Real-Time Code Execution Heatmap for Void IDE

Mason Zhang
22yjz@queensu.ca

Jack Atkinson
22hrsd@queensu.ca

Reyan Sehgal
23sw2@queensu.ca

Jeremie Trepanier
jeremie.trepanier@queensu.ca

Duncan Mackinnon
duncan.mackinnon@queensu.ca

December 1, 2025

Abstract

This report proposes a new feature for Void IDE called real-time code execution heatmap. Void IDE already uses large language models to explain and edit code, but it does not see how the code behaves when it runs. Our idea is to add a lightweight profiler that records which lines and functions execute most often and how long they take, and then show this information as a color overlay directly in the editor. The same profiling data is also sent to the existing AI assistant so it can explain performance problems and suggest fixes using real runtime measurements, not just static code. We first describe the feature and why it is useful for both new and experienced developers. Then we show how to integrate it into the current layered architecture of Void IDE and what changes are needed in key components such as Platform, Model Services, Void UI and Workbench. We also present and compare three design options for the feature using an SEI SAAM style analysis, and we discuss the impact on qualities such as performance, maintainability and extensibility.

1 Introduction

Void IDE is an open source code editor that extends the Visual Studio Code codebase with extra AI support. It already lets developers ask questions about their code and get suggestions from large language models. For this course we studied the Void IDE repository and built a conceptual and concrete architecture in earlier assignments. That work focused mostly on the existing features such as the workbench, the AI assistant and the way extensions are wired into the platform.

In this report we propose a new feature for Void IDE. The idea is to add a real-time code execution heatmap that shows how the program behaves when it runs. The heatmap will use profiling data collected at runtime and will paint hot and cold regions directly in the editor. On top of that, the existing AI assistant will be able to use this profiling data when it explains performance problems or suggests optimizations. Instead of looking only at static source code, the AI will see real measurements such as how often a function runs and how long it takes.

To support this feature we need to extend the current architecture. The layered structure from our previous work stays in place, but new responsibilities are added. The Platform and File System related parts need to collect and store profiling data. The Model Services layer needs to translate raw samples into a form that the editor and the AI can use. The Void UI and Workbench need to show the heatmap and provide commands that let the user explore hotspots and ask for explanations. In the report we explain where these changes fit and how they interact with existing components.

The report is organized as follows. First we give an overview of the new feature and why it is useful for Void IDE users. Then we describe three different architectural approaches to implement it: a layered style, an event based publish and subscribe style, and a client–server style. After that we look at the impact on subsystems and on specific files and dependencies in the concrete architecture. We also compare the three approaches using an SEI SAAM style analysis, focusing on different stakeholders and non-functional qualities such as performance, maintainability and testability. We close with a short discussion of risks and future work and include an AI collaboration report as required by the assignment.

2 Overview of the New Feature

Today Void IDE already talks to language models and can reason about code based on the text in the editor. However, it does not observe how that code executes. The new feature adds a small runtime profiler that tracks which lines and functions run most often and how long they take. This data is then shown directly inside the editor as a coloured heatmap over the source code. Hot areas that run frequently or take a long time appear with stronger colours, while cold areas stay faint. At a glance the user can see which loops are busy, which functions cause slowdowns, and which paths are rarely used.

The heatmap is not only a visual overlay. It also serves as the bridge between real execution data and the existing AI features in Void. When the user hovers over a hot section or clicks a flagged bottleneck, the IDE can package the runtime statistics together with the relevant source code and send it to the language model. Instead of guessing based only on static code, the model now receives precise information such as how many times a function was called, how much time it consumed, and how it compares to other parts of the program. The assistant can then explain why that section is slow, highlight suspicious patterns such as nested loops or repeated input and output operations, and propose concrete refactorings or algorithm changes.

This feature brings several benefits. It shortens the feedback loop for performance tuning because developers no longer need to run separate profiling tools or read complex logs. They can stay inside Void IDE and treat performance issues much like regular code issues, with inline highlights and contextual explanations. It also helps newer programmers who may not yet know how to profile a program, since the tool surfaces hotspots automatically and explains them in plain language. Finally, the feature lays a foundation for future enhancements such as tracking memory use, visualising call paths, or comparing runs over time, all using the same pipeline for collecting execution data and feeding it into the AI assistant.

3 Approach 1: Layered Architecture Style

In this approach we keep the current layered structure and thread the new feature through the existing layers. We add a small profiling module or subcomponent that translates raw samples into line based heatmap information. It knows how to map function names and file paths to editor lines and how to aggregate multiple runs. It also exposes a service for queries so that other parts of the system can ask for the heatmap of a file or for the top hotspots of the current project.

In the Void UI and Workbench components, we add the actual user facing parts. The Workbench contributes new views and commands such as "show execution heatmap for this file" or "explain this hotspot." The Void UI component adds the coloured overlay in the editor and the hover that shows a summary for each hot region. When the user asks for an explanation the Workbench calls into the AI Providers layer with both the source code and the profiling data from the Model Services. The AI Providers do not need to know how the data was collected; they just receive a richer prompt. This approach fits the current architecture well because each layer keeps its role. The main impact is that we add one new service in the lower platform layer and one new model and view in the upper layers and connect them through existing service interfaces.

