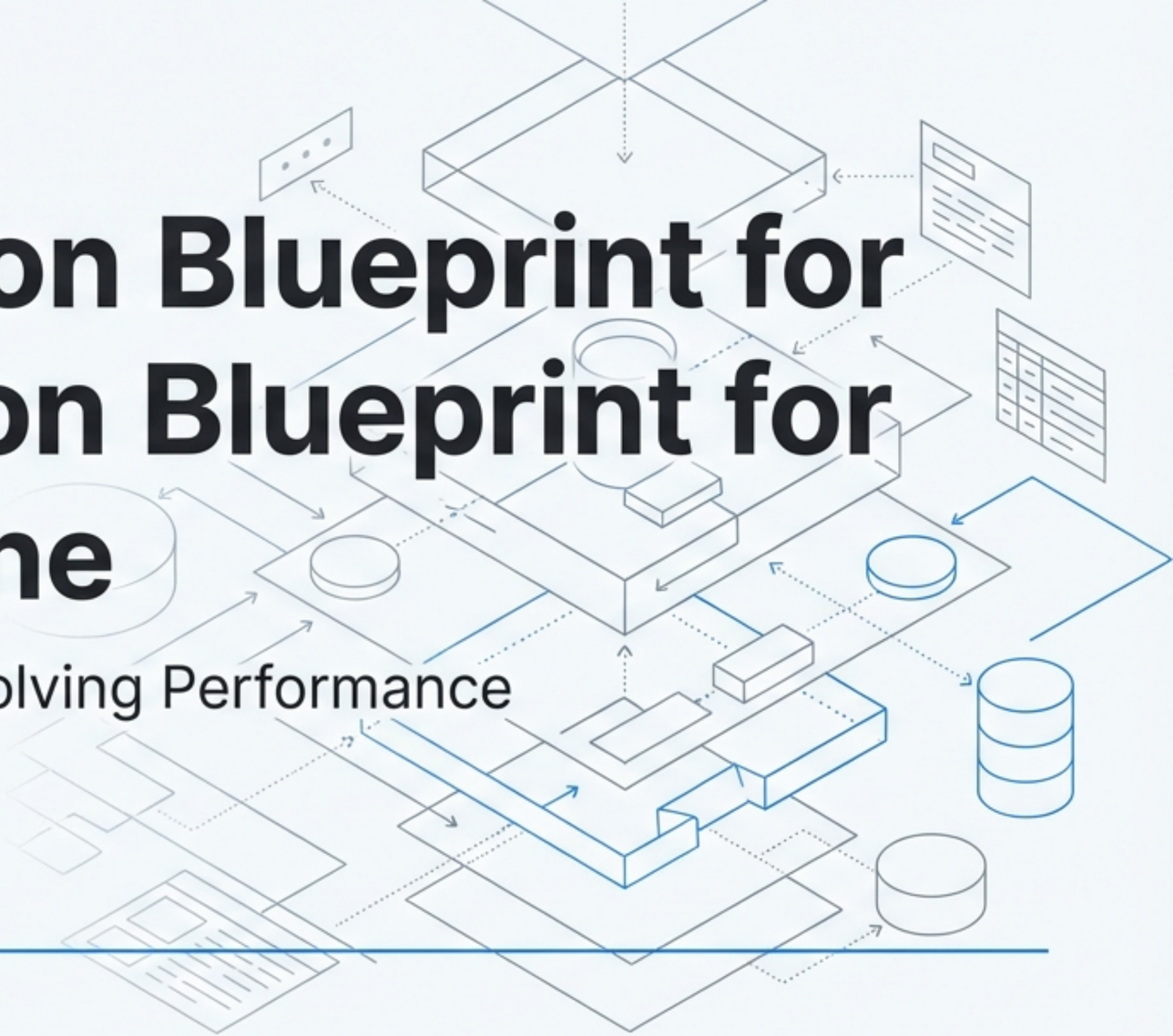
An Instrumentation Blueprint for An Instrumentation Blueprint for the MGraph Pipeline

