Void IDE - Conceptual Architecture (A1) Group 2

Video URL (YouTube):

https://youtu.be/KSJRmFyknHk

Group Members & Roles

Name	Role in Report	Role in Slides	Role in Presentation
Reyan Sehgal	Control & Data Flow, Concurrency and Evolution	Built section slides	Presenter
Mason Zhang (Team Lead)	Architecture and Overall Cohesion		
Duncan Mackinnon	Subsystems & Dependencies		
Jeremie Trepanier	Use Cases and Diagrams		Presenter
Jack Atkinson	AI Collaborative Report		

Derivation

We followed a clear, step-by-step process

1. Looked through public sources to find key parts of the system

2. Chose two main use cases — Quick Edit / Autocomplete and Chat ➜ Tool ➜ Apply

3. Made sequence diagrams using only the parts we found in those sources

4. Combined repeating parts into one big diagram with a legend and clear names

This gave us a clear link from what we read ➜ how it works ➜ how it’s built

Architecture Style

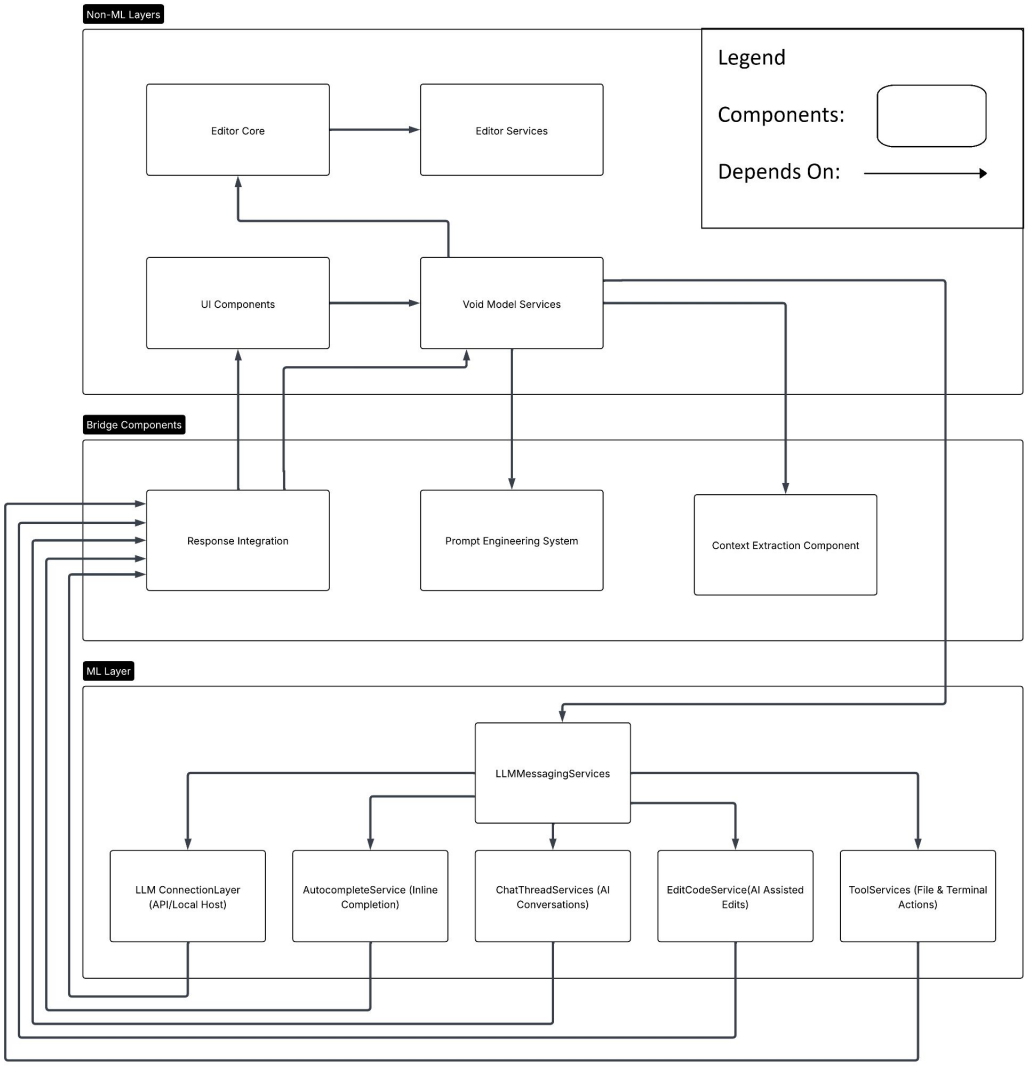
1. Layered Architecture
    Base Layer (Non-ML): Core editor, UI, and main services
    Bridge Layer: Connects editor context to AI systems
    ML Layer: Handles AI models, chat, and code suggestions

