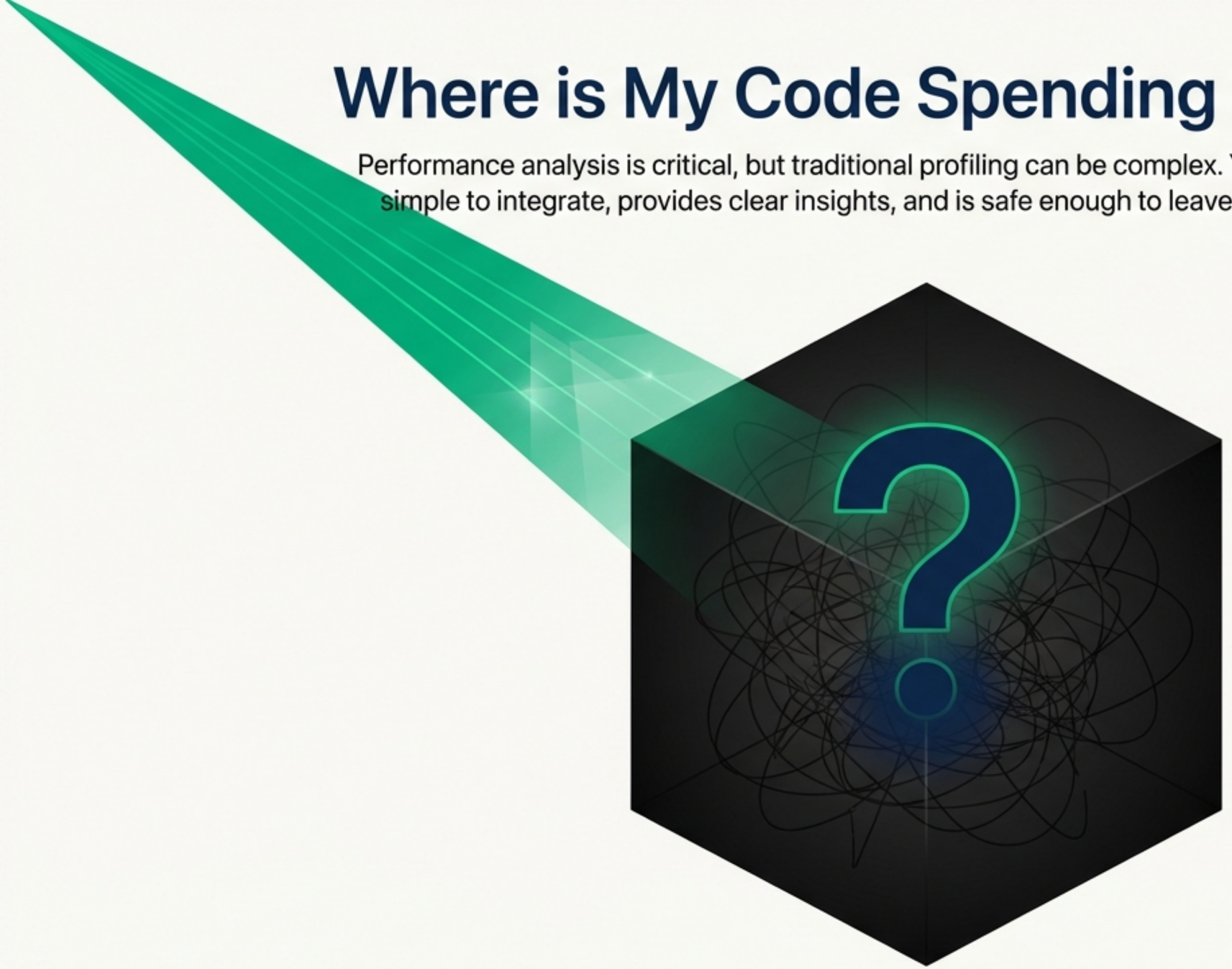
Where is My Code Spending Its Time?

Performance analysis is critical, but traditional profiling can be complex. You need a tool that is simple to integrate, provides clear insights, and is safe enough to leave in production code.