A Phased Plan for Identifying and Resolving Performance Bottlenecks



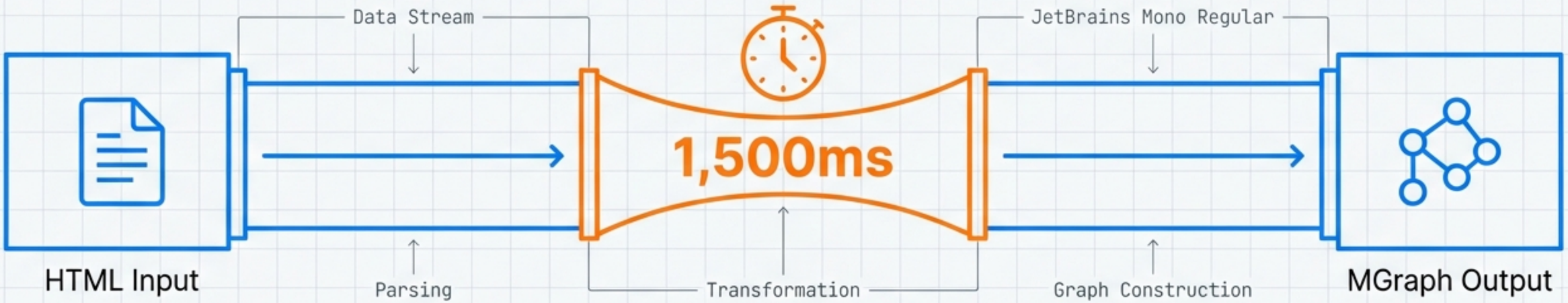
Project Version: v1.4.7

Instrumentation Framework: `osbot_utils.helpers.timestamp_capture` (v3.58.0+)

Affected Repositories: `MGraph-DB`, `MGraph-AI__Service__Html__Graph`

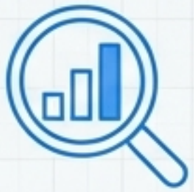
The Core Challenge: A 1,500ms Conversion Pipeline

- Our current HTML-to-MGraph conversion process exhibits a significant performance bottleneck, with conversion times reaching 1,500ms.



- This latency impacts user experience and limits system throughput.
- The complexity of the call tree makes it difficult to identify the true source of the delay (the 'self-time' of specific functions).
- Our objective is to precisely locate and measure these hotspots to enable targeted optimization.

Our Instrumentation Strategy Is Guided by Five Key Goals

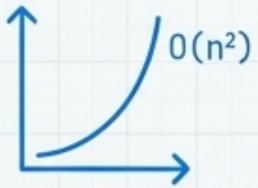


1. Identify Bottlenecks

Pinpoint the exact functions responsible for the **1,500ms** delay in the `HTML` → `MGraph` pipeline.

2. Measure True Self-Time

Move beyond simple call stacks to find the actual computational hotspots.

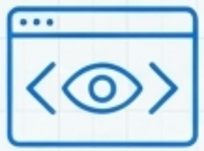


3. Track Scalability

Detect and analyse non-linear performance patterns, such as $O(n^2)$, as data complexity grows.

4. Enable Safe Production Monitoring

Implement a system with negligible overhead (**~3µs**) when not actively profiling.



5. Establish Permanent Visibility

Leave decorators in the codebase as a permanent tool for ongoing performance analysis.

The Solution: The `@timestamp` Decorator Framework

We will use the `@timestamp` decorator from the `osbot_utils` library to instrument our codebase. This framework is specifically designed for non-intrusive and low-overhead performance capture.

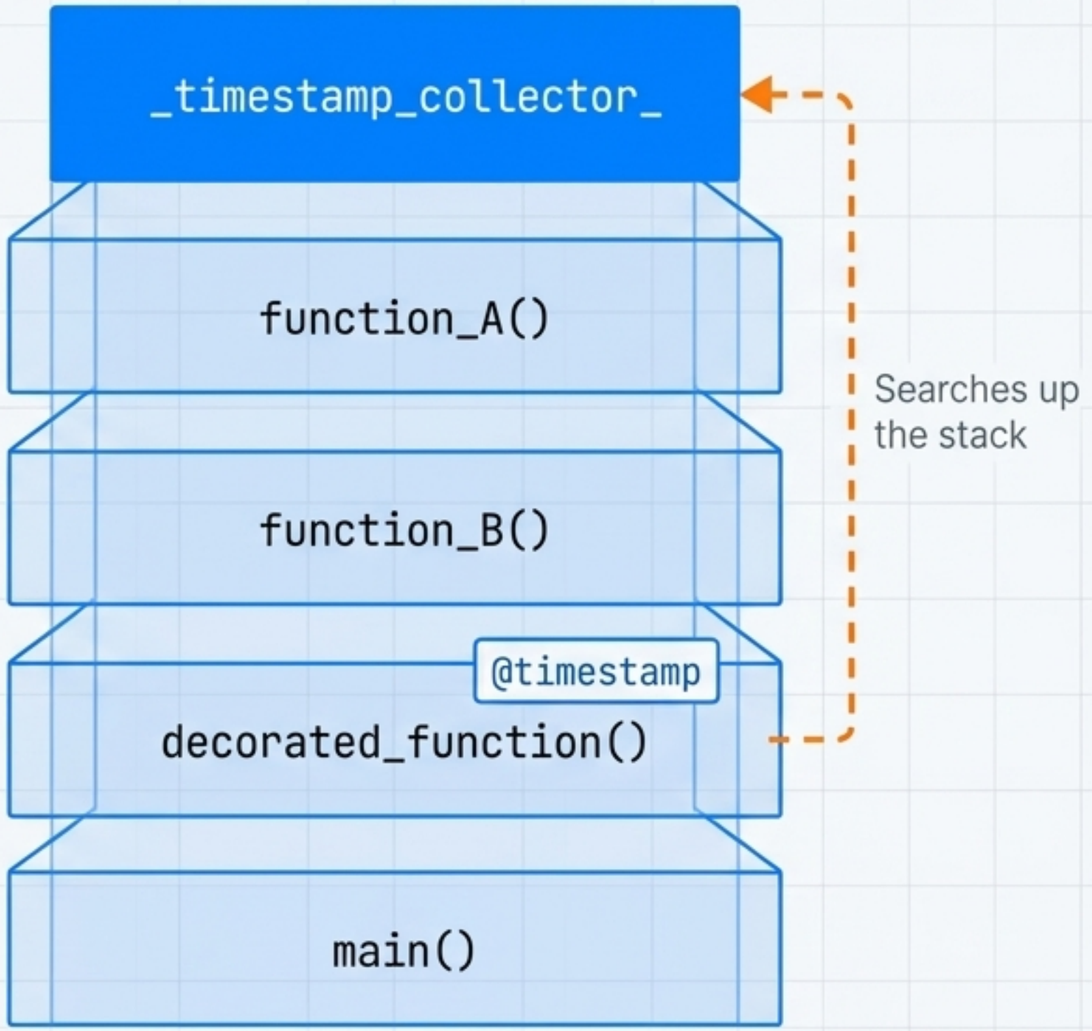
How It Works (High-Level)

