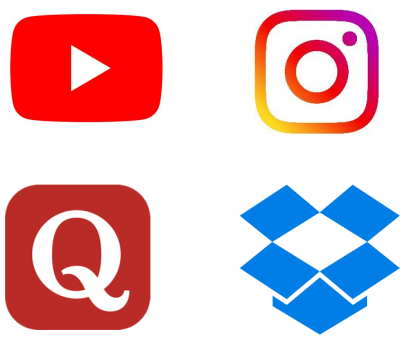
PYLIVE: On-the-Fly Code Change for Python-based Online Services

Haochen Huang*, Chengcheng Xiang*, Li Zhong, Yuanyuan Zhou

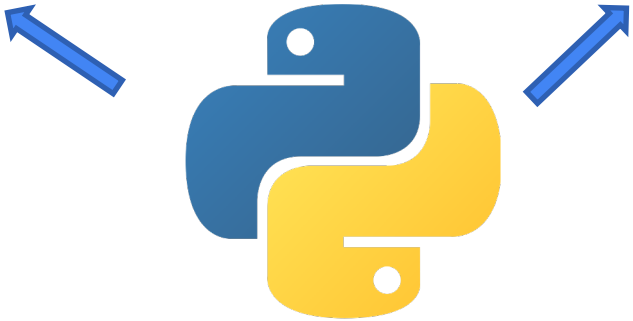
UC San Diego

* Co-first authors.

Python is widely adopted in ***online services***.



Commercial companies



Web framework

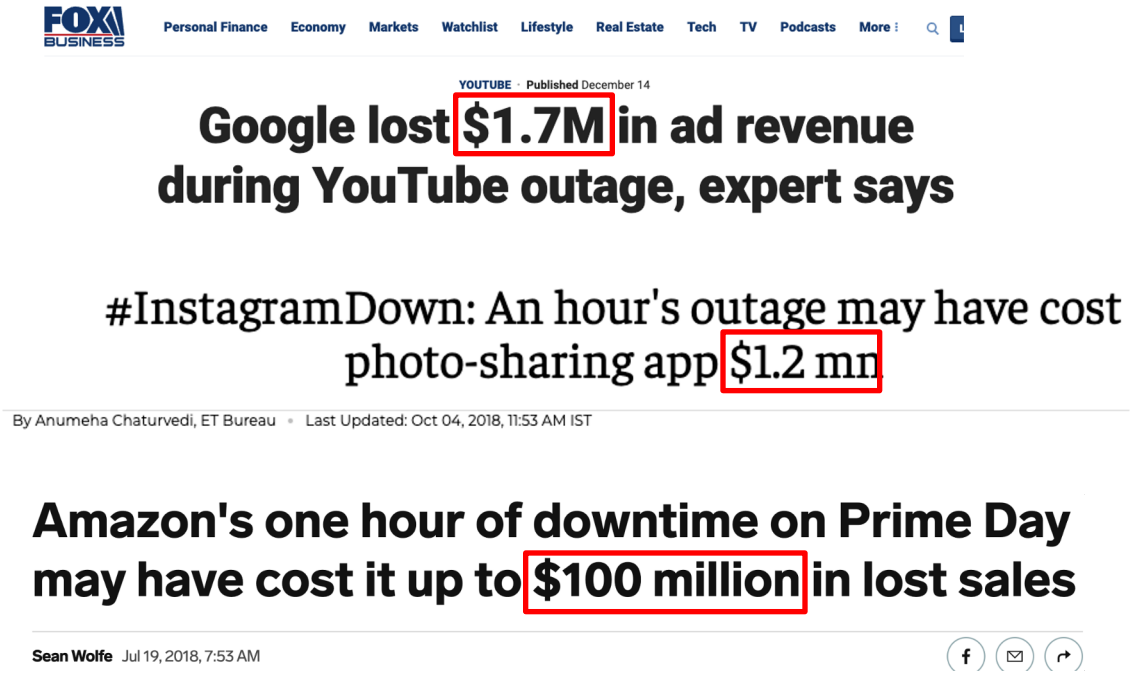
Web server

E-commerce

Message queue

Python-based frameworks

Online services have high requirements on *availability*



FOX BUSINESS Personal Finance Economy Markets Watchlist Lifestyle Real Estate Tech TV Podcasts More

YOUTUBE · Published December 14

Google lost $1.7M in ad revenue during YouTube outage, expert says

#InstagramDown: An hour's outage may have cost photo-sharing app $1.2 mn

By Anumeha Chaturvedi, ET Bureau • Last Updated: Oct 04, 2018, 11:53 AM IST

Amazon's one hour of downtime on Prime Day may have cost it up to $100 million in lost sales

