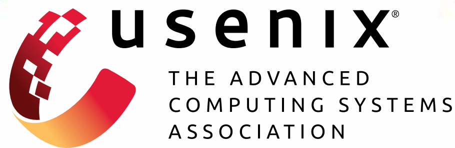


PyLive: On-the-Fly Code Change for Python-based Online Services

Haochen Huang, Chengcheng Xiang, Li Zhong, and Yuanyuan Zhou,
University of California, San Diego

https://www.usenix.org/conference/atc21/presentation/huang-haochen

This paper is included in the Proceedings of the
2021 USENIX Annual Technical Conference.

July 14–16, 2021

978-1-939133-23-6

Open access to the Proceedings of the
2021 USENIX Annual Technical Conference
is sponsored by USENIX.

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

Haochen Huang*, Chengcheng Xiang*, Li Zhong, Yuanyuan Zhou
University of California, San Diego

Abstract

Python is becoming a popular language for building online web services in many companies. To improve online service robustness, this paper presents a new framework, called PyLive, to enable on-the-fly code change. PyLive leverages the unique language features of Python, meta-object protocol and dynamic typing, to support dynamic logging, profiling and bug-fixing without restarting online services. PyLive requires no modification to the underlying runtime systems (i.e., Python interpreters), making it easy to be adopted by online services with little portability concern.

We evaluated PyLive with seven Python-based web applications that are widely used for online services. From these applications, we collected 20 existing real-world cases, including bugs, performance issues and patches for evaluation. PyLive can help resolve all the cases by providing dynamic logging, profiling and patching with little overhead. Additionally, PyLive also helped diagnose two new performance issues in two widely-used open-source applications.

1 Introduction

Python has gained wide adoption on developing online services. Many companies use Python to build their main online platforms. For example, Google's Youtube front-end server is built using Python [37, 51, 96]. Instagram's web services and Quora's main web services are also built with Python [36, 81, 94]. In addition to commercial companies, many open-source projects also use Python to build various frameworks for online services. Table 1 shows six categories of them, including web frameworks, e-commerce and message queues, etc.

These online services have critical requirements on high availability. A recent survey shows that 99.99% of uptime (i.e., 52.6 minutes per year) has become the minimum availability standard for most services [57]. A report from Statista shows that in 2020 one hour server downtime can cause more than $301K cost for 88% of companies and more than $5 million cost for 17% of companies [87].

However, high availability becomes challenging when there are code changes to apply to systems of running services. Such changes include adding logs to gain more diagnostic information, instrumenting programs to profile performance bottlenecks and applying patches to fix bugs.

*Co-first authors.

Category	Server Frameworks (Github stars)
E-commerce	Odoo (17.5k), Saleor (7.8k), Oscar (4.3k)
Web framework	Django (45k), Flask (48k), Pyramid (3.3k)
Web Server	Gunicorn (6.8k), Tornado (19.1k)
Message queue	Celery (14k), huey (2.7k), rq (6.6k)
Cache backend	Django-cacheops (1.1k), Beaker (429)
FTP server	pyftpdlib (1k), fbtftp (325), pyrexecd (202)

Table 1: **Popular Python-based frameworks for online services.**

Motivated by Python's popularity and the demand of applying code changes while keeping high availability, this paper presents a new idea that leverages the unique language features of Python to perform dynamic code changes. Specifically, we design and implement a framework, called PyLive, that enables dynamically changing Python programs for on-the-fly logging, profiling, and bug-fixing on *production systems* without restarting them.

PyLive's capability to change code during production runs can be used by online services for various purposes including:

(1) On-the-fly logging for diagnosing production-run errors. When an online service exhibits some abnormal behaviors, engineers can use PyLive to dynamically add logs at certain locations to collect debugging information from production-run. The logs can be enabled only during certain time (e.g. when the load is light) to minimize the performance impact. Production-run information is useful for diagnosing challenging issues (e.g., resource leaking) that are hard to reproduce in a testing environment and only manifest after a long-time (e.g., weeks) running.

(2) On-the-fly profiling for diagnosing production-run performance issues. When an online service has performance issues observed for certain types of requests, engineers can use PyLive to dynamically profile a set of functions for a short period of time to collect production-run timing information to troubleshoot the issues. The capability to (1) dynamically start and stop profiling during production runs and (2) only profile a small set of specified functions allows engineers to troubleshoot elusive performance issues without introducing much performance overhead. These performance issues may only emerge in production-run but are hard to reproduce during offline testing, making it necessary to perform in-production profiling.

(3) Urgent dynamic patching (bug-fixing or security

patch). PYLIVE allows dynamically applying urgent patches to fix some critical bugs or security issues without stopping and restarting an online service. These bugs and issues can either cause major failures or open up vulnerabilities to attackers and thus needs to be patched as soon as possible.

PYLIVE complements the commonly-used system update practice—rollout deployment [7, 42, 84]. Rollout is not the best choice for dynamic logging and profiling for two reasons. First, rollout still requires a restart of each service instance, which can clear the key program states for diagnosis. These states (e.g., resource leaks) may only be reproduced after a long production run, which is undesirable to be cleared by rollout. Second, a rollout deployment is heavyweight and an overkill for just collecting logging/profiling information. For instance, sometimes only a few servers of a fleet exhibit abnormal behaviors because of their unique memory states. If engineers want to diagnose the issue by rolling out a patch with new logging statements, this patch needs to be batched together with many other patches and will not be applied until the next deployment (wait for a few hours or even a few days). PYLIVE provides a better solution from both perspectives. It requires no restart of service instances so it retains the issue states for logging/profiling. In addition, it enables dynamically adding logging/profiling statements to a running program quickly and also removes the statements flexibly.

PYLIVE is designed based on the unique language features of Python. Python is an interpreted language that supports the meta-object protocol [22, 58] and dynamic typing. The meta-object protocol enables programs to dynamically modify their own metadata, including function bodies/interfaces and class attributes, while dynamic typing allows changing variable types during running. This makes dynamic code change much easier for Python than for compiled languages (e.g., C/C++) and other interpreted languages that do not support the full meta-object protocol or dynamic typing (e.g., Java):

- For compiled languages, dynamically changing a program requires many complex transformations to its code (e.g., patch functions) and memory layout (e.g., load new code into memory), as shown in previous work [38, 46, 54, 55, 68, 76, 77]. At run time, a compiled program's code is binary code and its memory layout is fixed in different segments. Modifying binary code or memory layout may introduce safety concerns.
- Some other interpreted languages, like Java, do not support the full meta-object protocol or dynamic typing. For example, Java does not support many types of dynamic changes, like adding/deleting methods or changing methods' signatures. Java is also statically-typed, making it hard to change variable types. For such languages, implementing dynamic code change requires modification to the language runtime (e.g., JVM), as in previous work [71, 78, 90]. This introduces portability concerns as different versions of runtime may be used by different systems.

PYLIVE makes two main contributions: (1) PYLIVE realizes safe and portable dynamic code change by leveraging Python's language features. PYLIVE is safe as it requires no low-level transformations of machine code or memory layout. PYLIVE is portable as it relies only on the interfaces provided by standard Python interpreters and thus can be easily adopted by existing Python-based systems. (2) Besides patching, PYLIVE also enables instrumentation-based code changes for dynamic logging and profiling. PYLIVE provides convenient interfaces to flexibly instrument customized code to a selected set of functions at running time. This is useful for collecting diagnostic information in production-run systems without causing significant performance degradation.

We evaluate PYLIVE with 20 *real-world* cases from seven widely-used Python-based applications (including the popular Django framework and Gunicorn used by Instagram [35]). All these applications have been deployed in various companies to serve millions of customers [5, 13, 23, 26, 27, 34]. PYLIVE successfully helps resolve the cases with on-the-fly logging, profiling and patching with little overhead. Additionally, PYLIVE has helped two widely-used open source e-commerce applications diagnose two new performance issues. We also measure the performance benefit of PYLIVE by comparing it with restarting services to apply code changes. During normal time (no code change), PYLIVE's overhead is negligible. Upon a code change, PYLIVE causes no downtime and has little performance degradation (<0.1%). In comparison, restarting the applications to apply changes can cause 2-17 seconds downtime and take up to 4.5 minutes to warm up (up to 90% performance downgrade during warmup).

2 Background

This section briefly describes the unique language features of Python that enables PYLIVE's dynamic code change.

