Production-time Profiling for Python

Julien Danjou

FOSDEM — 1st February 2020

DATADOG

Julien Danjou

Staff Engineer @ Datadog

DATADOG

@juldanjou

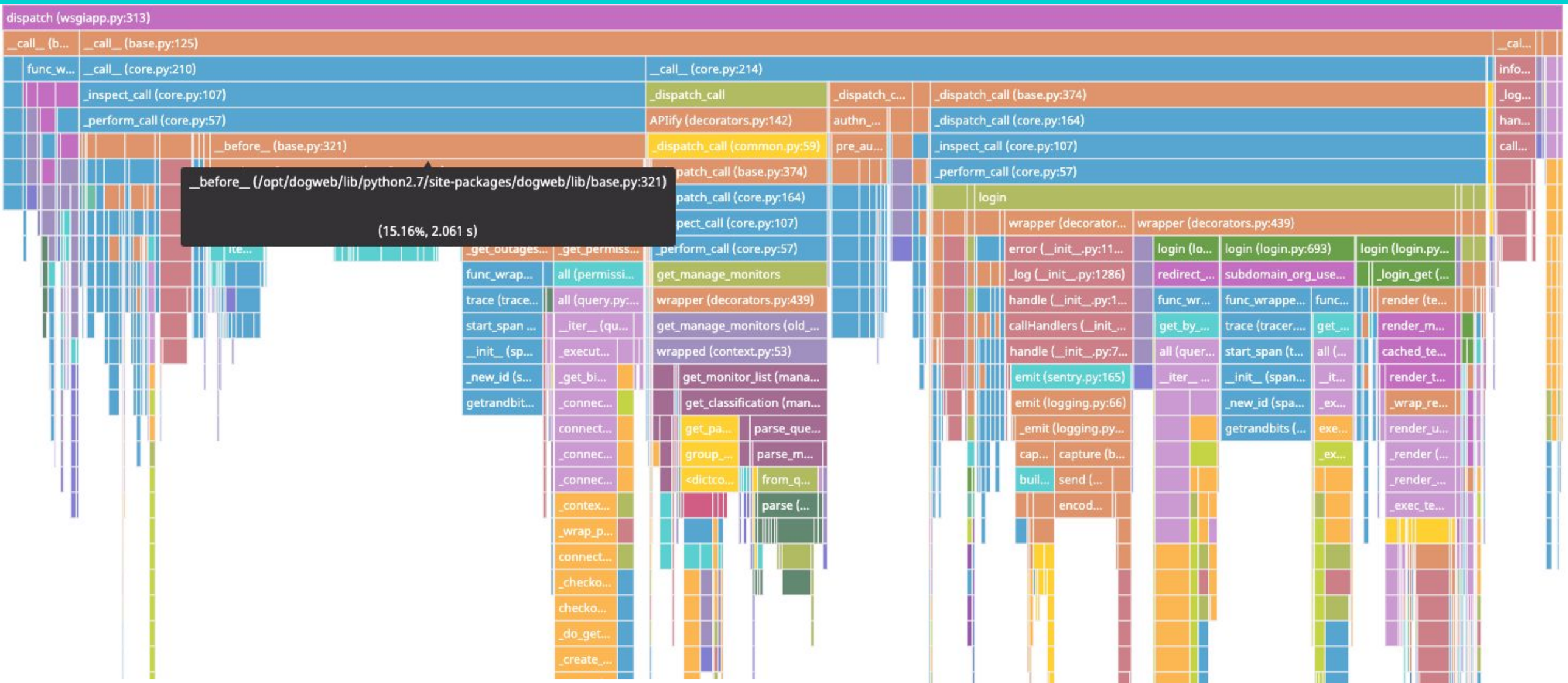
https://julien.danjou.info

Forward-looking Statements

This presentation may include certain "forward-looking statements" within the meaning of Section 27A of the Securities Act of 1933, as amended, and Section 21E of the Securities Exchange Act of 1934, as amended, including statements concerning our product offerings. These forward-looking statements reflect our current views about our plans, intentions, expectations, strategies and prospects, which are based on the information currently available to us and on assumptions we have made. Actual results may differ materially from those described in the forward-looking statements and are subject to a variety of assumptions, uncertainties, risks and factors that are beyond our control, including those risks detailed under the caption "Risk Factors" and elsewhere in our Securities and Exchange Commission filings and reports, including the final prospectus for our initial public offering filed with the Securities and Exchange Commission on September 19, 2019, as well as future filings and reports by us. Except as required by law, we undertake no duty or obligation to update any forward-looking statements contained in this release as a result of new information, future events, changes in expectations or otherwise.

