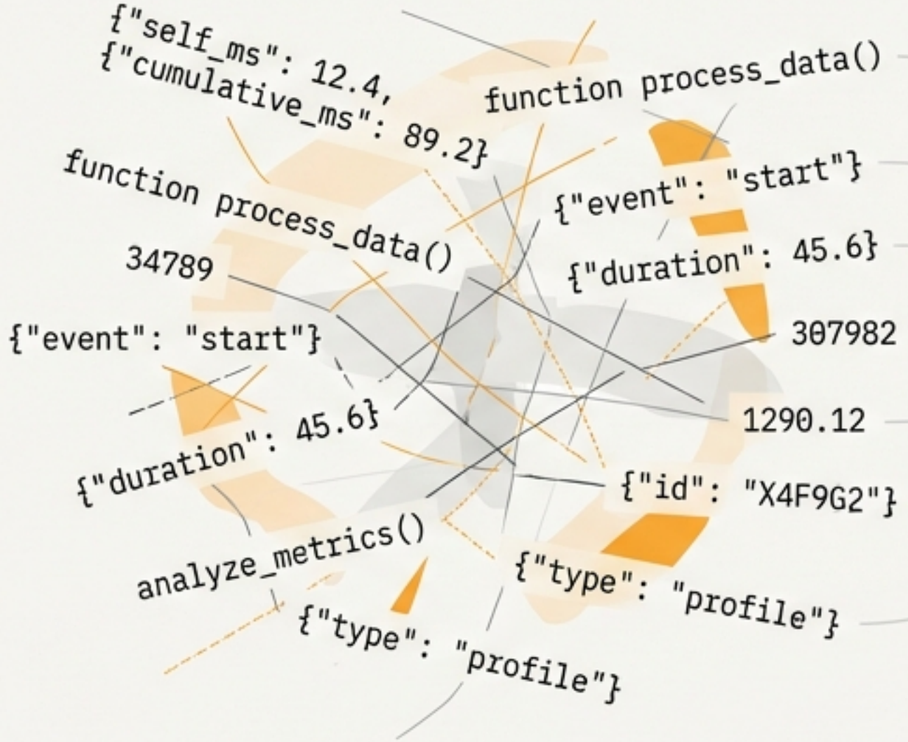
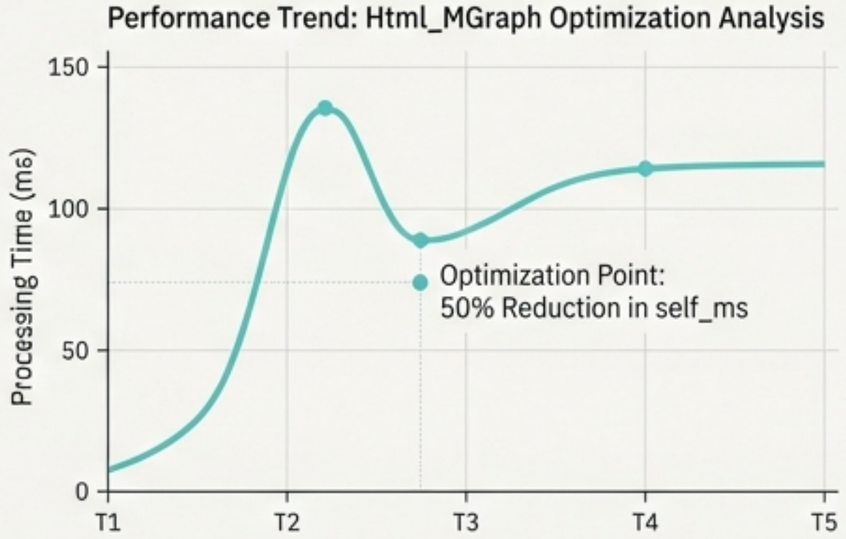
From Raw Data to Actionable Insight

A Case Study in Building a Purpose-Built Performance Analysis Workbench





Project: "Profile Analyzer"
Context: "Supporting Html_MGraph Performance Optimisation"
Stack: "A standalone, interactive React UI"