Illuminate Your Code's Performance in Two Steps.

Introducing `timestamp_capture`: Zero-boilerplate performance instrumentation.

1 Step 1: Decorate

Add the `@timestamp` decorator to the methods you want to profile.

```
from osbot_utils.helpers.timestamp_capture import
timestamp

@timestamp
def process_data(data):
    # ... complex logic ...
```

2 Step 2: Capture

Wrap your entry point with the `Timestamp_Collector`.

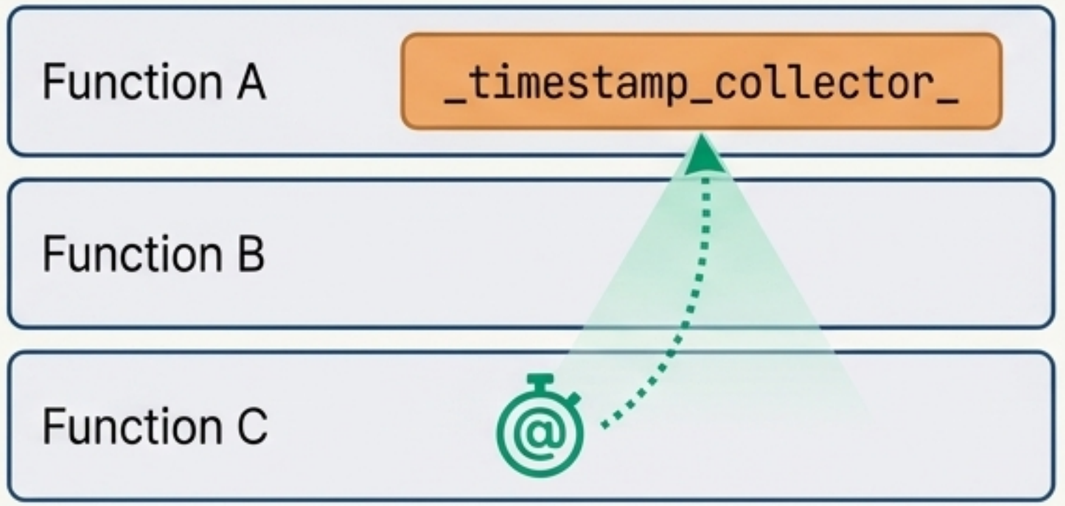
```
from osbot_utils.helpers.timestamp_capture import
Timestamp_Collector

with Timestamp_Collector() as _timestamp_collector:
    process_data(my_data)

_timestamp_collector.print_report()
```

That's it. The decorator automatically finds the collector via **stack-walking.**
No need to pass objects or modify function signatures.

The Elegance of ‘Magic’ and Production Safety.

How It Works: The Magic Variable

The `@timestamp` decorator walks the call stack looking for a local variable named exactly `_timestamp_collector_`. This design choice eliminates the need to pass a collector instance through your entire call chain.



Why It’s Production-Safe

If no `_timestamp_collector_` is found in the stack, the decorator is a near-zero-cost pass-through.

Scenario	Overhead	Production Safe?
@timestamp, no collector	~3 µs	✅ Yes
@timestamp, with collector	~8 µs	✅ Yes

You can leave the decorators in your production code. Only enable the collector when you need to profile.

Core Usage Patterns: Beyond a Single Method.

Pattern 1: Simple Method Profiling

The most direct way to measure a function's execution time.

```
@timestamp
def load_config_file():
    # ...
```

Pattern 2: Profiling Arbitrary Code Blocks

Use the `timestamp_block` context manager to time specific sections of code that aren't isolated in their own methods. Ideal for profiling phases within a larger function.

```
def process_pipeline():
    with timestamp_block("Phase 1: Data Extraction"):
        # ... extraction logic ...

    with timestamp_block("Phase 2: Data Transformation"):
        # ... transformation logic ...
```

From Function Names to Actionable Insights

Use custom metric names to organise your reports

By default, `@timestamp` uses the method's technical name (`__qualname__`). For clearer reports, provide a custom name using the `name=` argument. This allows you to create a logical hierarchy.

