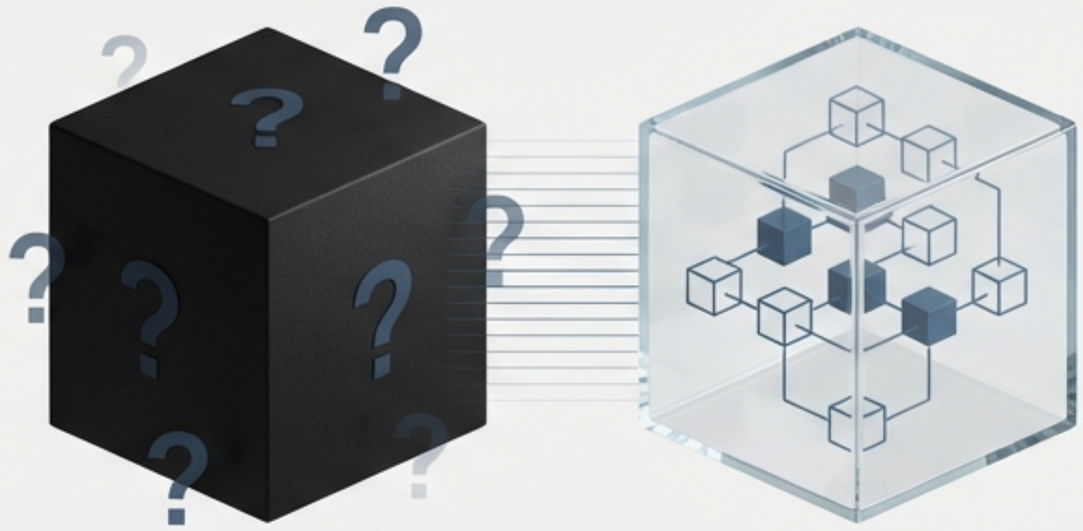
From Black Box to Glass Box

A Proposal for Non-Invasive, On-Demand Performance Tracing



Project: Timestamp Visualisation Routes | v1.4.10

When production services are slow, we need answers

Every developer has faced these critical questions when a transformation pipeline behaves unexpectedly:

- **Where is the time being spent?** Which specific methods are the performance hotspots?
- **What is the call flow?** How do the methods nest and in what sequence do they execute?
- **How does this run compare?** Is this specific execution slower than previous ones, and if so, why?

We need a solution that is non-invasive and on-demand

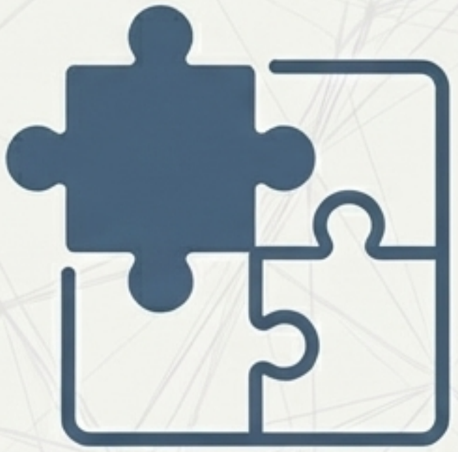
Traditional profiling often requires attaching debuggers, modifying application code, or restarting services. Our approach is designed to be fundamentally better.

Enable Per-Request

Activate detailed tracing for specific API calls without affecting the entire service.

Return Structured Data

Output traces in a serialisable, machine-readable format for analysis.

