Group 25
OK_COMPUTER

Video Link: https://youtu.be/ZXPancPpaR4

OK_COMPUTER

Thomas Schrappe - Presenter, slides, sequence diagrams, description

Adrian Yanovich - Presenter, slides, Risks & Mitigation, Impact Analysis

Colin Mclaughlin (Group Leader) - Intro, Abstract, AI report, Affected Files and Directories

Jaiman Sharma - State holder identification, NFR identification, SAAM table and Comparisons

Mantaj Toor - Architectural changes required, Conceptual Architecture

Lingwei Huang - Testing Plan

Proposed Enhancement Overview + Motivation

What We're Adding

- A Vim plugin that brings Void's AI capabilities into the terminal
- Supports macros (e.g., :void/[prompt]) and AI autocomplete inside Vim
- Allows Void to read/modify buffer text safely through controlled text-only edits
- Fully configurable to match each user's Vim workflow

Why This Matters (Motivation)

- Void's current CLI supports *only* basic editor operations — no AI features
- Developers who work in terminals, SSH sessions, or headless servers cannot access Void's AI
- Enabling AI in the CLI unlocks:
  - Automation (scripts, pipelines, DevOps workflows)
  - Remote development without a GUI
  - Broader adoption among Vim/CLI-first developers
- Fills a major gap in Void's ecosystem and expands where and how the tool can be used

Primary Use Cases

- AI Autocomplete in Vim using macros
- Code generation from prompts (:void/[prompt]) inserted directly into the buffer

Why This Enhancement Matters

Bring Void's AI Power to the Terminal

- Unlock all AI coding features *outside the GUI*
- Use code generation, refactoring, and analysis directly in Vim or any CLI workflow
- Perfect for developers who live in terminal-first environments

Supercharge Automation & CI/CD

- Run Void AI in scripts, build pipelines, and remote servers
- Enable headless execution with no need for a graphical workbench
  Integrate AI into testing, deployments, and automated refactors

Massively Expand Void's Reach

- Support for Vim (and future CLI editors like NeoVim/Emacs) grows the user base
- Attracts power users, DevOps engineers, and remote-only devs
- Makes Void usable anywhere VS Code's GUI can't run

Cleaner Architecture, Better Long-Term Flexibility

- Standalone Node runtime provides clean separation from the workbench
  More portable, easier to maintain, and simpler to evolve
- Ensures future features can extend smoothly into CLI environments

Architectural Alternatives

Alternative A integrates AI functionality directly into the existing Node and Rust-based CLI pipeline. It partially loads AI Services alongside the CLI runtime to handle commands like code generation or edits. The data flow moves from CLI parser to Node runtime and then through AI Services, returning results to the user. While it requires minimal new components, it depends on GUI-related subsystems, which can slow performance and reduce portability in headless scenarios.

Elon Musk

“Built ford tough” - Elon Musk

Alternative B introduces a standalone Node-based Model Server that handles all AI operations independently of the GUI. The CLI acts as a thin client, sending requests to the Model Server via a local API or IPC channel. Both GUI and CLI clients share the same backend, ensuring consistent API behavior and simplifying multi-editor integration. This approach improves performance, reliability, and scalability while maintaining clear separation of concerns between interface and AI logic.

Comparison + Selected Architecture

- Alternative A extends the existing CLI pipeline but depends on VS Code subsystems, limiting portability and introducing potential GUI-related overhead.

