

5. **Output:** Stable mapping (files ➜ components), dependency graph, and notes on exceptions

Evidence & Tools

- **SciTools Understand:** dependency graphs (packages/files), call graphs
- **Code sampling:** key services/interfaces (e.g., IEditCodeService, ConvertToLLMMessageService.ts, ToolsService.ts)
- **Sanity checks:** run-time entry points, registration/bootstrap code
- **Repo history:** commits/PRs to corroborate moved utilities (Void Platform consolidation)

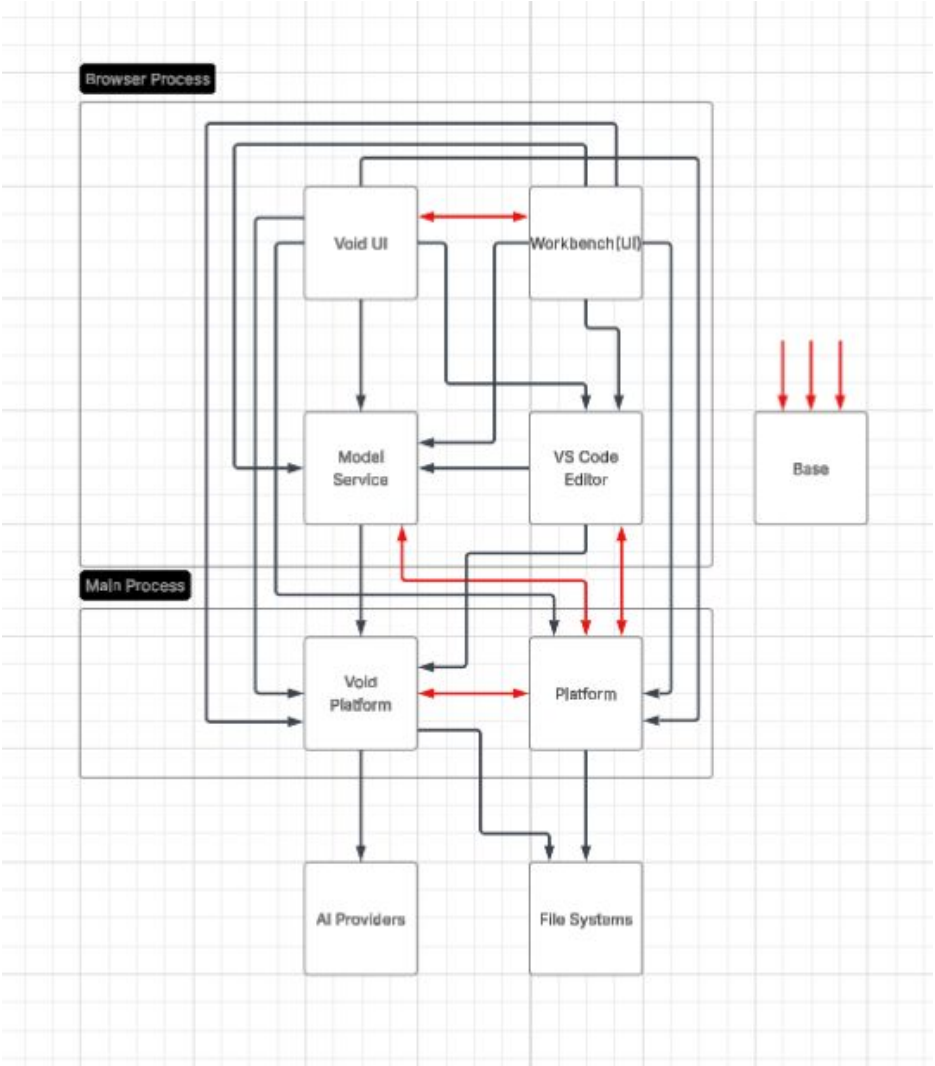
Mapping A1 ➜ Concrete

Conceptual ➜ Concrete directories / principal services

- Workbench ➜ workbench/*; view/service registration, shell bootstrap
- VS Code Editor ➜ editor/* (Monaco); edit orchestration
- Platform ➜ platform/*; OS/file/process/theming APIs
- Base ➜ base/*; events, disposables, data utils
- Void UI ➜ void/ui/*; chat panel, inline prompts, model picker, settings
- Model Service ➜ void/model/*; message prep, FIM prompts, editor context
- Void Platform ➜ void/platform/*; provider adapters, tools, snapshots
- File System ➜ platform/fs/*; read/write, dir tree
- AI Provider ➜ void/providers/*; model bridge(s)