Before (Default Name)

```
# Report shows: MyClass.process_stage_one
@timestamp
def process_stage_one(self): ...
```

After (Custom Hierarchical Name)

```
# Report shows: pipeline.stage1.extract
@timestamp(name="pipeline.stage1.extract")
def process_stage_one(self): ...
```

Naming Best Practices

Pattern	Example	Use Case
domain.operation	`parser.parse`	Simple categorisation
component.stage.action	`pipeline.stage1.extract`	Pipeline stages
feature.version.method	`algorithm.v2.sort`	A/B testing

The Challenge: Distinguishing Calls to the Same Method.

What happens when a decorated method is called multiple times in a loop or with different important arguments? With `@timestamp`, every call is aggregated under the same name, hiding the details.

Visual (Before)

```
Name                                | Calls
------------------------------------|-------
process_file                        | 100
```

The Solution: @timestamp_args

Use the `@timestamp_args` decorator to create dynamic metric names at call time by interpolating function argument values.

```python
@timestamp_args(name="process_file:{filename}")
def process_file(filename, config):
    # ...
```

Visual (After)

```
Name                                | Calls
------------------------------------|-------
process_file:config.yml             | 1
process_file:data.csv               | 1
process_file:schema.json            | 1
```

Advanced Dynamic Naming with @timestamp_args.

Since **@timestamp_args** uses Python's **.format()**, you can access powerful formatting features.

Accessing Instance Attributes

Use `{self}` to access instance attributes, perfect for distinguishing calls in polymorphic base classes.

```
# In a base class
@timestamp_args(name="{self.__class__.__name__}.execute")
def execute(self): ...
# Reports will show "SubclassA.execute", "SubclassB.execute", etc.
```

Powerful Formatting for Alignment

Use format specifications to create perfectly aligned columns in your reports.

```
@timestamp_args(name="process:{task_name:<20} | {status}")
def run_task(task_name, status): ...
```

Common Format Specs

Spec	Input	Result	Use Case
`{name:10}`	`"head"`	`"head      "`	Fixed width
`{name:>10}`	`"head"`	`"      head"`	Right align
`{id:04d}`	`42`	`"0042"`	Zero-padded numbers
`{val:.2f}`	`3.14159`	`"3.14"`	Decimal places

The Payoff: Understanding Your Comprehensive Report

`print_report()` gives you the full picture.

Name	Calls	Total	Self	Avg	%Total
pipeline.run	1	500ms	10ms	500ms	100.0%
pipeline.stage1.extract	1	150ms	150ms	150ms	30.0%
pipeline.stage2.transform	1	340ms	40ms	340ms	68.0%
parser.parse_data	10	300ms	300ms	30ms	60.0%

`Total`
Time spent in this method *and all its children*. Useful for understanding high-level impact.

`Self`
Time spent *only in this method's code*, excluding children. **This is what you need to optimise.**

`Avg`
Average time per call.

`%Total`
Percentage of the total captured time.

Find Your True Bottlenecks with `print_hotspots()`.

The Concept

A method can have a high **Total-Time** simply because it calls other slow methods. **Self-Time** reveals where the actual work is being done. To optimise effectively, focus on the methods with the highest **Self-Time**.

The Tool

`print_hotspots()` shows a sorted list of methods by their Self-Time, pointing you directly to the most expensive parts of your code.

```
Hotspots (by Self-Time)
-----------------------------------
1. parser.parse_data          | 300ms
2. pipeline.stage1.extract    | 150ms
3. pipeline.stage2.transform  |  40ms
```

Visualise Your Code's Execution Story with `print_timeline()`.

The timeline report provides a chronological, indented view of your program's execution. It's invaluable for understanding call hierarchy, spotting unexpected sequences, and seeing how nested operations contribute to overall duration.