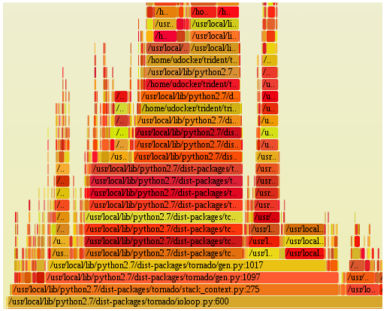
Sean Wolfe Jul 19, 2018, 7:53 AM

Requires >

99.99% of uptime!

Code changes are necessary for online services:

On-the-fly logging

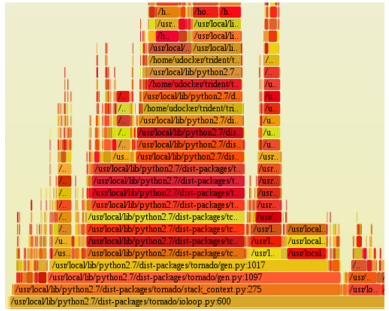


On-the-fly profiling

Urgent dynamic patching

Code changes are necessary for online services:

On-the-fly logging



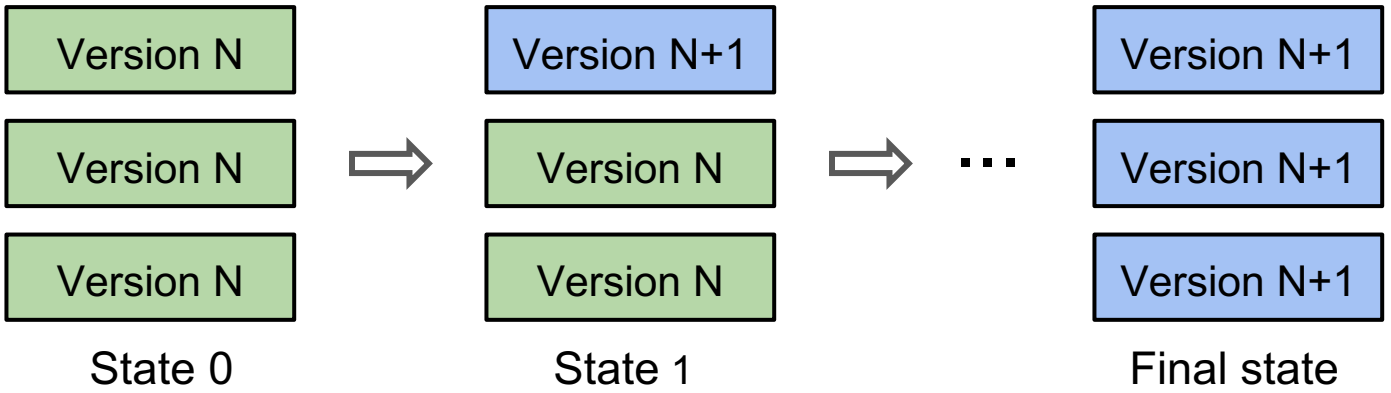
On-the-fly profiling

Urgent dynamic patching

\+ ***High availability***

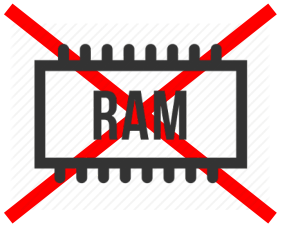
PYLIVE: dynamically change Python programs
in production *without restarting them*

A common system update practice — ***Rollout Deployment***

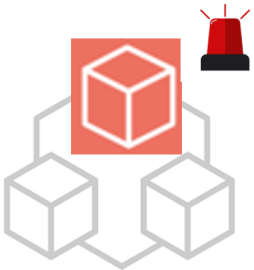


Rollout is not the best choice for ***dynamic logging and profiling***

Rollout requires restart & *loses states*.

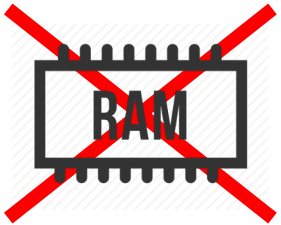


Rollout is *heavyweight* & *an overkill.*

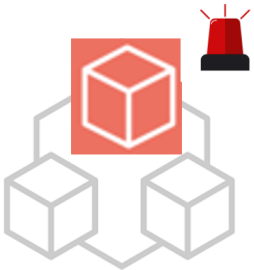


Rollout is not the best choice for ***dynamic logging and profiling***

Rollout requires restart & *loses states*.