Meta-object protocol. Python is designed with a full support of the meta-object protocol [22, 58]. A meta-object is an object that contains a program's metadata, including types, interfaces, classes and methods, etc. The meta-object protocol provides programming interfaces for programs to manipulate these metadata at runtime. For example, a new method named `A` can be dynamically added to class `C` simply with `C.A = D` (`D` is `A`'s definition). Similarly, an existing function's code body can also be changed at runtime with `A.__code__ = D.__code__`. Once it is changed, the new `D` is called when `A` is invoked. Other supported changes include changing a function's interface, adding a new field to a class, changing a variable's type, etc. All the above changes are supported by the standard Python interpreter [32].

Dynamic typing. Python defines and checks variable types at running time, which makes it easy to dynamically change them. For instance, a Python variable `a` can be changed from

a string to a bool. And at running time each time before `a` is used, a type checking is performed. A wrong typed call `compare(a,b)` is detected by the Python interpreter, which will throw a runtime exception. In comparison, for Java, it is impossible to dynamically change a variable's type without changing the underlying JVM. For C/C++, this is also almost impossible as a string is passed by pointer while a bool is passed by value.

Python bytecode. A Python interpreter stores and interprets programs in bytecode, providing an opportunity for dynamically instrumenting code. Compared to machine code, bytecode is much easier to analyze and change with automatic tools. First, it is architecture-independent. Although Python can run on X86-64, ARMv7 and other architecture, it has the same bytecodes for all architectures. So the same tool can be used for Python on all platforms. Second, bytecode also has fewer instructions than machine instructions. Python 3.8 bytecode only has 112 instructions, while X86 alone has 1503 instructions, let alone all various architectures. Third, Python bytecode retains more type information than machine code.

3 PyLive Framework

PyLive is a runtime framework that accepts dynamic change requests from engineers and applies them dynamically into production-run systems without a restart. PyLive can be used for on-the-fly logging, profiling and bug-fixing.

In this section, we begin with the design objectives (§3.1) and interfaces (§3.2) of PyLive. We then discuss the three challenges faced by PyLive: (1) How to support dynamic changes for function interface, function body and data structure? (§3.3) (2) How to identify safe change points to apply a change without causing inconsistency problems? (§3.4) (3) How to update programs with multi-threads and multi-processes? (§3.5)

3.1 Design Objectives

PyLive is designed with the following objectives:

(1) General. PyLive's generality comes from three aspects: (a) it requires no change to the standard Python interpreter. Therefore, engineers need not to download a modified interpreter, which may not be compatible with their systems. (b) PyLive is also general on the types of changes supported. It supports not only changes to function body, but also changes to function interfaces (e.g. add one more parameter) and data structures (e.g. add one more field). (c) Since most online services have multiple threads/processes, PyLive also provides support for multi-threads and multi-processes.

(2) Flexible. PyLive is flexible from two perspectives. First, PyLive is flexible in terms of *when* to apply a change and *when* to revert a change, all based on engineers' requirements. This can help engineers collect logs for a short amount of time to minimize the performance impact. For example, they can perform on-the-fly logging or profiling only during light-load time. Second, PyLive is also flexible with *where* to profile or log. PyLive allows engineers to specify which modules or functions to instrument logging/profiling code.

(3) Consistent and Safe. Dynamic changes to a running program need to be performed at a carefully selected execution point (aka, a safe point) to avoid inconsistency problems. For instance, changing an `unlock` function to its new implementation after an old `lock` function is already executed may cause inconsistency, leading to incorrect states. Unfortunately, choosing a safe point for general changes has proved to be undecidable [53]. So PyLive relies on engineers' knowledge to decide when a change is safe to happen: either when the changed functions are not executing, or a user-specified check function (e.g., specifying a lock is not held) returns true.

(4) Low Overhead. PyLive is designed to impose as little overhead as possible. At normal time when no change needs to be applied, the PyLive thread is sleeping and simply waiting for engineers' inputs. Once engineers instruct it to make code changes, PyLive's thread is woken up to perform the change. Once a change is already applied in this target program's metadata, PyLive gets to sleep again and is no longer involved in the execution of the target program.

(5) Little Human effort. PyLive aims to minimize engineers' efforts in using it. PyLive itself is downloaded as a small Python library that can be easily installed. Only two lines of Python code are needed to set up PyLive at the initialization of the target program. To insert a dynamic change, PyLive only needs engineers to write a small Python snippet to specify what needs to be changed. As shown in our evaluation of 20 real-world cases from seven widely-used Python software systems (cf. Table 3), each change specification needs only 7-13 lines of code).

3.2 PyLive's Interfaces

To make it easy to use, PyLive allows engineers to write the specification for dynamic code change in Python code. To enable dynamic changes for various purposes, PyLive supports two change interfaces: `instrument` and `redefine`.

Instrument. The `instrument` interface can instrument code to specified locations in certain functions or modules. It is useful for instrumenting log statements or profiling code to diagnose bugs or performance issues. The interface is:

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

`scope` is a list of function/module names that need to instrument. When only the module name is given, all functions in it are instrumented. `jointpoint_callback` is key-value dictionary of jointpoints (instrument location) and callback

code to instrument. PyLive supports different granularity of jointpoints: coarse-grained, such as function begin/end, and fine-grained, such as before/after a line and before/after a variable's definition. This allows engineers to flexibly customize the instrument locations. `time` allows engineers to specify when to perform an instrumentation and when to revert the instrumentation. `time` can be either a specific time or a function that decides if an instrumentation should be performed. Engineers may want to profile an online service only when it is lightly-loaded and only for a period of time.

Redefine. The `redefine` interface is for code changes that replace the definitions of existing functions and classes (data structures) with new ones. To perform such code changes, engineers use the following Python interface:

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

`prepFunc` is a user-defined Python function that engineers need to provide to execute before making the specified change. Inside `prepFunc`, engineers can import new modules and perform various initialization tasks. `old_new_map` is a key-value dictionary that specifies the changes. Each pair

```
{'old_func/old_class': new_func/new_class}
```

specifies an old function or class needs to be replaced by a new function or class. Engineers also need to provide the new function or class definition. To add a field to a class, just specify the field name and its initialization code with:

```
{'class.new_field': field_init}.
```

`safepoint` defines at what execution point it is safe to apply the specified change. It can be either "`FUNC_QUIESCENCE`" or a user-specified consistency check function (cf.§3.4).

3.3 Support Dynamic Changes

Change function interface and body. PyLive supports changes on both function interfaces and code bodies. Changing function interfaces includes altering the number of parameters, parameter types, and function names. To make these changes, PyLive utilizes Python's meta-object protocol and dynamic typing. For parameter number changes, Python functions' parameters are defined in a list (i.e. `co_varnames`), and PyLive directly edits the list to add or remove parameters. For parameter type changes, Python uses dynamic typing, and so PyLive needs not explicitly change anything. For function name changes, PyLive defines a new function and modifies the callers to call the new function.

For function body changes, PyLive supports two types of changes: redefine a function body with a new one and instrument the old function body with extra statements (e.g., for logging or profiling). For both types, PyLive replaces functions' code object (`__code__`) as a whole, as code objects are immutable and can only be replaced by reference change. For instrument changes, PyLive first copies the function's code object, builds an instrumented version by modifying its bytecode [10], and then sets the function's `__code__` to point to the instrumented code object.

PyLive may also need to change caller functions when changing the callee functions. For changes that modify callees' interfaces, PyLive needs to change all the callers' function body to call the new interface. PyLive expects that engineers include all changes to callers in the same patch as normal patching practice. For changes that only modify callees' function bodies, PyLive needs not to change the callers. This is because Python function calls are made by function names instead of addresses. Every time a function call happens, Python interpreters translate its name into address by looking up its metadata. Therefore, as long as the function metadata is updated (e.g. modify the `__code__` as discussed before), function calls can always be directed to the newest code objects. Note this differs from dynamic code changes in C/C++—they may need to update callers' function body as the function calls are made by address directly.

Change data structure. Data structure changes include changes to class attributes, object attributes and methods.

Class attributes are data fields defined in classes and shared by all the object instances. PyLive changes class attributes by modifying the namespace tables of the target classes. In order to hook class attributes access for profiling or debugging, PyLive adds `getter` and `setter` functions for attributes need to be changed. In Python, `getter` and `setter` are automatically called if an attribute is annotated as property.

Object attributes are more difficult to change since they are individually stored in different objects even though they are instantiated from the same class. In order to change an object attribute, it is necessary to go through all objects of the class and change each individually. Previous works typically need to refactor a system ahead of time so they can have Factory objects to keep track of all live objects at runtime [41]. PyLive utilizes Python's garbage collector (GC) to track live objects and modify each one when a change is requested. Specifically, PyLive calls `gc.get_objects()` to obtain a list of all live objects tracked by GC [14]. As Python uses reference counting to decide objects' liveness, this does not trigger a heap walk but returns a list immediately instead.