Our Starting Point: Rich Data Trapped in Raw JSON

The Profiling System

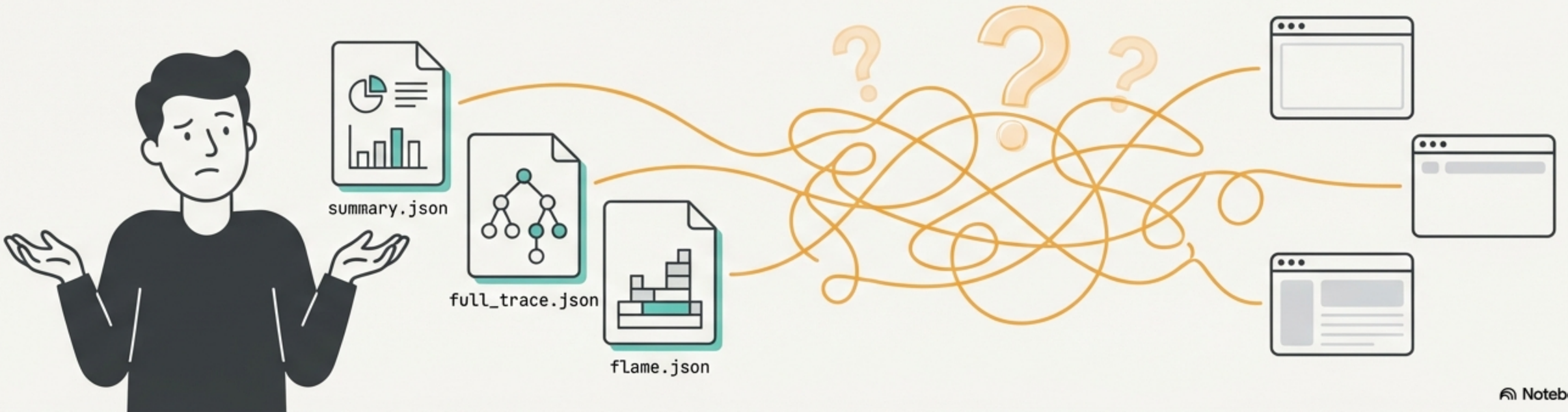
The source of the data: a sophisticated system using the `@timestamp` decorator from `osbot_utils`.

* **Summary files**: Aggregated hotspot analysis.
* **Full trace files**: Complete execution call trees.
* **Speedscope files**: Flame graph format.

The Analytical Bottleneck

The existing workflow was manual, fragmented, and inefficient, creating a barrier to insight.

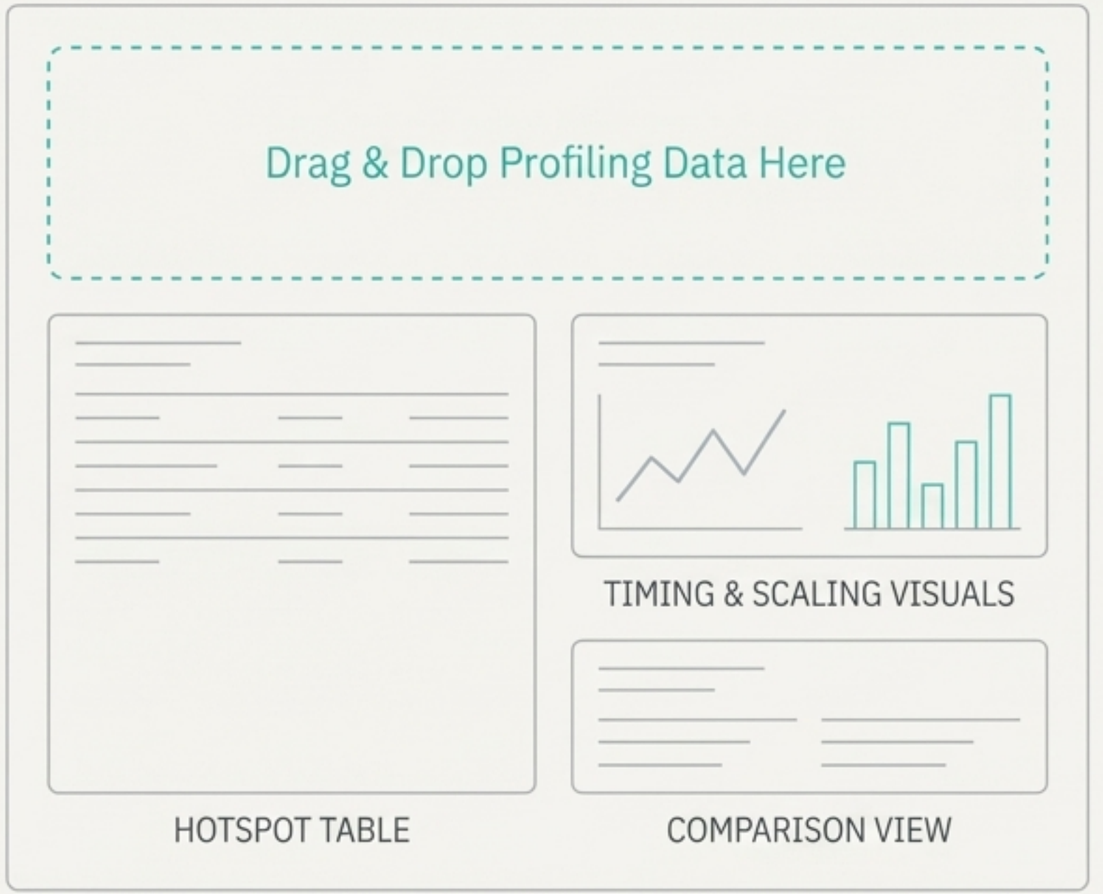
* Manually reading complex JSON files.
* Using generic tools that lacked specific context.
* A clear need for a tool that understood the data's structure and the user's workflow.



