Production-time Profiling for Python

Julien Danjou

FOSDEM — 1st February 2020



Julien Danjou Staff Engineer @ Datadog

🄰 @juldanjou https://julien.danjou.info



Forward-looking Statements

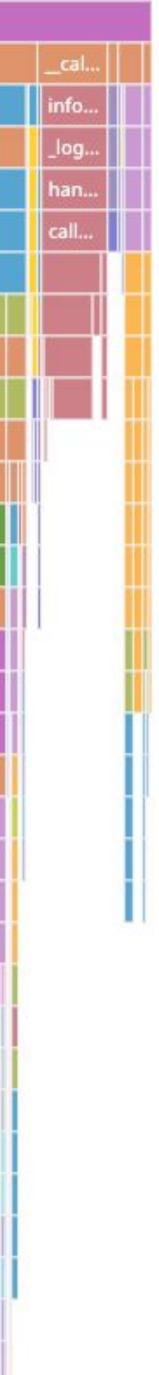
This presentation may include certain "forward-looking statements" within the meaning of Section 27A of the Securities Act of 1933, as amended, and Section 21E of the Securities Exchange Act of 1934, as amended, including statements concerning our product offerings. These forward-looking statements reflect 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?
CPU & Wall time profiling
Memory profiling
Threading profiling
Exporting & using the data

What is profiling?

dispatch (wsgiapp.py:313)												
_call(bcall(base.py:125)												
func_w	call (core.py:210)	_call_ (core.py:214)										
	_inspect_call (core.py:107)	_dispatch_call	_dispatch	_ C	_disp	patch_ca	ll (base.py:374)					
	_perform_call (core.py:57)	APIify (decorators.py:142)	authn		_dispatch_call (core.py:164)							
	before (base.py:321)	_dispatch_call (common.py:59)	pre_au		_inspect_call (core.py:107)							
		321) patch_call (base.py:374)			_perform_call (core.py:57)							
		patch_call (core.py:164)			login							
	(15.16%, 2.061 s)	pect_call (core.py:107)							rators.py:439)			
	_get_outagesget_permiss	_perform_call (core.py:57)					error (initpy:11	And in case of the local division of	login (login.py:693)	login (login.py		
	func_wrap all (permissi	get_manage_monitors					_log (initpy:1286)			_login_get (
	trace (trace all (query.py:	wrapper (decorators.py:439)					handle (initpy:1		func_wrappe func	render (te		
	start_spaniter(qu	get_manage_monitors (old					callHandlers (init	get_by	trace (tracer get	render_m		
1.000	init(spexecut	wrapped (context.py:53)					handle (initpy:7	1	start_span (t all (cached_te		
	_new_id (sget_bi	get_monitor_list (mana					emit (sentry.py:165)	_iter	init (spanit	render_t		
	getrandbitconnec	get_classification (man					emit (logging.py:66)		_new_id (spaex	_wrap_re		
	connect	get_pa parse_que					_emit (logging.py		getrandbits (exe	render_u		
5 C -	_connec	group parse_m					cap capture (b			_render (
	_connec	<pre><dictco from_q<="" pre=""></dictco></pre>					buil send (_render		
		parse (ų.		encod			_exec_te		
	_wrap_p connect											
	checko											
	_do_get											
	create											
			_								1 II	
					1							
	Getting frequency and											
											ίŦ	
											l III	
	sage of your co											
					1	1						

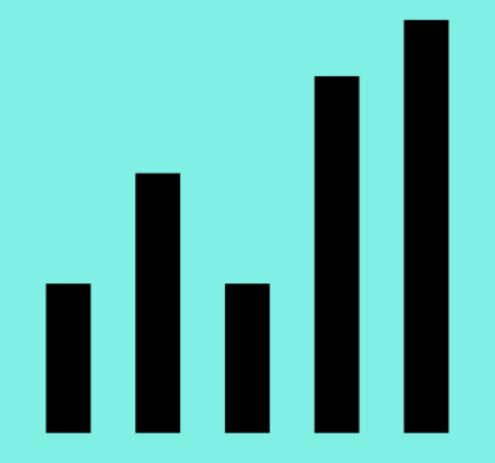


Two types of profiling

Deterministic

Run a scenario and meter all execution function by function

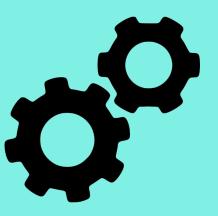
Statistical Sample your program periodically to see what it is doing