Table of Contents

What is profiling?



Getting frequency and usage of your code

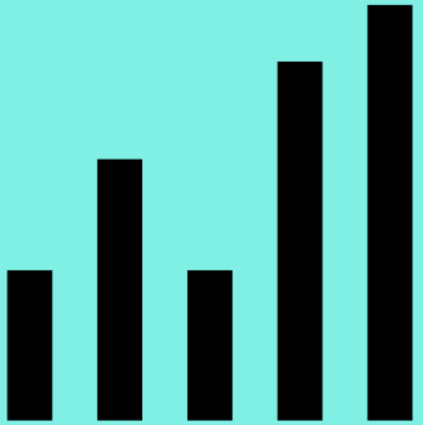
Two types of profiling

Deterministic

Run a scenario and meter all execution function by function

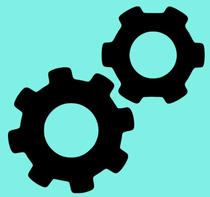
Statistical

Sample your program periodically to see what it is doing



Deterministic profiling

Register time before ***`function()`*** is called



Call ***`function()`***

Register time after ***`function()`*** is called

cProfile shortcomings

Only wall time

0–∞% overhead

Granularity to the function

Custom data format

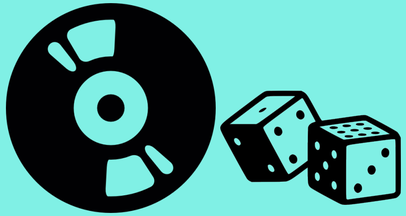
Production system

=

statistical profiling

Statistical profiling

Wake up



Register what the program’s doing right now (maybe)

Go back to sleep

Sampling strengths

Wall & CPU time

Low overhead

Granularity to the line

Can report raised exceptions

State of the art

CPU & Wall time

1

Sleep 10 ms.

`time.sleep()`

2

Get threads stacks.

`sys._current_frames()`

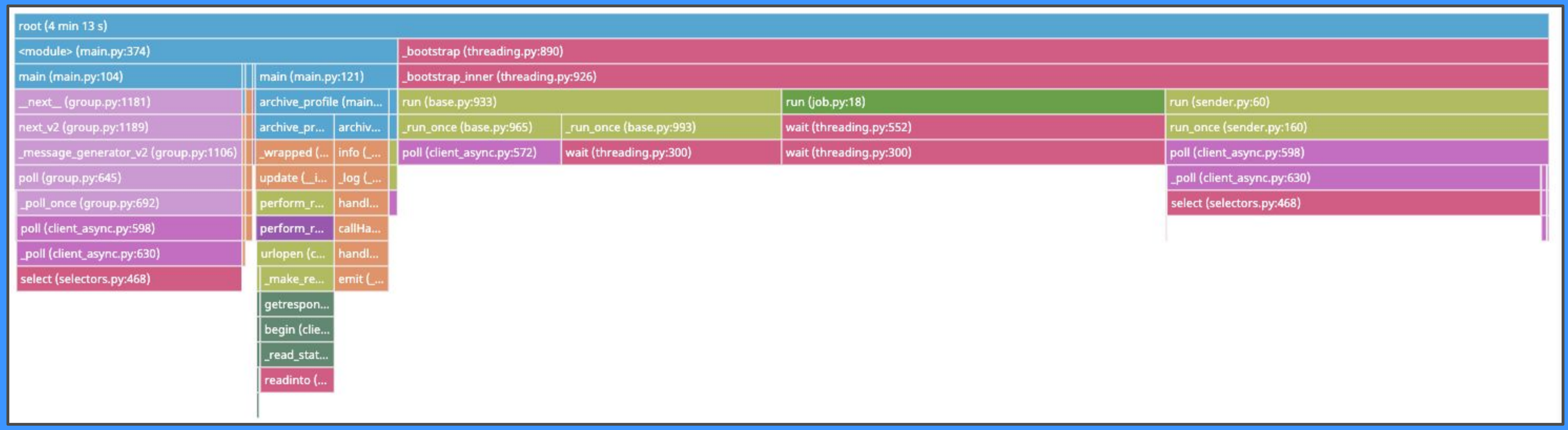
```
<Thread-1>
  a() myfile.py:123
    b() myfile.py:394
      c() mymodule.py:049
<Thread-2>
  d() myfile.py:123
    b() myfile.py:395
```