Rollout is *heavyweight* & *an overkill.*



PYLIVE requires *no restart.*

PYLIVE is *dynamic & flexible.*

PYLIVE complements ***Rollout deployment***

Python's ***language features*** ease the code change

- Build on **standard** Python interpreter:

Meta-object Protocol

Interfaces to dynamically modify ***metadata***

```
A.__code__ = D.__code__
```
(function body/interface)

```
C.A = D
```
(class attributes)

Python's ***language features*** ease the code change

- Build on **standard** Python interpreter:

Meta-object Protocol

Interfaces to dynamically modify ***metadata***

```
A.__code__ = D.__code__
```
(function body/interface)

```
C.A = D
```
(class attributes)

Dynamic Typing

Allows changing variable ***types***

```
A = "1"
A = 1
```

PYLIVE's Interfaces

Instrument

Instrument ***log/profiling*** code to ***specified locations***

```
instrument(scope,
    jointpoint_callback,
    time)
```

PYLIVE's Interfaces

Instrument

Instrument ***log/profiling*** code to ***specified locations***

```
instrument(scope,
    jointpoint_callback,
    time)
```

An example of ***on-the-fly profiling*** using PYLIVE --
diagnose a critical performance issue in e-commerce.

```
# instrument code to all functions of two classes
instrument(scope=['...Class_A.*',
                  '...Class_B.*'],
           jointpoint_callback={func_before: call_b,
                                func_end: call_a},
           time='24:00-2:00')
```

PYLIVE's Interfaces

Instrument

Instrument ***log/profiling*** code to ***specified locations***

```
instrument(scope,
    jointpoint_callback,
    time)
```

An example of ***on-the-fly profiling*** using PYLIVE --
diagnose a critical performance issue in e-commerce.

```
# instrument code to all functions of two classes
instrument(scope=['...Class_A.*',
                  '...Class_B.*'],
           jointpoint_callback={func_before: call_b,
                                func_end: call_a},
           time='24:00-2:00')
```

```
# profiling code to instrument
def call_b(start):
  start = time.time()
def call_a(start):
  logging.info(time.time()-start)
```

PYLIVE's Interfaces

Instrument

Instrument ***log/profiling*** code to ***specified locations***

```
instrument(scope,
    jointpoint_callback,
    time)
```

An example of ***on-the-fly profiling*** using PYLIVE --
diagnose a critical performance issue in e-commerce.

```
# instrument code to all functions of two classes
instrument(scope=['...Class_A.*',
                  '...Class_B.*'],
           jointpoint_callback={func_before: call_b,
                                func_end: call_a},
           time='24:00-2:00')
```

PYLIVE's Interfaces

Instrument

Instrument ***log/profiling*** code to ***specified locations***

```
instrument(scope,
    jointpoint_callback,
    time)
```

Redefine

Replace ***existing code*** with new ones

```
redefine(prepFunc,
    old_new_map,
    safepoint)
```

```
prepFunc:       from ... import ...

old_new_map:    {'old_func'        : new_func}
                {'class.new_field': field_init}

safepoint:      "FUNC_QUIESCENCE"
```

Three challenges with PYLIVE

Challenge 1: How to support dynamic changes for *function interface*, *function body* and *data structure*?

Challenge 2: How to identify *safe change points* to apply a change without causing *inconsistency* problems?

Challenge 3: How to update programs with *multi-threads* and *multi-processes*?

(Check paper for details)

Challenge 1: Support Dynamic Changes

Change function interface/body

- function interface

`A.__code__.co_varnames`
Dynamic typing

- function body

`A.__code__`
Bytecode rewriting (instrument)

- caller functions

Not necessary with interpreter's function look up mechanism.

Challenge 1: Support Dynamic Changes

Change data structure

- class attributes — Meta-object Protocol
- object attributes — Meta-object Protocol, Garbage Collection
- methods — Meta-object Protocol