High-Level Concrete Architecture (Box-and-Arrows)

- **Components:** Workbench, VS Code Editor, Platform, Base, Void UI, Model Service, Void Platform, File System, AI Provider(s)
- **Style:** Layered core; targeted client–provider pipeline for AI features
- **Reference:** onQ architecture mapping (link in report)



Components (Role & Rationale)

- **Workbench:** UI shell & view composition; registers VOID views
- **VS Code Editor:** Monaco editing behaviors; applies diffs/ghost text
- **Platform:** OS abstractions (fs, processes, webviews, theming)
- **Base:** shared primitives (events, disposables, helpers)
- **Void UI:** surface AI UX (inline, chat, model picker, settings)
- **Model Service:** prepare LLM messages (FIM/chat), gather editor context
- **Void Platform:** route to providers, tools API, snapshots, streaming
- **File System:** project reads/writes and tree queries
- **AI Provider(s):** model endpoints & adapters

Key Dependencies (Concrete)

- Most components use **Base** utilities (events, disposables)
- **Void UI** depends on Workbench APIs (view hosting), Platform (theme/workspace), and Void Platform (routing)
- **Workbench** wires Void UI + AI services at startup (registration, commands)
- **Model Service → Void Platform** (post-consolidation of utilities)
- **VS Code Editor** additionally calls **Void Platform** for low-latency completions
- **Void Platform** bridges to **File System** and **AI Providers**

Inner Architecture Deep-Dive: Void UI

Subcomponents:

1. **Chat Panel** — conversation threads, message list, input box
2. **Inline Prompting/Completions View** — ghost text, accept/undo, diff preview
3. **Model Picker** — provider/model selection, capability flags
4. **Prompt Composer** — context assembly UI (file scope, selection, instructions)
5. **Settings Panel** — Void features toggles, provider keys, limits

Interactions:

- Hosted by Workbench (webviews/panels)
- Talks to Model Service for message prep
- Calls Void Platform for tools/snapshots/streaming
- Pulls theme/workspace via Platform

Interaction Details (Void UI)

- **Chat Panel** → ChatThreadService.addUserMessageAndStreamResponse(prompt)
- **Inline View** → listens to editor events; triggers IEditCodeService.startApplying()
- **Model Picker** → updates provider selection in Void Platform registry
- **Prompt Composer** → requests file context via IToolsService.callTool('read_file'|'get_dir_tree')
- **Settings** → persists via Platform config APIs

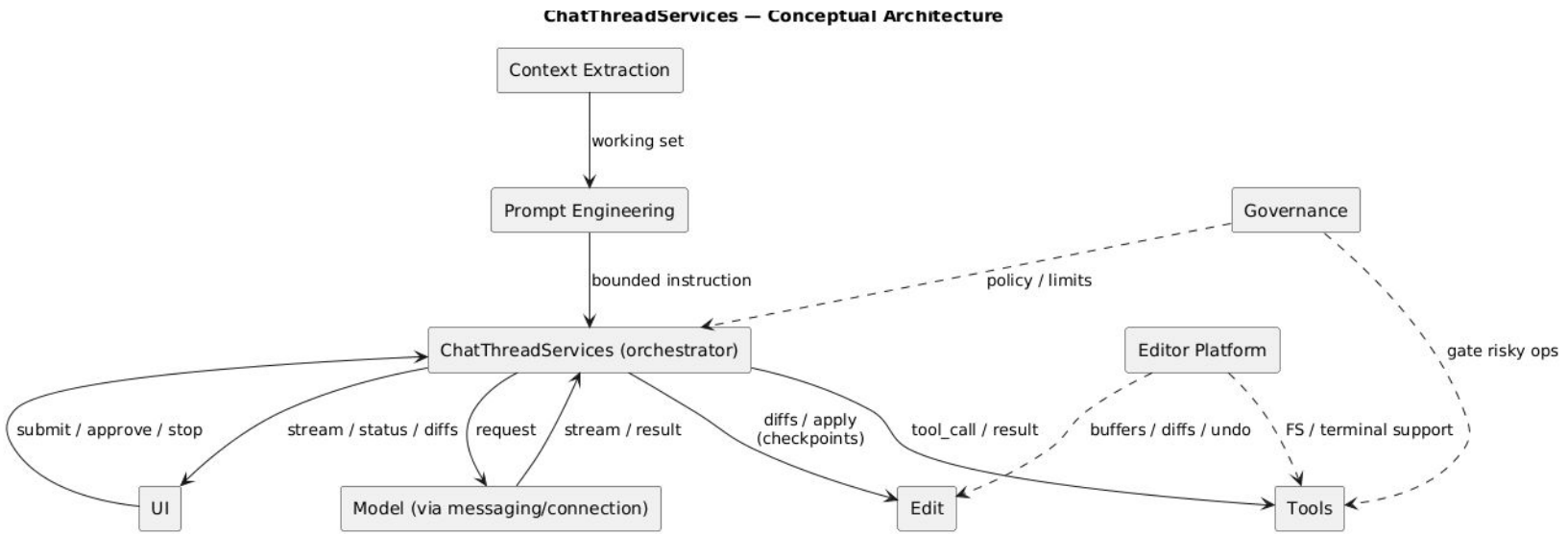
Reflexion Method (Evidence)