Methods are just functions defined in classes and so can be changed in the same way as global functions as described above. Methods' code is only stored in their classes instead of all instantiated objects, and so simply updating the classes' methods is sufficient to apply a change.

3.4 Identify Safe Change Point

Changing code at run time is not always safe. For instance, changing a function when it is executing may cause inconsistency problems. Therefore, dynamic code change systems

```
30 def file_move_safe(old_file_name, new_file_name):
     ...
58   fd = os.open(new_file_name,...)
59
60   try:
61     locks.lock(fd, ...)
       ...
65     os.write(fd, ...)
       ...
66   finally:
67     locks.unlock(fd)
       ...
```

Unsafe points for changing lock() / unlock()

Figure 1: **An example of unsafe change points for a patch from Django [11].** it is unsafe to change `lock()` or `unlock()` when the program is executing between line 61 and 67, as the change can cause that a new `unlock()` to be called against an old lock, which can leads to undefined behavior.

need to carefully choose a safe execution point to apply a change. Unfortunately, choosing a safe point for general changes has proved to be undecidable [53]. As a result, it is necessary to have engineers' knowledge to choose a safe change point. PYLIVE categorizes safe points into different types and lets engineers select one based on the changes they want to make. Note that choosing the safe update point is only necessary when applying patches. PYLIVE can always apply code changes for logging and profiling as they only add code but do not change the existing code. PYLIVE supports two kinds of safe points:

Quiescence of the changed functions. This requirement means a change is only applied when the changed functions are not under execution. This is also the update point used by many previous dynamic code change systems [38, 38, 39, 86, 91]. It ensures that no function is executed with a mixture of old and new code during changes. PYLIVE provides automatic support for this safe point. To specify it, engineers only need to specify `safepoint='FUNC_QUIESCENCE'`.

PYLIVE supports function quiescence for both changing one function and multiple functions. When changing one function, PYLIVE directly takes advantage of Python meta-object protocol to guarantee the quiescence. In Python, when a function's code is changed, the change only takes effect the next time it is called. When changing multiple functions, PYLIVE checks every thread's stack for any changed function. If any changed function is on a stack, PYLIVE defers the change, retries the checks later and applies the change when no changed function is on any stack.

Consistent state check. When the changed functions modify shared states between them, function quiescence may not be enough for safety. Consider an example shown in Figure 1, two functions `lock` and `unlock` need to be changed, and both of them modify the lock state. Applying the change when the program is executing between the calls to `lock` and `unlock` is not safe, even though the functions themselves are not executed. The new `unlock` may be called with an old lock state

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

Figure 2: **An example of state check function for the patch in figure 1.** It returns true when the lock is not currently held.

and the behavior is undefined.

To address this, PYLIVE allows engineers to provide a customized boolean function to decide when it is safe to apply a change. This is also noted as state quiescence in previous work [48]. Engineers can easily write such boolean functions in normal Python code. Figure 2 shows the state check function for changing `lock` and `unlock`. It checks if no lock is held before applying the change. PYLIVE periodically evaluates it and only applies the change when it returns true.

Note for most code changes, it does not require any customized consistency check. In our evaluation with 20 real-world cases from seven widely-used Python programs, only a few cases require a simple consistency check.

Guidance for engineers. We provide guidance to help engineers identify and specify safe change points for their needs:

First, if the changed functions have no side effects or negligible side effects on execution state, engineers can specify function quiescence as the safe change point. For example, if the changed functions modify no non-local variables, perform no database write and only write a few logs, it is safe to update them as long as they are not under execution.

Second, if the changed functions have some non-negligible side effects on execution states, engineers need to identify the states that the side effects of the old and new functions will not affect each other. Specifically, the variables V_{old} defined and propagated from old functions f_{old} will not be used by new functions f_{new}, and vice versa. To ensure this, the target consistent states are either no variable in V_{old} is defined or all of them are dead. An example of this is shown in Figure 2 that no lock is held at the point of change. Such states may not exist or may not be easy to express in state check functions, and in such cases it may be better to perform a restart than to use PYLIVE.

3.5 Support for Multi-threads and Multi-processes

Multi-threads. A server program may have multiple threads to serve different user requests. Different threads have different program counters while sharing the same code and global variables. Therefore, it is not straightforward to apply a change at a given safe change point for multiple threads.

To change multiple threads correctly, PYLIVE applies a given change synchronously. The synchronous change is ensured by Python global interpreter lock (GIL). At any execution point, only one thread can hold GIL and so can get

executed [15]. Therefore, when PyLive is actively applying a dynamic change, all other threads to be changed are blocked. When applying changes, PyLive also explicitly holds GIL lock to make sure no other threads can preempt it [17].

Based on the type of safe point, PyLive applies changes differently. If the safe point is function quiescence, PyLive either immediately applies the change when only one function needs to change or check program stacks to make sure no target function on stacks when multiple functions need to change. Applying one function change is simpler because Python's meta-object protocol ensures the change to not take effect during its execution. If the safe point is a consistent state check, PyLive first executes the check function provided by engineers. If the consistent check succeeds, PyLive then applies the change. If it fails, PyLive sets a timer *t* and goes to sleep to let other threads execute. The timer will wake up PyLive later to perform a state check again. The timer *t* is configurable by engineers. After several attempts, if it still fails, PyLive will give up and report an error to engineers.

Multi-processes. Online services may use multiple processes, and dynamic code change needs to be applied to all of them. Different Python processes reside in different address spaces and share code through copy-on-write. When code is changed in a process, a copy-on-write happens and other processes will continue to use the old code. As such, dynamic code change needs to be performed explicitly in all processes. PyLive adopts a controller-stub architecture to communicate changes to all processes. A stub is a change thread residing in a target process. PyLive starts one stub thread for each target process at its starting time. A stub thread listens to a controller for patches and applies the received patches at a safe change point. A PyLive controller is a standalone process that accepts engineers' change input and sends the specified code change to the stub thread in each process.

4 Use Cases

PyLive enables three types of use cases that require a running system to be dynamically changed.

4.1 On-the-fly Logging for diagnosis

Systems may exhibit abnormal behavior during running. To collect run time info for diagnosis, engineers may want to add new log messages dynamically without restarting services.

An example of this is the diagnosis of a bug [28] from the Shuup [24] e-commerce system. This bug is related to its shopping cart: when some users click "add to cart", the product is not added to the cart. This prevents users from purchasing products and causes direct revenue loss to businesses. Since the bug has no error logs, it is quite challenging for engineers to diagnose it off-line.

Figure 3 shows how engineers can use PyLive to add log messages to diagnose the issue. Engineers direct PyLive to add line-by-line logs in two functions `add_product` and `_find_product_line_data`. Engineers also specify to only collect logs during light-load time (24:00-2:00).

```
# callbacks to instrument
# logging right/left-hand variables in each line
def call_b(_righthands):
  logging.info(_righthands)
def call_a(_lefthands):
  logging.info(_lefthands)

# instrument code to every line in two functions
instrument(scope=['...add_product',
                  '..._find_product_line_data'],
          jointpoint_callback={line_before: call_b,
                               line_after: call_a},
          time='24:00-2:00')
```

Figure 3: **PyLive's dynamic logging spec for an urgent, real world bug in Shuup e-commerce system [28].** This spec tells PyLive to dynamically instrument code to log some variable values in two functions `add_product` and `_find_product_line_data` for a period of `time`. `line_before` and `line_after` are two joint-points PyLive provides to instrument code before and after each line in functions.

4.2 On-the-fly Profiling

Performance issues often occur in production as systems have more and more features and scale up to a larger size. When such an issue emerges, engineers may want to enable profiling to certain parts of a system during production run.

An example [21] of such issues is from the Oscar e-commerce system. This issue happens when there are a lot of product categories in Oscar. The issue causes a performance downgrade in many pages displayed to customers in Oscar, preventing customers from buying products.

Figure 4 shows how to use PyLive to dynamically instrument code to profile the system. Engineers instruct PyLive to instrument customized profiling code into the methods in two classes, `AbstractCatagory` and `CatalogueView`, that are speculated to be related to the issue.

4.3 Dynamic Patching

Online services frequently have urgent bugs (e.g., security bugs) that need to be patched as quickly as possible to minimize damages since they may cause information leakage/system compromise and prevent customers using online services.

An example [1] of such patches is from Django. It fixes a severe cross-site scripting (XSS) [8] issue, CVE-2019-12308 [9]. The issue is scored as "6.1" since it can expose malicious URLs as clickable links to victim users and direct them to vulnerable sites. Django developers quickly post a security release [12] to fix the vulnerability and encourage all online services that use Django to apply it as soon as possible.