Pros: Easy to test, organized, and flexible for updates
Cons: Can be slower across layers and limited by model size

2. Client–Server Architecture
    Client (Void IDE): Runs on the user's computer
    Server (LLM Models): Processes AI requests in the cloud

    Pros: Keeps the editor lightweight and fast; allows powerful AI processing remotely
    Cons: Depends on internet connection and requires careful retry and error handling

Subsystems & Dependencies



Subsystems

Non ML layer
UI Components — chat panel, diff overlays, approvals.
Editor Core — text model, selections, tokenization, rendering.
Editor Services — filesystem/SCM, diagnostics, config, storage.
Void Model Services — feature➜model mapping; limits/tool-use/streaming hints.

Bridge Components Layer
Context Extraction — builds context bundles with provenance.
Prompt Engineering System — bounds/structures prompts; declares allowed tools.
Response Integration - format response from LLM for usage in the non-ML layer

Subsystem - Continued

ML-Layer
LLMMessagingServices - send/stream/retry; pause for tools; resume.
LLM Connection Layer — provider routing/health/quotas.
Response Integration — stitches streams; lifts actions to diffs/commands with anchors.
EditCodeService — diff-first apply; per-hunk/batch; undo/redo checkpoints.
ToolServices — file/terminal actions with approval gates.
Non-ML Layer — allow/deny, rate/spend caps, secret-safe prompting.
We'll trace these exact names in the sequences next.

