Void IDE - Real-Time Code Execution Heatmap (A3)
Group 2: PHIL

Video URL (YouTube):

https://youtu.be/bLn-qVvytrA

Group Members & Roles

Name	Role in Report	Role in Slides	Role in Presentation
Reyan Sehgal	Control & Data Flow, Concurrency, Evolution	Built section slides	Presenter
Mason Zhang (Team Lead)	Abstract, Impact on Concrete Architecture (Files & Dependencies), Overall Cohesion		
Duncan Mackinnon	Use Cases and Diagrams, Potential Risks and Limitations		
Jeremie Trepanier	Testing	Built section slides	Presenter
Jack Atkinson	SAAM Analysis. New Feature Impact on Sub System, AI Collaboration Report		

Agenda

- Motivation & feature overview
- Two use cases + sequence diagrams
- Architectural alternatives
- SAAM comparison and decision
- Impact on architecture (high- & low-level)
- Testing & interactions with existing features
- Risks
- AI collaboration & conclusion

Void IDE & Motivation

- Void IDE = open-source VS Code fork with extra AI support
- Current AI: explains/edits code based on static source only
- Missing: insight into runtime behaviour (what actually runs, how often, how slow)
- Motivation:
  - Help devs quickly see hotspots inside the editor
  - Give the AI assistant real execution data for better explanations

Proposed Enhancement: Heatmap

- Lightweight runtime profiler tracks:
  - Which lines/functions execute most
  - How long they take
- Render profiling data as coloured heatmap directly in the editor
- Hot lines = high frequency / high time; cold lines = low activity
- Profiling data exposed as a model so AI can use it for performance explanations

Benefits of the Enhancement

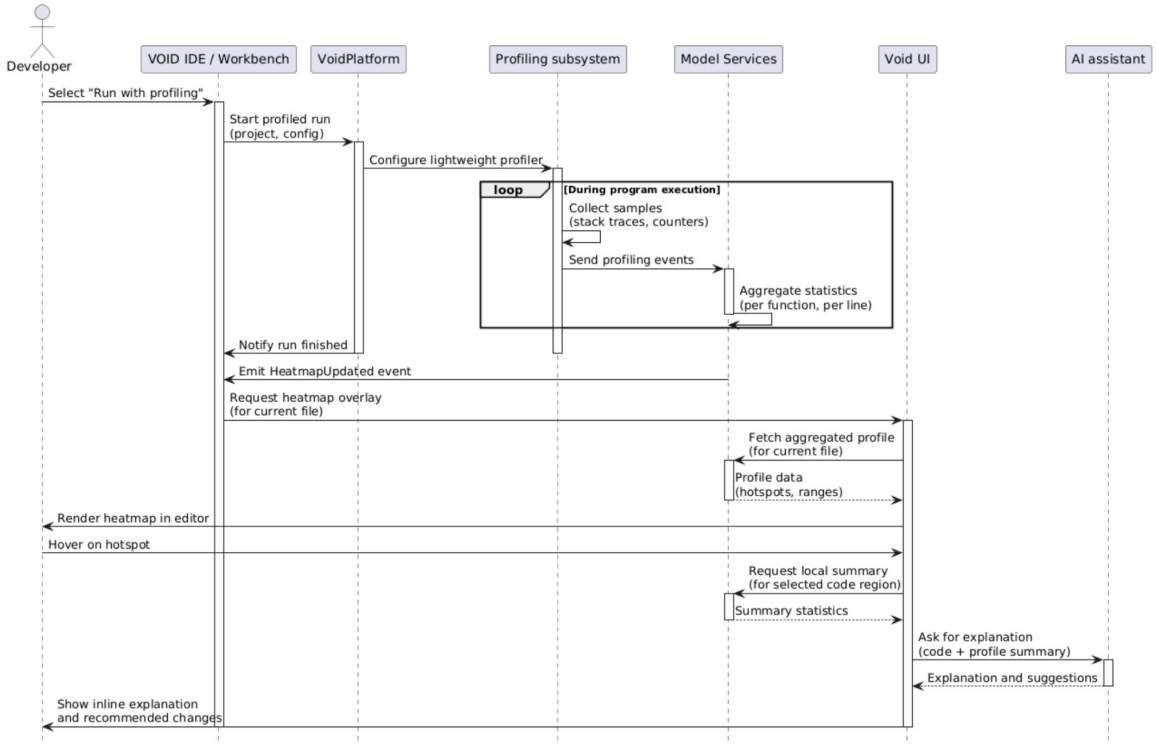
- Shorter feedback loop for performance tuning (no separate profiler UI needed)
- Helps newcomers understand which code actually matters at runtime
- Integrates with AI: “Explain why this is slow” using real measurements
- Reusable profiling pipeline for future visualizations (timelines, call graphs, etc.)

