From Uncertainty to Proof: Mastering Performance with the Profile Analyzer

A practical guide to verifying code optimisation and detecting algorithmic regressions.

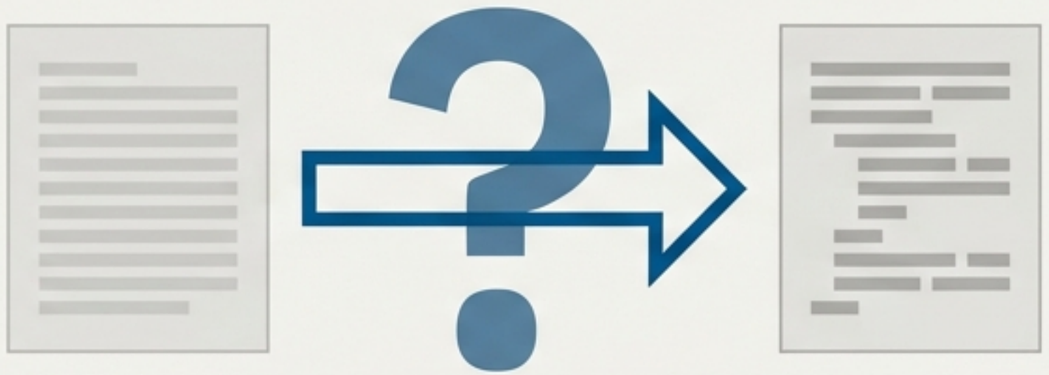
Version 1.0.0 | For internal use

Every developer has asked: “Did my change actually make things better?”

You’ve refactored a critical component, optimised a loop, or rewritten an algorithm. The code looks cleaner, and it *feels* faster. But is it? By how much? And did you inadvertently introduce a new, hidden bottleneck?

Answering these questions requires moving from gut feeling to empirical evidence.

How can you rigorously prove the impact of your work?



The Profile Analyzer enables a simple, four-step scientific workflow.

This tool transforms profiling data into a clear narrative, allowing you to systematically validate performance changes. The entire process is built around a robust before-and-after comparison.

1.

Capture Baseline

Profile the code *before* your changes using identical test data.

2.

Implement Change

Apply your optimisation or refactor.

3.

Capture After

Profile the code *after* your changes, using the same test data.

4.

Analyse & Verify

Load both profiles into the Analyzer to measure and prove the impact.

Your Analysis Starts with the Right Data Source.

The analyzer uses two types of JSON files generated by the `@timestamp` decorator system, each serving a distinct purpose in your investigation.

Summary Files (`summary_*.json`)

Use For:

- Quick hotspot identification
- Overall time tracking
- High-level A/B comparisons between runs

“The 10,000-foot view.”

Full Trace Files (`full_*.json`)

Use For:

- Per-call analysis
- Detecting O(n) complexity
- Deep algorithmic investigation
- Exploring the exact execution flow

“The cellular level.”

The First Answer: Did the total time improve?

Start in Summary Mode. The Comparison View provides an immediate A/B test result, showing the delta in milliseconds and percentage for every method. Colour-coding instantly reveals improvements and regressions.

Analysis Questions Answered:

- Did my change make things faster or slower overall?
- Which specific methods improved? Which regressed?
- What is the total time delta?

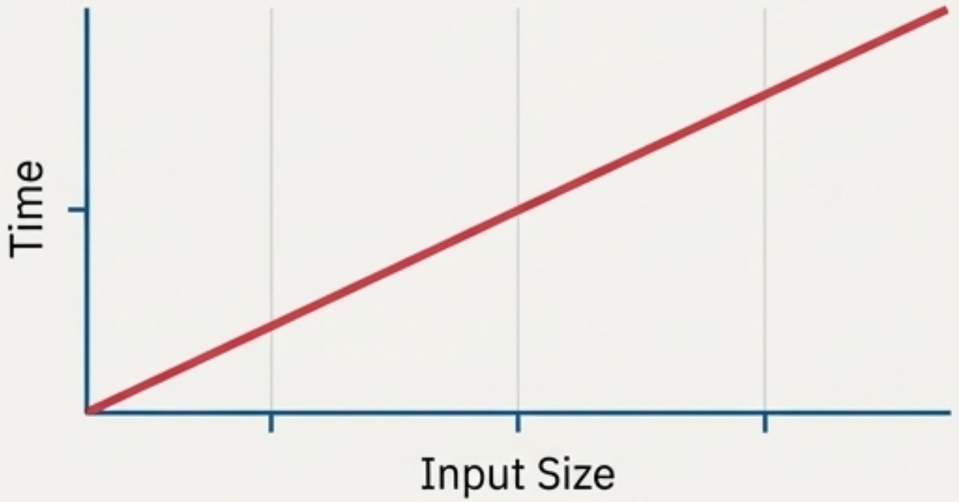
Delta Table

Method	Baseline (ms)	After (ms)	Delta (%)
process_items	1245.3	1055.9	-15.2%
validate_input	88.1	87.5	-0.7%
render_output	215.6	222.3	+3.1%
setup_context	5.2	5.1	-1.9%