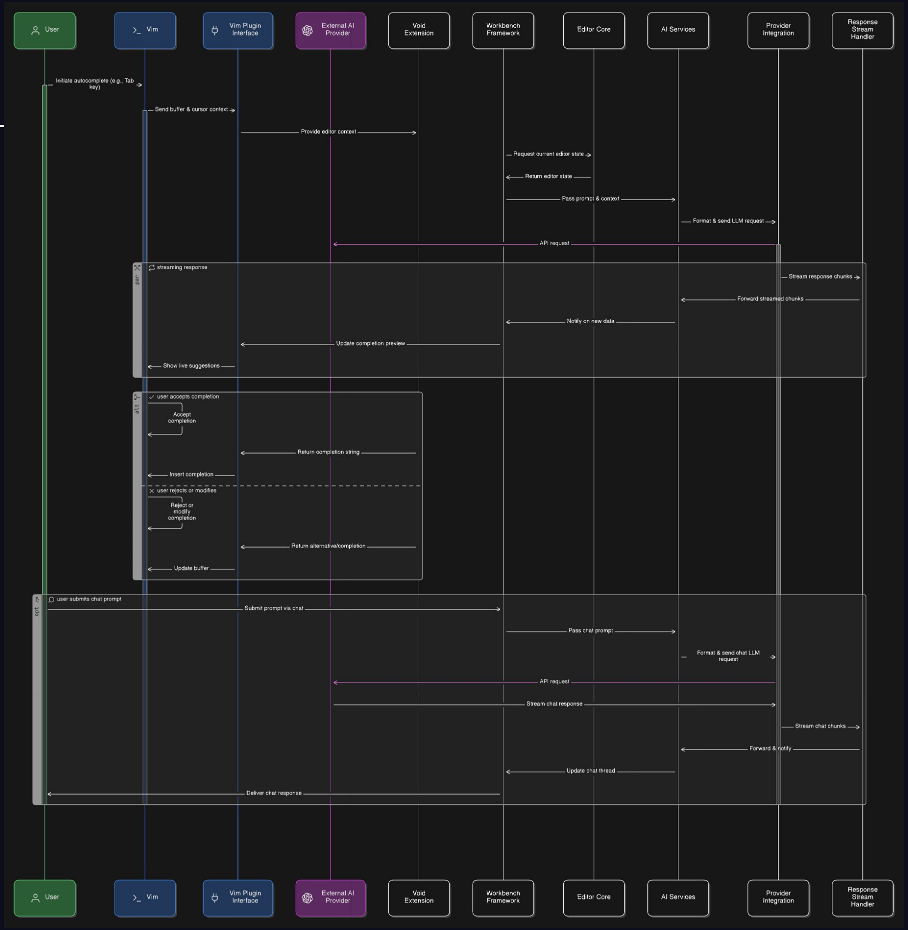
- Alternative B introduces a standalone Model Server, separating AI logic from the GUI and enabling headless operation with consistent API behavior.

- SAAM evaluation shows Alternative B outperforms A in maintainability, reliability, scalability, and cross-editor support while reducing risk of integration regressions.

- Alternative B is selected due to clear subsystem boundaries, better multi-editor compatibility, faster CLI response times, and long-term extensibility.

Use case

- The user triggers autocomplete in Vim via a bound macro or command, sending the current buffer and cursor context to the Void plugin.
- Void processes the context using its AI logic and generates a suggested completion string.
- The plugin inserts the completion directly into the active buffer without altering Vim's editor state or executing additional commands.
- Users can accept, reject, or refine the suggestion, with autocomplete operating continuously or on demand based on configuration.
-



Sequence Diagram of the Proposed Architecture

- CLI sends user prompt and editor context to the standalone Model Server via IPC or local API.
- Model Server processes input using AI services and returns generated or edited code.
- CLI receives AI response and applies changes to the active buffer or displays suggestions.
- GUI and other editor clients can interact with the same Model Server without interfering with CLI operations.
-

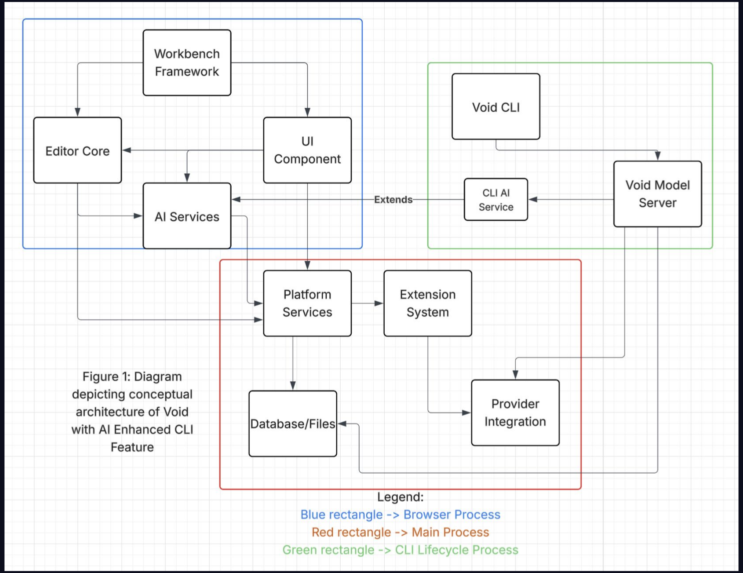


Figure 1: Diagram depicting conceptual architecture of Void with AI Enhanced CLI Feature

