THE QUADRATIC INCIDENT

A DOCUMENTARY RECONSTRUCTION OF A PERFORMANCE INVESTIGATION

The investigation took hours; the fix took minutes.
Four lines of code changed everything.

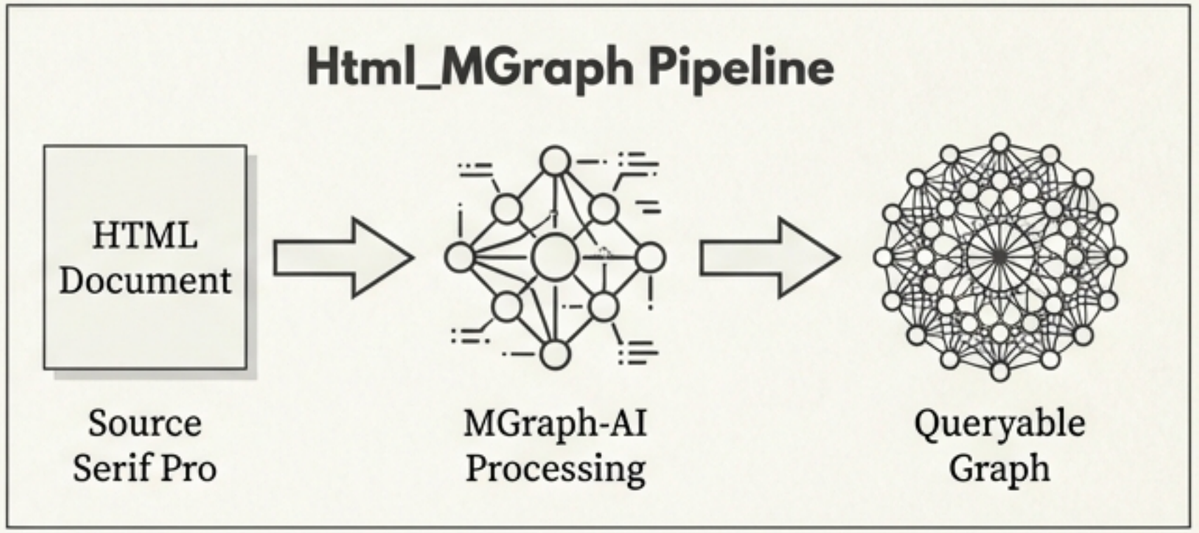
— Performance Investigation Debrief, December 2025

NotebookLM

A Perfect System, A Sudden Failure

The System: MGraph-AI, a powerful tool designed to transform any HTML document into a queryable graph structure. Every element, attribute, and relationship preserved and indexed.