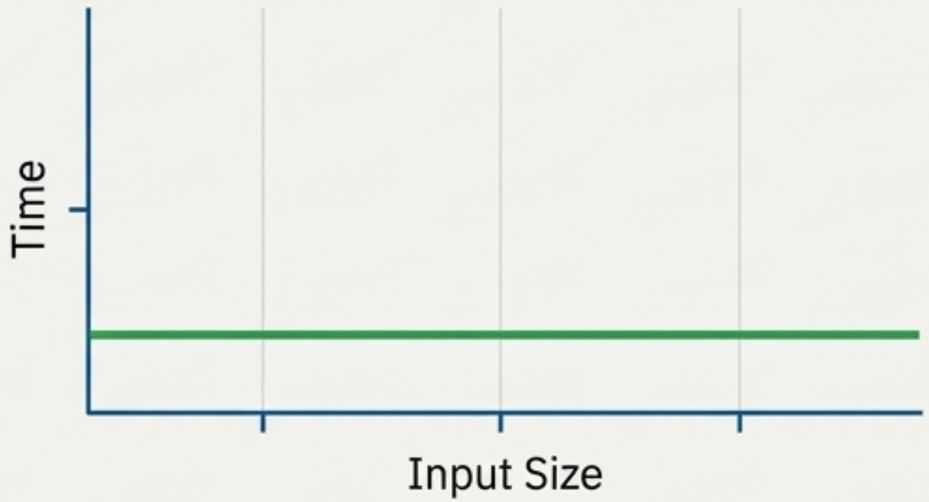
A faster runtime is good.
A better algorithm is transformative.

A simple speed-up might just be a constant-factor improvement. The most significant optimisations come from changing the *algorithmic complexity* of your code. The critical question is not just “is it it faster on this one input?” but “how does its performance change as the input size grows?”