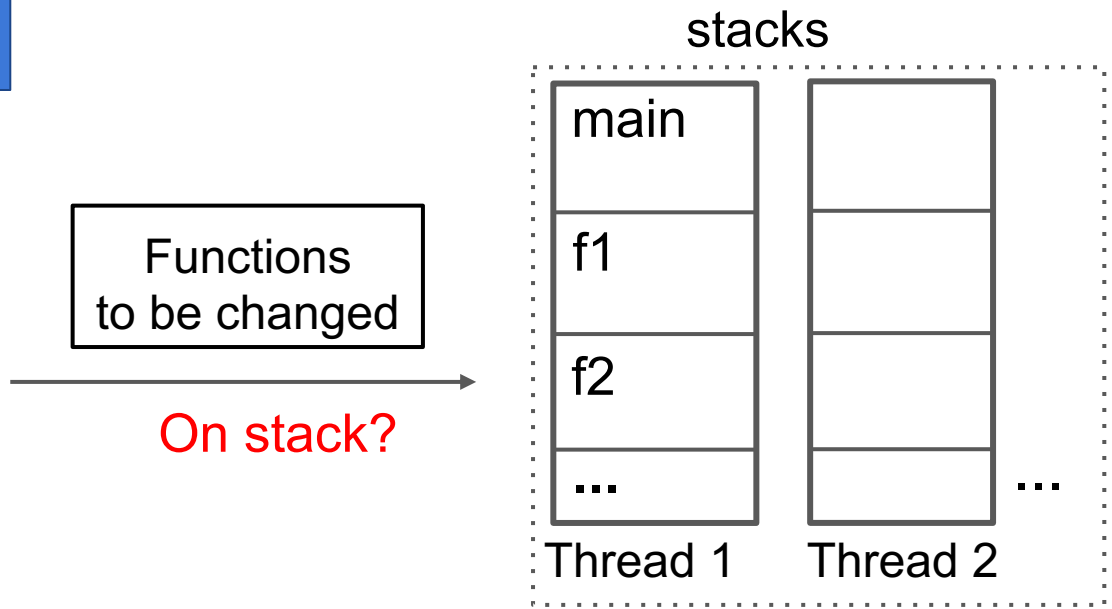
Challenge 2: Identify Safe Change Point

- Carefully choose a safe point to apply a change.

Challenge 2: Identify Safe Change Point

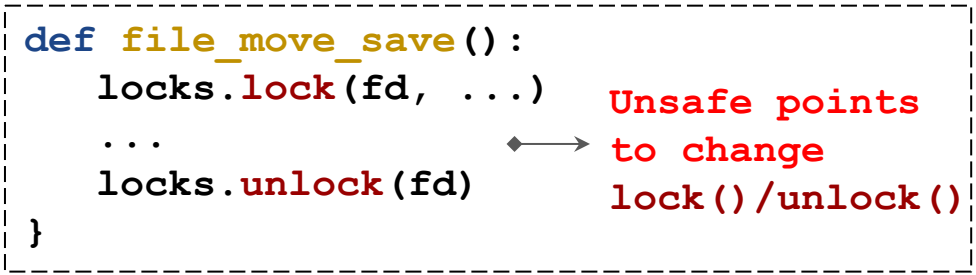
- Carefully choose a safe point to apply a change.

Type 1: ***Quiescence*** of the changed functions



Challenge 2: Identify Safe Change Point

- Carefully choose a safe execution point to apply a change.



An example of unsafe change points for a patch from Django

Challenge 2: Identify Safe Change Point

- Carefully choose a safe execution point to apply a change.

Type 2: ***Consistent*** state check

```
def file_move_save():
    locks.lock(fd, ...)
    ...
    locks.unlock(fd)
}
```

Unsafe points to change lock()/unlock()

An example of unsafe change points for a patch from Django

```
def state_check_func():
  for fd in all_fds():
    if locks.check_lock(fd) !=
locks.UNLOCK:
      return False
  return True
```

An example of state check function

Evaluation of PYLIVE

Application	Category	Logging	Profiling	Patching
Django	Web framework	1	0	2
Gunicorn	Web server	0	0	1
Oscar	E-commerce	1	2	1
Odoo	E-commerce	1	1	2
Shuup	E-commerce	1	0	1
Pretix	E-commerce	1	0	1
Saleor	E-commerce	1	1	2
Total:		6	4	10