Use Case 1: Investigating a Slow Feature

- Developer notices a slow command in their app
- In Workbench: chooses “Run with profiling”
- Platform / VoidPlatform enable profiler and collect samples
- Model Services aggregates samples into per-file, per-line stats

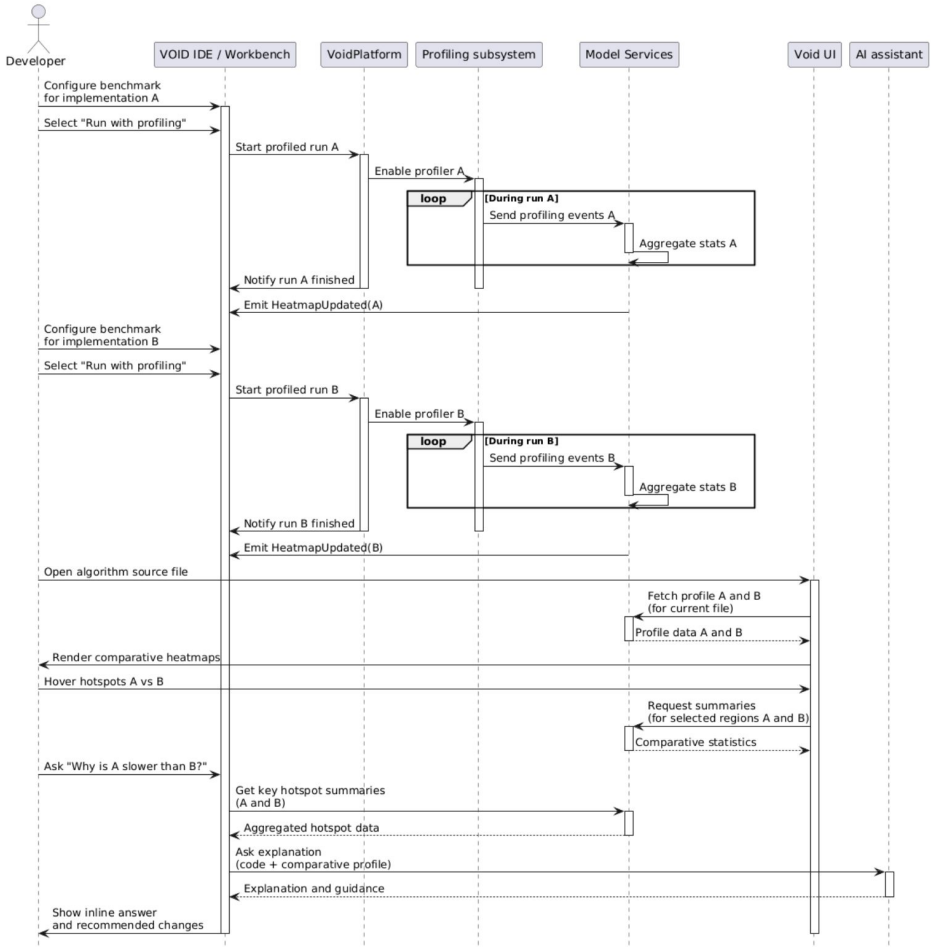
Void UI overlays heatmap on source file after run

- Developer hovers hotspot ➜ Workbench sends code + profiling summary to AI assistant ➜ AI explains why it’s slow and suggests fixes



Use Case 2: Comparing Implementations

- Developer has two algorithm implementations (A & B)
- Runs both under “Run with profiling” via Workbench benchmark/test harness
- Profiling subsystem streams samples to Model Services for each run
- Model Services exposes separate profiles for A and B
- Void UI overlays heatmaps for each implementation in the same source file
- Workbench asks AI:“Why is A slower than B?” with profiling summaries ➜ AI highlights hotspots and suggests more efficient patterns