The Goal: An Interactive Workbench for Performance Diagnosis

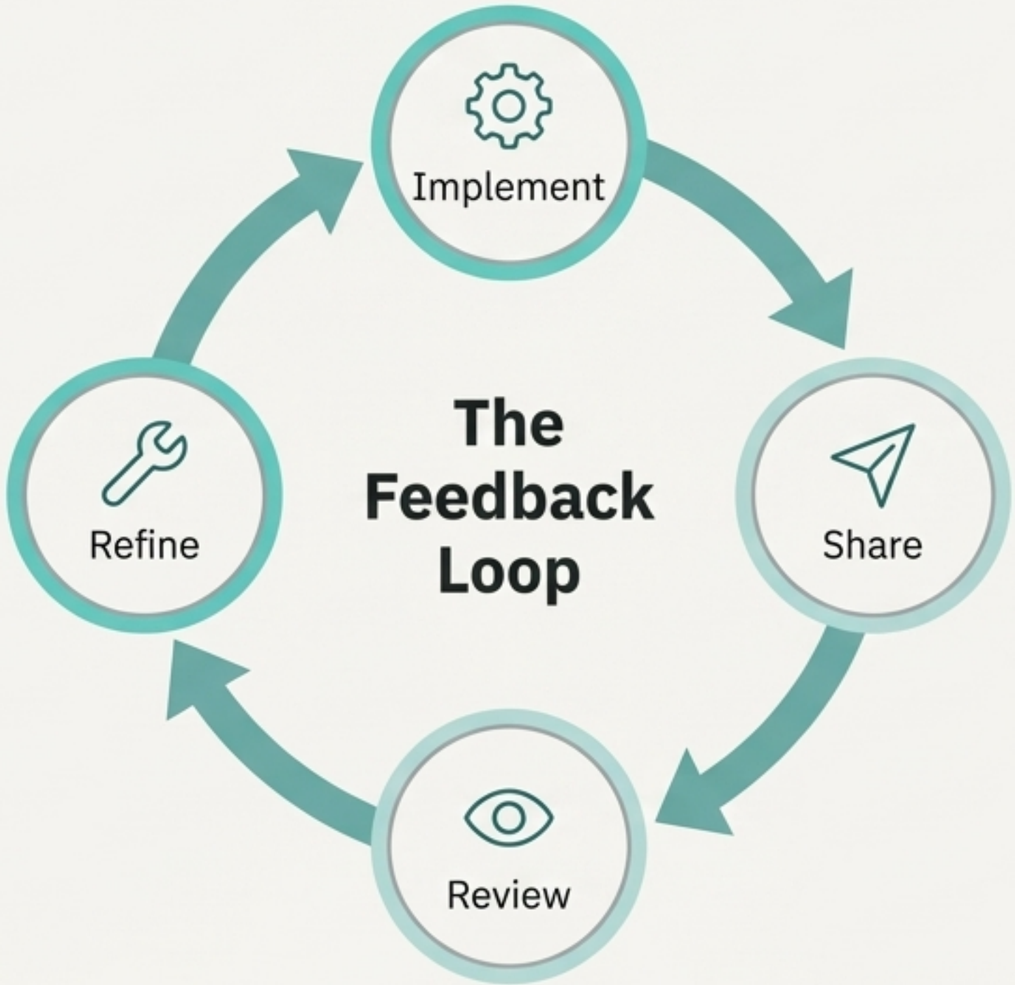
To create a web UI that could transform the analysis process from a chore into an exploration.