3

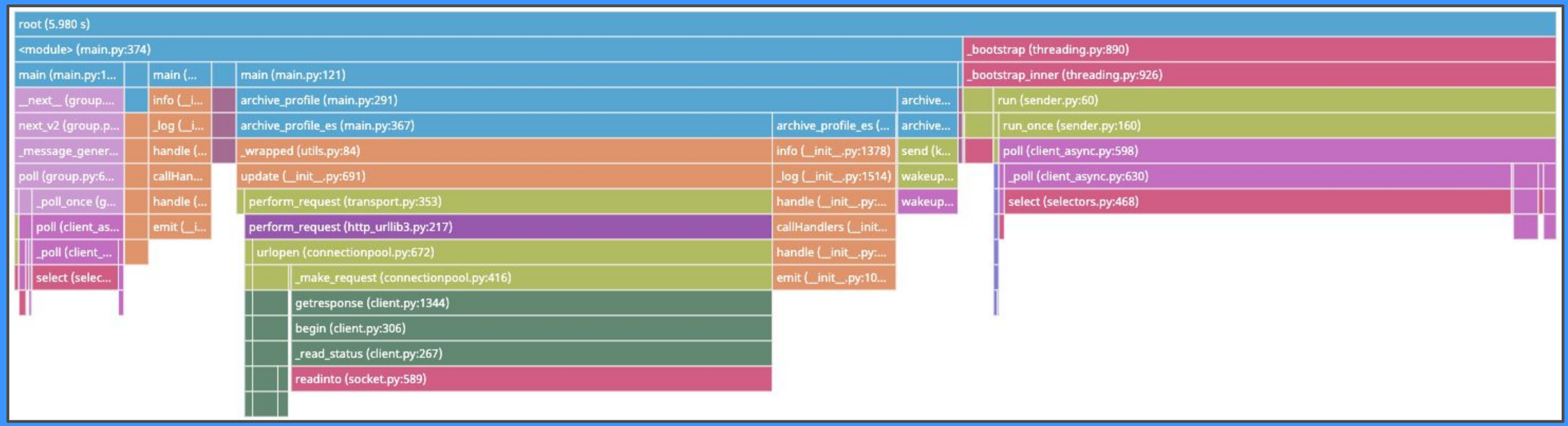
Get CPU time for each thread.