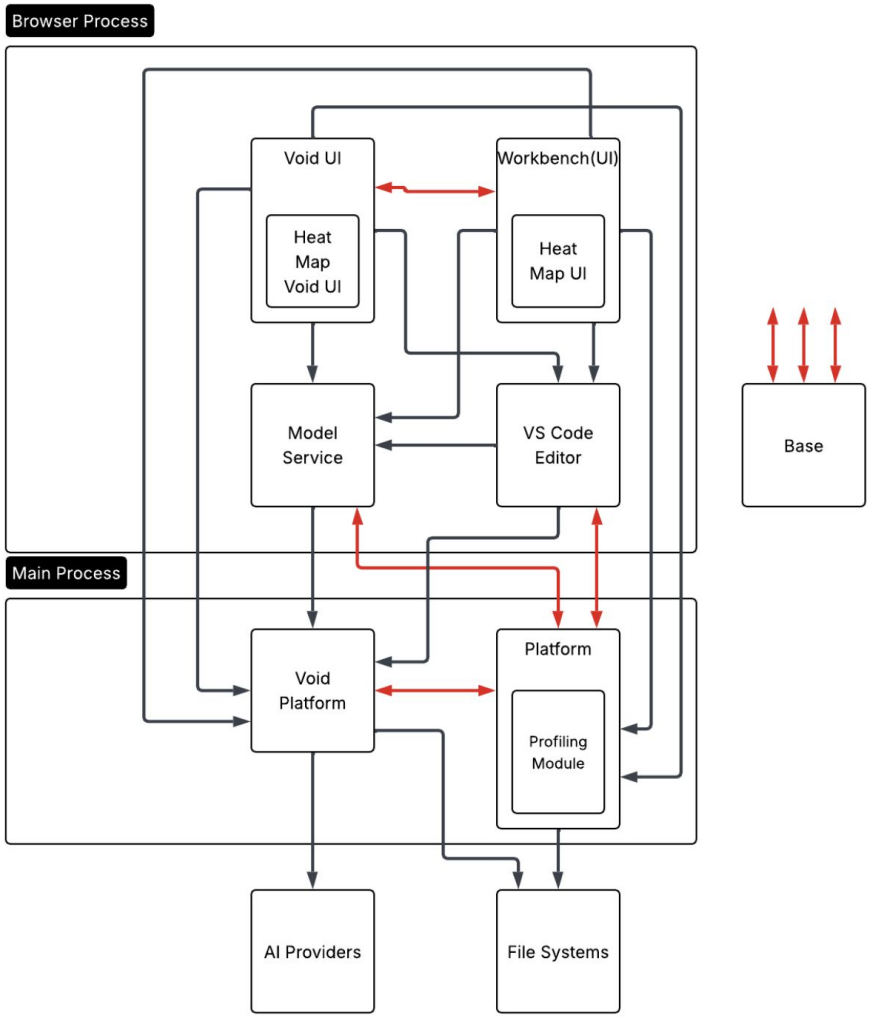


Architectural Alternatives

- **Approach 1 – Layered style:** profiling as a platform service; data flows up through existing layers
- **Approach 2 – Publish/Subscribe:** profiling events on event bus; multiple subscribers (UI, AI, etc.)
- **Approach 3 – Client–Server:** separate profiling service; IDE acts as client
- Goal: compare these using SAAM on performance, testability, modifiability, etc.