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

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

Figure 4: **On-the-fly profiling using PyLive to diagnose a critical performance issue occurred in Oscar eccomerce system [21].** This example requires PyLive to instrument code to profile the execution of every method in two classes `AbstractCategory` and `CatagolueView` for a period of `time`.

Figure 5 shows part of the patch and the change spec that engineers need to provide for PyLive to dynamically apply it. This patch is non-trivial to be dynamically applied, as it changes both function interfaces and data structures. It adds a new parameter `validator_class` to the `__init__` function and adds a new attribute `self.validator` to `AdminURLFieldWidget`. The change spec calls the `redefine` interface with three arguments: `preupdate` specifies that PyLive needs to import a new class `URLValidator` before applying the change; `old_new_map` indicates that the original `__init__` will be replaced with the new code. `safepoint='FUNC_QUIESCENCE'` tells PyLive to apply the change when the changed functions are quiescent. This requirement is safe enough in the case as there is no inter-dependency between the changed functions.

5 Evaluation

5.1 Methodology

We evaluate PyLive with 20 cases from seven Python-based real-world applications, as shown in Table 2. These applications are deployed in many companies, serving millions of customers [5, 13, 23, 34]. Django is a popular web framework that powered over 94,319 websites, of which many are for e-commerce [85]. Gunicorn is a production web server used by many big companies for their main services, such as Instagram [35]. All the online services need to be almost non-stop since any downtime can result in revenue loss.

To evaluate PyLive's benefit, we compare PyLive with a typical restart approach: modify code for logging/profiling/patching offline, stop the services and restart the services *immediately*. To precisely measure the restart impact, we only restart the Python part of a service, which does not restart other parts (e.g., database) to avoid the impact of warming up their cache. For profiling, we also compare PyLive with cProfile [47], which is Python's official profiling tool for collecting comprehensive profiling information in test environments.

```
# patch: add a parameter validator_class
# add an object attribute validator
class AdminURLFieldWidget(...):
  def __init__(self, attrs=None,
               validator_class=URLValidator):
    self.validator = validator_class()
    ...

# change specs.
def preupdate_call():
  from django.core.validators import URLValidator

redefine(
  preupdate = preupdate_call,
  old_new_map={
      '...AdminURLFieldWidget.__init__':__init__},
  safepoint='FUNC_QUIESCENCE')
```

Figure 5: **A real world security patch to Django [1] and PyLive's dynamic change spec for it.** This patch adds a parameter to function `__init__` and adds an object attribute `validator` to class `AdminURLFieldWidget`. Other part of the patch is omitted due to space limit. The change spec indicates: preupdate — import `URLValidator` before the change; old_new_map — replace `AdminURLFieldWidget.__init__` with a new one; safepoint — apply the change when the changed function is quiescent.

Applications	Category	Logging	Profiling	Patching
Django [33]	Web framework	1	0	2
Gunicorn [16]	Web server	0	0	1
Oscar [6]	E-commerce	1	2	1
Odoo [25]	E-commerce	1	1	2
Shuup [24]	E-commerce	1	0	1
Pretix [27]	E-commerce	1	0	1
Saleor [61]	E-commerce	1	1	2
Total		6	4	10

Table 2: **20 real-world cases evaluated in our experiments.** They are from seven widely used Python-based server applications that have powered many commercial e-commerce and ad-based web services including Instagram, serving millions of customers.

Note PyLive is not a substitute for cProfile as it collects less information than cProfile. However, as we will present in the results, some cases only need little dynamic information to diagnose. We conduct this comparison to study the benefit that PyLive can bring for such cases.

Each application is set up on a machine with a 2.30GHz CPU (6 core), 16GB Memory and 256GB SSD. Each application runs with 2 processes and 4 threads/process. Each application is initialized with ~2000 web pages. To mimic real-world workloads, JMeter [4] is used to generate random web page accesses. The JMeter client is started with 8 threads and can generate up to 15K requests/second.

We use throughput as the performance metric and normalize it to the max throughput of normal service run (41-752 requests/second). All the experiments are conducted within a LAN, which ensures that network is not the bottleneck.

5.2 Overall Performance Results

Overall, PYLIVE avoids 2-17 seconds downtime and avoid up to 4.5 minutes warmup time, during which the performance downgrade can be 55%-90%. PYLIVE causes negligible (<0.1%) overhead during normal run as well as applying changes. PYLIVE causes 0.1%-1.4% overhead during profiling. Compared with cProfile, PYLIVE's selective instrumentation avoids 10.5%-33.6% overhead.

Figure 6,7 show the results of eight representative cases. Two newly identified performance issues and the other twelve existing real-world cases have similar results and due to space limit are put online [3].

For **logging cases**, PYLIVE's benefits mainly come from avoiding the time to restart and warm up. The service restart is relatively fast (2-17 seconds), but the warmup takes much longer time. Our experiments set up applications with only ~2000 web pages, but the warmup still takes 2.3-3 minutes.

For **profiling cases**, PYLIVE makes the performance impact caused by profiling affordable in production-run systems. The benefit comes from two aspects. First, PYLIVE allows engineers to perform customized profiling, so they need not profile applications in a whole as with cProfile. The customized profiling is not a substitute for comprehensive profiling with cProfile because it collects less information. However, it's sufficient to diagnose many cases that only needs limited timing information, as shown later in our case studies (cf. §5.3). Second, PYLIVE avoids restart and warmup time (up to 4.5 minutes), which is needed by cProfile. With PYLIVE, the performance downgrade during profiling is 0.1%-1.4%. While with cProfile, the performance downgrade can be 11%-39%.

For **patching cases**, PYLIVE can apply them dynamically with almost no performance downgrades. This benefits urgent security patches, for which waiting for the next rollout can be dangerous. Our evaluation includes 5 security patches and PYLIVE successfully applied them on-the-fly.

5.3 Case Studies

This section dives into the details of eight representative cases. The remaining twelve cases evaluated are similar and due to space limit we put them online [3].

Case 1: Diagnose a purchase bug in Shuup [28]. This case is about diagnosing a bug related to the shopping carts of Shuup [24], a widely-deployed e-commerce website. As mentioned in §4.1, the bug causes an error in production and prevents customers from adding new items to shopping carts. To help diagnose it, PYLIVE dynamically instruments logging statements on the running application. Figure 6a shows that PYLIVE avoids 3 seconds downtime and 2.3 minutes warmup time. It imposes only < 0.1% performance overhead.

Case 2: Diagnose a payment bug in Odoo [74]. Odoo [25] is an e-commerce website and this bug prevents customers from paying an order. It is an "urgent" bug as it results in business loss. Odoo engineers diagnosed it by adding two logging statements and restarting the service. With PYLIVE, the logging statement can be added on-the-fly with 11 LOC. As shown in Figure 6b, PYLIVE avoids 4 seconds service downtime with < 0.1% overhead. Differing from other applications, Odoo does not have much cache and so requires little warmup time.

Case 3: Diagnose a purchase bug in Pretix [20]. Pretix [27] is an ticket-booking website that allows event organizers to sell event tickets online. In this case, when customers request a PayPal refund, it fails silently with no error messages. PYLIVE can dynamically instrument logging code to diagnose the reason. Figure 6c shows that PYLIVE successfully avoids 17 seconds downtime and 3 minutes warmup (by restarting Pretix) with < 0.1% performance overhead.

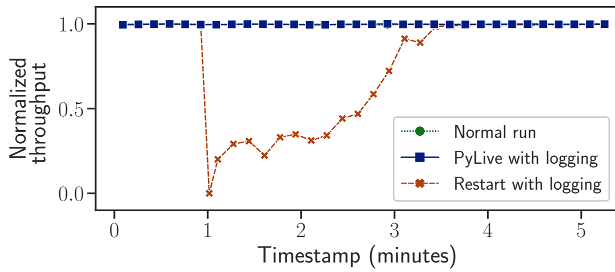
Case 4: Profile a main web page in Saleor [83]. This case is about diagnosing a slowly-loaded web page. This case is difficult to reproduce in testing as it only emerges when the product category grows to large. Currently, engineers use cProfile to profile the whole application [83]. Enabling cProfile needs an application restart, causing downtime and warmup time as shown in Figure 6d. Also, cProfile profiles every function, so after warmup it still imposes 35% performance downgrade.

PYLIVE can benefit the diagnosis in two ways. First, it can dynamically instrument profiling code into a running application. Figure 6d shows this can avoid 3 seconds downtime and 4.5 minutes warmup of Saleor services. Second, it can be customized to only profile the relevant functions suspected by engineers and thus reduces profiling overhead to only 1.4%.