Core Mechanism

It operates via **stack-walking**. The decorator searches up the call stack for a 'collector' object.

Key Advantage

This design eliminates the need to pass timing context variables through every function signature, keeping our business logic clean and unchanged.



A Framework Built on Deliberate Design Decisions

Decision	Rationale
Magic Variable (`_timestamp_collector_`)	A simple, explicit convention that clearly signals intent without polluting function scopes.
Stack-Walking Mechanism	The most critical feature: allows instrumentation with **zero changes to function signatures**.
No-op When Inactive	Designed to be production-safe. If no collector is found, the decorator becomes a near no-op.
High-Resolution Timer (`perf_counter_ns`)	Uses a monotonic clock to ensure accurate, high-precision timing immune to system time changes.
Built-in Self-Time Calculation	Automatically isolates a function's own execution time from the time spent in its children, revealing true bottlenecks.

Engineered for Production: Quantifying the Overhead

The framework's overhead is minimal and predictable, making it safe for continuous integration and production deployment.

Decorator present, no collector in stack

~3 µs

Production: Decorators remain in place, inactive.

Decorator present, collector is active

~8 µs

Debugging/Profiling: During active investigation sessions.

N nested decorated methods (active)

N × ~8 µs

Overhead scales linearly and predictably with call depth.

For any method that takes longer than **100µs** to execute, the **8µs** active overhead is less than 8% of its total runtime—an acceptable cost for detailed profiling.

Our Implementation Will Be Executed in Three Distinct Phases

Phase 1: Investigation

Goal: Find the root cause of the 1,500ms conversion time.

Phase 2: Optimization

Goal: Remediate the identified bottlenecks and validate performance improvements with the same instrumentation.

Phase 3: Production Monitoring

Goal: Finalise a set of permanent, key decorators for ongoing performance visibility in production.

Phase 1 Focus: Surgical Instrumentation for Discovery

Primary Goal: Locate the exact bottleneck(s) within the conversion pipeline.

Rules of Engagement for Phase 1:

- **Collectors in Tests Only:** The `_timestamp_collector_` will *only* be instantiated within test files. This guarantees zero behavioural changes or performance impact in production code.

- **Fine-Grained Instrumentation:** We will intentionally apply decorators at a granular level to ensure we can pinpoint the precise source of latency.

- **Iterative Approach:** The plan allows for easy addition or removal of decorators as we narrow down the investigation.