`time.pthread_getcpuclockid()`

Wall Time



CPU Time



High performance & precision

High Performance

Exception Profiling

Limit resources usage

~1% CPU usage @ 100 Hz
10 threads × 30 functions

No C profiling

(yet)

Memory

tracemalloc

1

```
time.sleep(0.01)
```

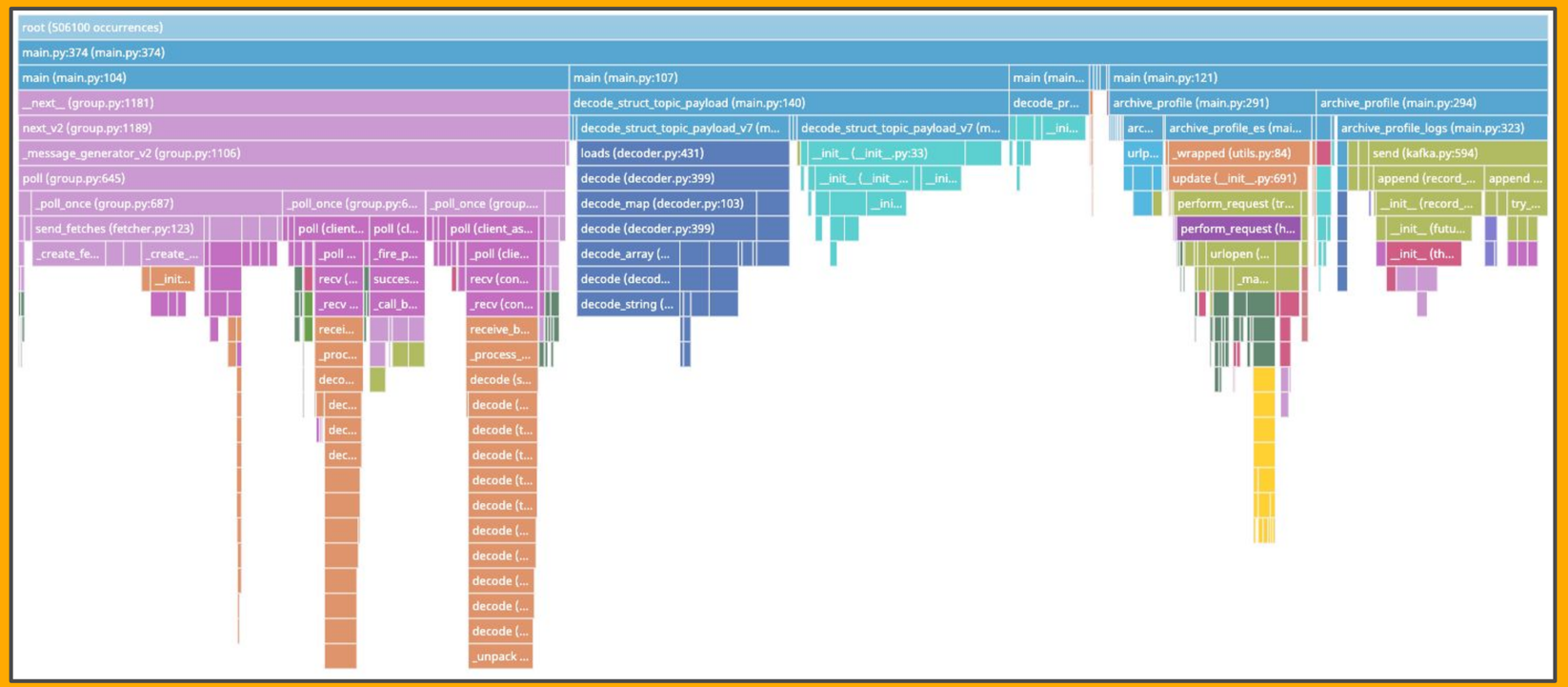
2

```
# 0 <= n <= 100
counter += n
```

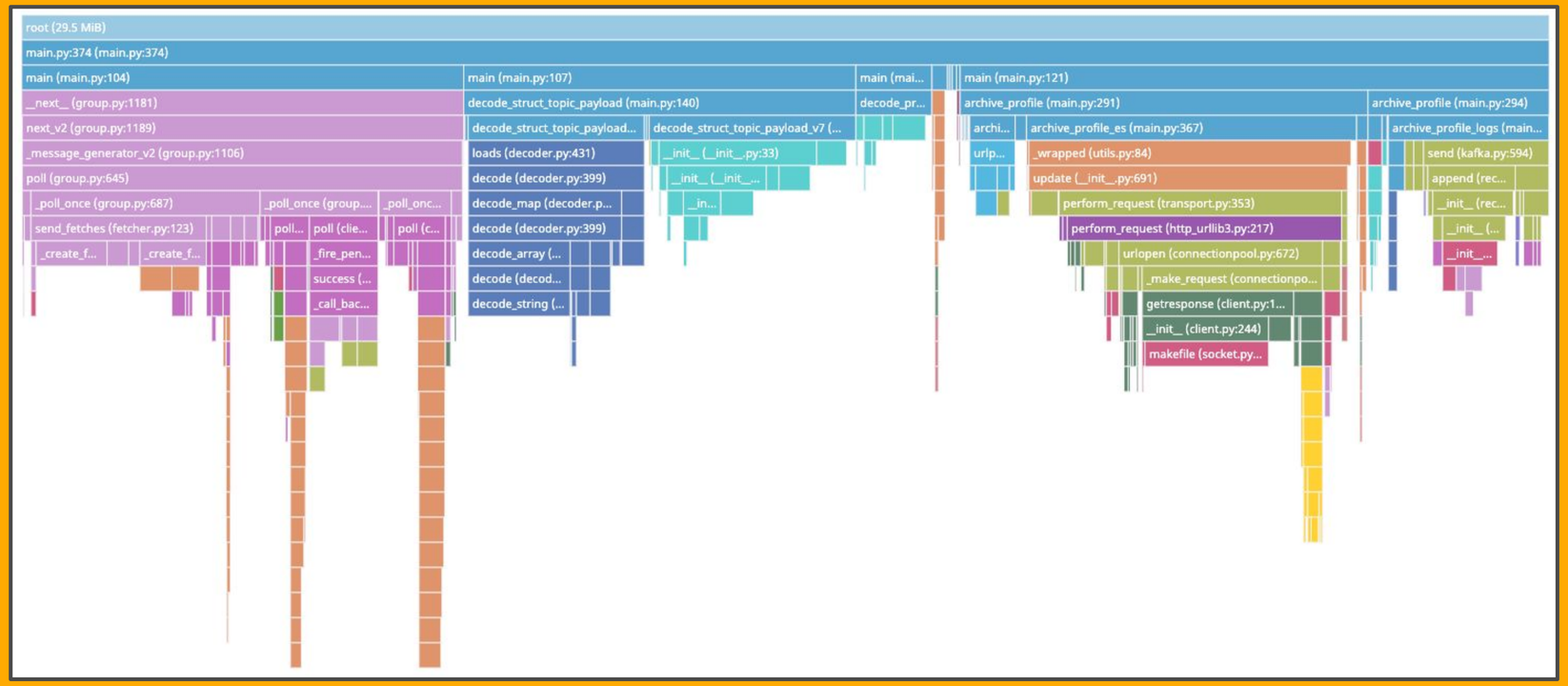
3

```
if counter >= 100:
    counter -= 100
    tracemalloc.start()
else:
    tracemalloc.stop()
```

Allocations Count



Allocations Size



Tracemalloc limitations

Overhead

No thread information

Only file names and line numbers

Threading

1

Intercept & wrap `threading.Lock` instances

2

Determine if `acquire()` is to be intercepted

3

Register `acquire()` and `release()` timings and stack traces

Lock Acquire Wait Time Total

Exporting and using the data

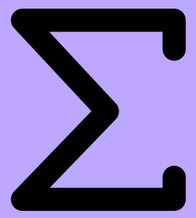
There is no real standard.