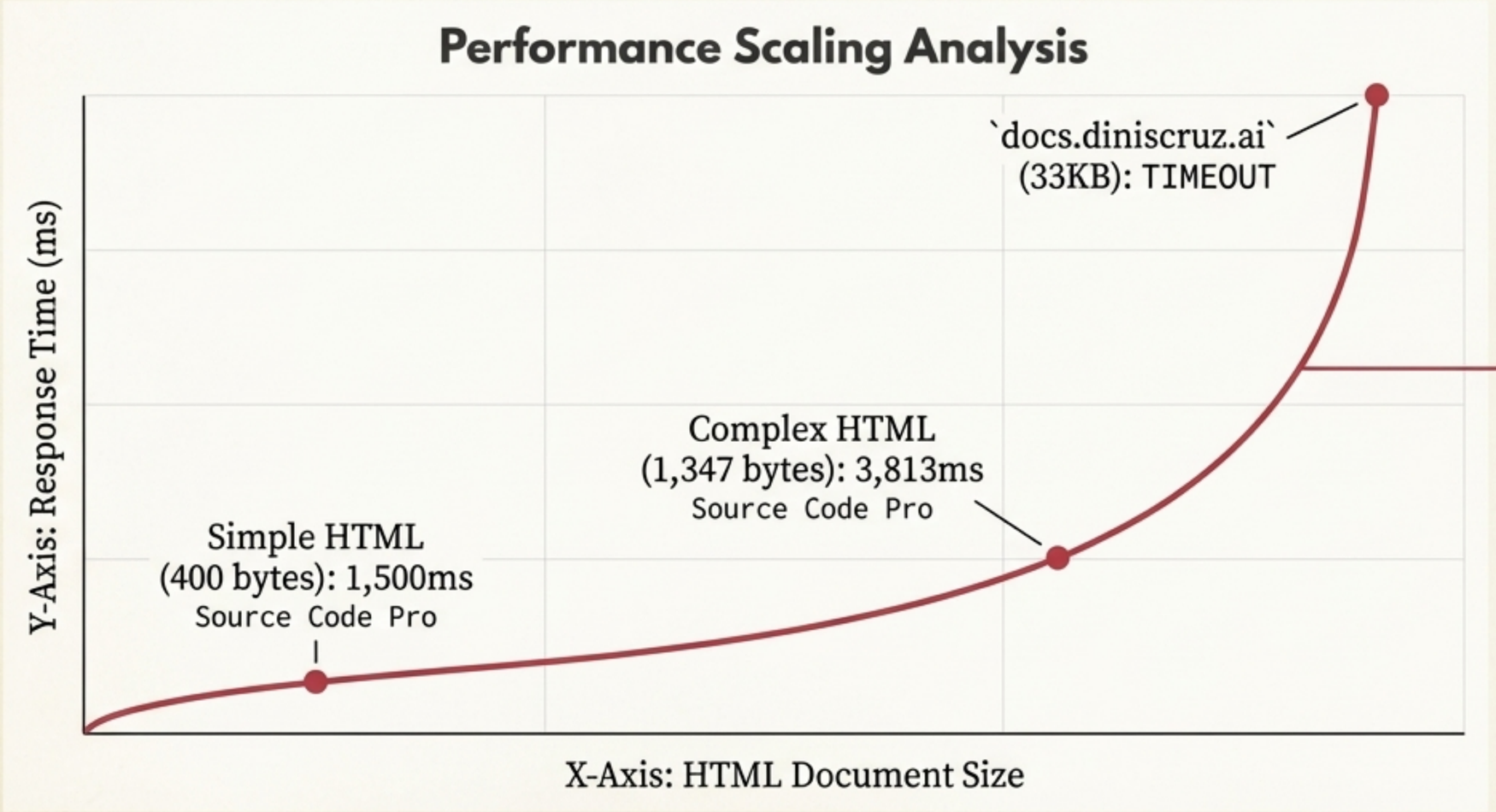
The Incident: A 33-kilobyte HTML file from a personal documentation site (docs.diniscruz.ai), which had processed successfully for weeks, suddenly caused a complete system timeout.



NotebookLM

Exhibit A: The Escalation Pattern

The initial performance metrics immediately ruled out simple **infrastructure issues.** The problem was intrinsic to the code, revealing a non-linear relationship between document size and processing time.