O(n) ⚠️



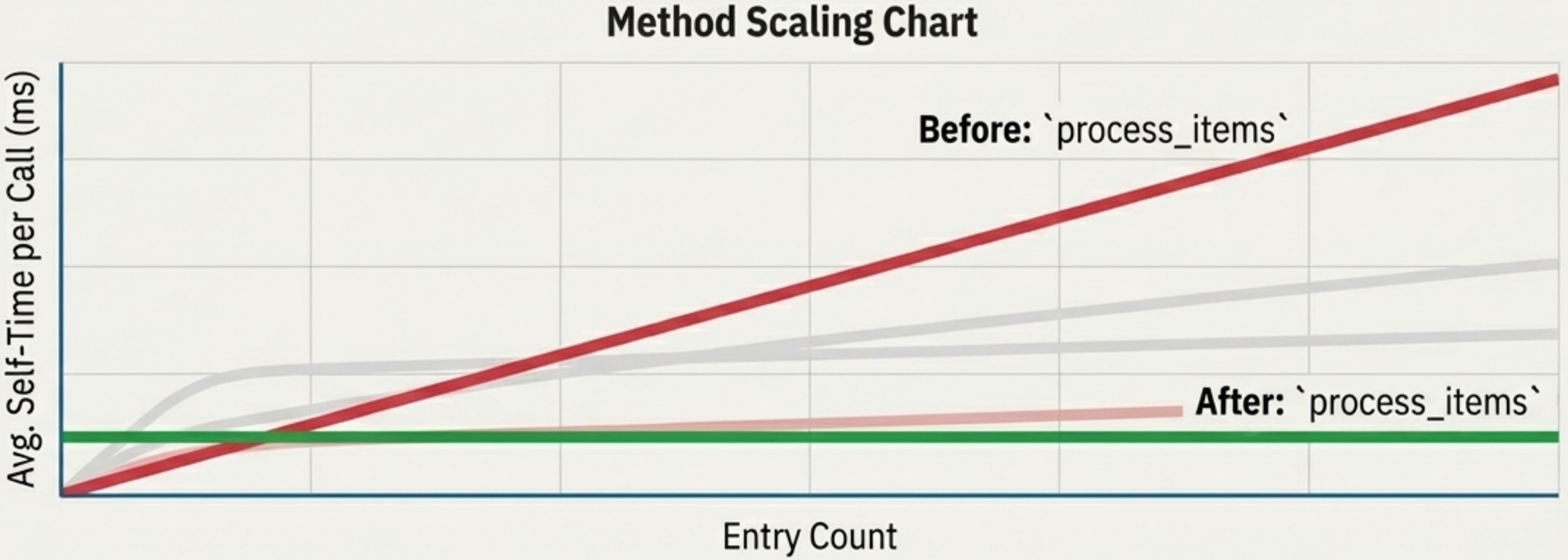
O(1) ✅

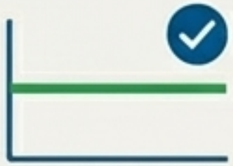


To find these patterns, we need to move from Summary Mode to Full Mode.

The Litmus Test: Verifying Complexity with Per-Call Scaling ⭐

This is the most diagnostic view for optimization work. It plots the average time per call against the number of operations (entry count), revealing the true scaling behaviour of each method.





Flat Line → O(1) per call: Excellent. The method performs a constant amount of work, regardless of input size. **This is the goal.**

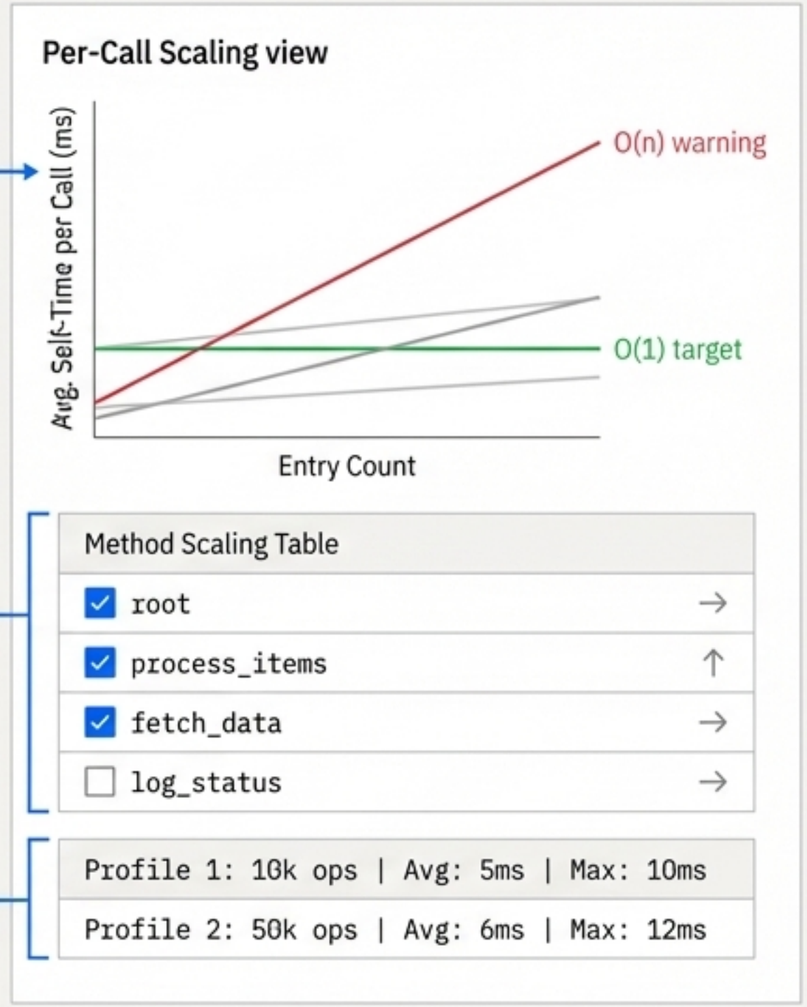
Rising Line → O(n) per call: Warning. The work done by the method grows as the input grows. **This is a primary optimization target.**

Decoding the Scaling Chart: A Practical Guide

1. ****Method Scaling Chart:** The main visual. Plots avg. self-time per call vs. entry count.

2. ****Method Scaling Table:** The legend. Use checkboxes to show/hide methods on the chart. Trend indicators (→, ↑) summarise behaviour.