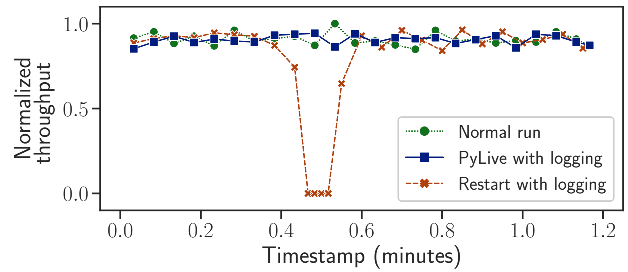
Case 5: Profile a slowly-loaded web page in Oscar [21]. This case is about diagnosing a slowly-loaded product-listing page. It happens when the number of products grows to large. PYLIVE enables dynamic profiling to Oscar with 9 LOC to specify the change. As shown in Figure 6e, PYLIVE causes only 0.5% performance overhead during profiling and nearly no overhead during normal run. In contrast, cProfile causes as much as 11% performance downgrades as well as 2 seconds of downtime and 3 minutes warmup time.

Case 6: Profile a slow action in Odoo [75]. This case is about diagnosing a slow receipt-validating action. It is hard to reproduce in testing as it only emerges when the database contains a large number of products and orders. PYLIVE enables dynamic profiling with 9 LOC. Figure 6f shows PYLIVE's performance benefit. PYLIVE causes only 0.1% performance overhead during profiling and nearly no overhead during normal run. In contrast, cProfile causes as much as 38.5% performance downgrades as well as 9 seconds of service downtime.

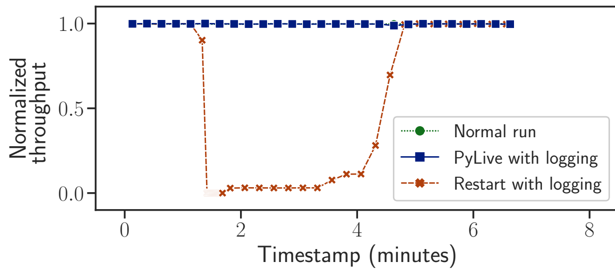
Case 7: Patch CVE-2019-12308 security vulnerability in Django [1]. This patch fixes a severe XSS security issue CVE-2019-12308 [9]. As we discussed in §4.3, it may lead users to click into malicious websites and can possibly affect



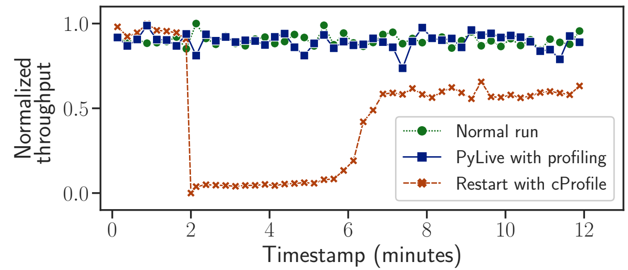
(a) **Shuup: on-the-fly logging to diagnose a payment bug [28].** PyLive causes < 0.1% overhead only when adding logs. In comparison, restarting causes 3 seconds of downtime and needs 2.3 minutes to warmup.



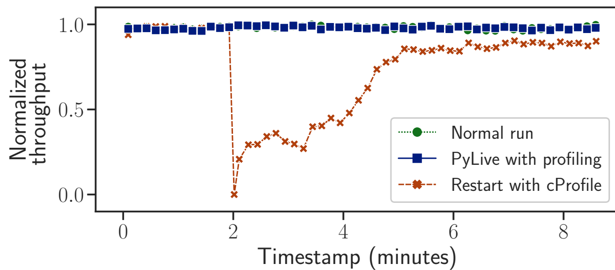
(b) **Odoo: on-the-fly logging to diagnose a shopping-cart bug [74].** PyLive causes < 0.1% overhead only when adding logs. In comparison, restarting causes 4 seconds of downtime. Odoo does not have much cache so has a short warmup time.



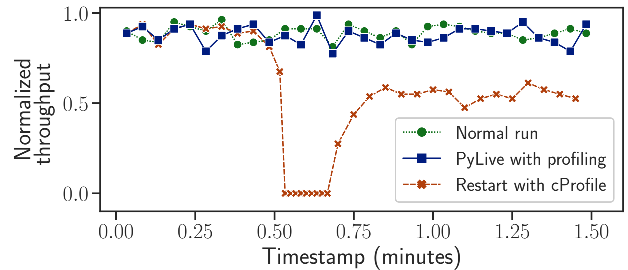
(c) **Pretix: on-the-fly logging to diagnose a payment bug [20].** PyLive causes < 0.1% overhead only when adding logs. In comparison, restarting causes 17 seconds of downtime and needs 3 minutes to warmup.



(d) **Saleor: on-the-fly profiling a long-loaded web page [83].** PyLive causes 1.4% overhead only during profiling. In comparison, restarting causes 3 seconds of downtime and needs 4.5 minutes to warmup. Using cProfile causes 35% overhead.



(e) **Oscar: on-the-fly profiling a long-loaded web page [21].** PyLive causes 0.5% overhead only during profiling. In comparison, restarting causes 2 seconds of downtime and needs 3 minutes to warmup. Using cProfile causes 11% overhead.

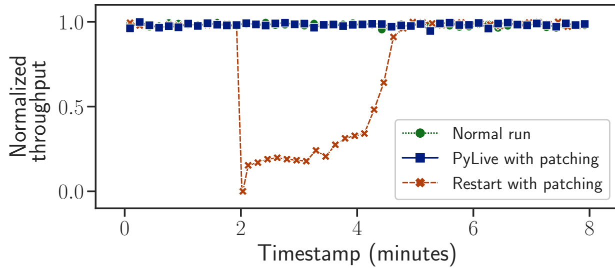


(f) **Odoo: on-the-fly profiling a long-loaded web page [75].** PyLive causes 0.1% overhead only during profiling. In comparison, restarting causes 9 seconds of downtime. Using cProfile to profile causes 38.5% overhead.

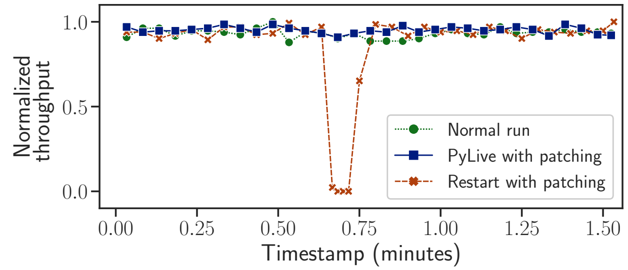
Figure 6: Throughput comparison of **three on-the-fly logging cases and three on-the-fly profiling cases** with PyLive in comparison with today's practices—stop and restart with logging added and profiling enabled.

many users. This patch is non-trivial to be dynamically applied, as it involves adding a parameter to a method interface and adding a new class attribute [1]. With PyLive, this patch is allowed to be applied safely with 8 additional LOC. Figure 7a shows the performance benefit of PyLive and PyLive avoids 2 seconds of downtime and 2.8 minutes warmup time. PyLive dynamically applies the patch with < 0.1% overhead.

Case 8: Patch CVE-2018-1000164 in Gunicorn [52]. This patch fixes a HTTP Response Splitting Vulnerability [73]. It has a severity score of "**7.5 High**" in the CVE system [72]. It can be exploited by various attacks, such as Cross-site Scripting (XSS), Cross-User Defacement, Hijacking [73]. The patch requires a modification to a function body. It can be dynamically applied with PyLive with only 13 additional LOC. As shown in Figure 7b, PyLive avoids 4 seconds of downtime, when a non-cached service runs on Gunicorn. PyLive introduces < 0.1% overhead while applying the patch.



(a) **Django: urgent security patching for CVE-2019-12308 [1].** Compared with restarting, PyLive avoids 2 seconds of downtime and 2.8 minutes warmup time, with < 0.1% performance overhead during patching.



(b) **Gunicorn: security patching for CVE-2018-1000164 [72].** Compared with restarting, PyLive avoids 4 seconds of service downtime, with negligible overhead during patching. Gunicorn's workload is Odoo, which has little cache, so it takes a short time to warmup.

Figure 7: Throughput comparison of **two representative patching cases** with PyLive and restarting services.