Instrumentation Map: `MGraph-AI__Service__Html__Graph`

Import Statement

```
from osbot_utils.helpers.timestamp_capture import timestamp
```

Targeted Methods

File: html_mgraph/converters/Html__To__Html_MGraph__Document.py

- convert → `@timestamp(name="html_mgraph.convert.to_document")`

File: html_mgraph/converters/Html_MGraph__Document__To__Html.py

- convert → `@timestamp(name="html_mgraph.convert.to_html")`

File: html_mgraph/graphs/Html_MGraph__Document.py

- setup → `@timestamp(name="html_mgraph.document.setup")`

File: html_mgraph/graphs/Html_MGraph__Body__Graph.py

- Main build method → `@timestamp(name="html_mgraph.body.build")`
- process_element → `@timestamp(name="html_mgraph.body.process_element")`

File: html_mgraph/graphs/Html_MGraph__Head__Graph.py

- Main build method → `@timestamp(name="html_mgraph.head.build")`

File: html_mgraph/graphs/Html_MGraph__Body__Graph.py

- Main build method → `@timestamp(name="html_mgraph.body.build")`
- process_element → `@timestamp(name="html_mgraph.body.process_element")`

File: html_mgraph/graphs/Html_MGraph__Head__Graph.py

- Main build method → `@timestamp(name="html_mgraph.attrs.build")`

File: html_mgraph/graphs/Html_MGraph__Attrs__Graph.py

- register_element → `@timestamp(name="html_mgraph.attrs.register_element")`
- set_attribute → `@timestamp(name="html_mgraph.attrs.set_attribute")`

File: html_mgraph/graphs/Html_MGraph__Scripts__Graph.py

- Main build/add method → `@timestamp(name="html_mgraph.scripts.build")`

File: html_mgraph/graphs/Html_MGraph__Styles__Graph.py

- Main build/add method → `@timestamp(name="html_mgraph.styles.build")`

Instrumentation Map: `MGraph-DB`

Targeted Methods

File: mgraph/actions/MGraph__Edit.py
  new_node → @timestamp(name="mgraph.edit.new_node")

File: mgraph/actions/MGraph__Edit.py
  new_edge → @timestamp(name="mgraph.edit.new_edge")

File: mgraph/actions/MGraph__Edit.py
  new_value → @timestamp(name="mgraph.edit.new_value")

File: mgraph/actions/MGraph__Edit.py
  delete_node → @timestamp(name="mgraph.edit.delete_node")
  delete_edge → @timestamp(name="mgraph.edit.delete_edge")

File: mgraph/actions/MGraph__Index.py
  Node indexing → @timestamp(name="mgraph.index.add_node")
  Edge indexing → @timestamp(name="mgraph.index.add_edge")

  Node removal → @timestamp(name="mgraph.index.remove_node")
  Edge removal → @timestamp(name="mgraph.index.remove_edge")
  Rebuild → @timestamp(name="mgraph.index.rebuild")

File: mgraph/actions/MGraph__Index__Values.py
  `get_or_create` value →
      @timestamp(name="mgraph.index.values.get_or_create")

  `compute_hash` →
      @timestamp(name="mgraph.index.values.compute_hash")
  Value `lookup → @timestamp(name="mgraph.index.values.lookup")

Consistent Naming and Actionable Outputs

Naming Convention

We will use a hierarchical dot-notation: `repo.module.class.method`.

```
@timestamp(name="mgraph.index.values.get_or_create")
```

This convention ensures that all collected data is automatically structured, sortable, and easy to analyse.

Expected Output: Hotspots Report

Name	Calls	Total Time (ms)	Self Time (ms)
html_mgraph.attrs.set_attribute	15,420	850.5	**710.2**
mgraph.index.values.get_or_create	15,420	412.3	**155.8**
html_mgraph.body.build	1	1450.1	45.1

Practical Guidance for Implementation

- **Verify Method Names:** The names in the plan are illustrative. Always confirm the exact method name in the current codebase before applying a decorator.
- **Start Top-Down:** Begin by instrumenting the high-level entry points (`convert`, `setup`, `build`). This will quickly isolate the most expensive sub-systems.
- **Add Granularity as Needed:** If a top-level method shows high self-time, add more decorators to its internal calls to drill down further.
- **Acknowledge Hot Loops:** Methods like `process_element` and `set_attribute` are expected to have very high call counts. This data is valuable for calculating average per-call cost.

Quick Reference: Key Files and Actions

Key Files to Modify

`MGraph-AI__Service__Html__Graph`

`.../Html__To__Html_MGraph__Document.py`

`.../Html_MGraph__Body__Graph.py`

`.../Html_MGraph__Attrs__Graph.py`

`MGraph-DB`

`.../MGraph__Edit.py`

`.../MGraph__Index.py`

`.../MGraph__Index__Values.py`

Core Workflow

1. Add Import

   from osbot_utils.helpers.timestamp_capture import timestamp

2. Add Decorator

   @timestamp(name="repo.module.method")

3. Create Collector in Test

```python
from osbot_utils.helpers.Timestamp_Capture import Timestamp_Capture

def test_conversion_performance():
    with Timestamp_Capture() as _timestamp_collector_:
        # ... your test code ...
    _timestamp_collector_.print_report()
```

This systematic approach will provide definitive data to guide our optimization efforts.