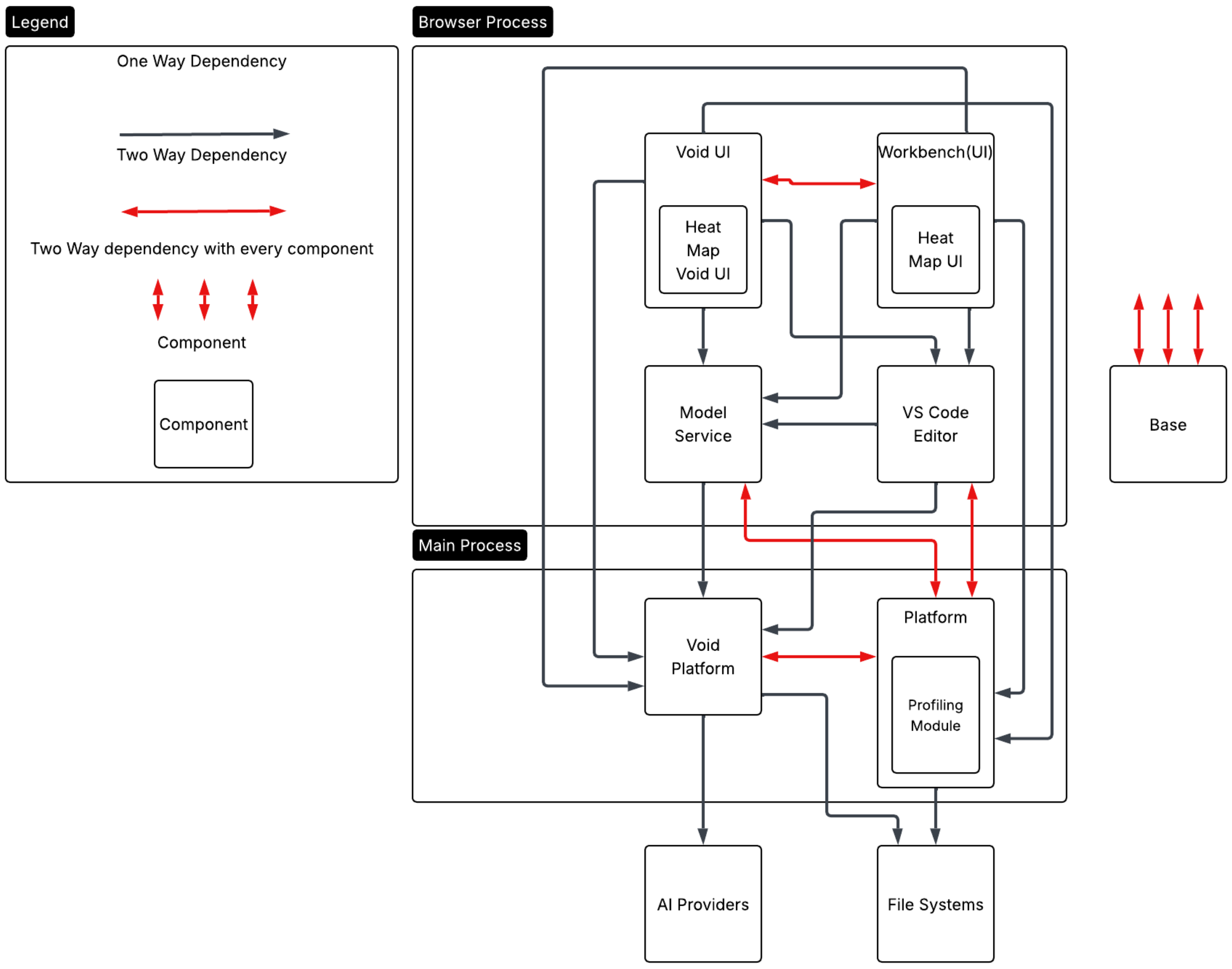


Figure 1: Conceptual view of the real-time execution heatmap in the layered architecture approach. Profiling is added as a dedicated service in the platform layer, and profiling data flows upward through existing model and user interface layers.

4 Approach 2: Publish and Subscribe Style

In this approach the profiling feature is driven by events following a publish and subscribe style. When the user starts a run with profiling enabled, lower level components in Platform and VoidPlatform publish events such as `ProfilingStarted`, `FunctionSampled`, `HotspotDetected` and `ProfilingFinished` on a shared event bus. They do not know who is listening. They simply report what happens during execution.

Higher layers subscribe to the events they care about. Void UI and Workbench subscribe to aggregate level events such as `HeatmapUpdated` or `HotspotListUpdated`. When these events arrive, the editor updates its overlay colours and side panels without directly calling into the profiler. The AI Provider layer can subscribe to `HotspotDetected` or `ProfilingFinished` and, when needed, combine the latest aggregate data with the current source code in a prompt for the language model.

This event based style keeps coupling low. Profiling producers can evolve without breaking the user interface, and additional listeners can be added later, for example a logging extension or an experiment collector. The trade off is that reasoning about the flow is more indirect, because effects are spread across subscribers, so care is needed in documentation and testing.

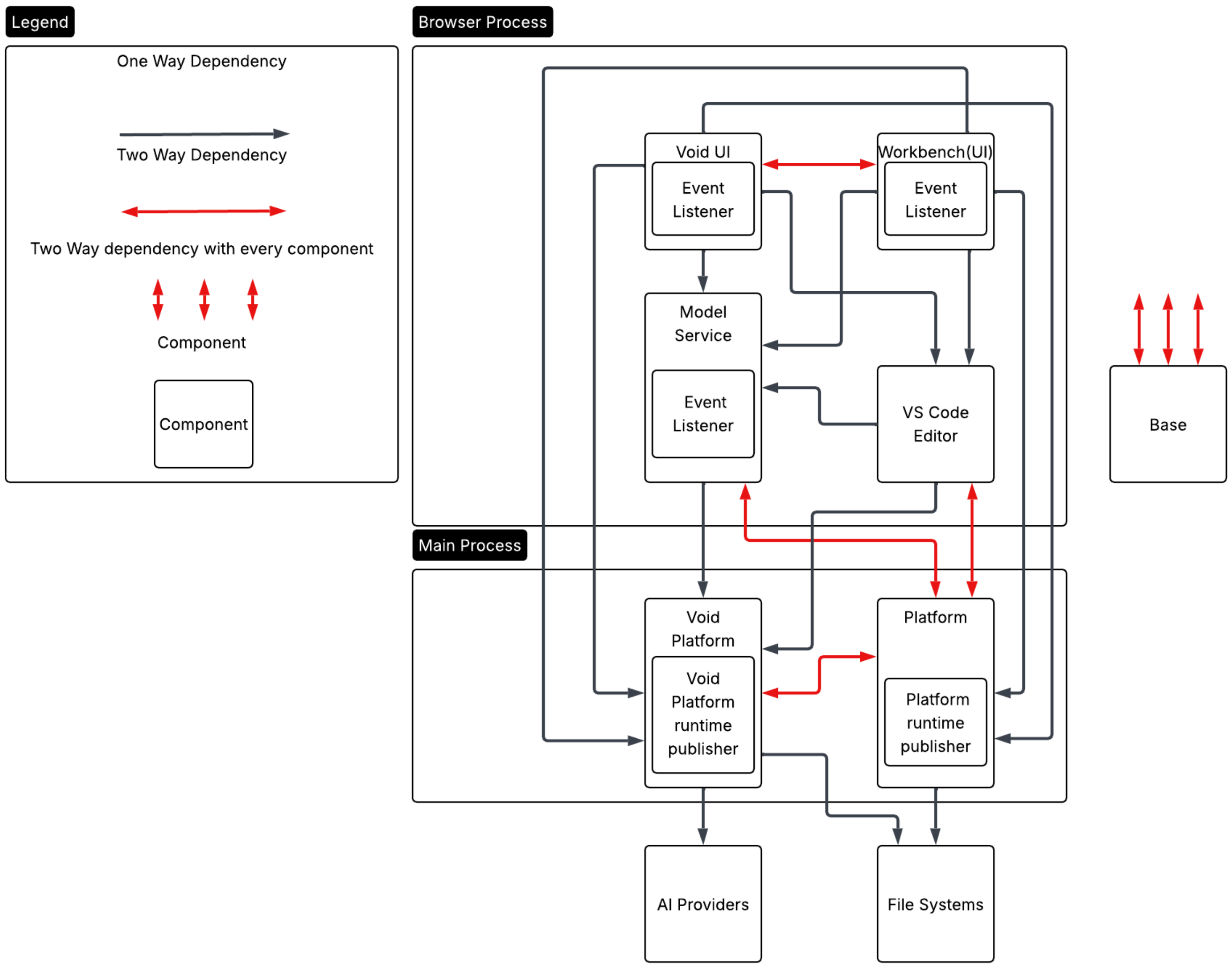


Figure 2: Conceptual view of the real-time execution heatmap in the publish and subscribe approach. Profiling components publish events on a shared bus, and multiple consumers such as the user interface and AI providers subscribe independently.