- Generated Understand graphs for **package/class dependencies** and **call graphs**
- Traced **registration/bootstrap** code paths to confirm runtime dependencies
- Used **git log/blame** to confirm migration of utilities into Void Platform and editor fast-path additions
- Recorded divergences with **explanations grounded in code locations** (screenshots in report)

High-Level Reflexion Analysis

- **Base:** concrete shows universal dependency (events/helpers) vs conceptual minimal depiction ➜ abstraction vs implementation granularity
- **Void UI:** gains deps on Workbench/Platform/Void Platform due to view hosting + shared services ➜ integration reality
- **Workbench:** depends on Void Platform/Model Service for AI bootstrap ➜ startup wiring is centralized
- **Model Service:** now only depends on Void Platform (Platform utilities consolidated) ➜ simplified service boundary
- **VS Code Editor:** direct Void Platform calls for latency ➜ perf-driven divergence
- **Void Platform:** depends on File System + Platform (storage, webviews, process control) ➜ middleware role
- **Platform:** now depends on Void Platform for AI init/webviews ➜ responsibility shift

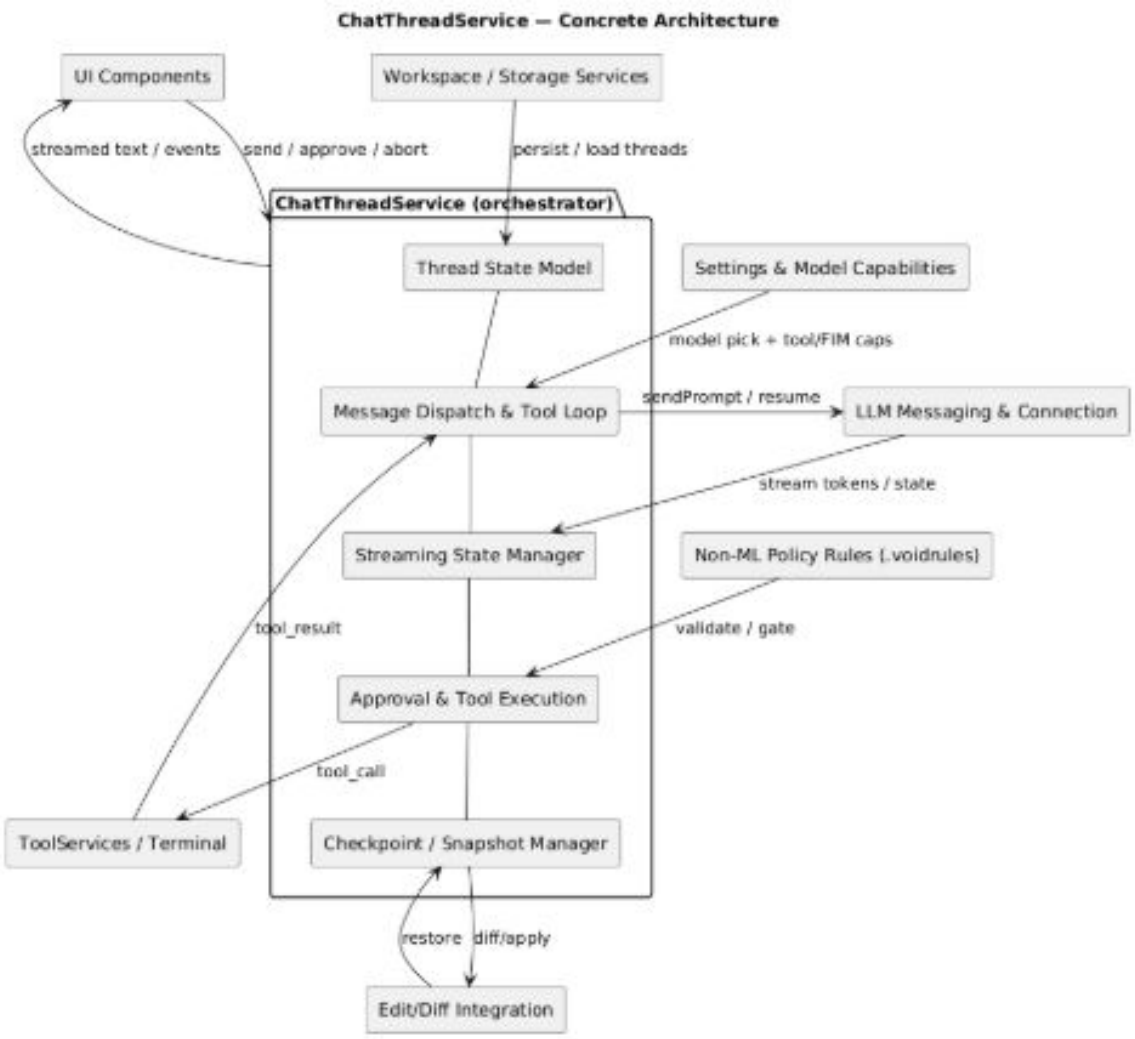
Low-Level: ChatThreadServices (Conceptual)



Low-Level: Conceptual

- **Role:** Orchestrates VOID's AI flow end-to-end: manages threads, times model calls, invokes tools, and shows **diffs before apply**; exposes clear **states** for UI; wraps edits with **checkpoints**.
- **Conceptual loop:**
    1. **Context Extraction** → working set from open buffers, recent edits, diagnostics, project tree (trim/summarize with provenance).
    2. **Prompt Engineering** → bounded instruction (objective, constraints, success criteria, allowed tools).
    3. **Ask model** via provider-agnostic messaging; stream partials or tool-calls.
    4. **Act** via ToolServices; resume stream with results; apply policy (approval gates, rate/cost, capability hints).

Low-Level: ChatThreadServices (Concrete)