3. ***Overall Scaling Table:** Profile-by-profile breakdown of key metrics.



Tips for Effective Analysis

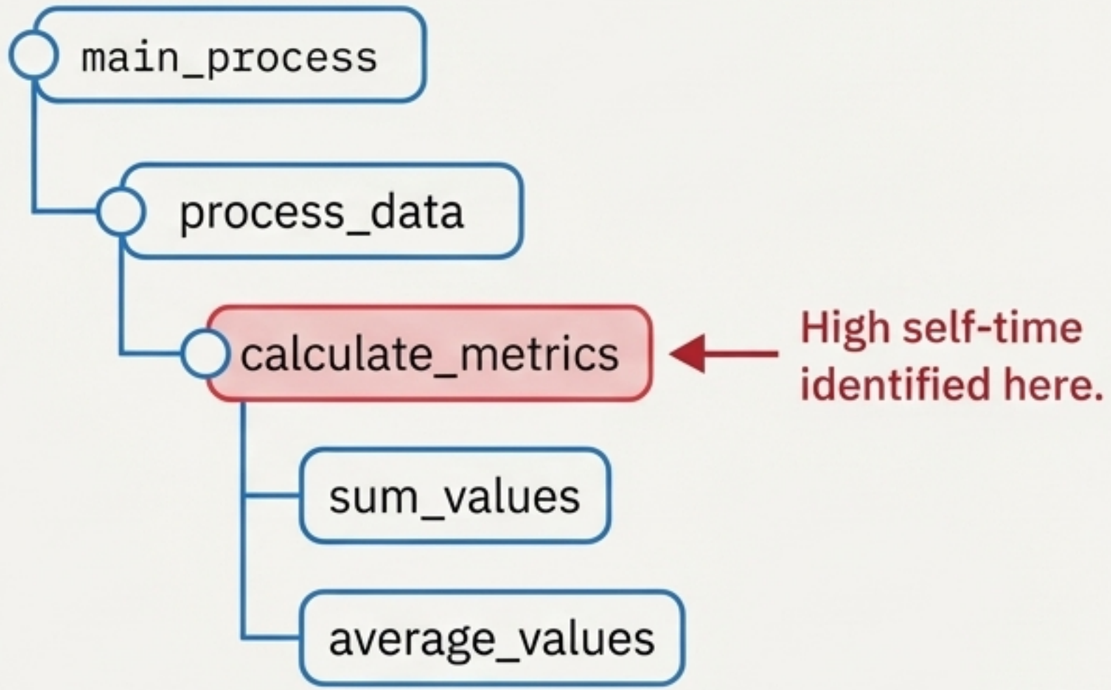
- **Start Here:** Always begin with this view when investigating algorithmic issues.
- **Declutter:** The root method often dominates the chart. Click its line or use the checkbox to hide it and reveal the behaviour of other methods.
- **Go Full Screen:** Use the 'Maximize' button for dense charts with many methods.
- **Cross-Reference:** If you find a rising line, use the Call Tree view to understand its context.

When a Method Scales Poorly, Find Out Why with the Call Tree.

The Call Tree View provides the ground truth of your code's execution. It shows the exact call sequence and hierarchy, allowing you to trace the execution path and understand the context of a problematic method.

Analysis Questions Answered:

- What is the actual call sequence?
- Where in the call stack is time truly being concentrated?
- What called this slow method, and what does it call?



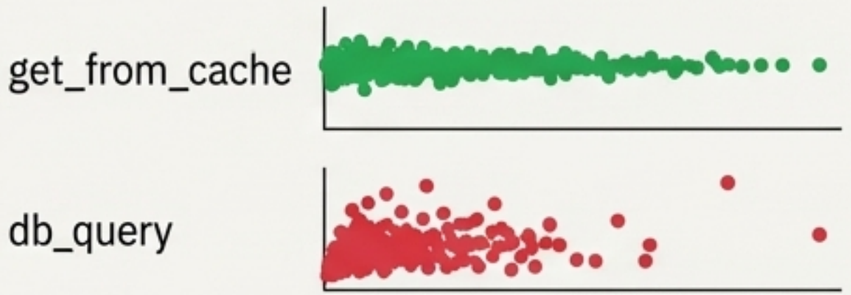
Investigate Inconsistent Performance with Method Stats.

Sometimes, a method is fast on average but has slow, outlier calls. The Method Stats view helps you spot this inconsistency by analysing the distribution of per-call timings.