Impacted Subsystems & Architectural Style

Maintainability

- Vim plugin cleanly separates editor logic from the core Void engine
- Modular boundary → plugin + core can be updated independently
- BUT: two editor environments means higher integration upkeep
- API/response format changes could break the plugin

Evolvability

- Provides a blueprint for future CLI plugins (NeoVim, Emacs)
  Uses text commands, macros, and buffer edits → easy to extend
- GUI-heavy future features may not translate to Vim
- Developers must consider both graphical & non-graphical editors

Testability

- Component-level tests easy via mockable communication layer
- Full system testing harder due to Vim's modal interface
- Requires scripted sessions or custom harnesses
- Regression tests complicated by Vim versions & user keybindings

Performance

- Added IPC + context handling introduces slight latency
- Large files increase CPU/memory usage
  CLI version generally faster than GUI (less rendering)
- Most delays still come from the LLM, not the plugin

Risks, Limitations, and Mitigation

Security Risks

- Plugin needs access to file buffers → risk of unintended writes
- API calls to Void may expose sensitive code
- Async responses may write to the wrong buffer (race conditions)

Performance Risks

- Provider/API latency may cause delays
- Large files → slow serialization + transfer
- Blocking calls could freeze Vim

Maintainability Risks

- Supporting VS Code + Vim creates divergent code paths
- Future Void API changes may break the plugin
- Vim's modal edge cases complicate logic

Test Coverage Risks

- Async responses can arrive late or interleave → inconsistent tests
- Streaming output hard to simulate
- Vim's modal behavior makes automated testing difficult

Mitigation Strategies

- Enforce strict text-only write boundaries
- Use non-blocking async calls with timeouts
- Define a stable, versioned API contract
- Create mockable IPC layers for reliable testing

Testing Strategy for Cross-Feature Interactions

Unit Testing

- Mock AI provider, mock editor, and DI boundaries
- Test command parsing: flags, missing args, malformed input
- Validate prompt request serialization/deserialization over IPC
- Verify error handling (bad models, provider down, network errors)
- Test streaming handlers for correct partial-token output

Integration Tests

- Simulate full workflow in Extension Host sandbox
- Validate flow: **CLI → Parser → IPC/Daemon → LLM Service → Provider → Output**
- Verify edits propagate correctly to workspace + file system
- Ensure model selection & context handling match GUI behavior
- Test multi-file prompt scenarios

UI / UX Tests

- Ensure CLI-generated edits appear correctly in GUI diffs/undo history
- Confirm visual stability of chat panel, decorators, history, and notifications
- Validate no unexpected UI artifacts from new IPC channels
- Test sync: GUI must show results of prior CLI actions
- Ensure GUI responsiveness remains unaffected during heavy CLI prompts

Regression Tests

- Re-run A2 scenarios to ensure no regressions
- Verify chat, autocomplete, and file-editing workflows still behave normally
- Test fallback behavior during provider failure
- Confirm consistent settings across CLI and GUI pathways

Performance Tests

- Measure CLI cold/warm start latency
- Benchmark provider round-trip times for varying prompt sizes
- Analyze streaming performance (token rate, chunk order, flush behavior)
- Stress test rapid CLI calls for leaks or IPC congestion
- Compare Alternative 1 vs. Alternative 2 to validate SAAM evaluation

AI Collaboration

- Used GPT-5 (Oct 2025) for structured writing, rubric alignment, and editing consistency.
- Used Claude Sonnet 4.5 (via Cursor) for file-system parsing and interpreting data from SciTools Understand and the Void GitHub repository.
  AI usage increased from Assignment 1 due to deeper code and dependency analysis needs.
- Other models (Gemini 2.5, Grok 4, o3) were tested but found less reliable for large-context code analysis.
- Finally, Eraser was used to convert a description of our use case diagram into a cleaner, more polished looking diagram.

AI mostly worked with technical details and touch ups, making clean diagrams, touching up text, and interpreting data.
Overall we believe AI contributed to around 35% of A2.

Lessons Learned & Wrap-Up

The enhancement enabled Void’s AI features to run in a headless CLI workflow via a standalone Model Server, supporting terminal editors like Vim and Neovim.

Separating the Model Server from the tightly coupled main and browser processes improved maintainability, subsystem clarity, and long-term extensibility.

The team learned the importance of clearly defining responsibilities in a layered architecture, realizing that existing AI Services could not be directly reused without creating dependencies.

Revisiting conceptual and concrete architecture diagrams helped maintain a clear system model, prevent accidental coupling, and guided the structured extension of the IDE’s functionality.