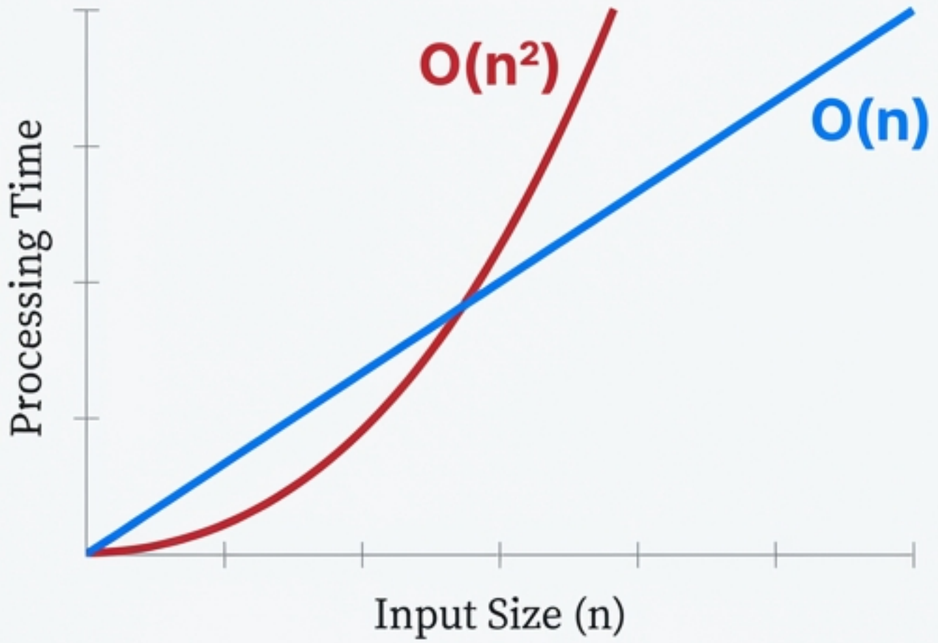
Hunting an O(n²) Bottleneck

How a 4-line fix unlocked impossible workloads in the Html_MGraph pipeline.



3-7× Faster
on typical workloads.

22% Faster
test suite execution.

TIMEOUT to <2s
on previously impossible jobs.

The System Was Failing Under Load

Simple HTML processing was taking over a second, with performance degrading exponenially as document size increased. Large, critical workloads were failing completely.

Initial Performance Measurements

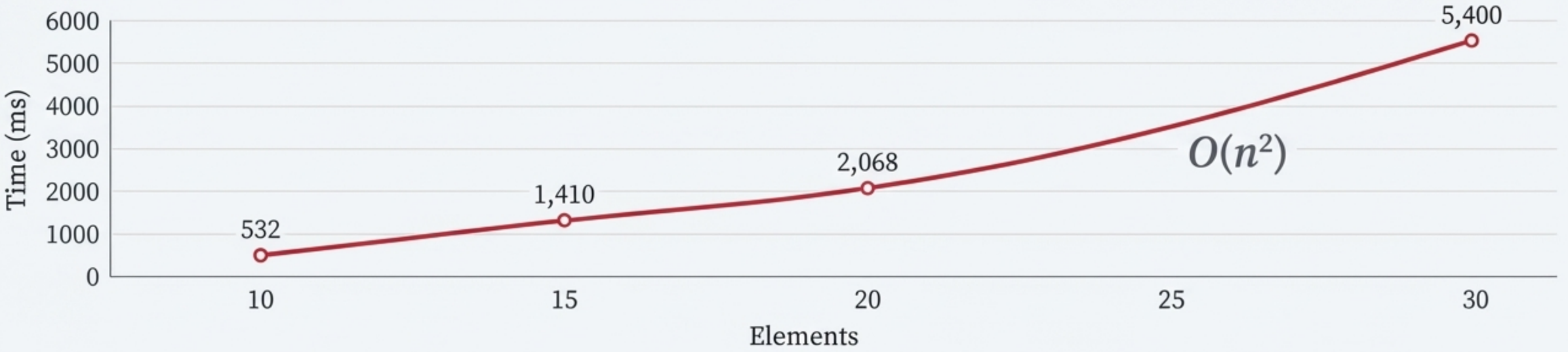
HTML Size	Processing Time	Expected Time
Minimal (39B)	111ms	<50ms
Simple (132B)	197ms	<50ms
With attrs (400B)	**1,500ms**	**<100ms**
Complex (1347B)	**3,813ms**	**<200ms**

Symptoms of Failure

- ⚠ User-facing API response times exceeding SLAs.
- ⚠ Processing time growing super-linearly.
- ⚠ Large websites timing out completely.