Low-Level: Concrete Architecture

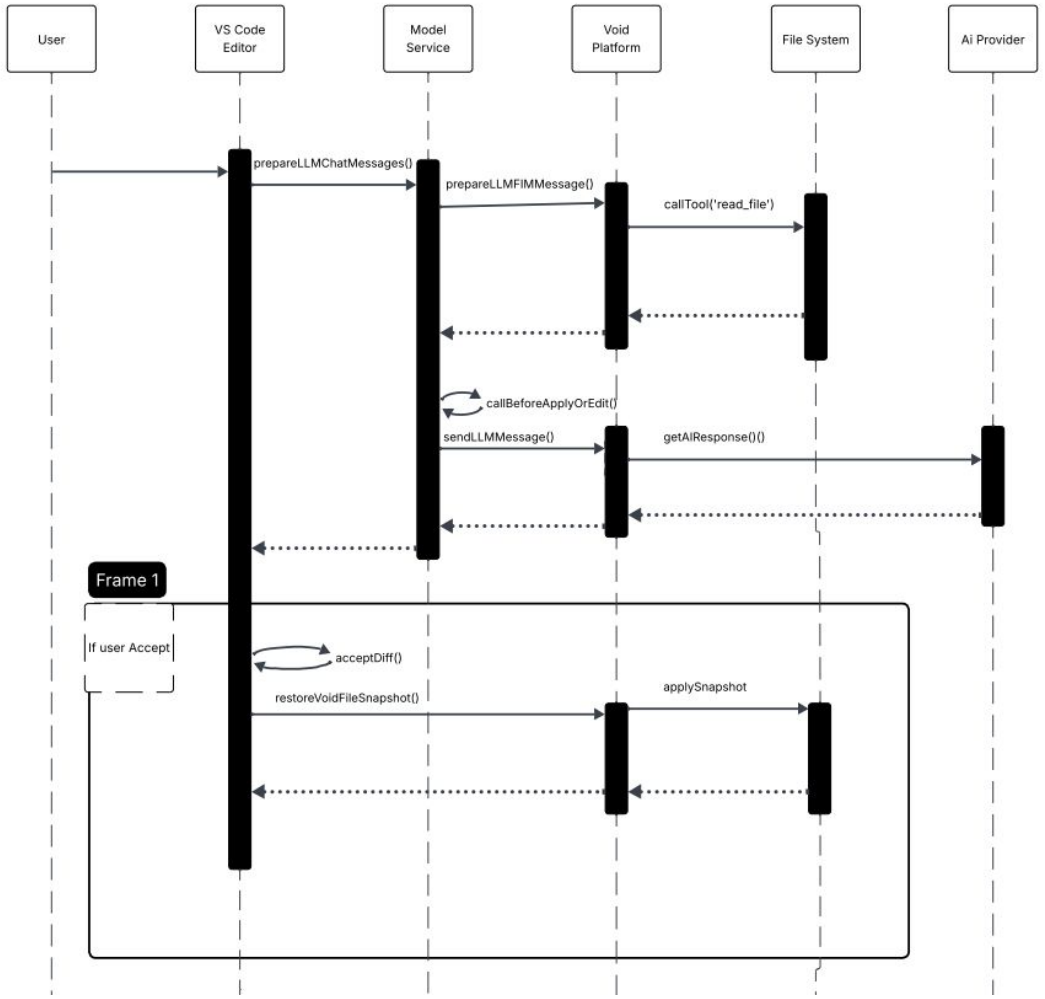
- **Per-thread state model:** active thread ID, history, step; persisted so thread switching reconciles in-flight work and keeps chat UI in sync.
- **Dispatch loop:** on submit → provider-neutral messaging → **stream tokens, pause on tool-call,** resume after tool result or stop.
- **State tracker:** transitions **idle → streaming → tool-running → awaiting-approval;** raises events for progress/abort/status (no UI guessing).
- **Concrete touchpoints:**
  - ChatThreadService.addUserMessageAndStreamResponse(text)
  - IToolsService.callTool('read_file' | 'get_dir_tree')
  - IEditCodeService.acceptDiff(diff) / editor snapshots for exact undo/redo

Low-Level: Safety, Interfaces & Reflexion

- **Approvals & checkpoints:** validate tool requests; inline Approve/Deny for risky/large edits; on approval run file/terminal actions ➜ typed payloads; denial halts with policy reason; snapshots bracket edits.
- **Interfaces/data:**
  - **Inbound:** UI intent & selections; compact bundle + bounded instruction from Context Extraction/Prompt Engineering.
  - **Model I/O:** all via messaging/connection layers (provider-agnostic).
  - **Tools I/O:** parameterized actions; results reinjectable/displayable; edits applied via standard apply/undo paths.
- **Reflexion (conceptual ↔ concrete):**
  - Intent➜Ask➜Act➜Diff ⇔ **thread state, dispatch loop, state tracker, approval+checkpoint** safety.
  - Aligns with VOID docs: provider-agnostic messaging, explicit approvals, **diff-first** edits.