5 Approach 3: Client Server Style

In this approach the profiling feature is treated as a separate service that acts like a small server. The rest of Void IDE behaves as a client. When the user starts a run with profiling turned on, the Platform layer sends a request to a profiling service with basic information about the program and project. The profiling service then attaches to the running program or wraps the runtime. It records function calls, line hits, and timing information, and keeps this data in its own model. On the client side the Platform only needs simple commands such as start profiling, stop profiling, and get results.

Model Services also act as a client of this profiling service. When Void UI or the Workbench need heatmap data, they ask Model Services, which then request a summary for a file from the profiling service. The service returns compact data such as hit counts and average time per line, and Model Services convert this into heatmap values for the editor. When the user clicks a hotspot and asks for an explanation, the Workbench calls the AI Providers with both the source code and a focused report from the profiling service. This client server style keeps profiling logic clearly separated and makes it easier to reuse or scale the profiler, at the cost of some extra communication and error handling between the IDE and the profiling service. Figure 3 shows the high level structure of this client server design.

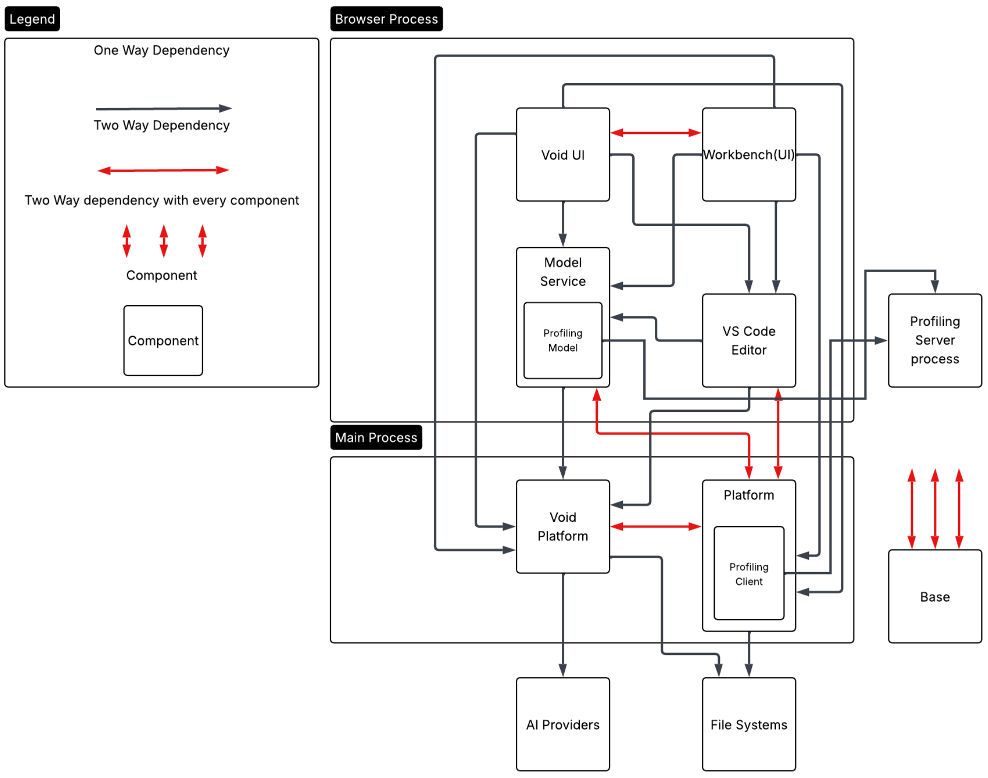


Figure 3: Client server approach for the profiling feature, with Platform and Model Services as clients and a separate profiling service as server.

6 SAAM Analysis

In this section we follow the Software Architecture Analysis Method (SAAM) to compare the three architectural approaches for the profiling and heatmap feature. We first identify stakeholders and quality attributes, then define concrete scenarios (benchmark tasks) and finally evaluate how well each architecture supports these scenarios.

6.1 Stakeholders and Non Functional Requirements

Table 1 lists the main stakeholder groups for the profiling and heatmap feature and the most important non functional requirements for each group.

6.2 Scenarios (Benchmark Tasks)

Following SAAM, we define a small set of concrete scenarios that stress the important qualities for our stakeholders. Each scenario is a realistic task that Void IDE must support once the profiling feature is in place.

Scenario S1: Interactive profiling run. A developer enables “Run with profiling” on a medium sized project, edits code while the program runs, and expects the editor and heatmap overlay to remain responsive. Stakeholders: application developers, UX team. NFRs: performance, responsiveness, reliability.

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.
New programmers	• Understandability. The profiling feature and heatmap should be easy to understand. • Learning support. Explanations should help users understand performance issues without overload.

Table 1: Stakeholders and non functional requirements for the profiling feature.

Scenario S2: Explain a hotspot with AI. A developer right clicks a hot loop and chooses "Explain this hotspot." Void IDE collects the relevant profiling and code context and sends it to the AI provider, which returns an explanation and suggestions. Stakeholders: application developers, AI provider developers, UX team. NFRs: data consistency, integration ease, clarity.

Scenario S3: Add a new visualization of profiling data. A maintainer wants to add a new "Performance Timeline" panel that also consumes profiling samples from the same runs. The rest of the IDE should not need major changes. Stakeholders: maintainers, extension developers. NFRs: modifiability, extensibility, API stability.

Scenario S4: Use profiling in automated regression tests. A QA engineer runs a test suite under profiling and compares heatmaps across builds to detect performance regressions. Runs must be repeatable and profiling must not distort timings too much. Stakeholders: QA engineers, maintainers. NFRs: testability, determinism, non distortion, reliability.

These scenarios act as our "benchmark tasks" in the sense of SAAM: they represent common and important ways the feature will be used and maintained over time.

6.3 Comparison of Architectural Approaches

We compare the three architectural approaches from the point of view of the qualities that matter to these stakeholders and scenarios. Table 2 summarises the main trade offs at a high level.

Table 2: Qualitative comparison of the three architectural approaches.

Attribute	**Approach 1 (Layered)**	**Approach 2 (Publish and subscribe)**	**Approach 3 (Client server)**
Performance and overhead	• Low runtime overhead and a simple call flow. • Work happens in clear steps inside the layers. • If too much work is done in a synchronous way, user interface updates can pause under heavy load, but this is easy to see and tune.	• Event bursts can increase CPU usage. • Overhead grows when many subscribers are attached. • Batching or throttling can reduce the impact but adds tuning work.	• Extra cost for calls between client and server. • Profiling work can run in a separate process so it does not block the main user interface. • Heavy analysis can be moved off the main machine or shared between projects if needed.
Modifiability	• Moderate modifiability. • Changes often need updates in several layers, but the paths are explicit and easy to trace. • Adding a new consumer usually means extending an existing service interface.	• High modifiability. • New subscribers can be added without changing the profiling core. • Components can evolve more independently when they agree on event types.	• The profiling service can change without touching most client code. • Changes to the service contract need matching changes in all clients. • It is easy to swap the profiling service for another one that uses the same interface.
Testability	• High testability. • The flow is deterministic and ordered. • It is easy to mock services and write regression tests.	• Lower testability. • Asynchronous timing and event ordering can introduce non determinism. • Tests may need event simulation and explicit control over timing.	• The profiling server can be tested in isolation. • Clients can be tested with mocked server replies. • Full tests also need to cover network or process errors and timeouts.