- cProfile → custom format
- Callgrind supports in some tools
- Many Python profilers focus on their output
- pprof to the rescue

The pprof format

Based on protobuf

Fast + schema

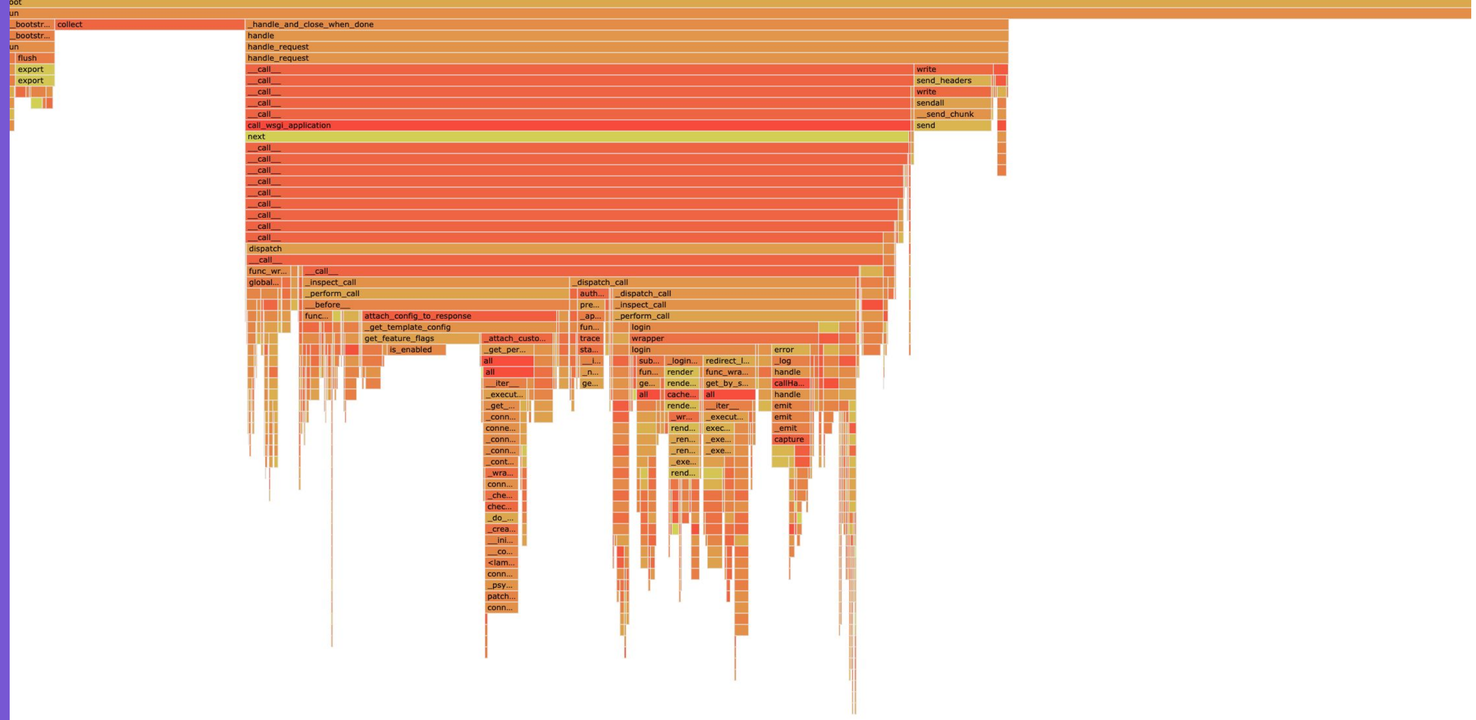


Aggregates data

Can compute KPM

Space efficient

String pool +
gzip
~20 KB / minute / process

