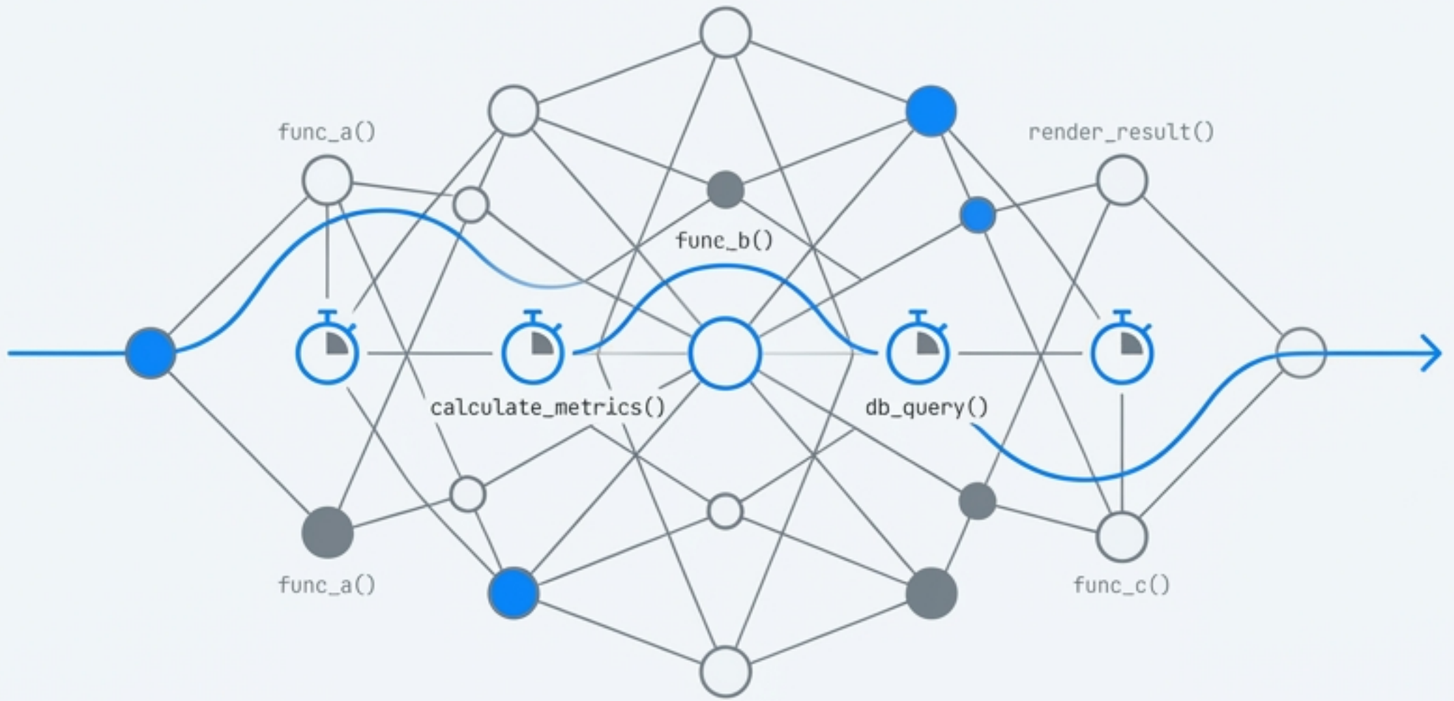
Profiling Without the Pain

A Lightweight Instrumentation Framework for High-Performance Python



The Developer's Dilemma: The Profiling Paradox.

We need to measure performance to optimise our code. But the very act of measuring often changes the code's behaviour or adds significant complexity. How do we gain insight without compromising the integrity of the system we are trying to measure?

Pristine Code

```python
def process_data(data: list[int]) -> list[int]:
    """Optimized data processing."""
    results = []
    for item in data:
        if item % 2 == 0:
            processed = item * 2
        else:
            processed = item + 1
        results.append(processed)
    return results

data_in = [1, 2, 3, 4, 5]
data_out = process_data(data_in)
```

Intrusive Profiling

```python
import time

def process_data(data: list[int]) -> list[int]:
    start_time = time.perf_counter()
    """Optimized data processing."""
    results = []
    for item in data:
        start_loop = time.perf_counter()
        if item % 2 == 0:
            processed = item * 2
        else:
            processed = item + 1
            results.append(processed)
        end_loop = time.perf_counter()
        print(f"Loop iteration took: {end_loop - start_loop} seconds")
    end_time = time.perf_counter()
    print(f"Total function time: {end_time - start_time} seconds")
    return results

data_in = [1, 2, 3, 4, 5]
start_overall = time.perf_counter()
data_out = process_data(data_in)
end_overall = time.perf_counter()
print(f"Overall time: {end_overall - start_overall} seconds")
```

Overhead!