Deterministic profiling



Register time before function() is called



Call function()



Register time after function() is called



cProfile shortcomings

Only wall time

Granularity to the function

0–∞% overhead

Custom data format

Production system

statistical profiling

Statistical profiling





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



Go back to sleep

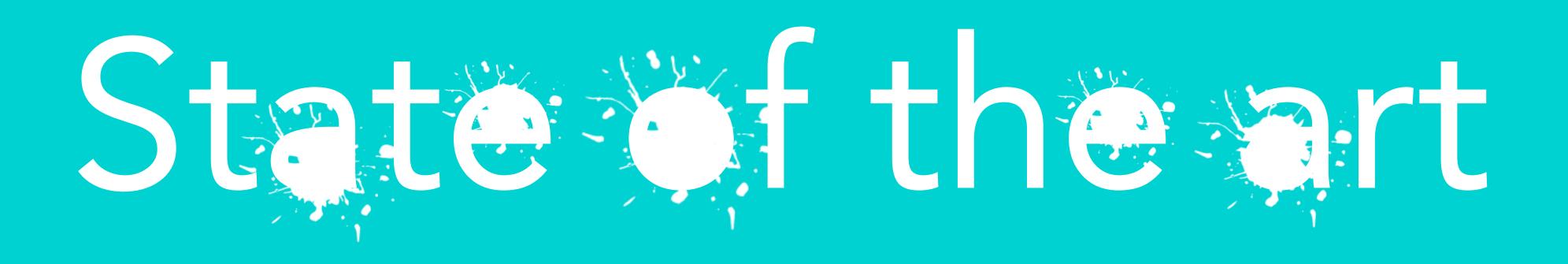
Sampling strengths

Wall & CPU time

Granularity to the line

Low overhead

Can report raised exceptions









Sleep 10 ms. time.sleep()

Get threads stacks.

sys. current frames()

<Thread-1> <Thread-2>

Get CPU time for each thread.

time.pthread getcpuclockid()

a() myfile.py:123 b() myfile.py:394 c() mymodule.py:049 d() myfile.py:123 b() myfile.py:395

Wall Time

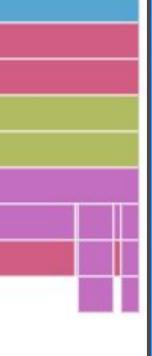
root (4 min 13 s)									
<module> (main.py:374)</module>	1421	_bootstrap (threading.py:890)							
main (main.py:104)	main (main.py:121)	_bootstrap_inner (threading.py:926)							
next (group.py:1181)	archive_profile (main	run (base.py:933)		run (job.py:18)	run (sender.py:60)				
next_v2 (group.py:1189)	archive_pr archiv	_run_once (base.py:965)	_run_once (base.py:993)	wait (threading.py:552)	run_once (sender.py:160)				
_message_generator_v2 (group.py:1106)	_wrapped (info (poll (client_async.py:572)	oll (client_async.py:572) wait (threading.py:300) wait (threading.py:300)		poll (client_async.py:598)				
poll (group.py:645)	update (ilog (_poll (client_async.py:630)				
_poll_once (group.py:692)	perform_r handl				select (selectors.py:468)				
poll (client_async.py:598)	perform_r callHa								
_poll (client_async.py:630)	urlopen (c handl								
select (selectors.py:468)	_make_re emit (
	getrespon								
	begin (clie								
	_read_stat								
	readinto (

CPU Time

root (5.980 s)								
<module> (main.py:374)</module>								
main (main.py:1	main (main (main.py:121)						
next (group, info (_i		archive_profile (main.py:291)						
next_v2 (group.plog (_i		archive_profile_es (main.py:367)						
_message_gener	handle (_wrapped (utils.py:84)						
poll (group.py:6 callHan		update (initpy:691)						
_poll_once (g	handle (perform_request (transport.py:353)						
poll (client_as	emit (_i	perform_request (http_urllib3.py:217)						
_poll (client		urlopen (connectionpool.py:672)						
select (selec		_make_request (connectionpool.py:416)						
		getresponse (client.py:1344)						
		begin (client.py:306)						
		_read_status (client.py:267)						
		readinto (socket.py:589)						

