



Peeking under the hood of instrumentation agents

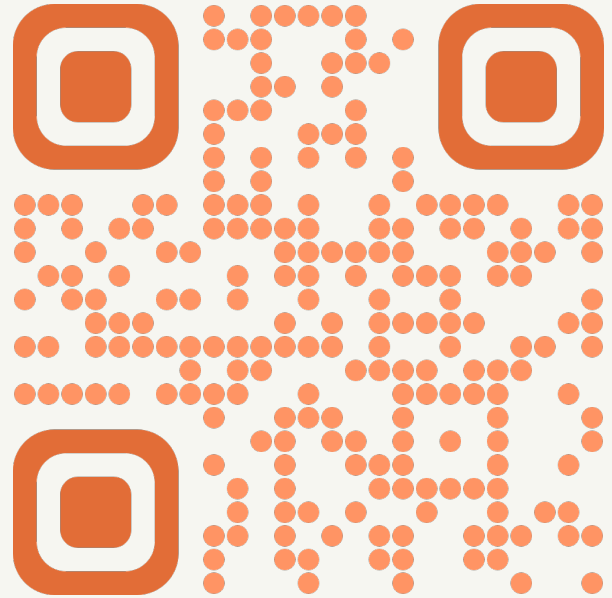
Amitosh Mahapatra
Computer Whisperer



\$ whoami



- Started coding early
 - First interaction with Python was in 2010 trying to make gDesklets
- Currently incanting prompts @ **CodeRabbit**
 - AIOps & AI O11y
 - Previously @ Gojek (GET!), Rippling
- Active in Open Source

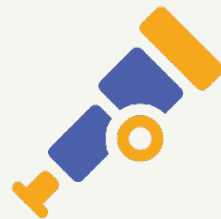


Scan to know more!



A recap of instrumentation tools

- Monitor, measure and analyze code execution, performance metrics, and system behavior during runtime
- Enable debugging, profiling, tracing and gathering operational insights without modifying application code
- APMs: Collect metrics, logs and traces from production systems
- Profilers: Measure how code runs, focused on code optimization