Use Case	Software	LOC
Case1	Shuup	7
Case2	Odoo	11
Case3	Pretix	9
Case4	Saleor	9

Use Case	Software	LOC
Case5	Oscar	9
Case6	Odoo	9
Case7	Django	8
Case8	Gunicorn	13

Table 3: **Lines of code (LOC) of change specification for PyLive.** For patches, this only count extra code for PyLive.

5.4 Human Effort

PyLive requires only a little human effort to adopt it in real-world applications. To enable PyLive in a Python-based application, it only needs to add two lines of code in the application's initialization stage. To apply dynamic change for different purposes, PyLive allows engineers to write Python code to specify the intended changes. Table 3 shows the lines of code (LOC) to specify the changes in the eight representative cases. For all cases, it requires only 7-13 lines of code to specify each change.

6 Limitations and Discussion

There are many kinds of code changes that PyLive cannot apply. First, PyLive cannot apply changes to long-running functions because dynamic changes only take effect next time when the functions are called. Fortunately, online services are usually request based and the major part is the request handling functions, which finish running in a short amount of time. Second, PyLive cannot apply patches that assume an initial program state. Patches for memory-leak bugs may need to reinitialize the program state to free the leaked memory, which needs a restart of the target program. Third, PyLive is not suitable for applying major changes to a target program. Such changes include adding new features, updating library versions, and refactoring the program structure. These changes may involve major changes to program states and code logic of many functions. Therefore, it is hard for engineers to write code to initialize the states and to specify a safe update point that considers all the dependencies between the changed functions.

PyLive relies on engineers to specifies the safe update points for dynamic patching. PyLive targets on simple bug-fixing and security patches that only update a few functions and data structures. For these patches, the safe update points can be specified as when the targeted functions are not executing or when a customized state check passes (e.g. a lock is not held as in Figure 1). However, for more complex patches that change many interdependent functions and data structures, the safe update point may not be easy to specify. For such cases, it is safer to restart the target program than to use PyLive. Note the safe update point is only necessary for applying patches but not for logging or profiling. Code changes for profiling and logging can always be safely applied as they only add code but do not change the existing code.

PyLive cannot prevent errors introduced by buggy patches. PyLive expects that engineers thoroughly test their patches in a testing environment before dynamically applying them to production-run systems. For logging and profiling cases, PyLive wraps the instrumented code in try-catch blocks so that buggy logging or profiling code does not affect the normal program execution.

PyLive has two security implications. First, in terms of the type of code changes that can be made dynamically, PyLive does not expand the attack surface of Python's own meta-object protocol. PyLive does not modify the Python interpreter to enable more types of code changes but just provides convenient interfaces purely based on Python's meta-object protocol. Second, the introduction of a change controller (cf. §3.5) expands the attack surface from one single process to two processes. The change controller is an additional process that commands a target program process to apply a change dynamically. Therefore, it would be dangerous if attackers gain access to the change controller. It can be mitigated by

setting the change controller's permission to make it only executable by a privileged user. We also plan to implement PYLIVE's own access control for the change controller in future work.

PYLIVE's design and implementation are generally applicable to Python variants and other interpreted languages as long as they support three language features:

- Meta-object protocol—PYLIVE uses this to modify a program's code at running time (cf. §2);
- Dynamic typing—PYLIVE relies on this to modify variable types at running time (cf. §2);
- Interpreter interfaces to freeze non-current threads—PYLIVE uses them to pause the execution of any other threads to safely apply changes (cf. §3.5).

All three features are supported by popular python variants including Pypy [29] and Pyston [2], and so PYLIVE can be easily ported to them. PYLIVE can also potentially be ported to two other popular interpreted languages: JavaScript [19] and Ruby [30]. The first two features are directly supported by JavaScript and Ruby. The third feature can also be implemented in JavaScript and Ruby in different ways. JavaScript uses a single-threaded event loop model—at any time only one event handler is running and it cannot be preempted before its completion. Therefore, when PYLIVE is running in JavaScript, any other thread is ensured not running at the same time. Ruby's official interpreter YARV [31] has a similar GIL lock as Python's GIL, which allows PYLIVE to hold GIL in Ruby to prevent preemption as in Python (cf. §3.5).

7 Related Work

Dynamic Code Change for C/C++ and Java. Many works have been done for dynamic changing C/C++-based operating systems [39–41, 45, 49, 64, 86] and applications [38, 46, 54, 56, 63, 69, 82, 89]. While simple dynamic change (e.g. patch only function bodies) to OS kernels has been used in production, more general change to applications has not been widely adopted. General dynamic change usually requires many unsafe transformations to target programs including modifying both machine code and memory layout. This may introduce safety concerns in production. Contrarily, PYLIVE realizes general changes by utilizing Python's standard language features—meta-object protocol and dynamic typing, making it safer to be adopted in production.

Works on dynamic changing Java either need to modify JVM [71, 90] or rely on some unsafe operations of JVM [78], introducing portability and safety concerns in production. When running a Java program, JVM maintains many metadata such as method signatures and class attributes as internal data but provides no safe operations to modify them. However, it is necessary to modify these metadata in order to support general dynamic change. In comparison, PYLIVE makes use of Python's meta-object protocol to safely modify related metadata when dynamically changing Python programs. This imposes no modification to a standard Python interpreter and so can be easily adopted in existing production systems.

Dynamic Code Change for Python. Pymoult [66] made a preliminary exploration on the feasibility of dynamically changing Python programs. As a preliminary proposal, it has no experimental result. More importantly, Pymoult relies on a special Python interpreter, Pypy, which is not fully compatible with the standard Python interpreter (i.e. CPython [32]). In order to use Pymoult, engineers need to port their systems to Pypy interpreter, which requires considerable human efforts. Contrarily, PYLIVE is based on the standard Python interpreter and so can be easily adopted in the field.

PyReload [93] is a dynamic code change tool based on the standard Python interpreter. However, it has two key limitations that prevent it to be practical. First, PyReload needs engineers to refactor a target program into modules, which requires huge human efforts. Second, PyReload only supports single-threaded programs. Considering that servers usually have multiple threads, PyReload is not suitable for server systems. In contrast, PYLIVE requires no refactor to the target program and supports updating multi-thread server programs.

Language level support for dynamic code change. Language level support for dynamic code change is not new. Besides Python, many other dynamically-typed programming languages, such as Common Lisp [88], Smalltalk [50], JavaScript [19] and Ruby [30], have provided support for meta-object protocol. Meta-object protocol can be directly used to update a single function/class; however, there are still many challenges in using meta-object protocol to practically update server programs, which usually needs to update multiple functions/classes and threads/processes. Very few works have been done on these challenges. Rivet [67] proposes interesting ideas to leverage JavaScript's reflection capabilities to debug single-threaded browser-side programs. But the ideas cannot be directly borrowed to update server programs, which usually have multiple threads/processes.

Focusing on Python, PYLIVE addresses three challenges of leveraging meta-object protocol to update server programs. First, to make it easy to update multiple functions and classes, PYLIVE provides two APIs: `Redefine` and `Instrument` (§3.2) and adopts the meta-object protocol and bytecode instrumentation to implement them (§3.3). Second, to make it safe to update multiple functions and classes, PYLIVE borrows ideas from previous works [38, 39, 48] and provides two different safe points (§3.4). Third, to support updating programs with multiple-threads and multiple-processes, PYLIVE proposes new synchronous update mechanisms based on Python GIL and a controller-stub architecture (§3.5).

Aspect-oriented programming. PYLIVE's `Instrument` interface is a type of aspect-oriented programming (AOP) [60]. AOP is a programming paradigm to break down independent program logic into different modules. A common usage of

AOP is to allow developers to write a function's main logic and its logging code separately. The AOP framework then "weaves" the code together at compile time or load time. Several works also aim to enable dynamic weaving at run time for AOP [79, 80, 92]. PyLive's `Instrument` interface is an AOP support for Python and is inspired by previous work on AOP interfaces for Java [59, 65]. However, AOP's main target is to insert additional code without modifying the existing code. PyLive also provides a `Redefine` interface to modify the existing code of a running program, which is challenging especially when the target program has multiple threads/processes. To realize `Redefine`, PyLive further considered the challenge of supporting safe update points and multiple-threads/processes programs.

Dynamic Instrumentation. PyLive's `Instrument` interface is related to previous works on dynamic binary instrumentation (DBI), including Pin [62], Valgrind [70], and DynamoRIO [43]. DBI enables modifying a running binary program at the machine instruction level and is usually used for logging and profiling a compiled program. However, DBI cannot be directly adopted for profiling a Python program in production. When DBI is used for a Python program, DBI is instrumenting the Python interpreter instead of the Python program. This can cause two folds of problems. First, the logging and profiling results are verbose and hard to understand by Python developers as they are mostly about the execution of the Python interpreter but not about the Python program. Second, this can introduce an unacceptable performance downgrade to the Python program as interpreting one line of Python code may need to run multiple lines of interpreter code. PyLive addresses these problems by instrumenting code at the granularity of Python bytecode. Therefore, the logging and profiling results can be directly mapped back to the Python program and so are easy to understand by Python developers. In addition, much less code is instrumented and so much less performance downgrade.