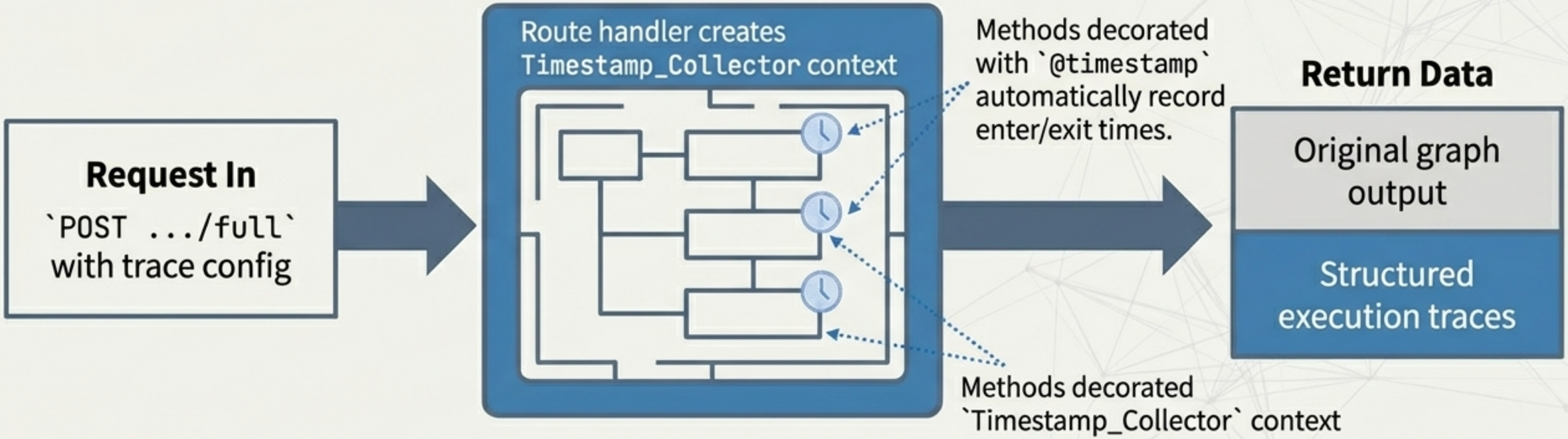


Integrate with Tooling

Natively support visualisation tools like speedscope and flame graphs.

The solution separates trace capture from core execution

The transformation pipeline remains completely unaware it is being traced. The route handler orchestrates the entire process, wrapping the execution and exporting the results.



Separation of concerns: The client chooses the format, the route handles the capture, and the core logic remains unchanged.

Three new routes provide flexible access to execution traces

We will introduce three new routes that mirror the existing graph endpoints, but are enhanced with the ability to capture and return timestamp data.

Route Suffix	Response Type	Primary Use Case
`.../full`	`Response__Full`	Complete trace data for storage and deep analysis.
`.../summary`	`Response__Summary`	A quick, high-level overview of performance hotspots.
`.../speedscope`	`Response__Speedscope`	An interactive format for immediate visualisation in speedscope.app.