The Diagnosis: Pathological O(n²) Scaling

The cost to process each new element was increasing as the graph grew.
This is the classic signature of a quadratic complexity bottleneck.



Elements	Time	ms/element	Growth Factor
10	532ms	53.3ms	*baseline*
15	1,410ms	94.0ms	1.76×
20	2,068ms	103.4ms	1.94×
30	~5,400ms	~180ms	~3.4×

The core problem: adding more work made ***all*** **existing work** slower.

Our Investigation Toolkit: The `@timestamp` Decorator

We used the OSBot-Utils *@timestamp* decorator system to instrument the codebase non-intrusively. This allowed us to precisely measure execution time at any point in the call stack.

Investigation Strategy

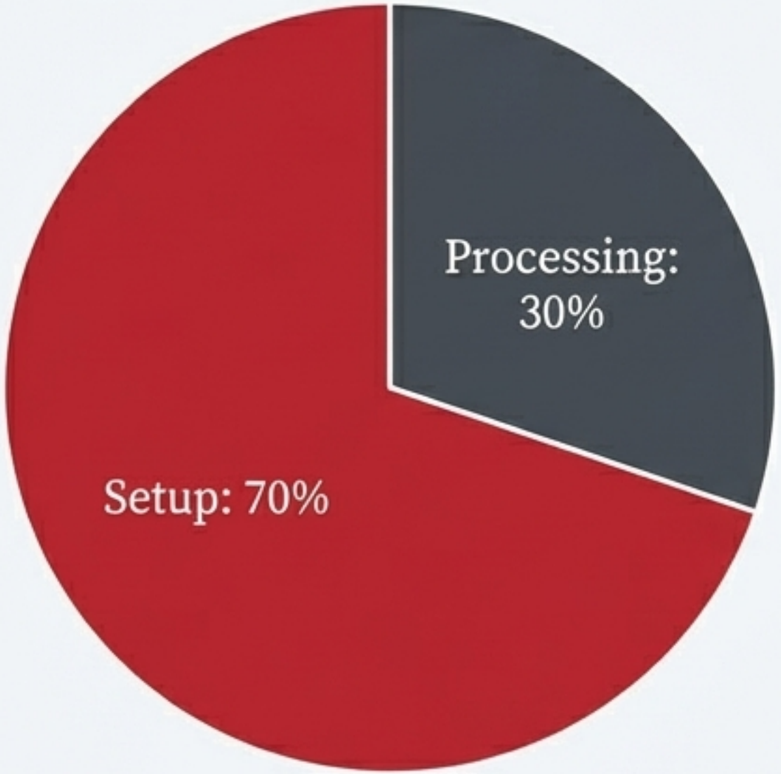
1. **"Start Broad":** Instrument top-level phases.
2. **"Follow the Time":** Drill into the function with the highest *self-time*.
3. **"Repeat":** Continue narrowing the focus.
4. **"Validate":** Confirm the fix using the same instrumentation.

Key Features

Feature	Purpose
`@timestamp(name="...")`	Basic method timing
`print_report()`	Summary by total time (call hierarchy)
`print_hotspots()`	Summary by **self-time** (actual work done)
`print_timeline()`	Chronological execution trace

First Clue: A Misleading Lead

Initial profiling on minimal HTML documents suggested that setup code was the main bottleneck, consuming over 70% of the execution time.