PyLive complements DTrace [44] on dynamic instrumentation. DTrace is a dynamic instrumentation framework for production systems. To enable DTrace for Python, it needs to embed "markers" in Python interpreters. This introduces additional compatibility requirements for Python interpreters to use DTrace. As noted by the official Python document [18], "DTrace scripts can stop working or work incorrectly without warning when changing CPython versions." PyLive takes a complementary approach to instrument Python application code without modifying Python interpreters, avoiding the compatibility concerns.

Rollout Update. An alternative way to avoid whole system downtime is rollout update [7, 42, 84]. In rollout update, a cluster of servers are restarted one by one or batches by batches so that during an update there are still servers alive to serve users' requests. However, for just collecting logging/profiling information, rolling out patches to a whole cluster at the next deployment is heavyweight and an overkill. It would be handy to quickly apply a simple patch that temporarily logs extra information or collects extra metrics to some servers on-the-fly. Furthermore, rollout update is less effective for collecting diagnostic or profiling information for certain types of issues because rollout update still requires restarting every service instance. As a result, errors that appear only after a long running time, such as resource leaks and concurrency issues, may not reappear quickly after restarting to provide diagnostic information [95]. Finally, rollout update can still result in a subset of servers restarting and warming up before providing service at their full capacity. This means during the rollout update, the entire system will suffer from certain levels of throughput degradation.

8 Conclusions

In this paper, we proposed a framework called PyLive that leverages Python's unique language features, meta-object protocol and dynamic typing, to support dynamic code change for on-the-fly logging, profiling and patching in production-run systems. PyLive only relies on standard Python interpreters and can be easily adopted by existing systems. We evaluated PyLive with seven widely-deployed Python-based systems for online services. PyLive successfully helped resolve 20 existing real-world cases from these systems with dynamically logging, profiling and patching. PyLive also helped two of the systems diagnose two *new* performance issues. In comparison to restart, PyLive avoids service downtime and warmup. PyLive imposes no overhead during normal run and negligible overhead during the change. For profiling, PyLive adds only 0.1%-1.4% overhead.

Acknowledgments

We greatly appreciate our shepherd, Larry Rudolph, and the anonymous reviewers for their insightful comments and feedback. We thank Stewart Grant, Rajdeep Das, Keegan Ryan, the Opera group as well as the Systems and Networking group at UCSD for helpful discussions and paper proofreading. This work is supported in part by NSF grants (CNS-1814388, CNS-1526966) and the Qualcomm Chair Endowment.

References

[1] [2.2.x] Fixed CVE-2019-12308 – Made AdminURLFieldWidget validate URL before rendering clickable link. https://github.com/django/django/commit/afddabf8428ddc89a332f7a78d0d21eaf2b5a673.

[2] A faster and highly-compatible implementation of the Python programming language. https://github.com/pyston/pyston.

[3] Anonymous repo for Pylive evaluation result. https://github.com/pyupdate/evaluation.

[4] Apache JMeter - Apache JMeter. https://jmeter.apache.org/.

[5] Butterfly Network Case Study – Saleor Commerce. https://saleor.io/case-study/butterfly-network/.

[6] Cases / Oscar - Domain-driven e-commerce for Django. http://oscarcommerce.com/cases.html.

[7] Continuous Deployment at Instagram. https://instagram-engineering.com/continuous-deployment-at-instagram-1e18548f01d1.

[8] Cross Site Scripting (XSS) Software Attack | OWASP Foundation. https://owasp.org/www-community/attacks/xss/.

[9] CVE-2019-12308 Detail. https://nvd.nist.gov/vuln/detail/CVE-2019-12308.

[10] Disassembler for Python bytecode. https://docs.python.org/3/library/dis.html.

[11] Django patch: Changed the use of fcntl.flock() to fcntl.lockf(). https://github.com/django/django/commit/195420259a5286cbeface8ef7d0570e5e8d651e0.

[12] Django security releases issued: 2.2.2, 2.1.9 and 1.11.21. https://www.djangoproject.com/weblog/2019/jun/03/security-releases/.

[13] edX. https://open.edx.org/blog/using-open-edx-ecommerce-module/.

[14] Garbage Collector interface. https://docs.python.org/3/library/gc.html#gc.get_objects.

[15] GlobalInterpreterLock - Python Wiki. https://wiki.python.org/moin/GlobalInterpreterLock.

[16] Gunicorn - Python WSGI HTTP Server for UNIX. https://gunicorn.org/.

[17] Initialization, Finalization, and Threads . https://docs.python.org/3/c-api/init.html#c.PyGILState_Ensure.

[18] Instrumenting CPython with DTrace and SystemTap. https://docs.python.org/3/howto/instrumentation.html.

[19] Javascript Programming Language. https://www.javascript.com/.

[20] Log the reason for failed PayPal refunds. https://github.com/pretix/pretix/commit/5400d26c60b7a4fceab2c832419e63abfd785f0d.

[21] Long rendered page when a lot of categories products 1910. https://github.com/django-oscar/django-oscar/issues/1910.

[22] Metaobject. https://en.wikipedia.org/wiki/Metaobject.

[23] Multivendor Marketplace Platform - Enterprise Commerce Software. https://www.shuup.com/.

[24] Multivendor Marketplace Platform - Enterprise Commerce Software. https://www.shuup.com/.

[25] Odoo. https://www.odoo.com/.

[26] Odoo Customer Reviews | Success Stories. https://www.odoo.com/blog/customer-reviews-6.

[27] pretix – Reinventing ticket sales for conferences, festivals, exhibitions, ... https://pretix.eu/about/en/.

[28] Product does not get added to basket if force_new_line = True #291. https://github.com/shuup/shuup/issues/291.

[29] Pypy. https://www.pypy.org/.

[30] Ruby Programming Language. https://www.ruby-lang.org/en/.

[31] ruby.git - The Ruby Programming Language. https://git.ruby-lang.org/ruby.git.

[32] The Python programming language. https://github.com/python/cpython.

[33] The Web framework for perfectionists with deadlines | Django. https://www.djangoproject.com/.

[34] Web Service Efficiency at Instagram with Python - Instagram Engineering. https://instagram-engineering.com/web-service-efficiency-at-instagram-with-python-4976d078e366.

[35] What Powers Instagram: Hundreds of Instances, Dozens of Technologies. https://instagram-engineering.com/what-powers-instagram-hundreds-of-instances-dozens-of-technologies-adf2e22da2ad.

[36] Adam D'Angelo. Why did Quora choose Python for its development? https://www.quora.com/Why-did-Quora-choose-Python-for-its-development-What-technological-challenges-did-the-founders-face-before-they-decided-to-go-with-Python-rather-than-PHP, Sep. 2014.

[37] Alex Martelli. Heavy usage of Python at Google? https://stackoverflow.com/questions/2560310/heavy-usage-of-python-at-google, Apr. 2010.

[38] Gautam Altekar, Ilya Bagrak, Paul Burstein, and Andrew Schultz. Opus: Online patches and updates for security. In *USENIX Security Symposium*, pages 287–302, 2005.

[39] Jeff Arnold and M Frans Kaashoek. Ksplice: Automatic rebootless kernel updates. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 187–198. ACM, 2009.

[40] Andrew Baumann, Jonathan Appavoo, Robert W Wisniewski, Dilma Da Silva, Orran Krieger, and Gernot Heiser. Reboots are for hardware: Challenges and solutions to updating an operating system on the fly. In *USENIX Annual Technical Conference*, pages 337–350, 2007.

[41] Andrew Baumann, Gernot Heiser, Jonathan Appavoo, Dilma Da Silva, Orran Krieger, Robert W Wisniewski, and Jeremy Kerr. Providing dynamic update in an operating system. In *USENIX Annual Technical Conference, General Track*, pages 279–291, 2005.

[42] Eric A Brewer. Lessons from giant-scale services. *IEEE Internet Computing*, 5(4):46–55, 2001.

[43] Derek Bruening and Saman Amarasinghe. *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering . . . , 2004.

[44] Bryan Cantrill, Michael W Shapiro, Adam H Leventhal, et al. Dynamic instrumentation of production systems. In *USENIX Annual Technical Conference, General Track*, pages 15–28, 2004.