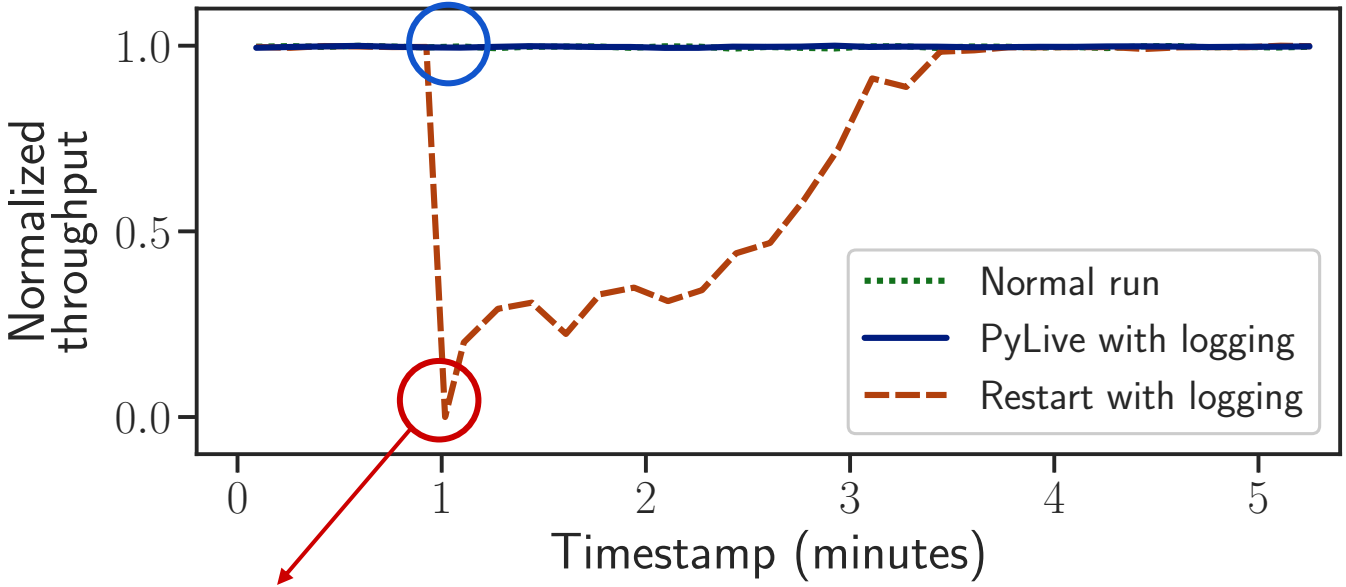
20 real-world cases evaluated in our experiments.

Evaluation of PYLIVE

- ***Performance benefit*** of PYLIVE to apply code changes.
  - Throughput as the performance metric.
  - Compare it with restarting services.
  - For profiling, also compare PYLIVE with cProfile.