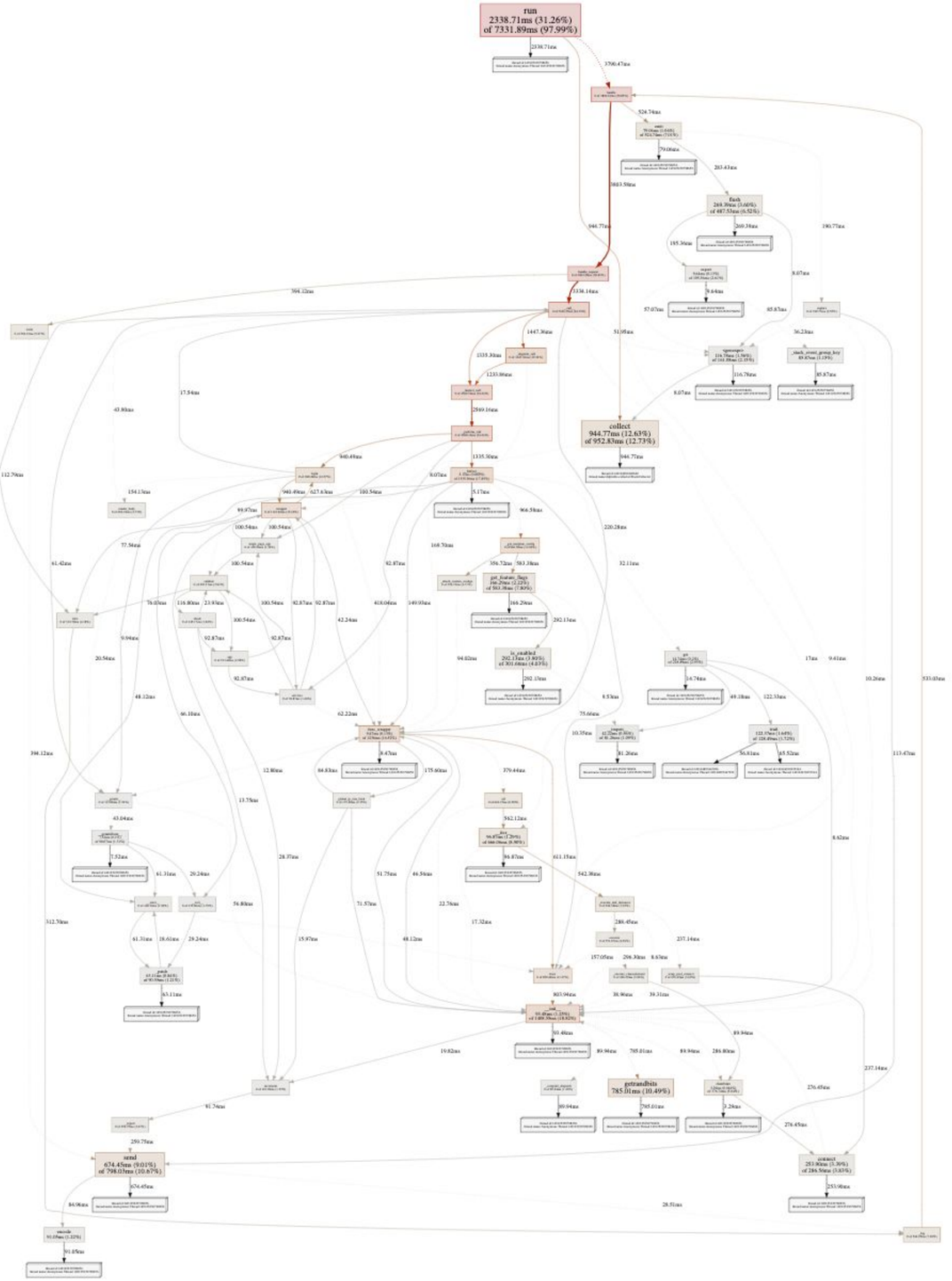


Visualization tool

– Also named pprof 🤦

– Fancy visualizations

– Written in Go 



Open-source library upcoming

(Apache / BSD)

`https://github.com/datadog/dd-trace-py`

DATADOG

pprof

Thank you

Follow me if you want to know when this gets released!

@juldanjou

Questions, feedback?

jd@datadoghq.com