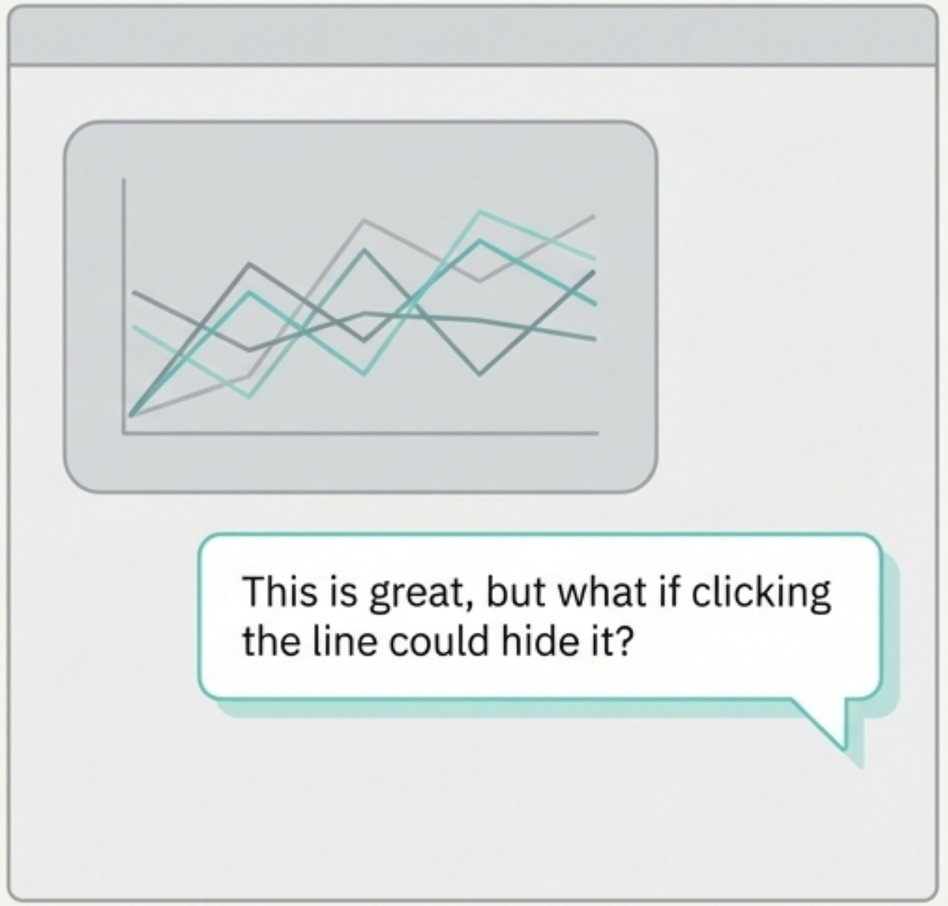
1. **Effortless Input**: Accept profiling data via drag-and-drop.
2. **Insightful Visuals**: Visualise hotspots and detailed timing data.
3. **Comparative Analysis**: Compare multiple profiling runs side-by-side.
4. **Scaling Behaviour**: Analyse performance across different input sizes.
5. **Zero Friction**: Work entirely in-browser with no backend required.



We Built the Tool Through Rapid, Feedback-Driven Iteration

What followed was a collaborative process spanning 15-20 refinement cycles.





A Key Insight Led to a Dual-Mode Architecture

We recognised that summary and full trace files serve fundamentally different analytical purposes. The UI should reflect this.

Summary Mode 💡

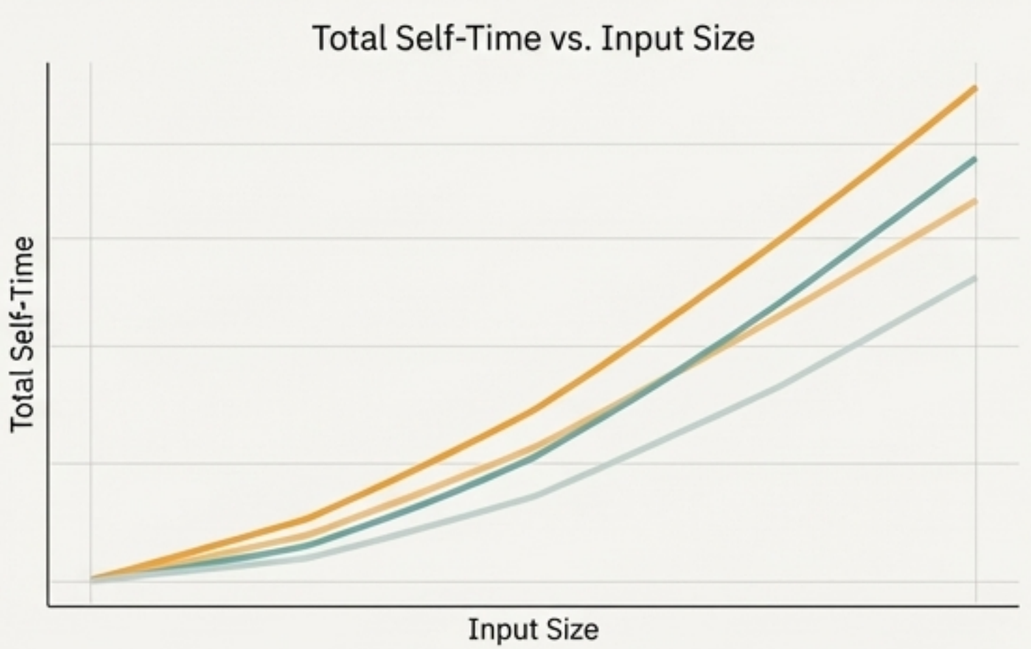
- **Purpose:** Quick hotspot identification and A/B testing.
- **For Files:** `summary_*.json`
- **Features:** Single Profile View, Comparison View, Scaling View.