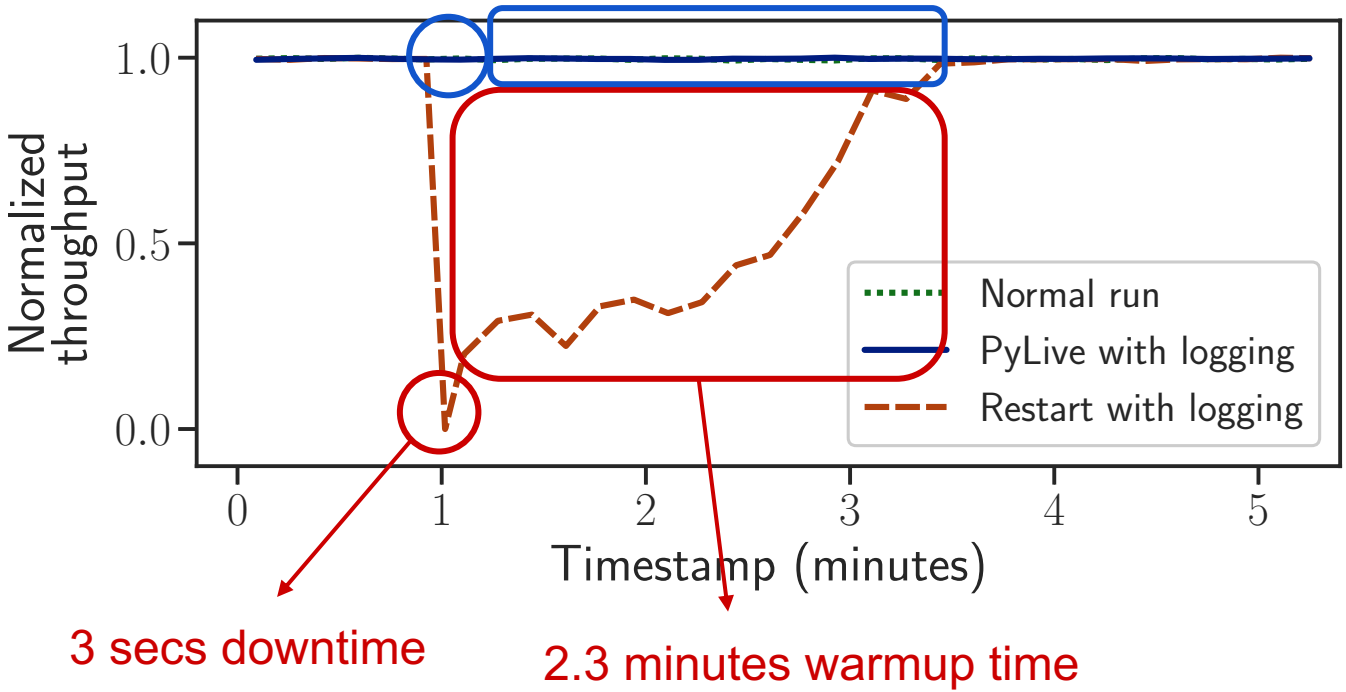
Evaluation of PYLIVE

- ***Performance benefit*** of PYLIVE when ***Logging***



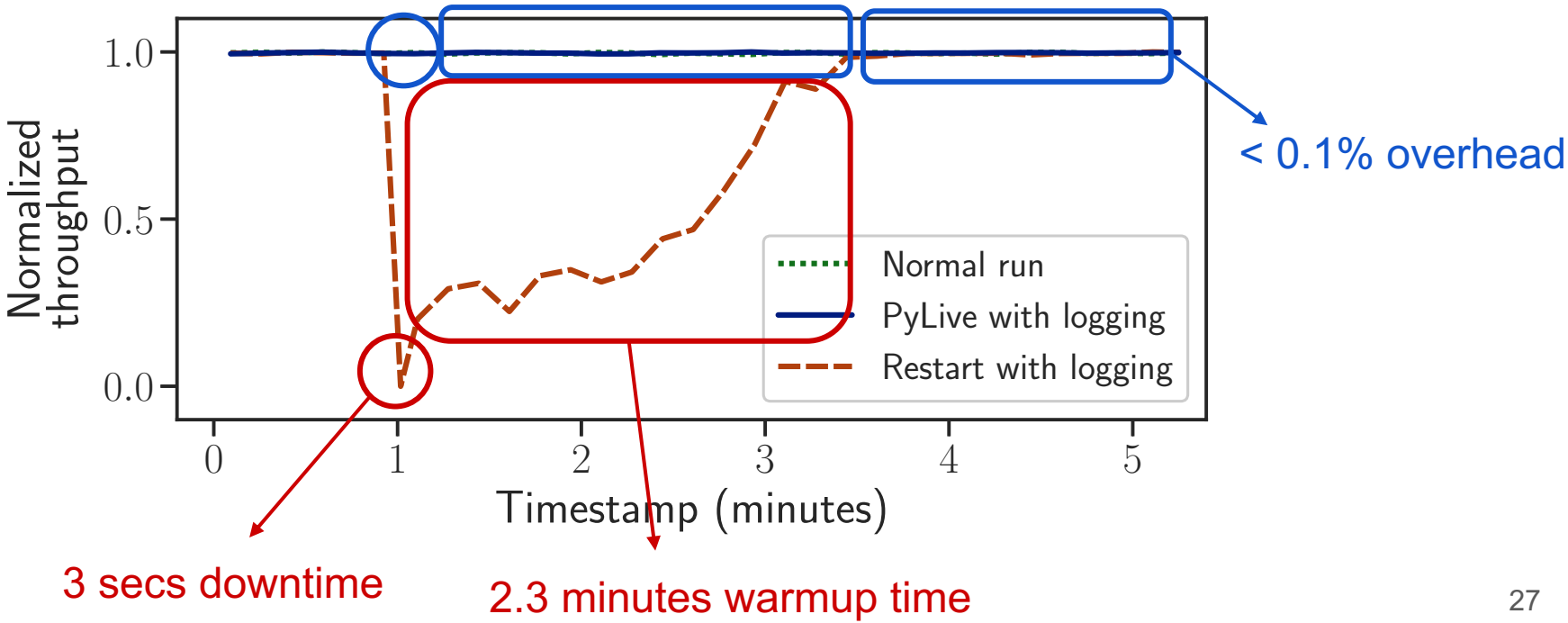
Evaluation of PYLIVE

- ***Performance benefit*** of PYLIVE when ***Logging***



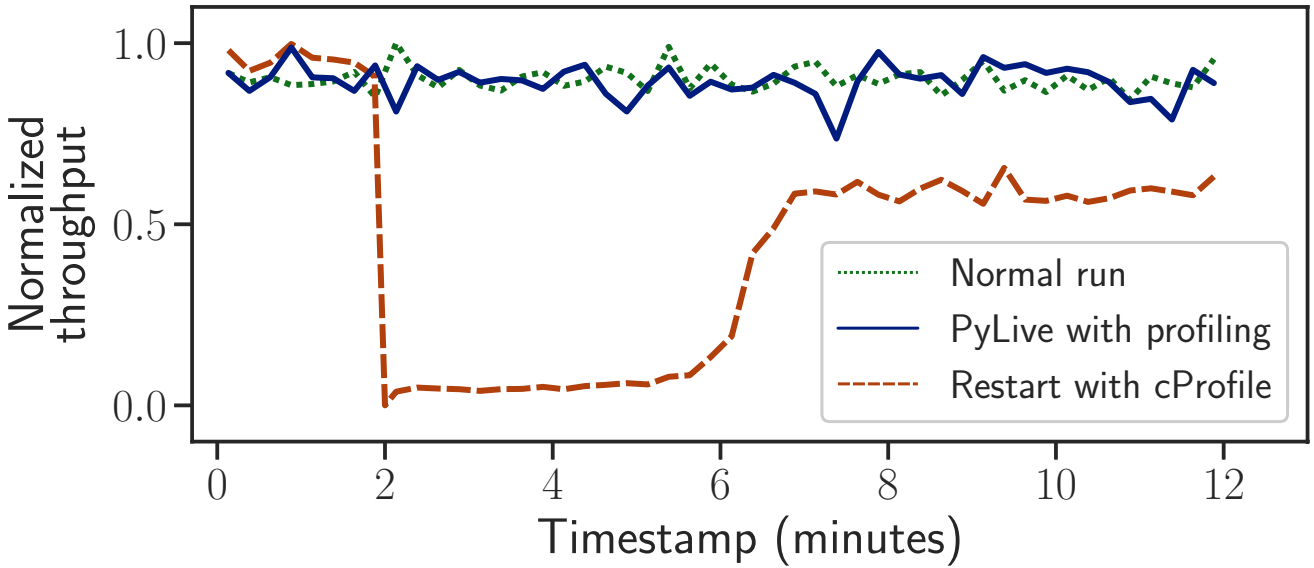
Evaluation of PYLIVE

- ***Performance benefit*** of PYLIVE when ***Logging***