Changes timing!

Clutter!

We Defined Our Ideal Profiler with Six Guiding Principles

Minimal Code Changes

Just add the `@timestamp` decorator to methods.

No Signature Changes

Methods should not need to be modified to accept a collector parameter.

Thread & Call-Specific

Each distinct invocation must have its own isolated data collector.

Extremely Low Overhead

The performance impact must be negligible, especially when not actively profiling.

Reusable & Generic

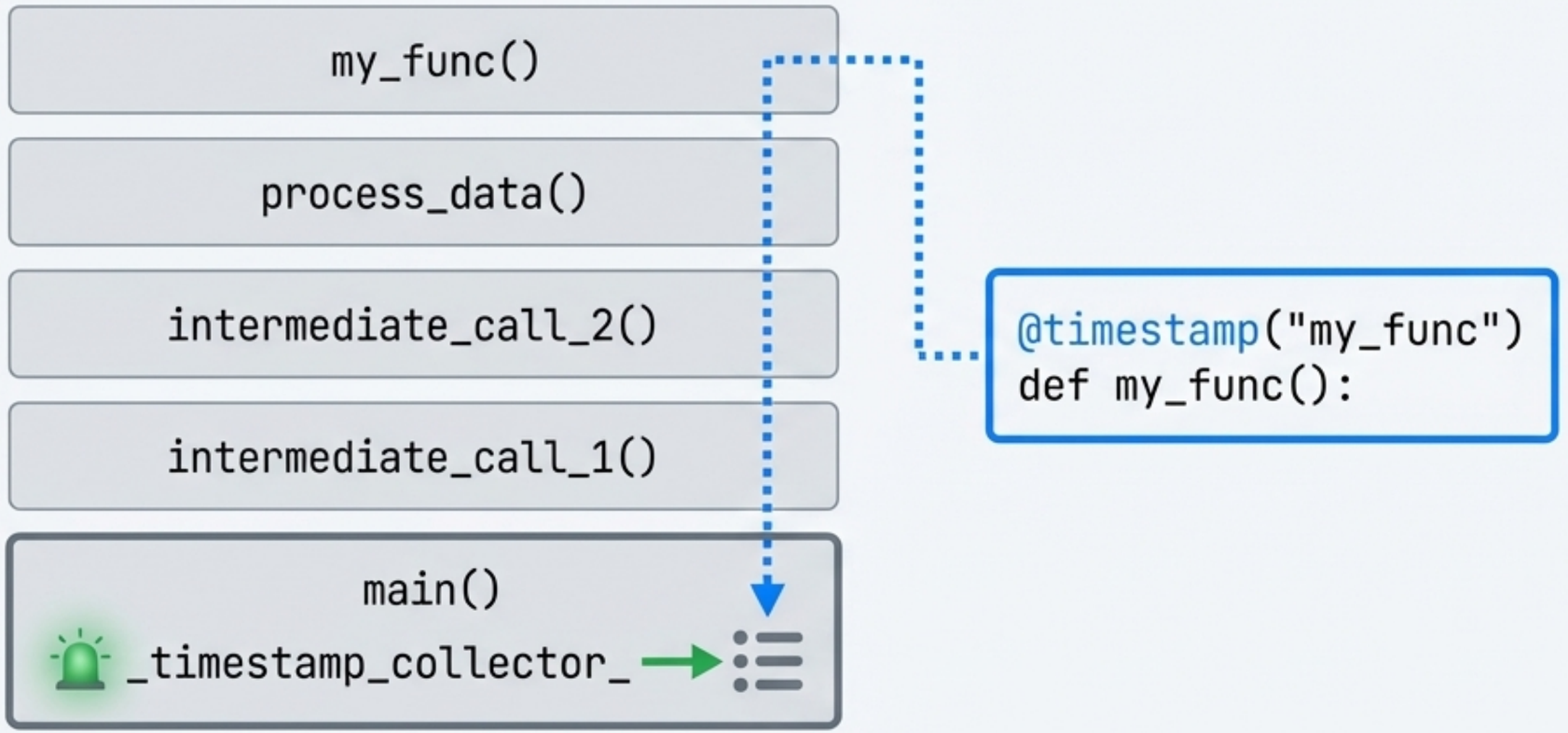
The system must not be tied to any single application (e.g., Html_MGraph).

Leave-in-Place

Decorators must be safe enough to remain in the codebase permanently.

The Breakthrough: Finding the Collector Without Being Told.

The key is a decorator that cleverly inspects the execution context. Instead of passing a collector object down through every function call, the decorated function looks *up* the call stack to find it.