Continued on next page

Table 2 – *Continued from previous page*

Attribute	**Approach 1 (Layered)**	**Approach 2 (Publish and subscribe)**	**Approach 3 (Client server)**
Coupling and separation of concerns	• Stronger coupling between layers. • Consumers depend directly on upstream services. • Some changes can ripple through the call chain.	• Very low coupling between publishers and subscribers. • Components do not know about each other directly. • Some complexity moves into the event bus.	• Clear split between client side code and the profiling server. • Clients are coupled to the service interface but not to its inner design. • The server can keep profiling concerns separate from user interface concerns.
Extensibility and scalability	• Good extensibility for our current needs. • New features can be added through existing service interfaces. • Very large growth in consumers could increase friction.	• Very high extensibility. • New listeners can be attached without changing the architecture. • Well suited to a large ecosystem of tools that consume profiling data.	• The profiling server can be scaled or moved to another host if needed. • Multiple IDE instances can share one profiling service in larger teams. • There is extra work to manage deployment and versioning of the server.
User interface responsiveness and predictability	• Stable and predictable user interface updates. • The sequence of calls is clear. • If processing becomes slow, we can move work off the main thread in specific places.	• Updates can become noisy or jittery if many events arrive quickly. • Throttling and buffering are needed for a smooth experience.	• The main user interface stays light because heavy work happens on the server. • Heatmap updates depend on server response time and network conditions. • We need clear timeouts and loading states to keep behaviour predictable.
Reliability and fault isolation	• Errors can propagate through service calls and affect several layers. • The call stack is clear, which helps with debugging.	• Better fault isolation. • If a subscriber fails, other subscribers can continue to run. • Failures can be harder to trace back through events.	• If the profiling server fails, the IDE can continue without profiling. • Errors are contained at the process or network level. • We must handle retries and partial data to avoid confusing the user.
Understandability and cognitive load for developers	• Easy to understand. • Data flow is linear and traceable. • Friendly for new contributors.	• Harder to understand. • Event based behaviour is less visible. • Developers often need logs or tracing tools to follow the flow.	• The client server split is clear at a high level. • There is extra mental load around network errors and remote calls. • Teams must think about two worlds: client code and server code.

6.4 Scenario Based Evaluation

We now use the scenarios S1 to S4 to compare all three approaches.

6.4.1 Scenario S1: Interactive profiling run

With the layered approach, profiling is a platform service that plugs into the existing run and debug flow. The Workbench makes one call to start profiling, and Model Services ask for summaries when the editor needs to draw or refresh the heatmap. The number of calls is small and clear, so it is easy to see where time is spent and to tune performance if the editor feels slow.

With the publish and subscribe approach, profiling events are sent all the time to any listener on the event bus. This makes extra live views easy to add, but many subscribers can cause bursts of events and short spikes in work. The client server approach keeps heavy work in a separate server process, which helps keep the user interface light, but every request and reply adds cost and may be delayed.

For S1, the layered approach gives the most predictable and simple behaviour. The client server approach is acceptable but more complex. The publish and subscribe style is weakest because of the risk of too many events during a run.

6.4.2 Scenario S2: Explain a hotspot with AI

In the layered architecture, the Workbench asks Model Services for a hotspot summary and then calls the AI Providers with the code and the summary. The data path is short and easy to follow. If we change what we send to the language model, we mostly update one model component.

In the publish and subscribe architecture, the AI code must either subscribe to profiling events or depend on another subscriber that builds summaries. This is flexible but spreads logic across several listeners. In the client server approach, the Workbench still has a simple flow: it asks Model Services to call the profiling server and then passes the result to the AI. The main extra work is handling server errors.

For S2, the layered and client server approaches both work well and are easy to understand. The layered approach is slightly simpler because everything stays inside the same process and service stack. The publish and subscribe approach is more flexible but harder to reason about for this focused AI use case.

6.4.3 Scenario S3: Add a new visualization of profiling data

For a new view of profiling data, such as a call tree or timeline, the publish and subscribe approach is very strong. A new view can simply subscribe to the needed events without changing the profiler or existing views.

With the layered approach, a new visualization usually needs changes to the profiling model in Model Services and a new Workbench view that calls into it. This touches more code but follows clear patterns. In the client server approach, new views call the profiling server through its API. If the API is rich enough, many views can be added without server changes, but sometimes the server contract must be extended.

For S3, the publish and subscribe approach is best if we expect many independent consumers. The client server approach is in the middle. The layered approach is least flexible but still fine for a small number of built in visualizations.

6.4.4 Scenario S4: Use profiling in automated regression tests

In the layered design, tests call the same services as the Workbench. A test can start a run, wait for it to finish, and ask Model Services for summary data to compare with a baseline. The order of calls is fixed and stays inside one process, so runs are more repeatable and easier to mock.

In the publish and subscribe architecture, tests must also control which subscribers are active. Extra listeners can change timing or consume events, making runs less stable. In the client server approach, we can test the server and clients separately, but full end to end tests must also handle network errors and timeouts, which adds more reasons for tests to fail.

For S4, the layered approach is easiest and most stable. The client server approach can work but needs extra care. The publish and subscribe approach is the hardest to test in a repeatable way.

6.5 SAAM Conclusion

Based on this SAAM analysis with three approaches, we still choose the layered platform approach, Approach 1, as our main design for the profiling and heatmap feature.

For our project the most important qualities are:

- Predictable performance and low extra cost for users during interactive profiling runs in Scenario S1.
- High testability and simple control of behaviour for regression tests in Scenario S4.
- Clear and easy to follow code paths for developers when wiring AI explanations in Scenario S2.

Approach 1 supports these key needs well. The control flow stays explicit and easy to trace. Profiling logic fits into existing platform services and model layers, so we do not need to add new processes or a new event bus to Void IDE.

Approach 2, the publish and subscribe style, brings strong long term benefits for extensibility, especially for Scenario S3 where many different visualizations or extensions may listen to profiling data. However, it makes the system harder to test and to understand, and it can hurt user interface smoothness if event traffic grows large.

Approach 3, the client server style, gives a clean separation between the editor and the profiling service and can help with scaling in larger teams. It is a good option if Void IDE later wants to reuse the same profiler across many tools. For our current project, this extra power comes with added complexity in deployment, error handling, and testing.

Given the current scope of Void IDE and the goals of this course, we judge that the layered approach offers the best overall balance between simplicity, reliability, and future growth. In the rest of the report we therefore focus on Approach 1 as the chosen architecture for the new profiling and heatmap feature.