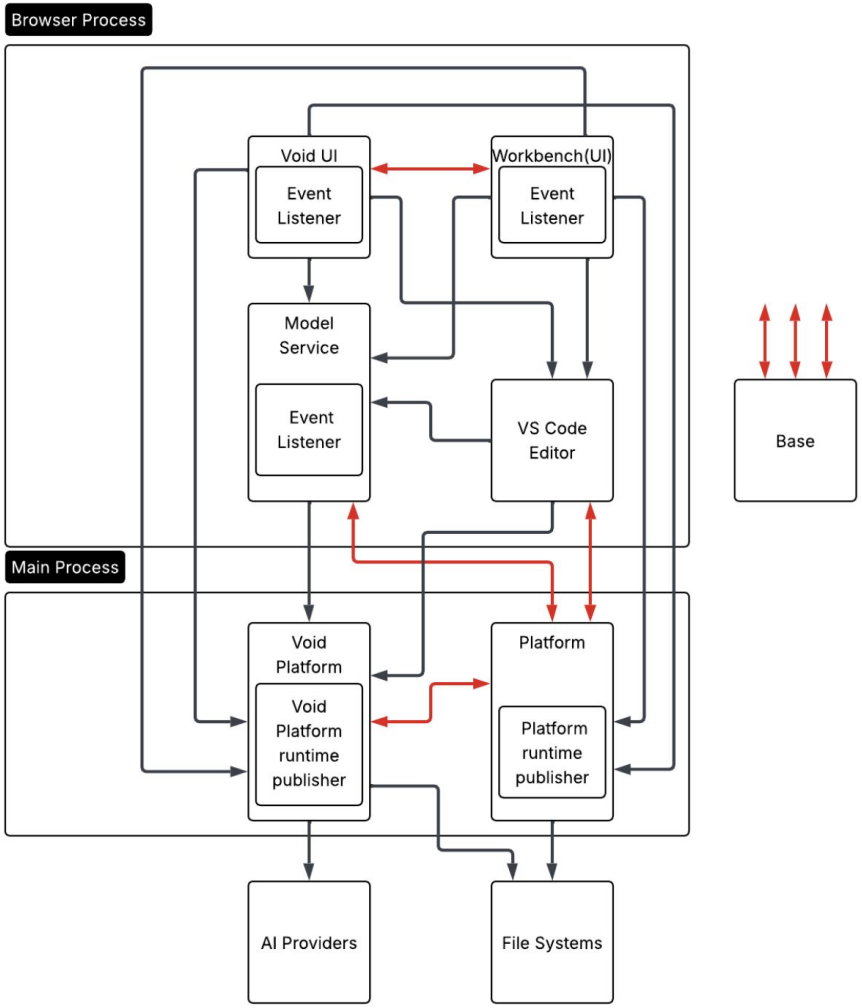
Approach 1: Layered Architecture

- Keep existing Void IDE layered structure
  (Platform ➜ Model Services ➜ UI/Workbench ➜ AI)
- Add **profiling service** in Platform
  (start/stop profiling, collect samples)
- Model Services gains **profiling model**
  (maps samples ➜ heatmap per file/line)
- Void UI & Workbench: new commands
  (Run with profiling, Show heatmap,
  Explain hotspot)
- AI providers receive richer prompts:
  code + profiling summary