Full Mode 🔬

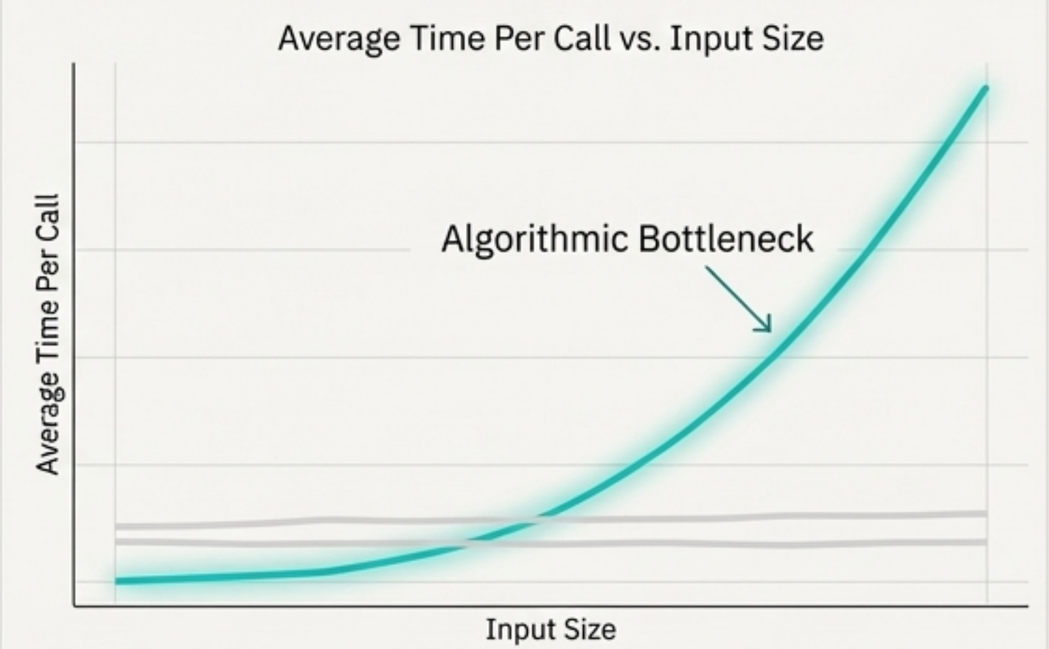
- **Purpose:** Deep algorithmic analysis and execution path exploration.
- **For Files:** `full_*.json`
- **Features:** Method Stats, Per-Call Scaling, Call Tree Explorer.

The Breakthrough: Realising Per-Call Averages Uncover True Algorithmic Complexity

Misleading



Insightful



The Flaw: Plotting **total self-time** vs. input size showed everything growing—expected but not useful.
The Insight: To find true O(n) issues, we plot **average time per call**. A method with O(1) complexity should show a flat line; a rising line is a red flag.

We Made Scaling Analysis More Robust by Letting the Data Describe Itself

The Problem 🧩

The X-axis for scaling charts was brittle, parsing input size from filenames like `with_size__30`. This failed on real-world test files and URL-encoded names.

The Solution 🎯

We pivoted to using the `entry_count` field from the file's own metadata, providing a universal measure of workload present in *all* profiling files.

Before

`...with_size__30.json`

After

```
{
  "metadata": {
    "entry_count": 30
  }
}
```

The Insight 📈

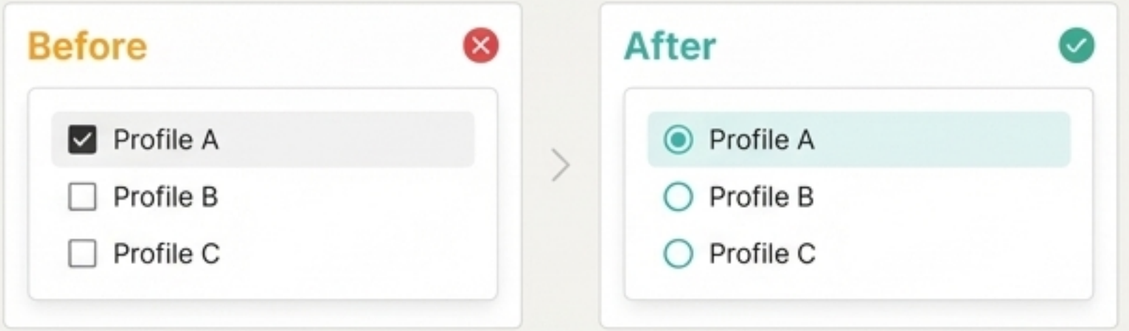
> Let the data describe itself rather than imposing external naming conventions.