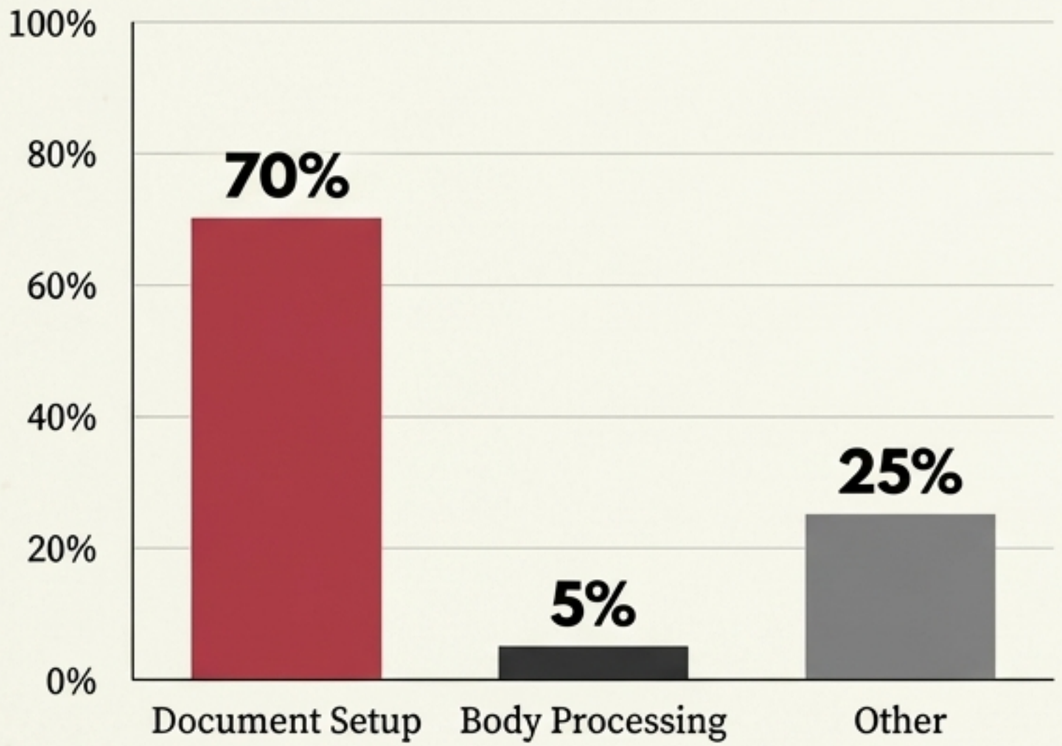
Key Insight:
Response time wasn't growing linearly. It was accelerating. This was the first sign of a deep, algorithmic flaw.

NotebookLM

The First Lead: A Dangerous Misdirection

The initial step in any investigation is to measure. The first profiling run seemed to point to a clear culprit. However, the test data was flawed.

Profiling a Minimal HTML Document



The Theory: The bottleneck is in the document setup phase.

The Flaw: *We were testing with a skeleton document. Of course setup dominated—there was nothing else to do. Real workloads would tell a different story.*

NotebookLM

When Your Tools Aren't Enough, You Build Better Ones

Standard profiling provided total execution times but couldn't pinpoint the *actual* work being done. The investigation required a more sophisticated tool, one that could distinguish between time spent in a function versus time spent in functions it calls.

Total Time

The entire duration of a method, including all sub-calls. (e.g., `main()` takes 100ms).

Self-Time

The time spent *exclusively* within a method's own code, excluding sub-calls. This reveals the true source of computational work.

```
# From OSBot-Utils v3.58+

@timestamp
def process_html_body(graph, body_element):
    # ... logic ...

# The decorator finds the collector automatically.
# No invasive changes. No passing contexts.
```

NotebookLM

Uncovering the Killer's Modus Operandi

The team generated synthetic HTML documents of increasing size to observe how the system behaved under load. The results were damning, revealing a pattern of pathological scaling.

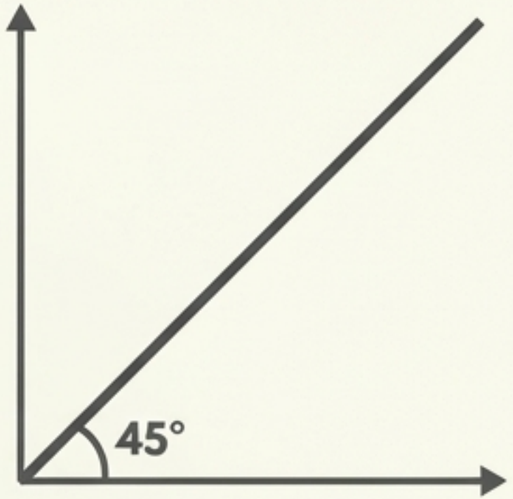
Elements	Total Time	Per Element Cost	Scaling Factor
10	532ms	**53.3ms**	baseline
15	1,410ms	**94.0ms**	1.76×
20	2,068ms	**103.4ms**	1.94×
30	~5,400ms	**~180ms**	~3.4×

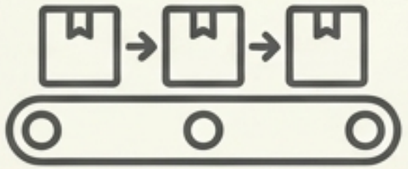
The per-element cost wasn't constant. It was tripling. Every new element made all the previous work more expensive. This is the signature of Quadratic Complexity: $O(n^2)$.