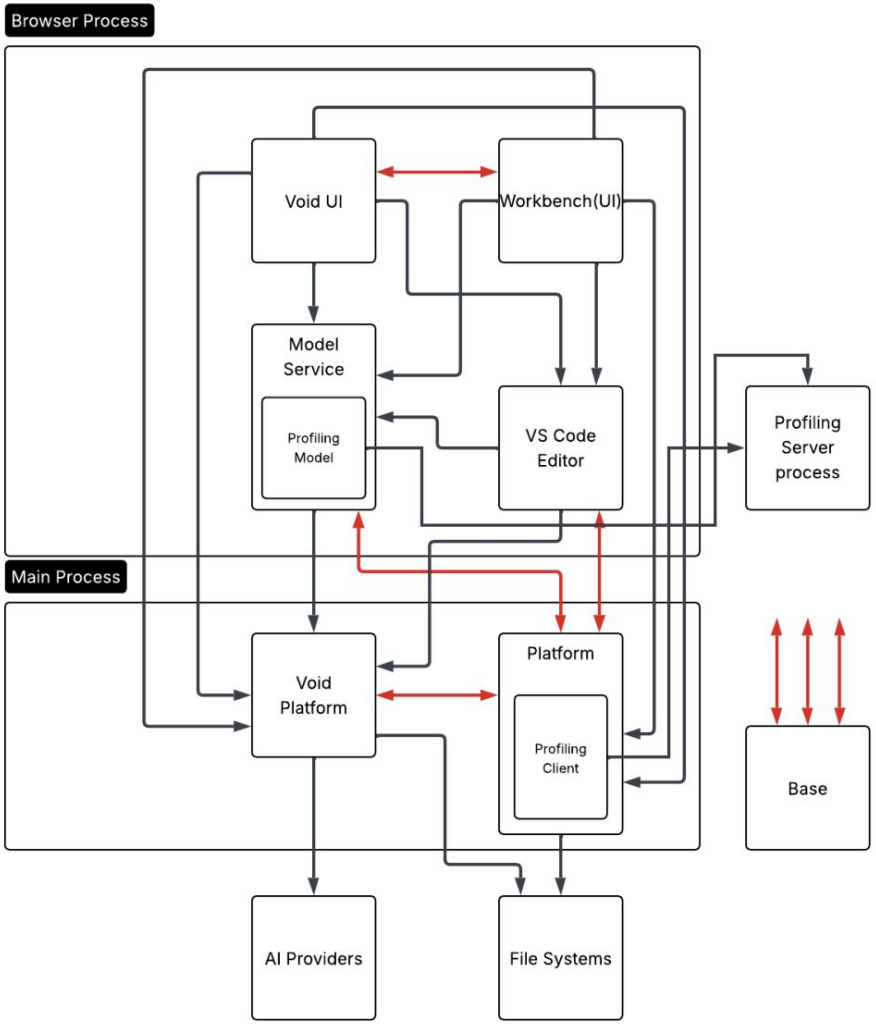
Approach 2: Publish/Subscribe

- Profiling producers publish events: ProfilingStarted, FunctionSampled, HotspotDetected, ProfilingFinished
- Void UI/Workbench subscribe to aggregate events like HeatmapUpdated
- AI providers subscribe to hotspot-related events as needed
- **Pros:** low coupling, easy to add new consumers (logs, experiments, new views)
- **Cons:** event storms, harder to reason about flow and test timing



Approach 3: Client–Server

- Profiling is a separate service;
  IDE (Platform, Model Services) are clients
- Platform requests profiling service to start/stop profiling runs
- Profiling service stores call counts, line hits, timing info
- Model Services and UI query the profiling service for summaries/heatmaps
- **Pros:** clear separation, can scale or reuse the service across tools
- **Cons:** added latency, error handling, deployment complexity



SAAM: Stakeholders & Quality Attributes

- Stakeholders:
  - Application developers
  - IDE maintainers
  - Extension developers
  - QA/performance engineers
  - AI provider developers
  - UX/product team
  - Students/new programmers
- Key quality attributes: performance, responsiveness, testability, modifiability, extensibility, data consistency, understandability

Stakeholder	**Key non functional requirements**
Application developers (end users)	• Performance and responsiveness. The profiler must not slow editing or coding. • Clarity and low cognitive load. Heatmaps and results should be easy to read. • Reliability. Output should be trustworthy and consistent for everyday work.
Void IDE maintainers and core engineering team	• Modifiability and maintainability. The system should be easy to change and debug. • Low complexity. Profiling should not make the IDE much more complex or add many new dependencies.
Extension developers	• Stable and simple APIs. Profiling data should be easy to access without deep knowledge of IDE internals. • Compatibility. Existing extensions should continue to work correctly when profiling is turned on.
Performance and quality assurance engineers	• Testability and determinism. Profiling behaviour should be repeatable and predictable. • Non distortion. The profiler should not change the performance of the program so much that measurements become misleading.
AI provider developers	• Data consistency. Heatmap and profiling data should have a stable format and meaning over time. • Easy integration. Developers should be able to consume profiling outputs without relying on hidden IDE details.
User experience and product team	• User interface responsiveness. Profiling should not make the interface feel slow or jittery. • Seamless experience. The profiling feature should feel like a natural part of the workflow.
Students and new programmers	• Understandability. The profiling feature and heatmap should be easy to understand. • Learning support. Explanations should help users understand performance issues without overload.

SAAM Scenarios (S1–S4)

- **S1 – Interactive profiling run:** run medium project with profiling, keep editor responsive
- **S2 – Explain a hotspot with AI:** right-click hot loop, get AI explanation using profiling + code
- **S3 – Add new visualization:** add a “Performance Timeline” view reusing same profiling pipeline
- **S4 – Profiling in regression tests:** QA runs tests with profiling, compares heatmaps across builds

SAAM Comparison Highlights

- **Performance & responsiveness (S1):**
  - Layered: predictable, simple call path
  - Pub/Sub: risk of event bursts
  - Client–Server: heavy work off UI, but network overhead
- **Testability & determinism (S4):**
  - Layered: easiest to test and mock
  - Pub/Sub: asynchrony & event ordering complicate tests
  - Client–Server: separate server tests + more E2E complexity
- **Modifiability/extensibility (S3):**
  - Pub/Sub best for many independent consumers
  - Client–Server okay with rich API
  - Layered sufficient for few built-in views

SAAM Conclusion

- Prioritised:
  - Predictable, low-overhead interactive profiling (S1)
  - High testability and simple control flow (S4)
  - Understandable architecture for student developers
- Layered approach:
  - Fits existing Void IDE structure
  - Minimal new concepts vs. pub/sub and client–server
  - Good enough extensibility for A3 scope
- Pub/Sub and client–server are viable future options if profiling ecosystem grows