Every Interaction Was Refined to Reduce Friction and Enhance Exploration

1. Smarter Selection

Problem: Checkbox-style selection was awkward for single-profile views.

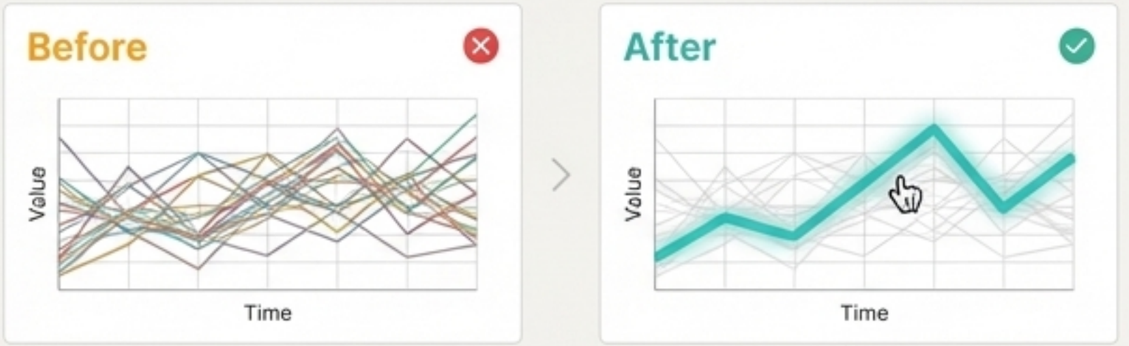
Solution: Implemented context-aware selection: radio-button behaviour in single views, checkbox behaviour in multi-select views.



2. Clearer Chart Lines

Problem: Charts with 12+ methods were cluttered and unreadable.

Solution: Added multiple interaction layers: table checkboxes, click-to-hide on lines, and hover highlighting where the focused line thickens and others fade.



3. Stable Tooltips

Problem: Default tooltip followed the cursor, obscuring data points.

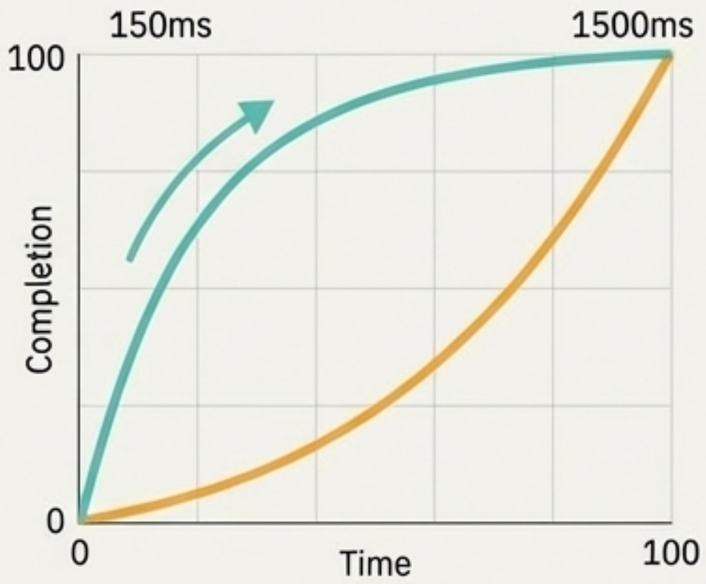
Solution: Implemented a fixed tooltip panel below the chart that is always visible but never blocks the view.

Final Polish Focused on Perceived Performance and Layout Stability

1. Fine-Tuning Animation

Problem: Default 1500ms Recharts animations felt sluggish when toggling line visibility.

Solution: Reduced duration to 150ms (10x faster) for a responsive feel that maintains visual continuity.



2. Ensuring Layout Stability

Problem: The tooltip panel's changing height caused the entire chart layout to 'jump' during interaction.