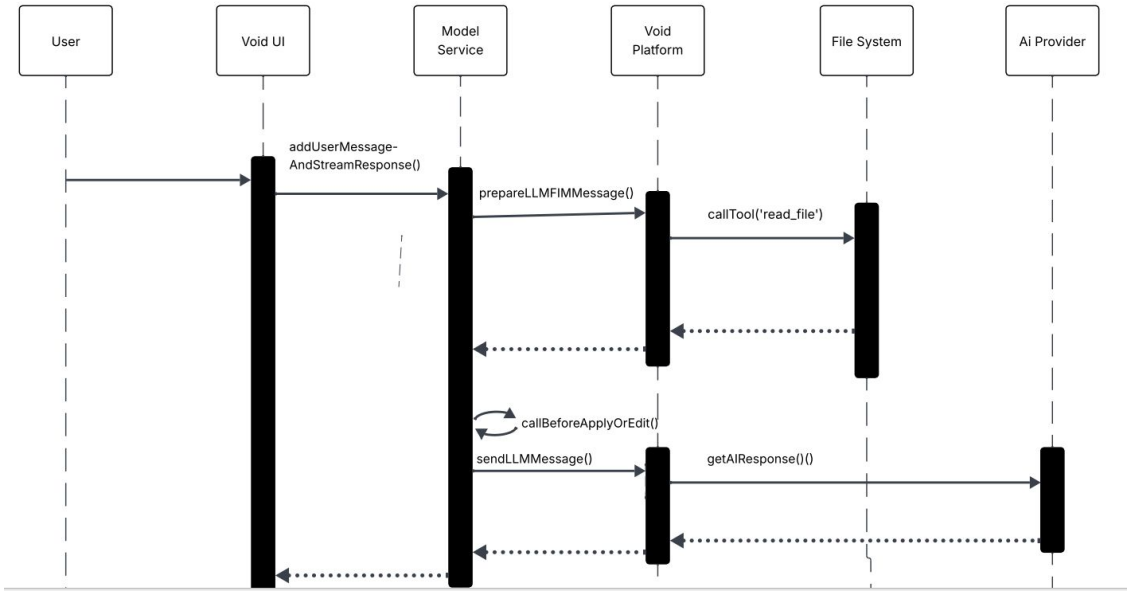
Use Case 1 Autocomplete

- Flow:
  UI → Context Extraction → Prompt Engineering → LLM Messaging → LLM Connection → Response Integration → EditCode → Editor Core
- What happens:
  - The user triggers autocomplete
  - Context Extraction gathers the surrounding code
  - Prompt Engineering builds a clean AI request
  - LLM Messaging sends it to the model and streams results
  - Response Integration shows inline code suggestions
  - EditCode Service safely applies accepted edits with undo/redo



Use Case 2: Chat

- UI ➜ ChatThread Services ➜ Context Extraction ➜ Prompt Engineering ➜ LLM Messaging ➜ LLM Connection ➜ Response Integration ➜ UI component
- What happens:
- ChatThread Services keeps track of the chat state and approvals
- Context Extraction gathers current code context
- Prompt Engineering shapes the AI request
- LLM Messaging streams the model's reply



Conceptual Architecture (Updated vs A1)

- ~10 components enumerated (see Slide 8) with roles & boundaries
- **Changes since A1:**
  - Consolidated utilities under **Void Platform** (reduced Model Service surface)
  - **Editor fast-path** to Void Platform for completions
  - **Platform ↔ Void Platform** bidirectional coordination during startup/webviews
  - **Rationale:** performance, reduced duplication, clearer provider boundary

Lessons Learned

- Understand first, then code reads: fastest path to a correct map
- Abstractions in A1 hide necessary wiring—expect extra deps in A2
- Performance needs (streaming/latency) reshape boundaries (editor fast-paths)
- Keep AI as an **analytical amplifier;** verify every claim in code/graphs

Limitations & Threats to Validity

- Partial code sampling risks missing rare paths
- Vendor/provider plugins may change interfaces over time
- Some dependencies are **build-time** only; we excluded non-runtime tooling
- Mitigations: multi-pass verification, independent reviewers, repo history triangulation

AI Teammate: Setup & Protocol

- **Model:** ChatGPT (GPT-5, Nov 2025)
- **Scope:** support/analysis (formatting, graph reading aid), not architectural authority
- **Protocol:** context-first prompts ➜ fact-based iterations ➜ human verification loop
- **QC:** every AI-aided statement checked vs Understand/code; git logs for refactors

AI Teammate: Impact

- **Quantitative:** ~12–15% contribution (verification, summarization, documentation polish)
- **Qualitative:** faster pattern spotting; required careful correction of over-generalizations
- **Takeaway:** human-AI pairing excels on repetitive analysis; humans own architecture

Conclusion

- Produced right-level concrete architecture with justifications per component
- Explained derivation and divergences with implementation-grounded reasons
- Demonstrated behavior with two method-accurate sequence diagrams
- Updated conceptual model with clear rationale (performance & consolidation)