NotebookLM

The Mathematics of Murder: Why $O(n^2)$ is a Silent Killer

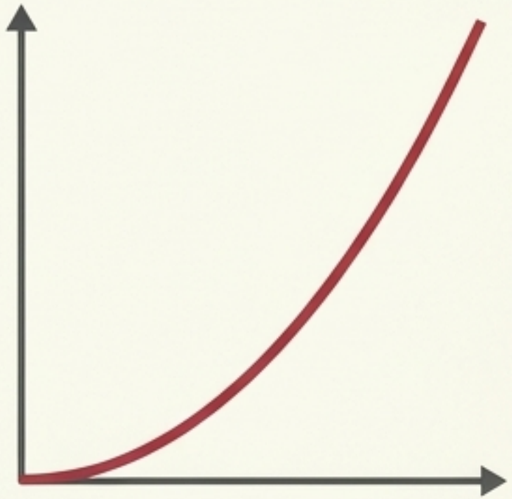
Linear Growth - O(n)

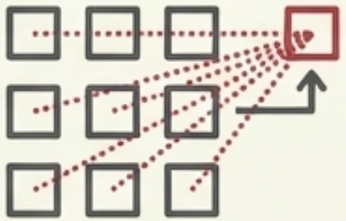




Healthy Scaling:
Double the input, double the work.

Quadratic Growth - O(n²)





Pathological Scaling:
Double the input, quadruple the work.

Total Work ≈ n × (n-1) / 2

NotebookLM

From Theory to Catastrophe

The 33KB document (**docs.diniscruz.ai`**) wasn't just a file; it was a crime scene of immense scale. The quadratic algorithm turned a manageable task into an impossible computation.

Input: 1 document (docs.diniscruz.ai`)

↓

Generates: 14,000+ Graph Objects

↓

Resulting in...

196,000,000

Approximate comparisons required by the O(n^2) loop.

NotebookLM

The Smoking Gun: Four Lines of Innocent-Looking Code

After instrumenting deeper into the `register_attributes` function (which accounted for 52% of self-time), the timeline view revealed the final clue: the first attribute lookup took **1.4ms**, the last took **14.4ms**. The cause was a simple, intuitive loop.

```
# The Culprit: A linear scan inside a creation loop

def get_or_create_value_node(graph, value):
    for node in graph.get_all_nodes():  # This loop runs for every new value...
        if node.value == value:         # …scanning a graph that is constantly growing.
            return node
    # ... create new node ...
```

"Perfectly readable. Perfectly intuitive. It's what any junior developer would write. And it was killing us." — Dinis Cruz

NotebookLM

The Fix: A Two-Line Change Replaces the Loop

The solution was not a complex refactor but a fundamental change in data structure. By replacing the linear scan (O(n)) with a dictionary lookup (O(1)), the quadratic behaviour was eliminated entirely.

```
  # ...
  def get_or_create_value_node(graph, value):
-     for node in graph.get_all_nodes():
-         if node.value == value:
-             return node
+     if value in graph.value_nodes_by_value:
+         return graph.value_nodes_by_value.get(value)
  # ... create new node and add to dictionary...
```

Investigation: 4 days. Coding the fix: minutes.