High-Level Architectural Impact

- Keep main subsystems: Platform, VoidPlatform, Model Services, Void UI, Workbench, AI Providers, File Systems
- New responsibilities:
  - Platform + profiler: collect runtime data
  - VoidPlatform + Model Services: aggregate into heatmap model
  - Void UI & Workbench: expose commands and visualizations
  - AI Providers: consume profiling summaries in prompts
- No new top-level layers; profiling is a **cross-cutting concern** threaded through existing ones

Concrete Architecture: Impacted Components

- **Platform / VoidPlatform:**
  - New profiling service interface in shared “common” area
  - Implementations in node and Electron main parts
  - Run/debug code gains small dependency to enable/disable profiling
- **Model Services:**
  - New profiling model (heatmap per file/line, hotspot queries)
  - Existing models import it when they need heatmap data
- **Workbench / Void UI:**
  - New controller/view-model for profiling commands
  - Editor overlay, hover, and optional “hotspots” panel
- **AI Providers:**
  - Prompt builder extended to include profiling summaries
  - Dependencies remain directional: UI ➜ models ➜ platform

Effects on Key Qualities

- **Maintainability:**
  - Profiling logic encapsulated in dedicated service + model
  - Narrow APIs (“run with profiling”, “get heatmap”) isolate changes
- **Evolvability:**
  - Profiling model reused for new views (timeline, call tree, etc.)
  - Low impact on existing components when adding new visualizations
- **Testability:**
  - Deterministic layered control flow; easy to mock each layer
  - Unit tests for profiling service, aggregation logic, and UI wiring
- **Performance:**
  - Lightweight sampling; aggregation off UI path where possible
  - Profiling is opt-in and can be limited by sampling rate/run duration

Interactions with Existing Features

- **Run / Debug:**
  - Profiling as optional flag on existing run flows
- **Editor decorations:**
  - Heatmap overlays coexist with diagnostics, breakpoints, code lenses
  - Need to manage decoration priorities/visual clarity
- **AI tools:**
  - Some AI commands upgraded to use profiling when available
  - Must degrade gracefully when no profiling data exists
- **File/Snapshot handling:**
  - Optional caching of profiles in workspace; no changes if disabled

Testing Plan

- **Unit tests:**
  - Platform/VoidPlatform APIs: start/stop profiling correctly
  - Model Services profiling model: correct mapping to lines/files & hotspot calc
  - UI: commands (Run with profiling, Toggle heatmap, Clear results) update decorations
- **Integration & regression tests:**
  - End-to-end: Workbench ➔ Platform ➔ Model Services ➔ UI heatmap
  - Compare behaviour with/without profiling (no behavioural regressions)
  - Run diagnostics, completion, AI explanations while heatmap active
- Goal: verify both **feature correctness** and **interactions** with existing features

Potential Risks

- **Security & privacy:**
  - Profiles contain stack traces and behaviour; risk of leaking proprietary logic
  - Mitigation: workspace-scoped access, explicit prompts before sending data to AI
- **Scalability & capacity:**
  - Large projects / many runs ➜ big profiles & heavy aggregation
  - Mitigation: sampling caps, run duration limits, optional profile persistence
- **Performance & resource usage:**
  - Sampling overhead; UI lag on large files if heatmap frequently refreshed
  - Mitigation: opt-in profiling, careful refresh strategy, downsampling

AI Collaboration Report

- **Model selection:**
  - Primary: ChatGPT (GPT-5, Nov 2025)
  - Others (Claude, Gemini) tried but not used in final text
- **Role & tasks:**
  - Summarisation, rewriting, table formatting, checking sequence diagram logic
  - No architectural decisions delegated to AI
- **Interaction protocol:**
  - Context-first prompts with your own drafts + explicit instructions
  - AI used to refine, not invent, content
- **Quality control & impact:**
  - All AI-generated text reviewed/edited by team
  - SAAM scores and architectural choices validated manually
  - Rough quantitative impact: ~15–18% of textual formatting/writing effort

Conclusion

- Real-time execution heatmap ties **runtime behaviour** and **AI** directly into Void IDE
- Layered architecture chosen via SAAM for:
  - Predictable performance
  - High testability
  - Clear integration with existing structure
- Enhancement extends both conceptual & concrete architectures without rewriting them
- Solid testing plan, risk analysis, and AI collaboration approach in place