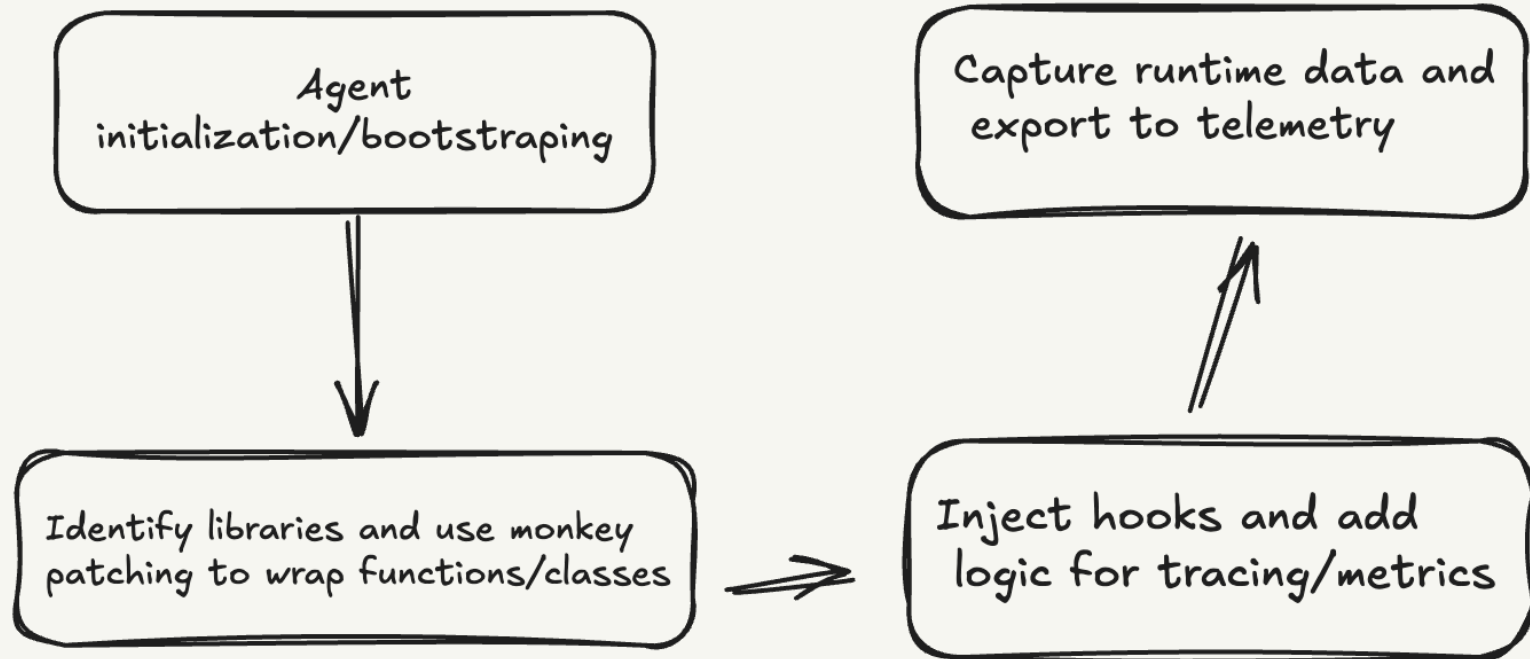




Programmatic (manual) vs Zero-Code (auto)

- Implemented as a library SDK
 - Manually define metrics and tracepoints with decorators, context-managers etc
 - Fine-grained and supports bespoke business metrics
 - Lot more work!!
- Injects instrumentation code and runtime automatically
 - No need to modify code
 - Lots of generic metrics out-of-the box
 - Automatically integrates with libraries
 - Run command -> get data!

How does automatic instrumentation work?





Automatic injection

- Instrumentation code needs to start before any application code is run
- Python provides multiple ways of running something before application code runs
 - Very first import in entrypoint of application (not so automatic)
 - "[sitecustomize.py](#)" under any directory of PYTHONPATH
- Read configuration from environment variables or files



Environment / Library dependent behaviors

Customize behavior for different frameworks and libraries

- Modular “instrumentor” plugins for each supported framework.

Examples:

- Detecting frameworks (Flask, FastAPI, Django, etc.) and applying framework-specific hooks
- Adapting to async vs sync code (e.g. asyncio, threading)
- Using library introspection (via `importlib.metadata`, `sys.modules`) to apply compatible hooks



Monkey Patching

Exploiting dynamic languages – replacing original functions with wrapped versions

Preserves original behavior while adding new behavior transparently

```
1 import requests
2 _orig = requests.Session.request
3
4 def wrapper(self, method, url, *a, **kw):
5     # pre
6     try:
7         return _orig(self, method, url, *a, **kw)
8     finally:
9         ... # post
10 requests.Session.request = wrapper
11 return go(f, seed, [])
12 }
```



Better Monkey Patching with Wrapt

- Problems with manual monkey patching:
 - signature drift, bound vs unbound methods, double-patching, attribute loss.
- wrapt fixes these
 - Preserves signatures/metadata and binding semantics.
 - Works with instance, class, module functions.
 - Offers import hooks to patch when a module is imported (order-independent).

```
1 import wrapt
2 from .engine import tracer
3
4 # Example: wrap requests.Session.request
5 @wrapt.patch_function_wrapper('requests', 'Session.request')
6 def _patched_request(wrapped, instance, args, kwargs):
7     method, url = args[0], args[1]
8     with tracer.record("http.client", attributes={"http.method": method, "http.url":
9         url}):
10         # Optionally inject headers for distributed tracing
11         return wrapped(*args, **kwargs)
```



Maintaining state

- Store instrumentation data without polluting global scopes
- Using contextvars - stdlib module (PEP 567)
- Mark events, request-response life cycles, timing info
- Store custom metadata from code
- Automatically works across async/await, tasks, and modern schedulers



Building your own instrumentation

Wrap modules

Record data

Profit!!