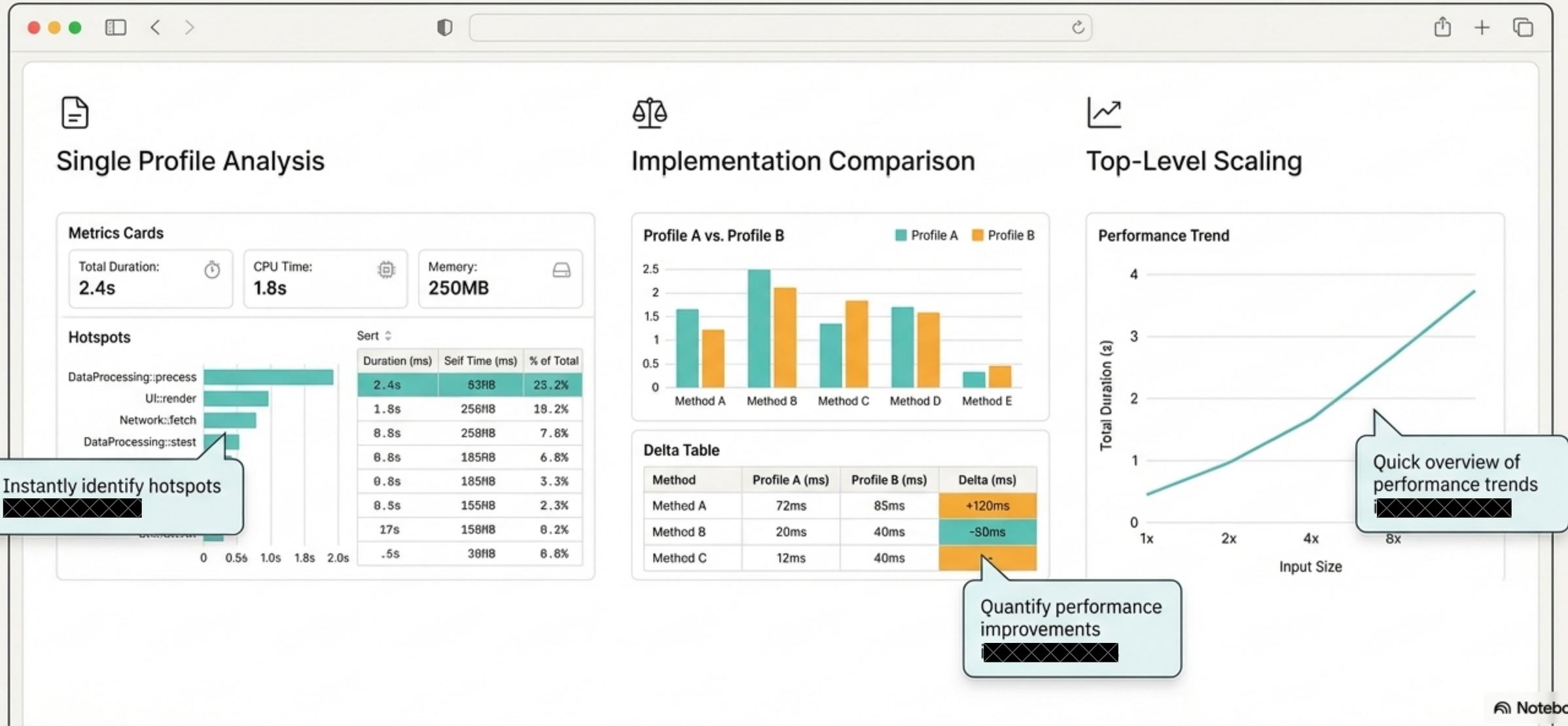
Solution: Gave the panel a fixed height with overflow hidden to eliminate visual disruption.

Insight: Predictable layouts build user confidence; unexpected movement creates cognitive friction.