[45] Haibo Chen, Rong Chen, Fengzhe Zhang, Binyu Zang, and Pen-Chung Yew. Live updating operating systems using virtualization. In *Proceedings of the 2nd international conference on Virtual execution environments*, pages 35–44, 2006.

[46] Haibo Chen, Jie Yu, Rong Chen, Binyu Zang, and Pen-Chung Yew. Polus: A powerful live updating system. In *29th International Conference on Software Engineering (ICSE'07)*, pages 271–281. IEEE, 2007.

[47] Python Software Foundation. The python profilers. https://docs.python.org/3.5/library/profile.html, 2019.

[48] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. Safe and automatic live update for operating systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, page 279–292, New York, NY, USA, 2013. Association for Computing Machinery.

[49] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. Safe and automatic live update for operating systems. *SIGPLAN Not.*, 48(4):279–292, March 2013.

[50] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., 1983.

[51] Quinta group. Python at google. https://quintagroup.com/cms/python/google.

[52] gunicorn. Potential http response splitting vulnerability. https://github.com/benoitc/gunicorn/issues/1227, 2016.

[53] Deepak Gupta, Pankaj Jalote, and Gautam Barua. A formal framework for on-line software version change. *IEEE Transactions on Software engineering*, 22(2):120–131, 1996.

[54] Christopher M. Hayden, Edward K. Smith, Michail Denchev, Michael Hicks, and Jeffrey S. Foster. Kitsune: Efficient, general-purpose dynamic software updating for c. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, page 249–264, New York, NY, USA, 2012. Association for Computing Machinery.

[55] Michael Hicks, Jonathan T. Moore, and Scott Nettles. Dynamic software updating. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, page 13–23, New York, NY, USA, 2001. Association for Computing Machinery.

[56] Gisli Hjalmtysson and Robert Gray. Dynamic c++ classes-a lightweight mechanism to update code in a running program. In *USENIX Annual Technical Conference*, volume 98, 1998.

[57] Information Technology Intelligence Consulting Corp. ITIC 2020 Global Server Hardware, Server OS Reliability Report. https://www.ibm.com/downloads/cas/DV0XZV6R, April 2020.

[58] Gregor Kiczales, Jim Des Rivieres, and Daniel Gureasko Bobrow. *The art of the metaobject protocol*. MIT press, 1991.

[59] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G Griswold. An overview of aspectj. In *European Conference on Object-Oriented Programming*, pages 327–354. Springer, 2001.

[60] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European conference on object-oriented programming*, pages 220–242. Springer, 1997.

[61] Mirumee Labs. A modular, high performance, headless e-commerce storefront built with python, graphql, django, and reactjs. https://saleor.io/, 2020.

[62] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices*, 40(6):190–200, 2005.

[63] Kristis Makris and Rida A Bazzi. Immediate multi-threaded dynamic software updates using stack reconstruction. In *USENIX annual technical conference*, volume 2009. San Diego, CA, 2009.

[64] Kristis Makris and Kyung Dong Ryu. Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 327–340, 2007.

[65] Lukáš Marek, Alex Villazón, Yudi Zheng, Danilo Ansaloni, Walter Binder, and Zhengwei Qi. Disl: a domain-specific language for bytecode instrumentation. In *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*, pages 239–250, 2012.

[66] Sebastien Martinez, Fabien Dagnat, and Jérémy Buisson. Pymoult : On-Line Updates for Python Programs. In *ICSEA 2015*, 2015.

[67] James Mickens. Rivet: browser-agnostic remote debugging for web applications. In *Presented as part of the 2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)*, pages 333–345, 2012.

[68] Iulian Neamtiu, Michael Hicks, Gareth Stoyle, and Manuel Oriol. Practical dynamic software updating for c. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, page 72–83, New York, NY, USA, 2006. Association for Computing Machinery.

[69] Iulian Neamtiu, Michael Hicks, Gareth Stoyle, and Manuel Oriol. Practical dynamic software updating for c. *ACM SIGPLAN Notices*, 41(6):72–83, 2006.

[70] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices*, 42(6):89–100, 2007.

[71] Angela Nicoara, Gustavo Alonso, and Timothy Roscoe. Controlled, systematic, and efficient code replacement for running java programs. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 233–246, 2008.

[72] NVD. Cve-2018-1000164 detail. https://nvd.nist.gov/vuln/detail/CVE-2018-1000164, 2018.

[73] NVD. Cve-2018-1000164 detail. https://owasp.org/www-community/attacks/HTTP_Response_Splitting, 2018.

[74] odoo. [payment_paypal] 500 error when process order. https://github.com/odoo/odoo/issues/39406, 2019.

[75] odoo. [13.0] performance issue on validating receipts. https://github.com/odoo/odoo/issues/46900, 2020.

[76] Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 87–102, New York, NY, USA, 2009. Association for Computing Machinery.

[77] Luís Pina, Anastasios Andronidis, Michael Hicks, and Cristian Cadar. Mvedsua: Higher availability dynamic software updates via multi-version execution. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 573–585. ACM, 2019.

[78] Luís Pina, Luís Veiga, and Michael Hicks. Rubah: Dsu for java on a stock jvm. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14,

page 103–119, New York, NY, USA, 2014. Association for Computing Machinery.

[79] Andrei Popovici, Gustavo Alonso, and Thomas Gross. Just-in-time aspects: efficient dynamic weaving for java. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 100–109, 2003.

[80] Andrei Popovici, Thomas Gross, and Gustavo Alonso. Dynamic weaving for aspect-oriented programming. In *Proceedings of the 1st international conference on Aspect-oriented software development*, pages 141–147, 2002.

[81] Romain Komorn. Python in production engineering. https://engineering.fb.com/production-engineering/python-in-production-engineering/, May 2016.

[82] Florian Rommel, Christian Dietrich, Peng Huang, Daniel Friesel, Sangeetha Abdu Jyothi, Karan Grover, Marcel Köppen, Nina Narodytska, Muthian Sivathanu, Christoph Borchert, et al. From global to local quiescence: Wait-free code patching of multi-threaded processes. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 651–666, 2020.

[83] saleor. Category index renders extremely slow when there are many discounts. https://github.com/mirumee/saleor/issues/1314, 2017.

[84] Tony Savor, Mitchell Douglas, Michael Gentili, Laurie Williams, Kent Beck, and Michael Stumm. Continuous deployment at facebook and oanda. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 21–30. IEEE, 2016.

[85] similartech. Market share and web usage statistics of Django. https://www.similartech.com/technologies/django, Jan. 2020.

[86] Craig AN Soules, Jonathan Appavoo, Kevin Hui, Robert W Wisniewski, Dilma Da Silva, Gregory R Ganger, Orran Krieger, Michael Stumm, Marc A Auslander, Michal Ostrowski, et al. System support for online reconfiguration. In *USENIX Annual Technical Conference, General Track*, pages 141–154, 2003.

[87] Statista Inc. Average cost per hour of enterprise server downtime worldwide. https://www.statista.com/statistics/753938/worldwide-enterprise-server-hourly-downtime-cost/, 2021.

[88] Guy Steele. *Common LISP: the language*. Elsevier, 1990.

[89] Gareth Stoyle, Michael Hicks, Gavin Bierman, Peter Sewell, and Iulian Neamtiu. Mutatis mutandis: Safe and predictable dynamic software updating. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, page 183–194, New York, NY, USA, 2005. Association for Computing Machinery.

[90] Suriya Subramanian, Michael Hicks, and Kathryn S. McKinley. Dynamic software updates: A vm-centric approach. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, page 1–12, New York, NY, USA, 2009. Association for Computing Machinery.

[91] Suriya Subramanian, Michael Hicks, and Kathryn S McKinley. Dynamic software updates: a vm-centric approach. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, 2009.

[92] Davy Suvée, Wim Vanderperren, and Viviane Jonckers. Jasco: an aspect-oriented approach tailored for component based software development. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 21–29, 2003.

[93] Wei Tang and Min Zhang. Pyreload: Dynamic updating of python programs by reloading. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, pages 229–238. IEEE, 2018.

[94] thenewstack.io. Instagram Makes a Smooth Move to Python 3. https://thenewstack.io/instagram-makes-smooth-move-python-3/, Jun. 2017.

[95] Joseph Tucek, Shan Lu, Chengdu Huang, Spiros Xanthos, and Yuanyuan Zhou. Triage: diagnosing production run failures at the user's site. *ACM SIGOPS Operating Systems Review*, 41(6):131–144, 2007.

[96] Wikipedia. Youtube. https://en.wikipedia.org/wiki/YouTube, 2020.