The code we want to trace

```
# ...  
  
client = OpenAI(api_key=api_key)  
    response = client.chat.completions.create(  
        model="gpt-5-mini",  
        messages=[{"role": "user", "content": "What is the capital of France?"}])  
  
assistant_message = response.choices[0].message.content  
  
print(f"\nAgent response: {assistant_message}")
```

Full code at: <https://github.com/recrsn/llm-observability>



Tracing code

```
def wrap_openai_chat_completions():
    from openai.resources.chat import completions
    original_create = completions.Completions.create

    @wrapt.decorator
    def traced_create(wrapped, instance, args, kwargs):
        response = wrapped(*args, **kwargs)

        print(f"OpenAI Call: input: {response.usage.prompt_tokens} output:
        {response.usage.completion_tokens}")

        return response

    completions.Completions.create = traced_create(original_create)
```



Tracing entrypoint

```
#!/bin/bash
```

```
SCRIPT_DIR="$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd)"
```

```
export PYTHONPATH="${SCRIPT_DIR}:${PYTHONPATH}"
```

```
# Enable LLM observability
```

```
export LLM_OBSERVABILITY_ENABLED=1
```

```
# Run the Python script with all arguments passed to this script
```

```
python "$@"
```



```
Workspace/regrsn/llm-observability | main
> uv run main.py
Sending request to OpenAI agent...

Agent response: The capital of France is Paris.
```

Running without tracing



```
Workspace/recrsn/llm-observability ʘ main
> uv run ./tracer.sh main.py 2s

=====
LLM Observability: ENABLED
=====

[2025-10-18 08:30:20.319] [TRACE] OpenAI chat.completions.create patched successfully
[2025-10-18 08:30:20.319] [TRACE] requests library not found, skipping patch
=====

Sending request to OpenAI agent...
[2025-10-18 08:30:20.347] [TRACE] OpenAI API Call Started
[2025-10-18 08:30:20.347] [TRACE]   Model: gpt-5-mini
[2025-10-18 08:30:20.347] [TRACE]   Messages: 2 message(s)
[2025-10-18 08:30:20.347] [TRACE]     [0] system: You are a helpful assistant.
[2025-10-18 08:30:20.347] [TRACE]     [1] user: What is the capital of France?
[2025-10-18 08:30:22.125] [TRACE] OpenAI API Call Completed
[2025-10-18 08:30:22.125] [TRACE]   Duration: 1.778s
[2025-10-18 08:30:22.125] [TRACE]   Input Tokens: 23
[2025-10-18 08:30:22.125] [TRACE]   Output Tokens: 16
[2025-10-18 08:30:22.125] [TRACE]   Total Tokens: 39
[2025-10-18 08:30:22.125] [TRACE]   Estimated Cost: $0.000013
[2025-10-18 08:30:22.125] [TRACE]   Response: The capital of France is Paris.
[2025-10-18 08:30:22.125] [TRACE] -----

Agent response: The capital of France is Paris.
```

Tracer in action



What's missing

- Provide a userland API for configuration
- Record more data
- Send to an actual database
- Allow adding custom labels from application code
- Trace a logical group of operations



Case study 1: Open Telemetry



Architecture

- Has a dedicated entry point (opentelemetry-instrument)
- Static configuration of library instrumentation
 - Uses wrapt for monkey-patching internally (auto-instrumentation)
- Exposes importlib.metadata entry points
- Context propagation via contextvars
- Configurable via environment variables
- Library specific injectors through distributed tracing
 - Automatically adds W3C trace headers to libraries like request
- Supports multiple destinations through exporters



How it works

1. Application code or auto-instrumented library calls `start_span()`
2. Span processors handle start/end lifecycle events
3. Exporters batch and send data asynchronously



Case Study 2: Datadog Agent



Architecture

- Has a dedicated entrypoint (dd-trace run)
- manipulates PYTHONPATH to load before user code
- Auto-loads using [sitecustomize.py](#)
- Initializes products like tracing, profiling, AppSec, runtime metrics
- Uses a module watchdog pattern based on `wrapt.importer.when_imported()` intercept library imports.
 - When a module (e.g., Flask) is imported:
 - Integration patch.py is dynamically loaded
 - Functions like `Flask.wsgi_app` and `dispatch_request` are wrapped
- Uses an event hub to reuse interceptors across all products



How it Works

- Monkey-patched code generates events on the event hub
- Event Hub dispatches and listens for framework events (flask.request.started, etc.)
- Context Management tracks per-request data and span relationships
- Other products (tracing, profiling, AppSec) register listeners independently
- Tracing headers, span ID injection, log injection etc are performed in the handler event itself



Q&A