No parameters passed. No global state. Just a simple, powerful convention.

Four Deliberate Design Choices Make This Possible

1. Stack-Walking to Find the Collector

Why: This is the core mechanism that allows us to avoid passing the collector through every function signature, satisfying the ‘No Signature Changes’ goal.

2. Using the `_timestamp_collector_` Magic Variable

Why: A simple, unambiguous convention that works reliably with Python’s frame inspection tools. It is the ‘beacon’ the stack walk looks for.

3. No-Op Behaviour When Collector is Absent

Why: Guarantees safety. If no collector is found in the call stack, the decorator does almost nothing, making it safe to 'Leave-in-Place' in production code.

4. Timing with `perf_counter_ns`

Why: We use a monotonic, high-resolution clock. This ensures measurements are accurate and not affected by system time changes (e.g., NTP updates).

The Cost of Insight is Measured in Microseconds.

We rigorously analysed the performance overhead. The results confirm that the framework is exceptionally lightweight and suitable for permanent integration.

~1-2 µs

per call **When not profiling.** The cost of a **decorated call** when no collector is active. This is the 'leave-in-place' cost.

~3-5 µs

per call **When actively profiling.** The total cost, including the stack walk and recording two timestamps.

Our Approach Delivers the Best Balance of Overhead and Granularity.

Approach	Granularity	Key Limitation
`This Approach` JetBrains Mono Bold, #212529	Method-level Inter Regular, #212529	Requires decoration Inter Regular, #212529
`cProfile` JetBrains Mono Regular, #212529	Function-level Inter Regular, #212529	All-or-nothing, not for production Inter Regular, #212529
`sys.settrace` JetBrains Mono Regular, #212529	Line-level Inter Regular, #212529	Extremely high overhead Inter Regular, #212529

We Considered and Rejected Other Common Approaches

- ✕ **`sys.settrace` / `sys.setprofile`**
  Reason Rejected: Prohibitively high overhead (~100-500µs per call) and captures far too much information for targeted analysis.

- ✕ **`cProfile`**
  Reason Rejected: An "all-or-nothing" profiler that cannot be left in place and is unsuitable for fine-grained, conditional profiling in production environments.

- ✕ **Context Variables (`contextvars`)**
  Reason Rejected: A viable alternative, but still requires a mechanism to set the context. Our stack-walking approach was deemed more elegant and explicit.

- ✕ **Thread-Local Storage**
  Reason Rejected: Introduces implicit global state, which can lead to complex cleanup issues and is less explicit than a lexically-scoped collector.

A Simple and Powerful API with Three Core Tools. Core Tools.

The `@timestamp` Decorator

For instrumenting entire methods or functions.

```
from timestamp_capture import
timestamp

@timestamp("data_processing")
def process_large_dataset(data):
    # ... heavy computation ...
```

The `timestamp_block` Context Manager

For profiling specific, granular blocks of code within a larger method.

```
from timestamp_capture import
timestamp_block

def complex_function():
    # ... setup code ...
    with timestamp_block("file_io"):
        # ... read from disk ...
    # ... teardown code ...
```

The `Timestamp_Collector`

The object that activates profiling and gathers all timestamp data.

```
from timestamp_capture import
Timestamp_Collector

collector = Timestamp_Collector()
with collector:
    process_large_dataset(my_data)
collector.print_report()
```

Ready for Production: Integration
Integration in Three Steps.

1. **Decorate.**
   Add the `@timestamp` decorator to key methods throughout your codebase that you might want to profile in the future.

2. **Profile.**
   In your top-level application entry point, test harness, or a specific debug route, wrap the relevant code block with a `Timestamp_Collector`. Profiling is now active for that specific call tree.

3. **Deploy.**
   Commit and deploy your code with the decorators still in place. They are designed to be a permanent part of the codebase.

Remember: The dormant overhead is only **~1-2 µs**. The decorators are effectively invisible until a collector is present.

The Timestamp Capture System: An Elegant Solution to a Complex Problem.

Feature	Implementation
Finds Collector without Parameters	Stack walk looking for `_timestamp_collector_`
Minimal Code Changes	`@timestamp` decorator and `timestamp_block`
Extremely Low Overhead	**~3-5µs** (profiling), **~1-2µs** (dormant)
Thread-Safe & Call-Specific	Each call tree finds its own collector instance
Safe to Leave-in-Place	Decorators are a no-op when no collector is found
Calculates Exclusive Time	Automatically tracks and subtracts time spent in children
Aggregates Results	Sums multiple calls to the same method in reports

The Paradox is Solved.

The Timestamp Capture system provides deep performance insight without forcing developers to compromise on code simplicity or production safety.

Insight without Intrusion.

Find the code and full documentation at: [link-to-internal-repo]