Rationale for Interactions

Context Extraction ➜ Prompt Engineering

Collects code context and turns it into a clear AI prompt

Prompt Engineering ➜ LLM Messaging ➜ LLM Connection

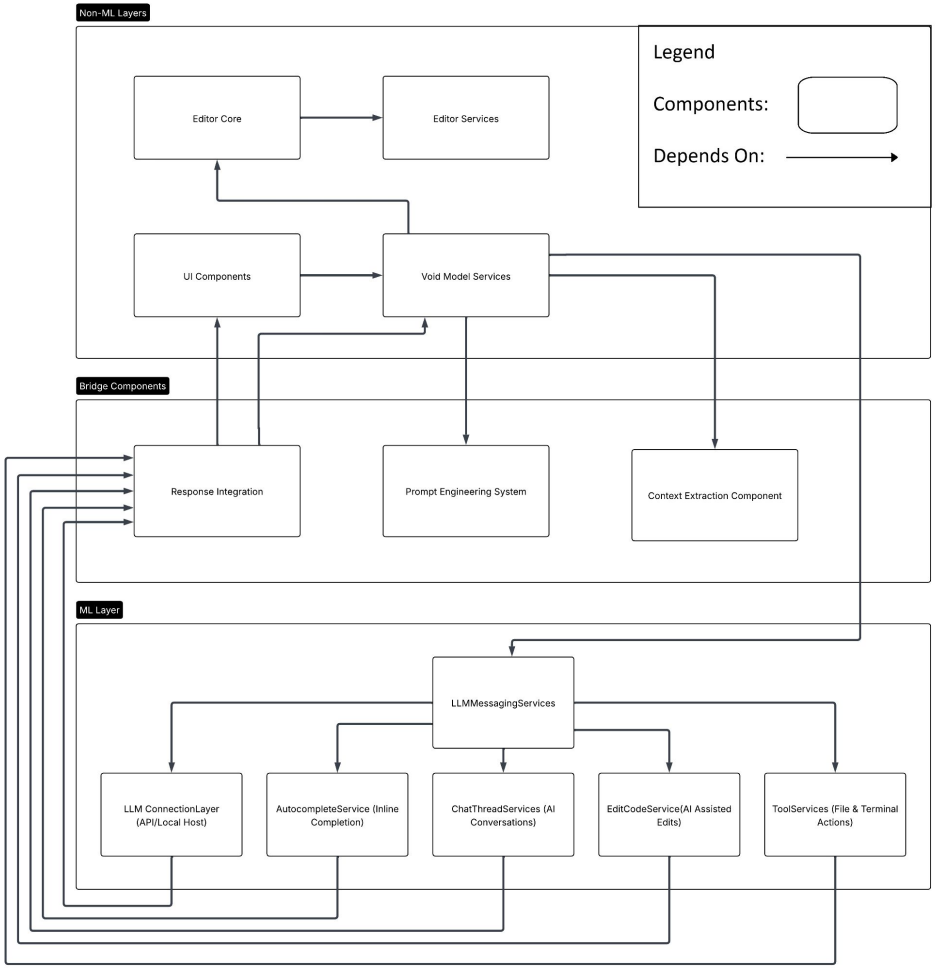
Sends requests to the right AI model and manages responses safely

Response Integration ➜ EditCode / Tool Services

Turns AI output into real code changes or tool actions, with undo/redo checkpoints

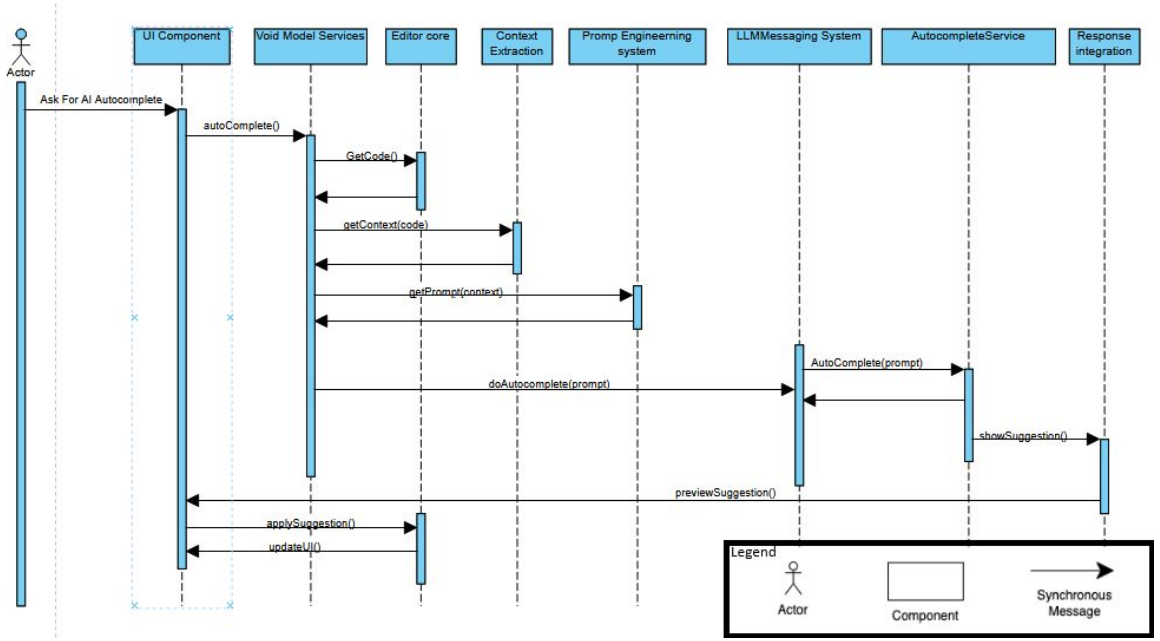
Non-ML Layer

Adds safety checks and limits for expensive or risky operations



Sequence 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



Sequence 2: Chat

Flow (matches Fig. 1):
UI ➔ ChatThread Services ➔ Context Extraction ➔ Prompt Engineering ➔ LLM Messaging ➔ LLM Connection ➔ Response Integration ➔ UI component