3. Maximising Focus

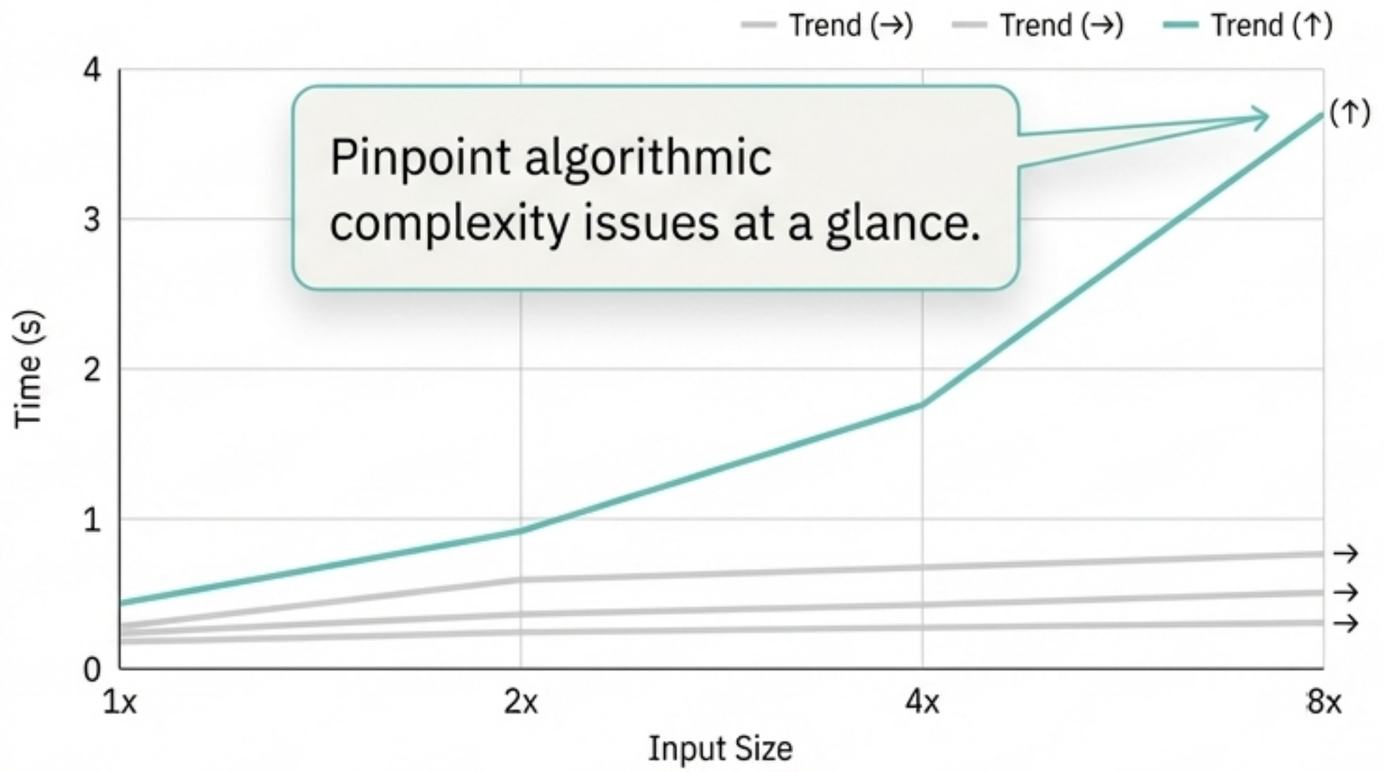
Feature: Added a full-screen overlay mode (⊕) for any chart, allowing for detailed analysis of dense visualisations without distraction.

The Result: A Summary Mode for High-Level Hotspots and A/B Testing



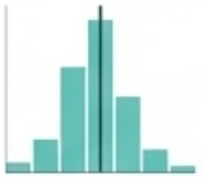
The Result: A Full Mode for Deep Algorithmic Investigation

Per-Call Scaling (The O(n) Detector)



Method Statistics

Method	Min (ms)	Max (ms)	Avg (ms)
DataProcessing::process	1.2	4.5	2.8
...			



Call Tree Explorer

- Main
  - DataProcessing::process (2.8s) [redacted]
  - UI::render (1.5s) [redacted]
  - Network::fetch (0.5s) [redacted]
    - fetch::data (0.2s) [redacted]

The Final Artefact is a Single, Portable HTML File

The project was packaged for maximum portability and zero installation friction.

1. Single `index.html` `index.html` File

Contains the entire application shell.

2. CDN Dependencies

Loads React, Recharts, and Babel directly from a CDN. No local dependencies.

3. In-Browser Transformation Transformation

JSX logic is transformed by Babel in the browser. No build step required.

4. Static Server

Works with any simple static file server (e.g., `python -m http.server`).

Key Benefit: An incredibly simple distribution model that allows anyone to run the tool instantly.

Key Lessons from the Development Journey

- **Start with the data:** Its structure should drive every design decision.

- **Iterate with real feedback:** Rapid feedback loops are essential for refining interactions.

- **Separate concerns by use case:** Distinct interfaces for distinct analytical tasks (Summary vs. Full).

- **Make everything interactive:** Enable hypothesis testing, don’t just display static visuals.

- **Prioritise layout stability:** Unexpected visual movement creates cognitive friction.

- **Use the data’s own metadata**: It is always more robust than external conventions like filename parsing.

A Purpose-Built Workbench is More Than a Visualiser; It's a Workflow Enabler

What began as a need to "visualise some JSON" evolved into a specialised performance analysis workbench. The iterative process was essential in transforming a generic viewer into a diagnostic tool aligned with a real-world workflow.

Looking Ahead: Future Opportunities

- Speedscope format integration for flame graphs.
- Persistent storage of user preferences.
- Automated anomaly detection and regression tracking.

The tool is immediately useful for the Html_MGraph project and generalisable to any codebase instrumented with the `@timestamp` system.