Key Metric: Spread (max / min ratio)
A low spread indicates consistent, predictable timing. A high spread (e.g., 10x or more) suggests performance is being affected by external factors like caching, garbage collection, or I/O.

Method	Avg Self (ms)	Min (ms)	Max (ms)	Spread
get_from_cache	0.8	0.7	55.4	✅ 1.2x
db_query	15.3	2.1	95.9	⚠️ 45.7x

Call Distribution



A Field Guide to Interpreting Performance Patterns

✅ Healthy Patterns

- **Observation**: Flat lines in Per-Call Scaling → **Interpretation**: O(1) per-call complexity.
- **Observation**: Green cells in Comparison delta → **Interpretation**: Performance improved.
- **Observation**: Low spread in Method Stats → **Interpretation**: Consistent, predictable timing.

⚠️ Warning Patterns

- **Observation**: Rising line in Per-Call Scaling → **Interpretation**: O(n) per-call behaviour. Investigate.
- **Observation**: Red cells in Comparison delta → **Interpretation**: Performance regressed.
- **Observation**: High self-time in orchestrator methods → **Interpretation**: Unexpected; these should typically be near zero.

🔴 Critical Patterns

- **Observation**: Steeply rising line that was previously flat → **Interpretation**: A new O(n) bug has been introduced.
- **Observation**: Total time growing faster than linear → **Interpretation**: Possible $O(n^2)$ complexity lurking.

Which View Should I Use? A Quick Reference Guide.

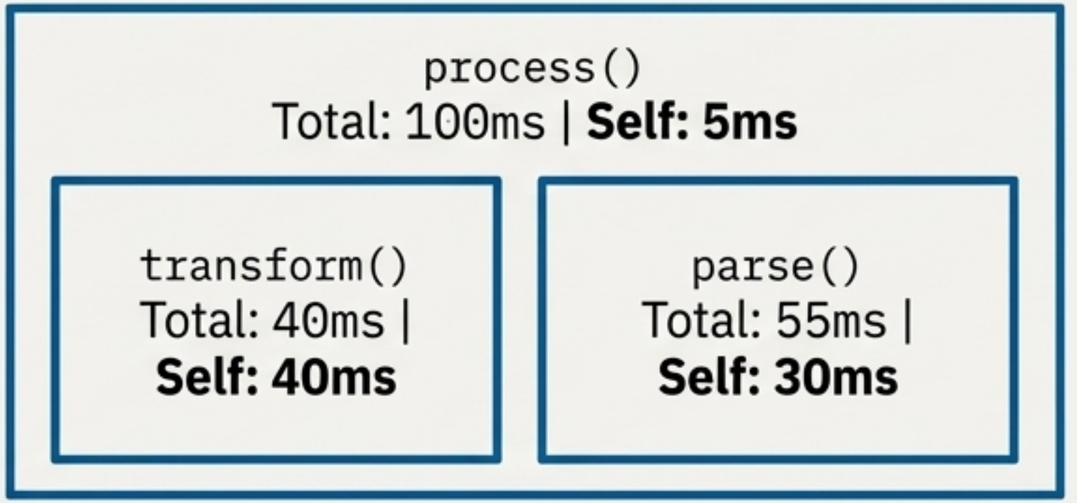
Use this guide to jump directly to the view that best answers your current question.

Question	Mode	View
Where is most of the time being spent?	Summary	Single Profile
Did my change actually help or hurt?	Summary	Comparison
Does my code have O(n) behaviour?	**Full**	**Per-Call Scaling**
How does performance scale overall?	Summary	Scaling
What is the exact sequence of calls?	Full	Call Tree
Are my method timings consistent?	Full	Method Stats

Appendix: Understanding the Core Metrics

What ‘Self-Time’ Means.

Self-time is the time spent executing code *within* a method, excluding any time spent in methods it calls. This is the critical metric for identifying where the actual work is happening and what to optimise.



Why We Use ‘Entry Count’.

The X-axis on scaling charts is `entry_count`—the total number of instrumented method calls. This serves as a reliable, built-in proxy for workload size, as it’s directly proportional to the actual work performed by the system.

Stop Guessing. Start Proving.

The Profile Analyzer provides the framework and evidence to confidently answer the question, “Did my change actually make things better?”

Key Takeaways

- Always capture **before and after** profiles with identical test data.
- Use **Per-Call Scaling** to verify algorithmic fixes and prove O(1) behaviour.
- Use **Comparison View** to quantify overall improvements in milliseconds.
- **Document your findings** with screenshots of the key charts as evidence of your impact.