Profile for Minimal HTML

The Crucial Insight

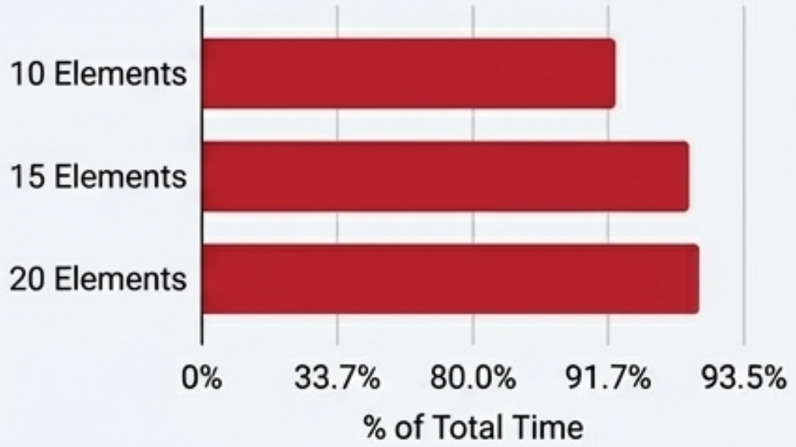
- This result was a **red herring.**
- Real-world documents have significant content.
- The performance characteristics of a system can change dramatically when moving from minimal test cases to scaled, realistic workloads. We needed to test with more data.

The Real Bottleneck Emerges with Scaled Data

When tested with HTML containing 10-20 elements, the performance profile inverted. The body.process method became the overwhelming bottleneck.

The Picture Changes Completely

Elements	body.process Time	% of Total Time	Cost Per Element
10	532.69ms	80.7%	53.3ms
15	1,410.18ms	91.7%	94.0ms
20	2,068.60ms	93.5%	103.4ms



Key Finding: The cost-per-element was *increasing*, confirming the problem was inside body.process.

Drilling Down: `register_attrs` is the Offender

By instrumenting the methods within `body.process`, we discovered that a single function was responsible for the majority of the slowdown.

52.1%

of total execution time was spent in `register_attrs` for a document with only 17 elements.

Confirmation Test

To confirm our suspicion, we temporarily disabled the `register_attrs` method.

Result:

Processing time dropped dramatically. A previously timing-out page now loaded in just **1.2 seconds**.

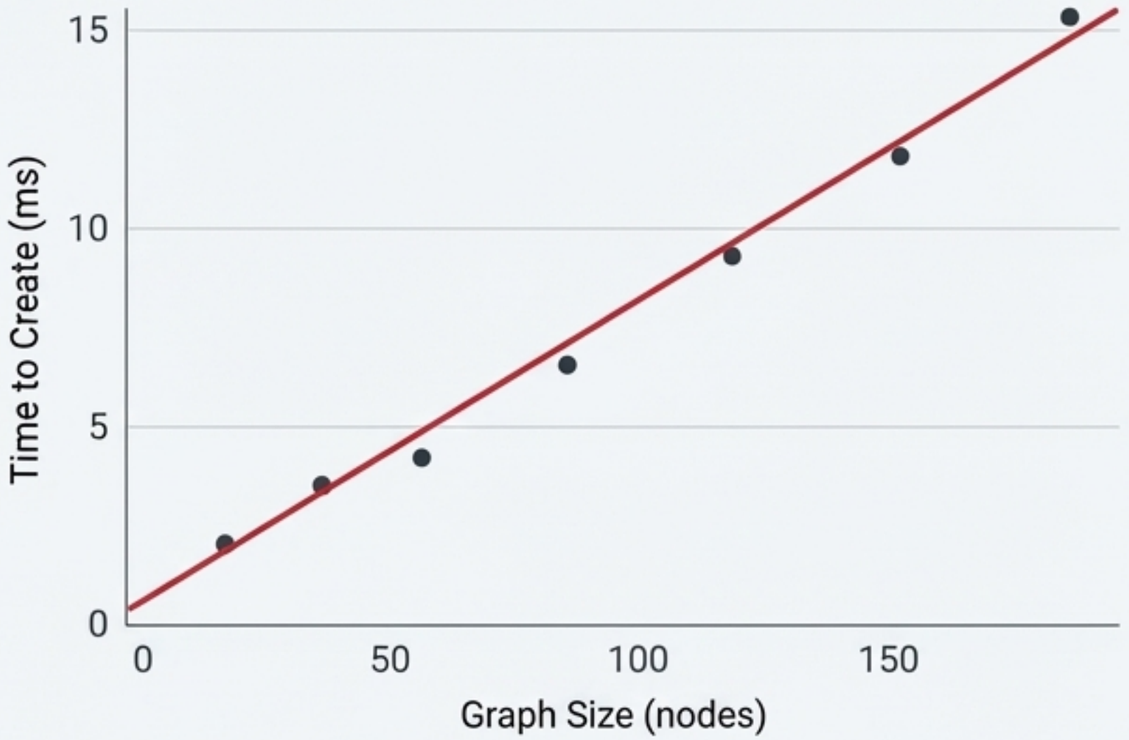
This confirmed `register_attrs` as the location of the bottleneck.