7 The New Feature's Impact on Subsystems

At the high level, the introduction of runtime profiling and heatmap visualization creates a new cross cutting concern that spans multiple major subsystems: Platform, Model Services, VoidPlatform, Workbench, UI, AI Providers, and (optionally) File Systems.

7.1 Execution Profiling (New Subsystem Inside Platform)

The biggest new piece is a small profiling subsystem. It sits close to where programs are run. When the user chooses "Run with profiling," this subsystem attaches to the running program. It records basic facts like which function ran, how often it ran, and how long it took. It does not know anything about the editor or the AI. Its only job is to gather this data efficiently and offer a simple way for other parts of the IDE to read it.

7.2 Platform and VoidPlatform

The existing Platform subsystem is where runs and debugging are already managed. We extend it so it can turn profiling on and off for a process and pass the right flags when starting a program. Platform also passes lifecycle events to the profiler so data collection starts and stops at the right time.

VoidPlatform then adds a layer on top of this raw data. It groups samples by file, function, and line range and keeps track of which workspace and run they belong to. VoidPlatform offers a clearer API such as "give me the heatmap for this file" instead of low level events.

7.3 Model Services

Model Services already knows about projects, files, and symbols. With this feature it gets a new performance model. This model stores heatmap data keyed by file and line range and keeps it in sync with updates that come from VoidPlatform. Other subsystems can now ask Model Services things like "what are the hottest functions in this file" or "give me stats for the currently selected lines." This keeps all knowledge about how to map runtime data to source code in one place.

7.4 VS Code Editor

The editor needs to be able to draw the heatmap. It already supports decorations and minimap coloring, so we mainly add a new kind of decoration that uses profiling intensity to pick a color. The editor also needs a way to ask Model Services for data when the user moves the cursor or selects a region. The core editor stays generic. Profiling is an optional feature that is turned on by the Workbench and Void UI when profiling data is available.

7.5 Base

The Base subsystem holds common helpers and shared types. For this feature it will likely gain simple utilities for working with profiling numbers. Examples include mapping raw times to a 0 to 1 scale for colors, formatting times in milliseconds, and defining TypeScript interfaces for profiling metrics. This lets Platform, Model Services, Void UI, and AI Providers all agree on the same data shape without depending on each other directly.

7.6 Void UI and Workbench

Void UI is where the user sees the heatmap. It uses the editor's decoration APIs and the data from Model Services to color lines and minimap regions. When the user hovers over a hot area the UI asks Model Services for the stats and shows a small popup with counts and timings. When the user right clicks and chooses "Explain this hotspot," the UI sends a command to the Workbench with the location of that hotspot.

The Workbench then plays its usual role as coordinator. It defines new commands like "Run with profiling," "Toggle heatmap," and "Explain bottleneck." It wires these commands into menus and buttons. It calls Platform to start or stop profiling, calls Model Services to get the latest snapshot, and calls AI Providers to get explanations. The Workbench may also host a new "Performance" view that lists the worst hotspots and lets the user jump to them.

7.7 AI Providers

AI Providers already build prompts from code and user requests. With the new feature they can optionally add profiling summaries to those prompts. They get this data from Model Services, not from Platform directly. For example, when asked to explain a hotspot the AI provider receives the code and a small JSON object with call counts and timings. It then uses this to give a more grounded explanation and more precise suggestions. The AI layer still does not know how data was collected. It only knows that performance metrics are available.

7.8 File Systems

The File Systems subsystem only needs small changes, and only if we decide to persist profiles. If we store runs on disk, File Systems needs to be aware of a new profile file type and keep it in sync when files are moved or renamed. If we choose to keep profiles only in memory until the IDE closes, then File Systems is almost unchanged. It may only be used to cache temporary profile data in a workspace folder if that is needed later.

Overall the impact of the new feature is spread across several subsystems, but each change is small and keeps to the existing responsibilities. Platform plus the new profiler collect the data. Model Services and VoidPlatform turn it into a usable model. Void UI and Workbench present it and trigger AI explanations. AI Providers consume it to improve their answers. The existing architecture remains valid, with one new cross cutting concern threaded through it in a controlled way.

8 Impact on Concrete Architecture (Files and Dependencies)

The new profiling and heatmap feature changes the concrete architecture in a few focused places, but it does not rewrite how Void IDE is structured. Most changes happen inside the Platform /

VoidPlatform, Model Services, Workbench / Void UI, and AI Providers components.

At the lowest level, we add profiling support inside Platform and VoidPlatform. This means introducing a new profiling service in the platform area, with an interface in the shared "common" part and implementations in the node and electron main parts. These new files start and stop profiling when a program is run from Void IDE, measure how long functions take, and count how often they are called. They store this data per function and per file. To connect this to the rest of the system, some existing "run" or "debug" code in Platform gains a small extra dependency on this profiling service so it can turn profiling on or off based on user settings. This addition is local to the lower layers and does not affect UI code directly.

We extend Model Services and introduce a profiling model that knows how to work with the raw data coming from the platform profiler. This model maps function names and file paths back to actual lines in the editor, aggregates data across runs, and produces heatmap data for each file, for example a score for each source line. It provides simple functions such as "get heatmap for a given file" or "list the top hotspots for the current workspace." Some existing model files that already deal with editors, files, or projects will import this new profiling model so they can ask for heatmap information when needed. The File System component may only need very light changes or none at all, unless we choose to persist profiling results on disk, in which case a small helper can be added to save and load profiling snapshots using existing file APIs.

The most visible changes are in Workbench and Void UI. In the Workbench component we add a small controller or view model dedicated to profiling. This new code calls the Model Services profiling model to get heatmap and hotspot data and wires it to commands such as "Show Execution Heatmap" or "Explain Hotspot." These commands live alongside the existing Void specific workbench actions and follow the same patterns. The Workbench also calls into the AI Providers layer when the user asks for an explanation, passing both the source code and the profiling summary.

In Void UI we add the visual parts: an editor overlay that colours lines based on how "hot" they are, a simple hover that shows profiling numbers, and possibly a small panel listing hotspots. These UI pieces use existing VS Code editor APIs from the VS Code Editor component to draw decorations and controls. They depend on the new Workbench controller to receive data, but they do not talk directly to the lower platform layers. In the dependency graph this appears as new arrows from Workbench and Void UI down to the profiling model in Model Services, but the overall direction of dependencies stays consistent: UI depends on models, models depend on platform.

The AI Providers component also gets a small extension. Some existing prompt building code is updated so it can accept profiling data as another input, along with the usual code snippet and file context. We may add a small helper function that converts heatmap values into short text descriptions such as "this function ran many times and took a large share of total runtime." This helper only depends on the profiling data types defined in Model Services, not on any node or Electron internals. The lower level networking code that sends requests to the language model can stay the same, since it just receives a richer prompt.

9 Concrete Files & Dependencies

This section lists the main files and folders that must be added or changed to support the profiling and heatmap feature. The list matches the conceptual architecture: Platform collects data, Model Services stores and processes it, Workbench and UI display it, and AI Providers use it when explaining hotspots.