		;						
				_bootstrap (threading.py:890)				
			_boo	_bootstrap_inner (threading.py:926)				
		archive		run (sender.py:60)				
	archive_profile_es (archive		run_once (sender.py:160)				
	info (initpy:1378)	send (k		poll (client_async.py:598)				
	_log (initpy:1514)	wakeup		_poll (client_async.py:630)				
	handle (initpy:	wakeup		select (selectors.py:468)				
	callHandlers (init							
	handle (initpy:							
	emit (initpy:10							





High performance & precision



High Performance



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

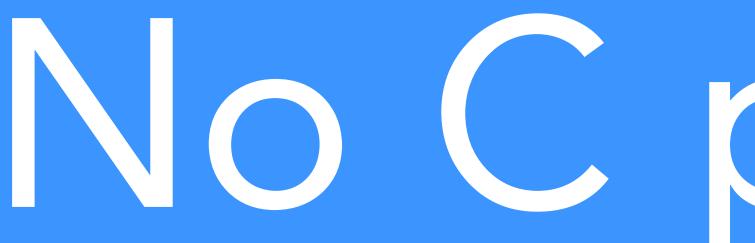


Exception Profiling



Limit resources usage





No C profiling

Memory



tracemaloc

time.sleep(0.01)

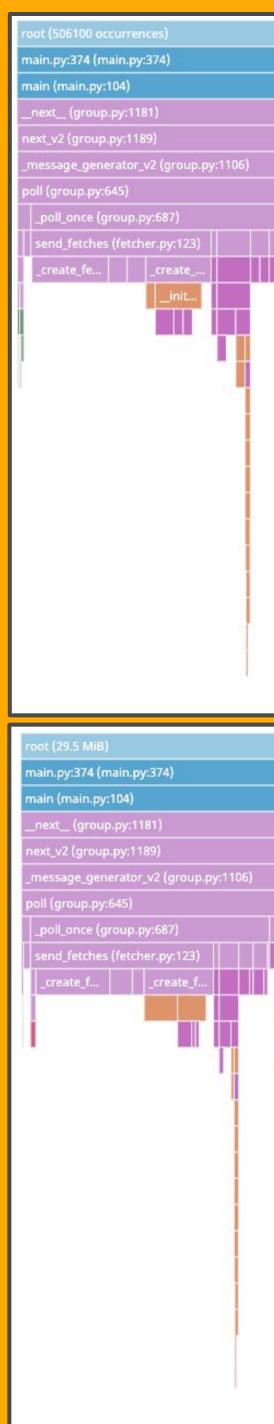
counter += n

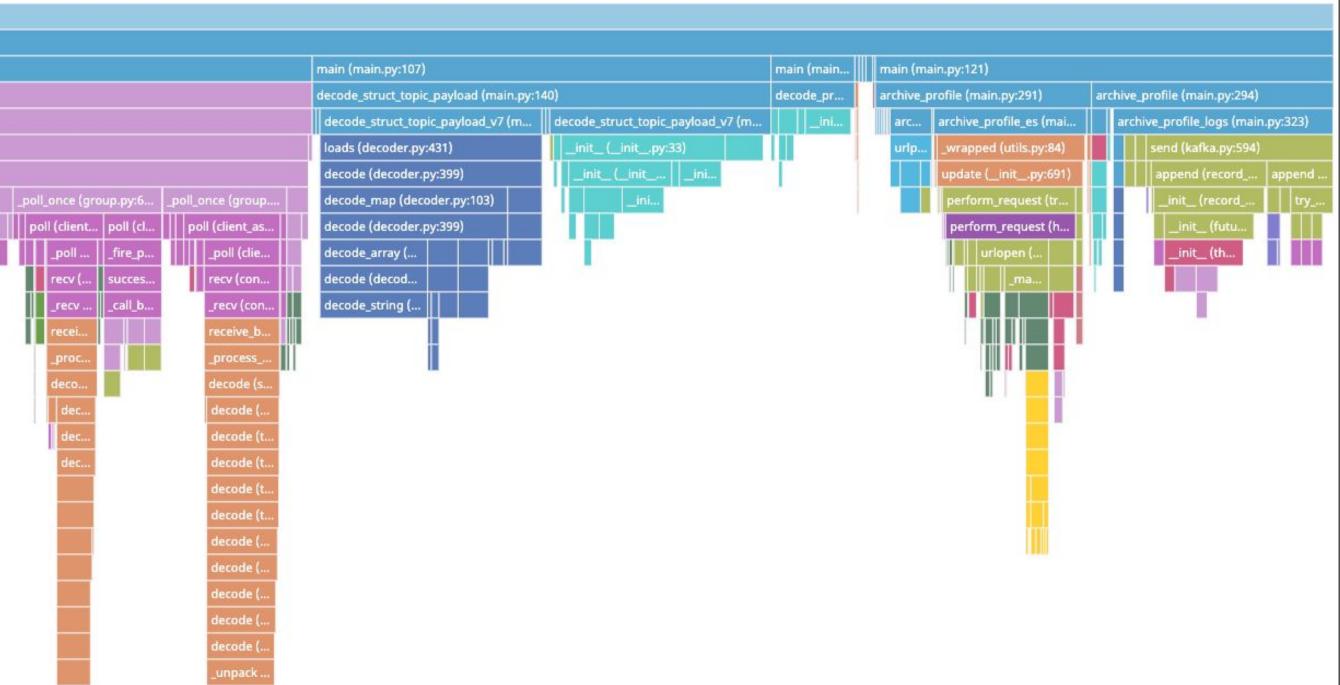
0 <= n <= 100

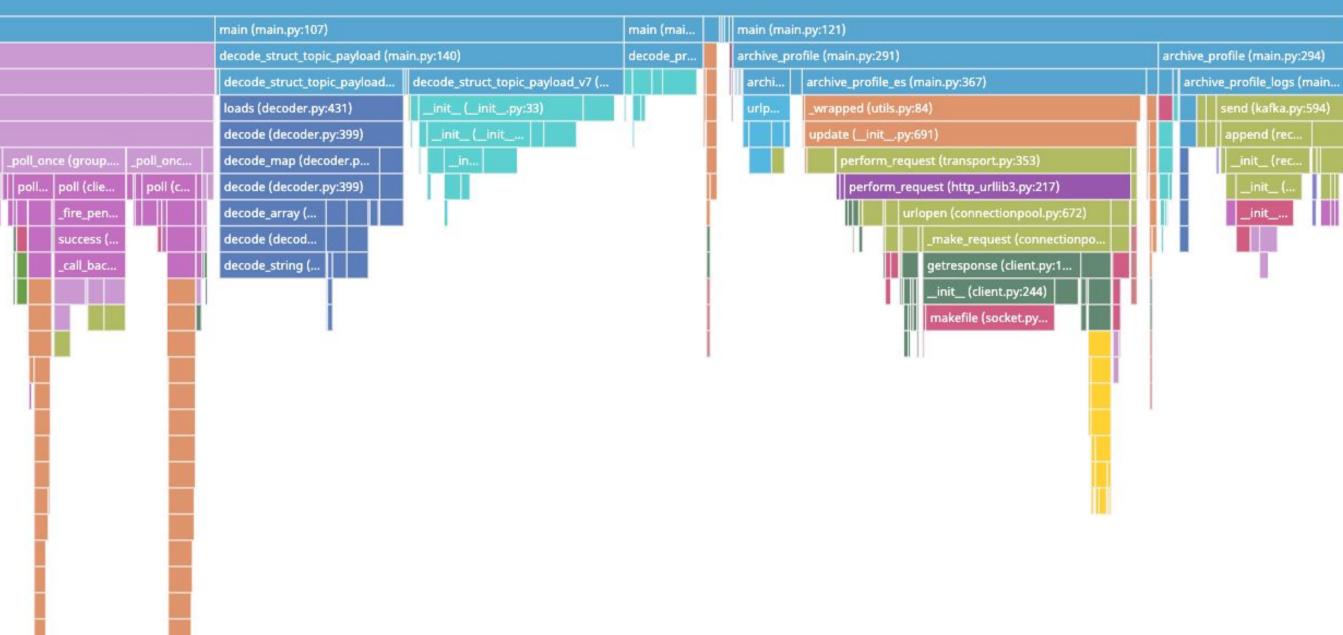
if counter >= 100: counter -= 100 tracemalloc.start() else: tracemalloc.stop()