The Smoking Gun: Lookup Time Grew with the Graph

We instrumented the `get_or_create` operations within attribute processing to capture the execution time for each unique attribute. The results were conclusive.

Lookup Time vs. Graph Size

Attribute Value	Time to Create	Graph Size When Created
`en`	1.43ms	~10 nodes
`utf-8`	2.64ms	~20 nodes
`viewport`	3.30ms	~30 nodes
`stylesheet`	5.27ms	~50 nodes
`nav`	7.57ms	~70 nodes
`/about`	9.65ms	~90 nodes
`app.js`	**14.39ms**	**~130 nodes**



Each call was becoming progressively slower.
The operation's cost was a function of the graph's *current size*.

The Root Cause: A Linear Scan Hiding in Plain Sight

The `get_or_create_value_node` and `get_or_create_name_node` functions were performing a linear scan. To check if a value already existed, they iterated through every single node in the graph.

```
# Simplified for clarity
def _get_or_create_value_node(self, value):
    # This loop is the O(n) operation
    for node in self.graph.nodes:
        if node.value == value:
            return node
    # ... code to create new node ...
```

O(n) operation

This O(n) lookup is called for every attribute (N times).

↓

The graph grows with each call.

↓

Result: N calls × O(n) operation = O(n²) complexity

The Fix: Replacing Scans with Hash Indexes

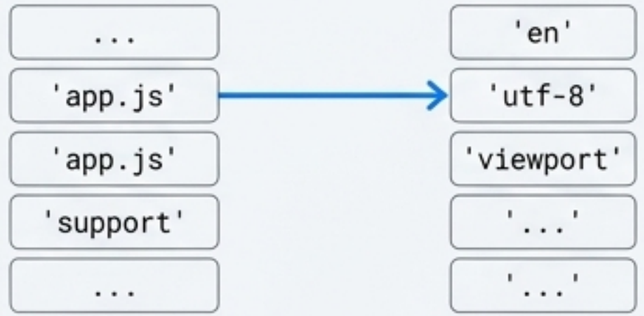
The solution was to replace the O(n) linear scans with O(1) dictionary lookups. We introduced two dictionaries to serve as hash indexes for value and name nodes.

Before (O(n))

Loop through all N nodes to find a match.

After (O(1))

Look up the node directly in a dictionary.



4 lines of actual logic were changed to fix the entire performance bottleneck.

The investigation took hours; the fix took minutes.

Validation: Immediate and Staggering Speedups

The impact of the fix was instantly visible in the instrumented `get_or_create` functions. Lookup times became negligible.

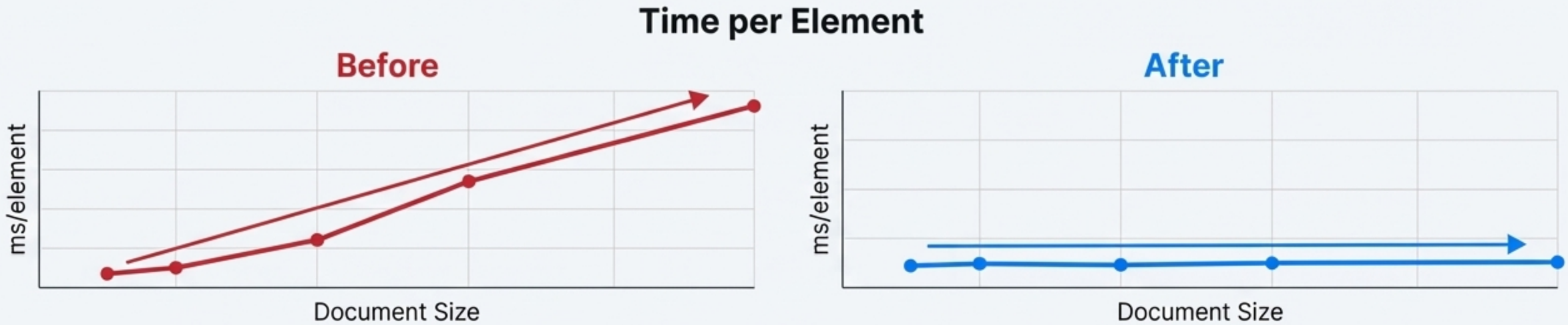
Lookup Performance, Before vs After			
Operation (14 calls)	Before	After	Speedup
_get_or_create_value_node	110 ms	0.64 ms	172×
_get_or_create_name_node	85 ms	6.65 ms	13×