9.1 Platform / VoidPlatform

- `platform/profiling/` (new folder)

  - `index.ts` — simple interface for start/stop profiling.
  - `profilerNode.ts`, `profilerElectron.ts` — actual implementations.
  - `sampler.ts` — small module that collects samples.
- `platform/runManager.ts` (updated) — adds the “run with profiling” flag and calls the profiling service.

9.2 Model Services

- `model-services/profiling/` (new folder)
  - `profilingModel.ts` — stores per-file and per-line data.
  - `lineMapper.ts` — maps raw samples to editor line numbers.
  - `api.ts` — provides `getHeatmap()` and `getHotspots()`.
- `model-services/index.ts` (updated) — registers the profiling model.

9.3 Workbench and Void UI

- `workbench/profilingController.ts` — connects commands to Model Services.
- `workbench/commands.ts` (updated) — adds “Run with profiling,” “Toggle heatmap,” and “Explain hotspot.”
- `void-ui/editor/heatmapDecorator.ts` — draws coloured lines in the editor.
- `void-ui/editor/hoverProfiling.ts` — shows line counts and timings on hover.
- Optional: `workbench/views/performanceView.tsx` — simple list of hotspots.

9.4 AI Providers and Common Code

- `ai-providers/promptBuilder.ts` (updated) — adds profiling data to AI prompts.
- `common/types/profiling.ts` — shared types (line stats, hotspot summaries).
- `common/utils/formatProfiling.ts` — formats numbers and colours (simple helper).

9.5 Link to Conceptual Architecture

- **Platform** → profiling service files.
- **Model Services** → profiling model + mapping code.
- **Workbench/UI** → controllers, commands, editor decorations.
- **AI Providers** → updated prompt builder.

10 Use Cases and Sequence Diagrams

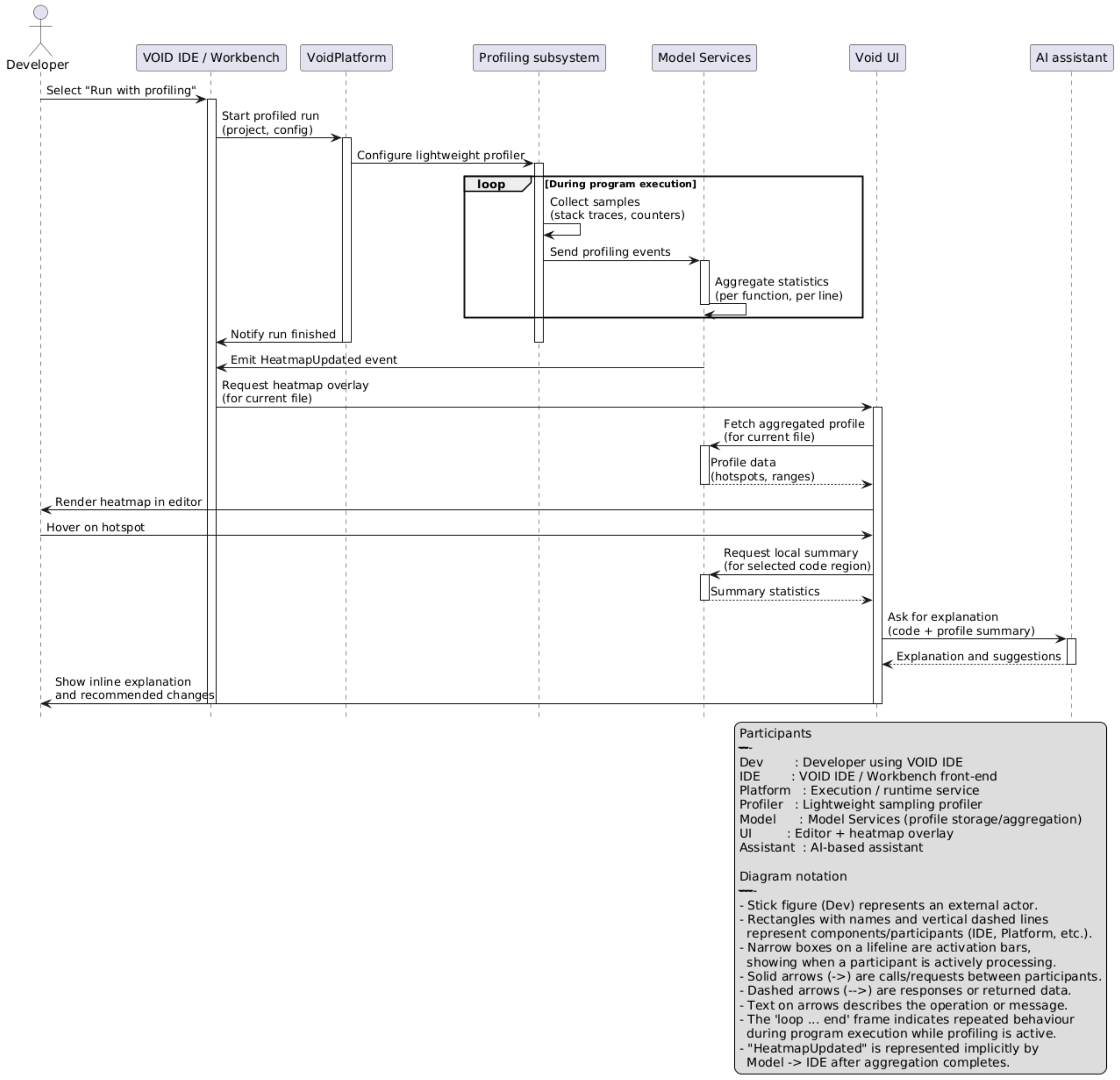
This section describes two representative ways the real-time code execution heatmap is used inside Void IDE, using a platform approach in which shared profiling capabilities are provided centrally. Together, they show how profiling data flows through VoidPlatform, Model Services, Void UI, and the Workbench, and how the new feature supports both day to day development and deeper code understanding.

10.1 Use Case 1: Investigating a Slow Feature

A developer notices that a particular command in an application feels slow. From the Workbench in Void IDE, they choose “Run with profiling” for the project. The Workbench forwards this request to VoidPlatform, which starts the program with the lightweight profiler enabled.

As the program runs, the profiling subsystem periodically samples the call stack and other performance counters, emitting profiling events. Model Services collects these events, aggregates them into per function and per line statistics, and updates its profiling model. When the run completes, Model Services exposes the aggregated profile through its API and raises a `HeatmapUpdated` event.

The Workbench and Void UI subscribe to this event. Void UI requests the relevant profile from Model Services and overlays the heatmap on the source file for the current project, colouring hotter regions more strongly than colder ones. When the developer hovers over a hotspot, the editor requests a local summary for that region (for example, execution count and total time). The Workbench combines this summary with the surrounding code and sends it to the AI assistant. The assistant responds with an explanation of why the code is slow and suggests refactorings or alternative approaches. The developer can then address the performance issue directly in Void IDE without switching to a separate profiling tool.