The API provides the graph output alongside the trace data

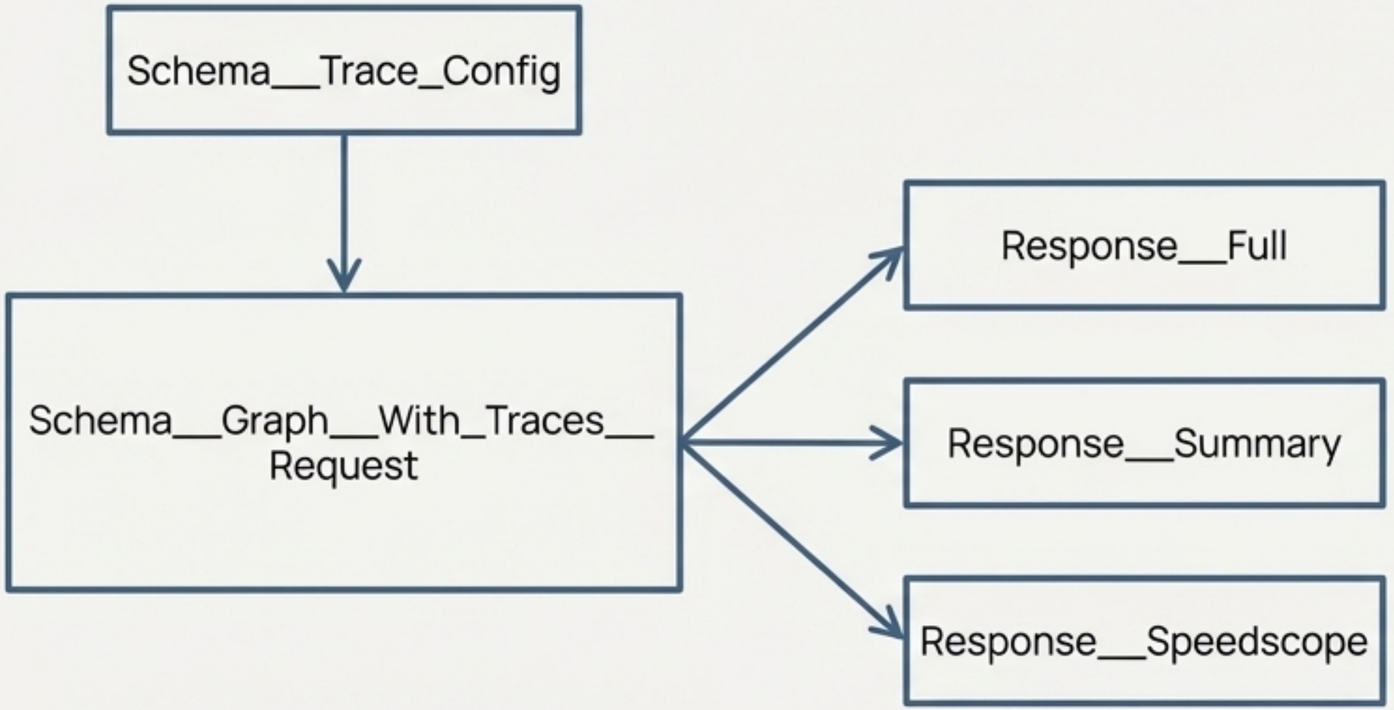
`POST …/speedscope`

```
{
  "graph_data": { ... },
  "transform_config": { ... },
  "trace_config": {
    "capture_traces": true,
    "trace_format": "speedscope"
  }
}
```

Response (speedscope format)

```
{
  "graph_output": { ... },
  "traces": {
    "$schema": "https://www.speedscope.app/file-format-schema.json",
    "shared": { ... },
    "profiles": [ ... ]
  }
}
```

Type-safe schemas ensure data integrity and clear contracts



Key Schemas

Enums*: `Enum__Trace_Format`, `Enum_ _Trace_Output`

Configuration*: `Schema__Trace_Config`

Request*: `Schema__Graph__With_Traces__Request`

Responses*: Full, Summary, and Speedscope schemas.

All data structures are defined and validated using Type_Safe schemas, eliminating ambiguity.

Interacting with the new endpoints is straightforward

cURL

```
curl -X POST "http://127.0.0.1:8000/.../
summary" \
  -H "Content-Type: application/json" \
  -d '{
   "graph_data": { ... },
   "trace_config": { "capture_traces": true }
}'
```

Python Client

```
import requests

url = "http://127.0.0.1:8000/.../speedscope"
payload = {
    "graph_data": { ... },
    "trace_config": {
        "capture_traces": true,
        "trace_format": "speedscope"
    }
}
response = requests.post(url, json=payload)
trace_data = response.json().get("traces")

# Now visualise trace_data in speedscope.app
```

The result is a powerful, integrated, and low-friction profiling system

Non-Invasive

- Core logic is untouched.
- No need for code changes or restarts to enable tracing.
- Leverages existing `@timestamp` decorators.

On-Demand

- Profiling is enabled via a simple API flag per-request.
- Zero performance overhead when not in use.
- Developers control when and what to trace.

Integrated

- Returns structured, serialisable data.
- Works out-of-the-box with tools like speedscope.app.
- Designed for distributed systems: capture here, visualise there.

This is the foundation for a comprehensive observability platform

The initial implementation unlocks several powerful future extensions. We have a clear, phased plan to build upon this work.

Phase 2: External Storage

Automatically push trace data to a dedicated observability backend (e.g., S3, OpenSearch) for historical analysis.

Phase 3: URL Source

Enable graph transformations directly from a URL, with tracing enabled via query parameters.

Phase 4: Trace Comparison

Build tooling to programmatically diff two execution traces to pinpoint the source of performance regressions.