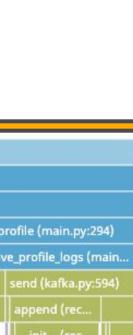
Allocations Count

Allocations Size









Tracemalloc limitations



Overhead





No thread information

Only file names and line numbers

Threading





Intercept & wrap threading.Lock instances

Determine if

acquire() is to be intercepted



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

Lock Acquire Wait Time Total

0%							
root (11.749 ms)							
<module> (main.py:374)</module>							
main (main.py:121)							
archive_profile (main.py:291)							
archive_profile_es (main.py:367)	archive_profile_es (main.py:367)						
_wrapped (utils.py:84)							
update (initpy:691)							
perform_request (transport.py:353)							
perform_request (http_urllib3.py:217)	perform_request (http_urllib3.py:217)						
urlopen (connectionpool.py:654) urlopen (connectionpool.py:740)							
_get_conn (connectionpool.py:251) _put_conn (connectionpool.py:291)							
get (queue.py:181) put (queue.py:151)							
notify (threading.py:345) notify (threading.py:345)							
_is_owned (threading.py:258)is_owned (threading.py:258)							

	100% (11.749 ms)	<		Thre		Total Lo
	Þ					
	_bootstrap (threading.py:890)			MainTh	read	
	_bootstrap_inner (threading.py:926)		<	kafka-py	/thon-	produc.
	run (sender.py:60)		<	prof-arc	hiver	heartbe
	run_once (sender.py:160)			Thread-	1	
	poll (client_async.py:602)					
	_fire_pending_completed_requests (client_async.py:734)					
	success (future.py:36)					
	_call_backs (future.py:79)					
	_handle_produce_response (sender.py:206)					
	_complete_batch (sender.py:243)					
	done (record_accumulator.py:80)					
	success (future.py:18)					
	set (threading.py:522)					
	notify_all (threading.py:365)					
	notify (threading.py:345)					
	_is_owned (threading.py:258)					

Lock	Acqui	٠
	8 ms	
c	3 ms 📕	
e	0 ns	
	0 ns 📃	

Exporting and using the data

There is no real standard.