Developer
VOID IDE / Workbench
VoidPlatform
Profiling subsystem
Model Services
Void UI
AI assistant
Select "Run with profiling"
Start profiled run (project, config)
Configure lightweight profiler
loop
[During program execution]
Collect samples (stack traces, counters)
Send profiling events
Aggregate statistics (per function, per line)
Notify run finished
Emit HeatmapUpdated event
Request heatmap overlay (for current file)
Fetch aggregated profile (for current file)
Profile data (hotspots, ranges)
Render heatmap in editor
Hover on hotspot
Request local summary (for selected code region)
Summary statistics
Ask for explanation (code + profile summary)
Explanation and suggestions
Show inline explanation and recommended changes

Participants

Dev : Developer using VOID IDE
IDE : VOID IDE / Workbench front-end
Platform : Execution / runtime service
Profiler : Lightweight sampling profiler
Model : Model Services (profile storage/aggregation)
UI : Editor + heatmap overlay
Assistant : AI-based assistant

Diagram notation

- Stick figure (Dev) represents an external actor.
- Rectangles with names and vertical dashed lines represent components/participants (IDE, Platform, etc.).
- Narrow boxes on a lifeline are activation bars, showing when a participant is actively processing.
- Solid arrows (->) are calls/requests between participants.
- Dashed arrows (-->) are responses or returned data.
- Text on arrows describes the operation or message.
- The 'loop ... end' frame indicates repeated behaviour during program execution while profiling is active.
- "HeatmapUpdated" is represented implicitly by Model -> IDE after aggregation completes.

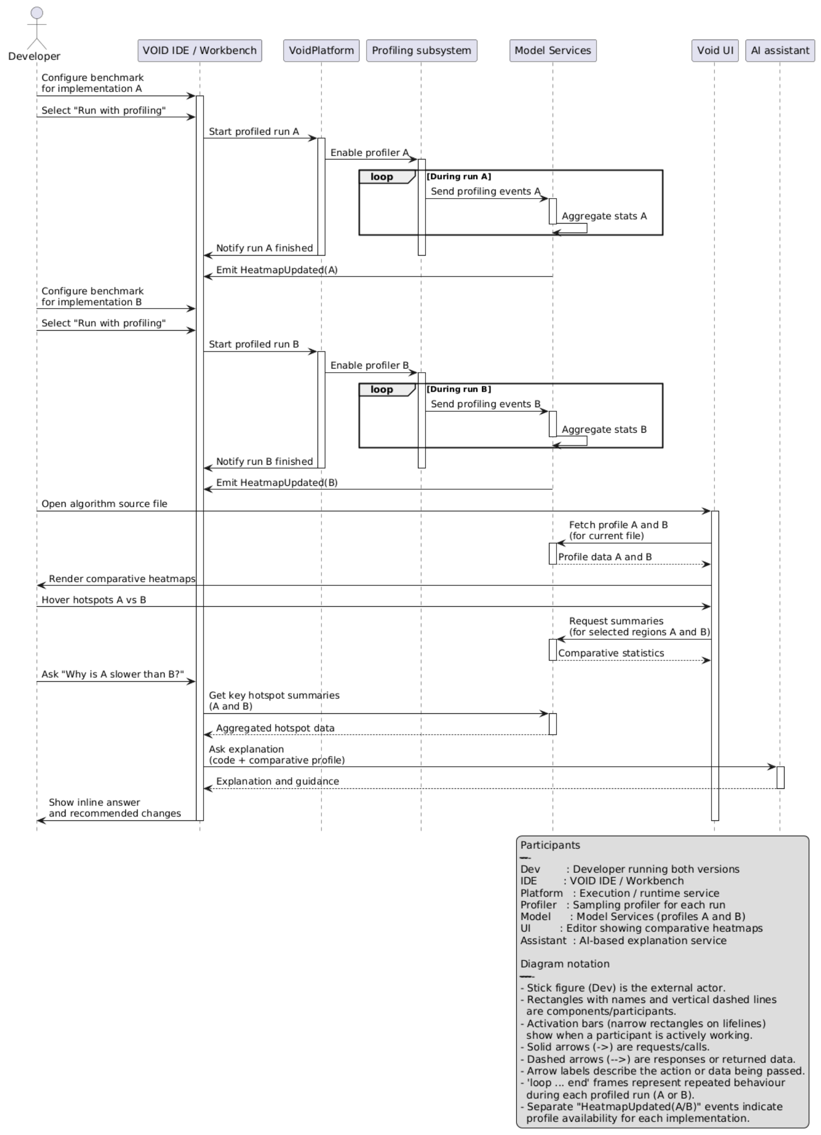
Figure 4: Sequence diagram for Use Case 1: investigating a slow feature.

10.2 Use Case 2: Comparing Alternative Implementations

A developer is evaluating two implementations of the same algorithm (for example, a naive and an optimised version) and wants to understand why one is slower. They configure a small benchmark or test harness in the Workbench and choose “Run with profiling” for each implementation in turn.

During each run, VoidPlatform enables the profiler and the profiling subsystem streams execution samples to Model Services. Model Services aggregates data per file, function, and line, then exposes a profile for each run. After the runs complete, the developer opens the relevant source files. Void UI overlays heatmaps for the chosen runs, making inner loops or frequently called helper functions appear as hot regions while rarely executed branches remain cool.

When the developer asks, “Why is implementation A slower than implementation B?”, the Workbench retrieves profiling summaries for the key hotspots in both runs from Model Services and includes them in a request to the AI assistant. The assistant uses these summaries to highlight where most of the time is spent and to suggest more efficient patterns. This keeps performance comparisons tightly integrated with the code in Void IDE.



Participants

```
Dev        : Developer running both versions
IDE        : VOID IDE / Workbench
Platform   : Execution / runtime service
Profiler   : Sampling profiler for each run
Model      : Model Services (profiles A and B)
UI         : Editor showing comparative heatmaps
Assistant  : AI-based explanation service
```

Diagram notation

- Stick figure (Dev) is the external actor.
- Rectangles with names and vertical dashed lines are components/participants.
- Activation bars (narrow rectangles on lifelines) show when a participant is actively working.
- Solid arrows (->) are requests/calls.
- Dashed arrows (-->) are responses or returned data.
- Arrow labels describe the action or data being passed.
- 'loop ... end' frames represent repeated behaviour during each profiled run (A or B).
- Separate "HeatmapUpdated(A/B)" events indicate profile availability for each implementation.

Figure 5: Sequence diagram for Use Case 2: comparing alternative implementations.

11 Potential Risks

11.1 Security and Privacy

Profiling captures source locations, stack traces, and runtime behaviour for user projects. If this data is kept longer than necessary, logged in shared locations, or included in prompts to the AI assistant without clear limits, there is a risk of leaking proprietary logic or sensitive context across projects or users. Access to profiling data should be limited to the owning workspace, and the IDE should make it clear when information is being sent outside Void IDE, for example to the AI service.