Timestamp	Level	Event	Name
10:00:00.000	0	▶ Enter	pipeline.run
10:00:00.005	1	▶ Enter	pipeline.stage1.extract
10:00:00.155	1	◀ Exit	pipeline.stage1.extract
10:00:00.158	1	▶ Enter	pipeline.stage2.transform
10:00:00.160	2	▶ Enter	parser.parse_data
10:00:00.190	2	◀ Exit	parser.parse_data
...			
10:00:00.498	1	◀ Exit	pipeline.stage2.transform
10:00:00.500	0	◀ Exit	pipeline.run

Visual Nesting

Points to the indented Name column.

Entry/Exit Markers

Points to the ▶ and ◀ icons.

Timestamp Alignment

Points to the Timestamp column, highlighting the chronological order.

Best Practices for Effective Instrumentation

DO ✓

- **Decorate Entry Points & Key Methods:** Start with high-level functions to get a broad overview.
- **Use Descriptive Collector Names:** e.g., `_ts_pipeline_run_collector_` if you need multiple collectors (though `_timestamp_collector_` is the default magic name).
- **Use `timestamp_block` for Phases:** Profile logical stages within a single method.

DON'T ×

- **Decorate Micro-Functions in Hot Loops:** The overhead can become significant. Profile the loop itself or a higher-level function.
- **Forget the Magic Variable Name:** The collector must be named `_timestamp_collector_` to be found automatically.

Common Troubleshooting Scenarios

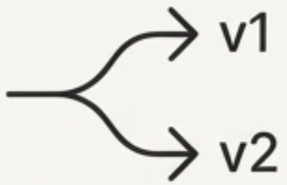
Problem	Likely Cause & Solution
No data is being captured.	**Cause:** The collector variable is not named exactly `_timestamp_collector_`. Check for typos.
A method shows 0ms Self-Time.	**Cause:** The method does no work itself and only calls other decorated methods. This is normal. Look at its children to find the real work.
Report is too noisy (too many entries).	**Cause:** Decorating a low-level function that is called thousands of times inside a loop. **Solution:** Decorate the higher-level method containing the loop instead.
Can’t distinguish multiple calls to the same method.	**Cause:** Using `@timestamp` where each call needs to be tracked separately. **Solution:** Switch to `@timestamp_args` and use argument values in the name.

Common Integration Scenarios

Profiling a Data Conversion Pipeline

Goal: Understand which stage (extract, transform, load) is the bottleneck.

Implementation: Use `@timestamp(name="pipeline.stage.action")` on key methods for each stage. Use `print_hotspots()` to identify the slowest stage by Self-Time.



Comparing Two Algorithm Implementations

Goal: A/B test two versions of an algorithm.

Implementation: Use `@timestamp(name="algorithm.v1.sort")` `@timestamp(name="algorithm.v1.sort")` and `@timestamp(name="algorithm.v2.sort")`. Run both and compare the `Total-Time` from `print_report()`.

Continuous Performance Monitoring

Goal: Track performance over time in a CI/CD environment.

Implementation: Leave decorators in the code. In CI, enable the collector, use `get_method_timings()` for programmatic access to the data, and assert that key metrics do not exceed a performance budget.

Your timestamp_capture Integration Checklist

A step-by-step guide to adding powerful, production-safe profiling to your project.

- [ ] Import `@timestamp`, `@timestamp_args`, and `Timestamp_Collector`.
- [ ] Add `@timestamp` to key methods (entry points, major processing steps).
- [ ] Use `@timestamp(name="...")` for clear, hierarchical naming.
- [ ] Use `@timestamp_args(name="...{arg}")` to distinguish calls by their arguments.
- [ ] Use `timestamp_block` for code phases not in separate methods.
- [ ] Name your collector variable exactly `_timestamp_collector_`.
- [ ] Use `print_hotspots()` to find bottlenecks (via Self-Time).
- [ ] Use `print_timeline()` to understand execution flow.
- [ ] Remember: Decorators are safe to keep in production code.

Get Started Now.

Install

```
pip install osbot-utils
```

Import

```
from
osbot_utils.helpers.timestamp_
capture import timestamp,
Timestamp_Collector
```

Source Repository

github.com/owasp-sbot/OSBot-Utils