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

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

* Co-first authors.
Python is widely adopted in online services.

Commercial companies

Python-based frameworks

Web framework

Web server

E-commerce

Message queue
Online services have high requirements on availability.

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.

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

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.*

PYLIVE requires *no restart.*

Rollout is *heavyweight & an overkill.*

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**
  - **Dynamic Typing**
  - Interfaces to dynamically modify *metadata*
  - Allows changing variable *types*
  - \[ \texttt{A.\_\_code\_\_ = D.\_\_code\_\_} \]
    (function body/interface)
  - \[ \texttt{C.A = D} \]
    (class attributes)
  - \[ \texttt{A = "1"} \]
  - \[ \texttt{A = 1} \]
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.

```python
# 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

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

```python
# 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
```python
def call_b(start):
    start = time.time()

def call_a(start):
    logging.info(time.time()-start)
```
PYLIVE’s Interfaces

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)`

**Parameters**
- `prepFunc`: `from ... import ...`
- `old_new_map`: `{"old_func": new_func, "class.new_field": field_init}`
- `safepoint`: "FUNC QUIESENCE"
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

- **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
  - object attributes
  - methods

Meta-object Protocol

Garbage Collection
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

![Diagram showing stacks and functions](image)
Challenge 2: Identify Safe Change Point

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

```python
def file_move_save():
    locks.lock(fd, ...)  # Unsafe points
    ...  # to change
    locks.unlock(fd)  # lock()/unlock()
```

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**

An example of unsafe change points for a patch from Django:

```python
def file_move_save():
    locks.lock(fd, ...)  # Unsafe points to change
    ...
    locks.unlock(fd)
```

An example of state check function:

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

<table>
<thead>
<tr>
<th>Application</th>
<th>Category</th>
<th>Logging</th>
<th>Profiling</th>
<th>Patching</th>
</tr>
</thead>
<tbody>
<tr>
<td>Django</td>
<td>Web framework</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Gunicorn</td>
<td>Web server</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Oscar</td>
<td>E-commerce</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Odoo</td>
<td>E-commerce</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Shuup</td>
<td>E-commerce</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Pretix</td>
<td>E-commerce</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Saleor</td>
<td>E-commerce</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td><strong>Total:</strong></td>
<td></td>
<td><strong>6</strong></td>
<td><strong>4</strong></td>
<td><strong>10</strong></td>
</tr>
</tbody>
</table>

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*

3 secs downtime
Evaluation of PYLIVE

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

![Graph showing normalized throughput over time with timestamps and downtime/warmup time annotations.](image-url)
Evaluation of PYLIVE

- *Performance benefit* of PYLIVE when *Logging*

- 3 secs downtime
- 2.3 minutes warmup time
- < 0.1% overhead
Evaluation of PYLIVE

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

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

3 secs downtime

4.5 minutes warmup time
Evaluation of PYLIVE

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

- 1.4% overhead
- 35% overhead for CProfile
- 3 secs downtime
- 4.5 minutes warmup time
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