New Profile Summary

After the fix, the hotspot report showed a healthy profile. No single operation pathologically dominated execution time.

The Result: O(n) Scaling Confirmed, Impossible Workloads Unlocked

The system now scales linearly. The per-element processing cost remains constant, regardless of the document size.



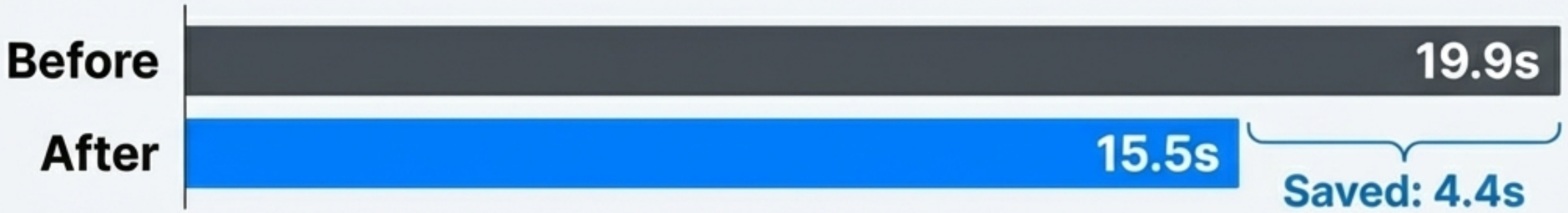
Real-World Workloads

Size (Elements)	Before Fix	After Fix	Speedup
15	1,410ms	417ms	**3.4×**
30	~5,400ms	802ms	**6.7×**
100	**TIMEOUT**	2,166ms	**∞**
500	**IMPOSSIBLE**	10,750ms	**∞**

Real Site Example: **docs.diniscruz.ai** (33.5 KB) went from **TIMEOUT** to processing in **1.8s**.

A Healthier System: 22% Faster Test Suite

The performance improvement was so significant it measurably accelerated our entire test suite, improving developer velocity and CI/CD efficiency.



Metric	Before	After	Improvement
Total Test Time	19.9s	15.5s	**-4.4s (22%)**
Regressions	-	0	✓ All tests pass

> “Faster feedback loops for every developer on every commit.”

Lessons from the Hunt

Don’t Trust Minimal Tests

Initial profiling was a red herring. Always test with realistic, scaled workloads to find true bottlenecks.

Follow the Self-Time

Total time shows you the call stack. Self-time shows you where the actual work is being done. Use hotspot analysis to pinpoint the real culprit.

O(n²) Hides in Simple Code

A simple loop, called repeatedly within another loop, is a classic pattern for quadratic complexity. Be vigilant during code reviews.

The Fix is Often Simple

The investigation is the hard part. The root cause is often a small, localized logic error with a simple, elegant solution.

A Lasting Improvement: Instrumentation as a Capability

This was more than a one-time fix. The instrumentation that enabled this investigation remains in the codebase, providing a powerful tool for future performance monitoring.

Why Leave Instrumentation In Place?

- **Minimal Overhead:** ~3µs overhead when no data collector is attached.
- **On-Demand Profiling:** Enables performance analysis in any environment without code changes or redeployment.
- **Self-Documenting:** The decorators act as a permanent marker for performance-critical code paths.

3-7x faster workloads

22% faster test suite

Impossible workloads now routine

All with a 4-line code change.