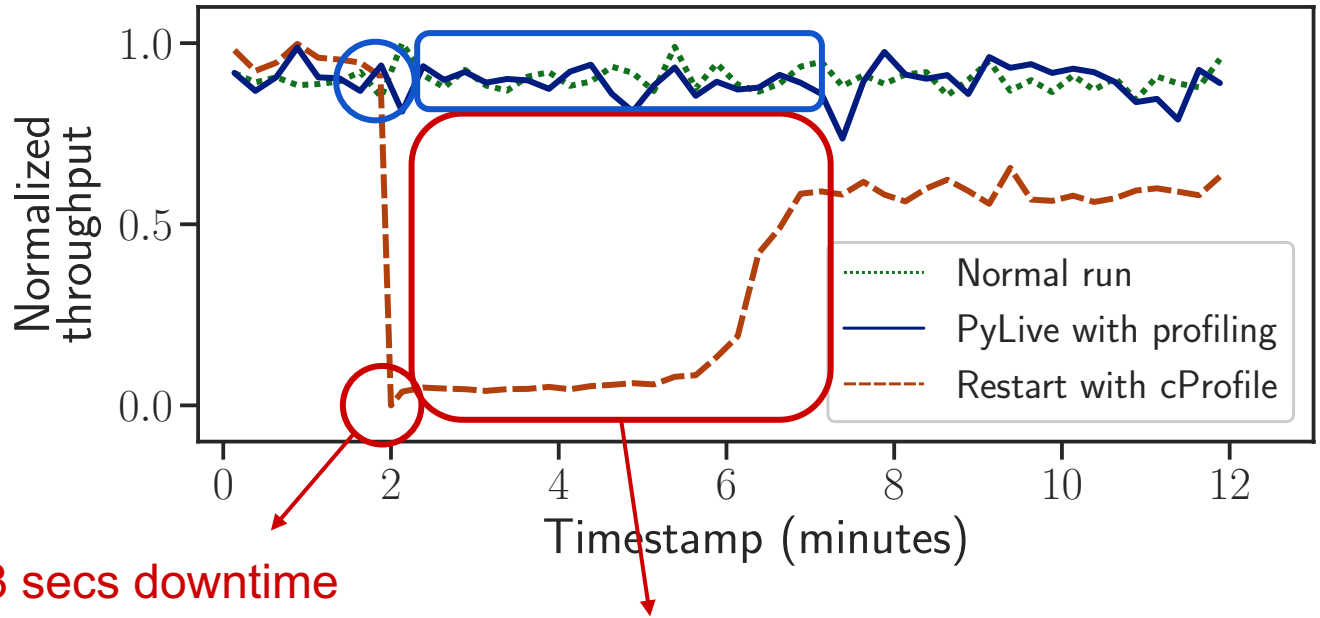
Evaluation of PYLIVE

- ***Performance benefit*** of PYLIVE when ***Profiling***



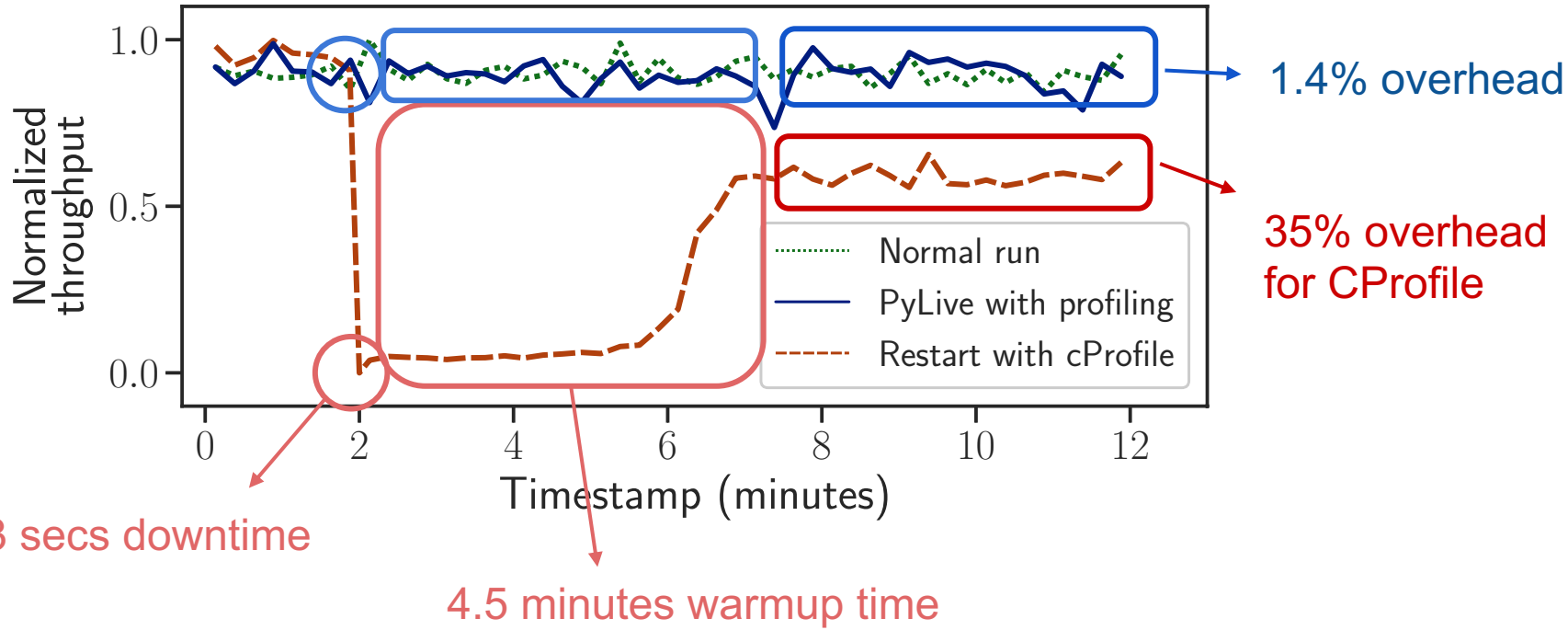
Evaluation of PYLIVE

- ***Performance benefit*** of PYLIVE when ***Profiling***



Evaluation of PYLIVE

- ***Performance benefit*** of PYLIVE when ***Profiling***



Summary

- Build PYLIVE to support on-the-fly logging, profiling and patching in production-run systems without restarting.
  - Relies on standard Python interpreters.
  - Avoids service downtime and warmup time. Little overhead for profiling.
- Evaluate PYLIVE on 20 existing real-world cases and two new performance issues.

Thank you!

Contact: hhuang@ucsd.edu