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

The pprof format

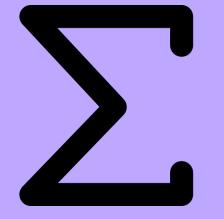


Based on protobuf



Fast + schema



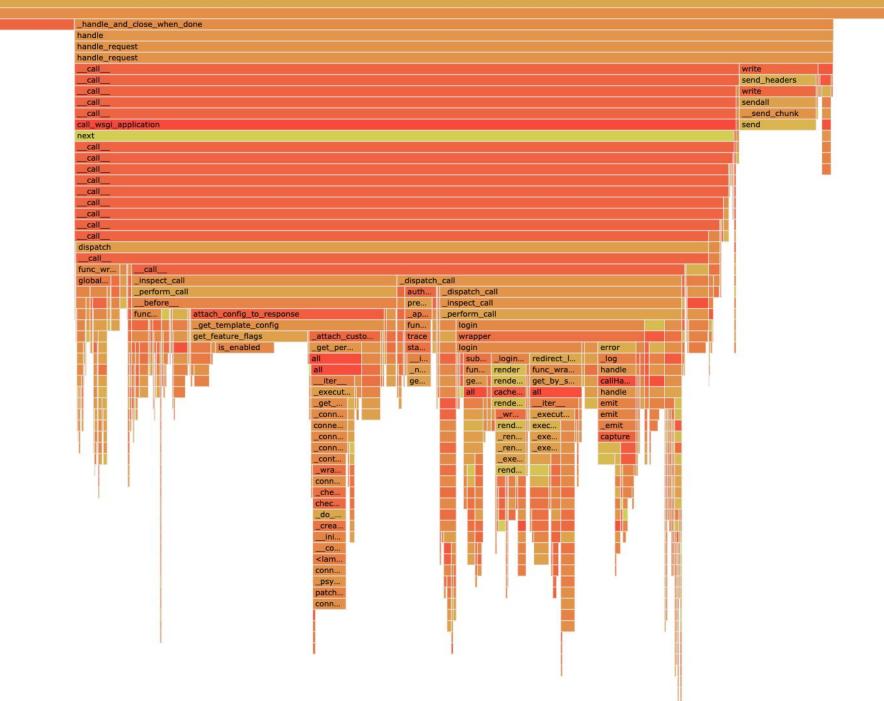


Aggregates data

Space efficient

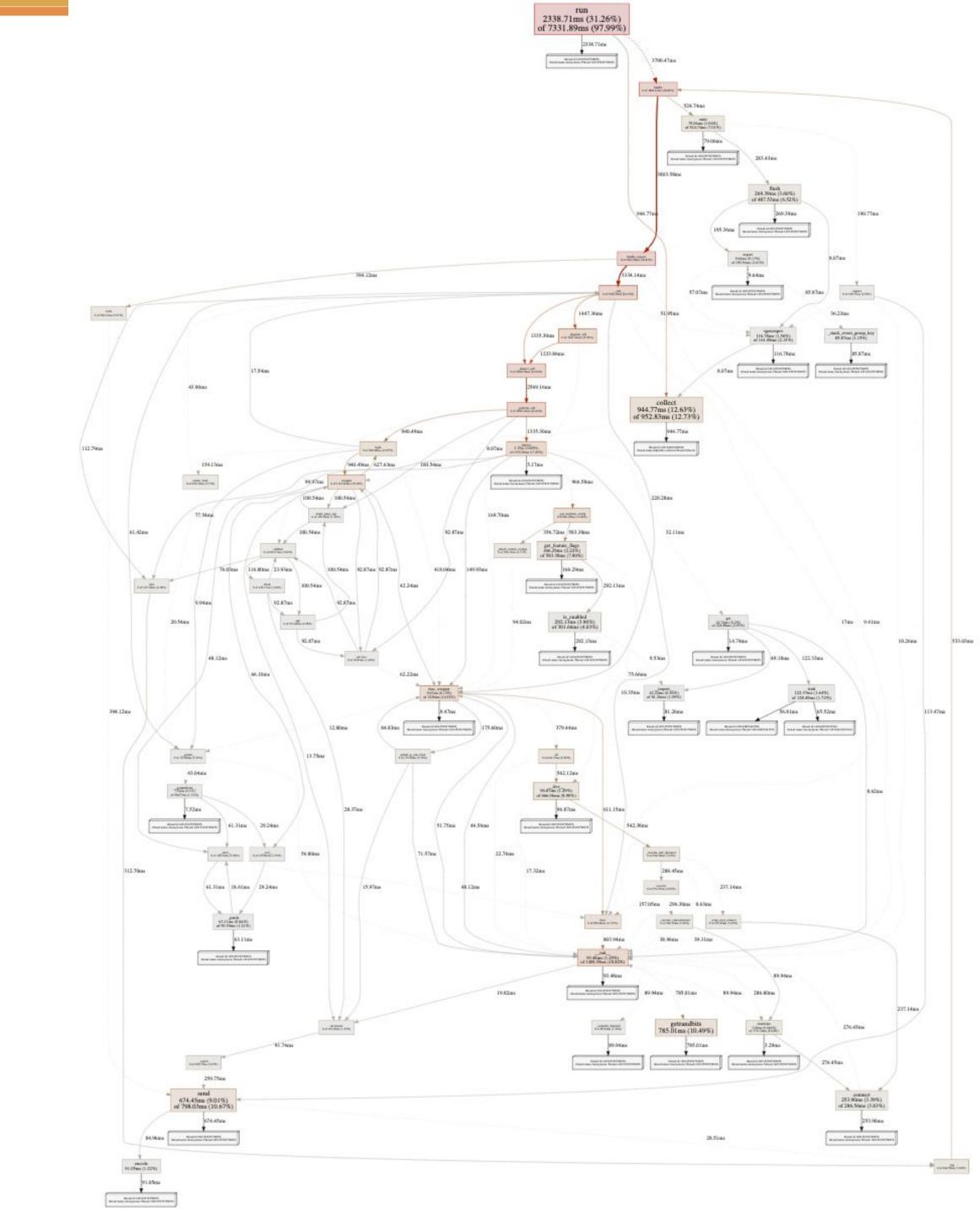
Can compute KPM

String pool + gzip ~20 KB / minute / process



Visualization tool

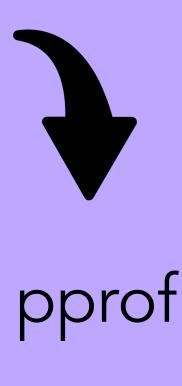
- Also named pprof 😰
- Fancy visualizations
- Written in Go



Open-source library upcoming (Apache / BSD)

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





Questions, feedback?

jd@datadoghq.com

Thank you

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

Øjuldanjou