NotebookLM

Validation: From Quadratic Collapse to Linear Health

Micro-Benchmark Speedup

Function: _get_or_create_value_node (14 calls)

Before: 110 ms

After: 0.64 ms

172x

New Scaling Test Results

Elements	Before Fix	After Fix	Speedup
30	5,400ms	802ms	6.7×
100	TIMEOUT	2,166ms	∞
500	IMPOSSIBLE	10,750ms	∞

The new per-element cost remained constant at ~25ms, regardless of document size. This is what healthy software looks like.

NotebookLM

Lessons From the Case File, Part 1

Lesson 1: Don't Trust Minimal Tests

- **Finding:** Our initial profiling was dangerously misleading because it used trivial input data.

Takeaway: Performance testing must always be done with realistic, scaled workloads. Minimal tests can hide catastrophic scaling issues.

Lesson 2: "Self-Time" Reveals the True Culprit

Finding: “Total Time” shows you where the application is spending time overall, but “Self-Time” shows you which function is doing the actual computational work.

Takeaway: When hunting performance bugs, a profiler that can report on self-time is an indispensable tool. A hotspot report sorted by self-time points directly to the code that needs fixing.

NotebookLM

Lessons From the Case File, Part 2

Lesson 3: O(n^2) Hides in Innocent-Looking Code

- **Finding:** The root cause was not complex or obfuscated code, but a simple, readable loop.

Takeaway: The most dangerous performance bugs are often the ones that look harmless. Always question loops that iterate over a growing collection.

Lesson 4: Build the Tools You Need

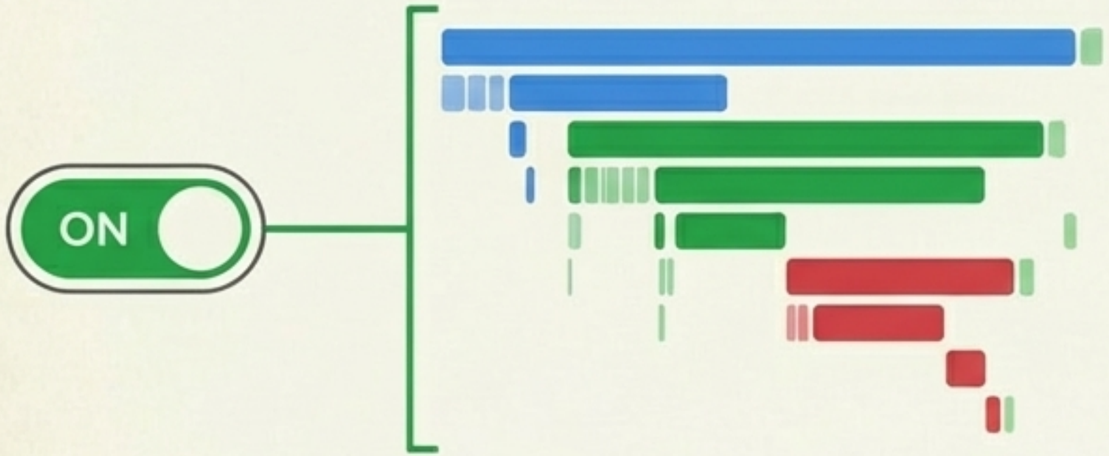
- **Finding:** We lacked the necessary instrumentation and built it during the crisis. The `@timestamp` system was forged by necessity.

Takeaway: Investing in internal tooling and observability pays for itself. The right tool can turn an impossible problem into a solvable one.

NotebookLM

Six Months Later: An Instrument for the Future

The `@timestamp` decorators remain in the production codebase. With a near-zero overhead of ~3 microseconds when inactive, they represent a permanent "flight recorder" for the system. When the next performance incident occurs—and one always does—the tools are already in place to solve it.



System Throughput: 14,000+ graph objects processed per second.

Test Suite Improvement: 22% faster.

The `docs.diniscruz.ai` **Site:** Now loads in **under 2 seconds.**

NotebookLM

Self-time never lies.

The @timestamp instrumentation system is open source. Available in OSBot-Utils v3.59+ on GitHub.
github.com/owasp-sbot/OSBot-Utils

NotebookLM