11.2 Scalability and Capacity

On large repositories or with many profiled runs at the same time, the number of sampling events and size of aggregated profiles can grow quickly. Without limits on sampling rate, run duration, and data retention, Model Services and VoidPlatform may become a bottleneck. This can delay heatmap updates and compete with existing build and test work. The client side can also be affected: repeatedly pulling and rendering large profiles for big files can stress the editor and reduce responsiveness.

11.3 Performance and Resource Usage

Even lightweight sampling adds overhead in CPU, memory, and sometimes network usage. If profiling is enabled by default or used heavily during normal workflows, test runs and local checks can feel noticeably slower, and profiled executions may differ from production timings. On less powerful developer machines, maintaining and updating heatmap overlays in the editor may cause user interface lag, especially when working with large files or several open profiled views.

12 Changes from A1/A2 Architectures

For A3 we stay with the same overall conceptual and concrete architecture that we used in A1 and A2. Our proposed profiling and heatmap feature is an extension of the existing structure rather than a replacement.

At the conceptual level, we still rely on the same main subsystems: Platform, VoidPlatform, Model Services, Void UI, Workbench, AI Providers, and File Systems. The new feature adds responsibilities inside these subsystems, but it does not introduce new top level layers or remove any existing ones.

At the concrete level, we add new services and helper modules inside the current components instead of creating a separate profiling framework. For example, Platform gains a profiling service next to the existing run and debug services, Model Services gains a profiling model next to existing file and project models, and Workbench and Void UI gain new commands and views that follow existing patterns. The result is that the concrete architecture is extended to support profiling, but the core structure from A1 and A2 is still valid and reused.

13 Testing

We need to test two things. First, that the profiling and heatmap feature itself works as intended. Second, that it does not negatively affect existing Void IDE behaviour such as runs, debugging, editor features, and AI tools.

13.1 Unit Testing

For the new profiling logic, we will add unit tests around the following pieces:

- The APIs in the Platform and VoidPlatform layers that start and stop profiling for a run.
- The code in Model Services that turns raw profiling events into the data structure used by the heatmap, for example per line and per function data.

- The user interface components that show and hide heatmap information in the editor and in the Workbench.

Small test programs with predictable behaviour, such as simple functions with loops or fixed delays, will be used to check that call counts and timings are recorded correctly and that the results are attached to the right files and lines. User interface and unit tests will simulate actions like "Run with profiling," "Toggle heatmap," and "Clear results," and verify that the right API calls are made and that editor decorations update or disappear as expected.

13.2 Integration and Regression Testing

On the integration side, we will run end to end scenarios in the IDE. For example, we will start a project from the Workbench with "Run with profiling," let it finish, and then check that profiling events travel through VoidPlatform to Model Services, heatmap data is published, and the correct editor buffers show the expected hotspots. We will also run the same project without profiling and confirm that behaviour matches the original system apart from not showing a heatmap.

We will test the feature alongside existing editor and AI functionality. This includes running diagnostics, code completion, and AI explanations while the heatmap is visible, to make sure decorations do not overlap in a confusing way and that profiling updates do not make the editor feel sluggish. For AI features that use profiling data, such as "Explain why this code is slow," tests will check that the AI provider receives a summary that matches the current heatmap and that failures in the AI path do not break the rest of the IDE.

Taken together, these tests should give confidence that the enhancement behaves correctly and that its interactions with other features remain controlled and predictable over time.

14 Conclusion and Future Work

In this report we proposed a real-time execution heatmap feature for Void IDE and compared three architectural approaches: a layered design, a publish and subscribe style, and a client–server style. Using SAAM, we evaluated how each option supports key scenarios such as interactive profiling runs, AI explanations of hotspots, new visualisations, and automated regression tests. Based on this analysis, we selected the layered approach as our primary design because it offers predictable performance, high testability, and straightforward control flow that fits well with Void IDE's existing architecture.

As future work, we would like to explore:

- Adding additional visualisations (for example, timelines or call trees) on top of the same profiling pipeline.
- Revisiting the publish/subscribe or client–server styles if Void IDE gains more extensions that consume profiling data or if profiling is shared across tools.

References

[1] M. Zhang, J. Atkinson, R. Sehgal, J. Trepanier, D. Mackinnon"PHIL A2 Report," [Online]. Available: `https://duncanmackinnon.github.io/PHIL/assets/a2/a2-report-322.pdf`. Accessed: Nov. 28, 2025.

[2] R. Kazman, L. Bass, G. Abowd, and M. Webb, "SAAM: A method for analyzing the properties of software architectures," in *Proc. 16th Int. Conf. Software Engineering (ICSE)*, 1994. [Online]. Available: `https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.65.8786&rep=rep1&type=pdf`.

A AI Collaboration Report

A.1 AI Model Selection

We used ChatGPT (GPT–5, November 2025) as our primary AI assistant. We chose it for consistency with A1 and A2 and for its strong support in summarisation, rewriting, and table construction. Other models such as Claude and Gemini were briefly tested but produced less consistent formatting and were not used for content in the final deliverable.

A.2 Role and Task Assignment

For A3, AI was used only for support tasks and not for architectural decision making. The main tasks included:

- Helping draft and format SAAM comparison tables.
- Rewriting non functional requirement descriptions to be clearer.
- Suggesting clearer wording for subsystem impact explanations.
- Formatting stakeholder tables and ensuring column alignment.
- Helping check sequence diagram logic for the main use cases.
- Reorganising paragraphs for readability.
- Providing example phrasing for trade offs in modifiability, performance, and testability.

Formatting assistance helped unify our writing styles and led to smoother transitions between sections. All architecture design, SAAM reasoning, selection of the final approach, and subsystem impact analysis were performed manually by the team.

A.3 Interaction Protocol and Prompting Strategy

We used a structured prompting approach:

- **Context first prompting:** Each request included our enhancement description, architectural alternatives, and draft subsystem impacts.
- **Fact bound queries:** AI was asked to restate or clarify information we already provided, rather than to invent new architectural decisions.
- **Iteration with human verification:** Any AI generated text was reviewed and edited by team members to ensure technical correctness and consistency with our design.

A.4 Quality Control and Validation

We applied the following checks to keep AI support under control:

- All SAAM assessments and trade offs were verified manually by the team.
- The AI was not allowed to choose the final architecture; this conclusion was made only after team discussion.
- Subsystem impact explanations were cross checked with our A1 conceptual and concrete architectures.
- All diagrams were reviewed by at least two team members before inclusion.

A.5 Quantitative Impact

We estimate that AI contributed roughly fifteen to eighteen percent of the textual formatting and rewriting effort. All architectural reasoning, trade off evaluation, and final design decisions remained human generated.

A.6 Reflection on Human and AI Collaboration

Compared to A2, our use of AI for A3 focused more on clarity and structure than on idea generation. GPT–5 helped us refactor long sections, especially the non functional requirement tables and architectural trade off descriptions. However, we found that relying on AI for architecture comparison can lead to over general or incomplete reasoning. This reinforced our practice of using AI as a language assistant rather than as an architecture assistant. The collaboration worked best when AI handled wording and layout and humans handled design and analysis.