Phase 5: LLM Analysis

Use Large Language Models to analyse trace data and provide natural language summaries of performance characteristics.

The implementation plan is well-defined and ready for execution

Schema & Enum Definition

- ☐ `Enum__Trace_Format` , `Enum__Trace_Output`
- ☐ `Schema__Trace_Config`
- ☐ all Request/Response schemas

Route Implementation

- ☐ Create `Routes__Timestamps` class
- ☐ Register routes in the FastAPI application

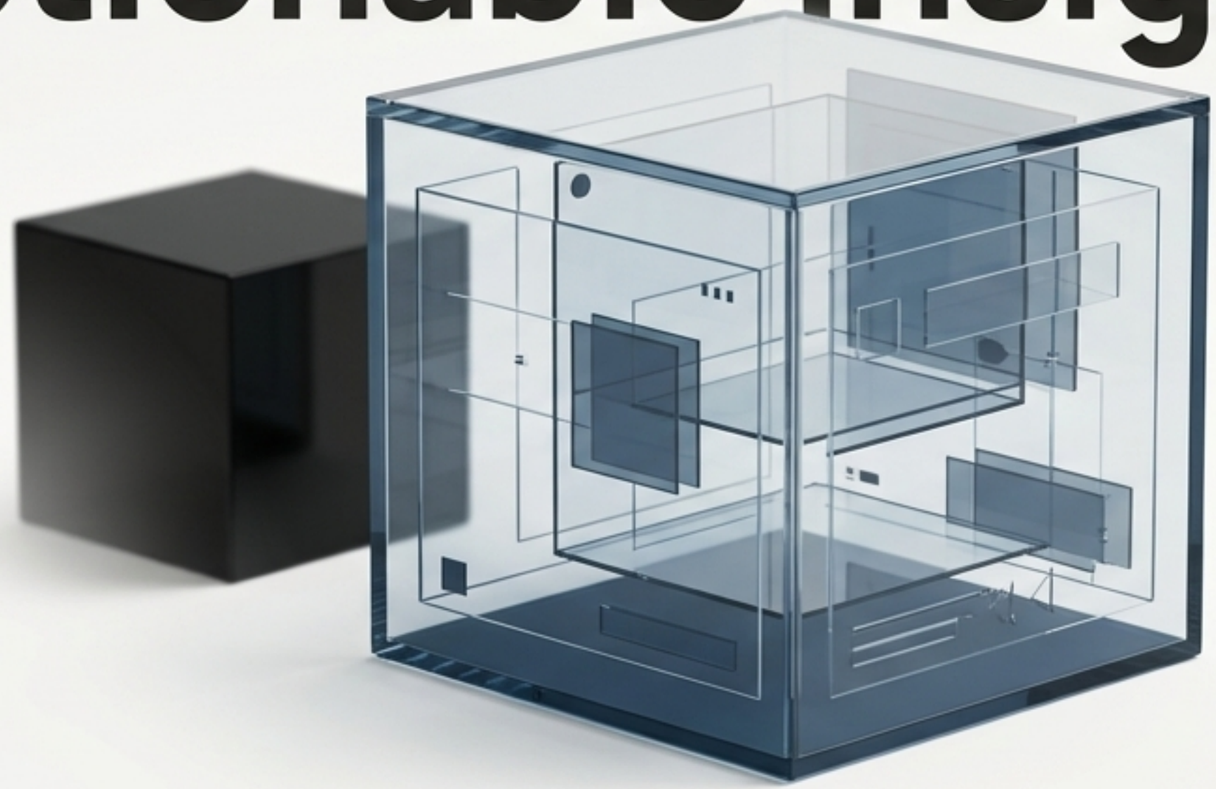
Instrumentation

- ☐ Add `@timestamp` decorators to key service methods

Validation

- ☐ Write comprehensive integration tests
- ☐ Test end-to-end with speedscope.app

Turning opaque behaviour into actionable insight



By providing non-invasive, on-demand tracing, we empower developers to understand, debug, and optimise our services more effectively than ever before.