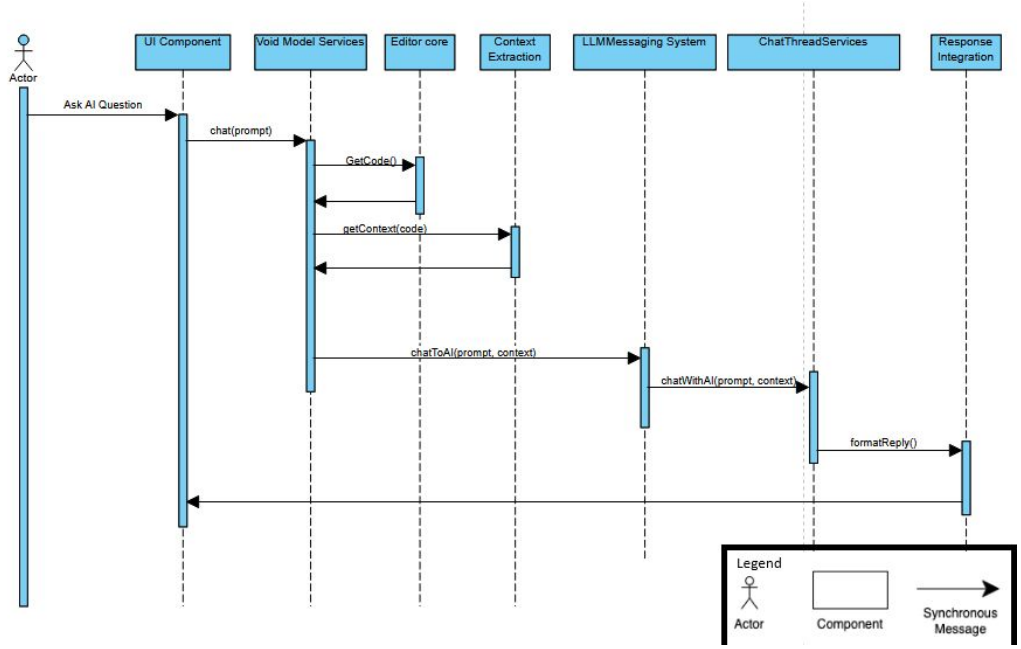
What happens:

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



Concurrency & Team Impact

Renderer handles the UI, Main handles I/O and AI — connected with async messages to keep the editor fast

Only one edit per file at a time; cross-file edits checked and grouped safely

Rapid triggers slowed down to prevent overlap

Clear layer roles and diff-first reviews let the team work safely in parallel

Architectural Alternative : Repository Style

- All parts share one main data store instead of talking directly to each other
- Could make data more consistent and simplify tracking of prompts, context, and AI results
- Helps with versioning and undo/redo across the system
- But: creates a single point of dependency — harder to change safely
- The layered design we chose keeps parts separate and easier to update or expand

Limitations & Lessons

Limitations:

Some docs were missing — parts had to be inferred
AI–editor links not fully public
Performance data unavailable
Diagrams show conceptual, not exact, design

Lessons:

Teamwork filled gaps in unclear docs
Early task planning improved accuracy
AI sped work up, but we verified everything
Layered design clarified system structure
Regular check-ins solved technical issues

AI Teammate

Model: GPT-5 Thinking (Oct 2025)

Role: helped outline slides and write legend text

Quality control: every AI claim tied to a source; unverifiable items removed; humans validated all content

Contribution: ≃ 20% — AI assisted, humans made final decisions

Conclusion