PyRR: Record-Replay and Multi-Version Execution for Unmodified CPython

Farhan Saif

Master's Project Report[1]

Committee Members

Professor Dr. Luis Pina (Advisor)
Professor Dr. William Mansky

October 2026

[1]Submitted in partial fulfillment of the requirements for the Master of Science degree, project track.

Abstract

Record-replay (R/R) systems capture the nondeterministic inputs of a program so that a failing execution can be reproduced exactly, and multi-version execution (MVX) systems run a second copy of a program in lockstep with the first so that it can take over when the first fails. Existing systems operate at native-process boundaries or within managed runtimes such as the Java virtual machine. We investigate adapting language-boundary interception to CPython. We present PyRR, an R/R and MVX system for CPython that extends the design of JMVX [7] with the mechanisms a dynamic language requires. PyRR intercepts the C-implemented functions through which CPython reaches the kernel by rewriting the interpreter's own function tables at runtime, so it needs no interpreter changes, no bytecode rewriting, and no custom build. It handles Python-specific sources of nondeterminism such as the seeding of the global random number generator and the readiness order of the asyncio event loop. In MVX mode the follower holds the leader's live socket descriptors and can take over the same TCP connections when the leader is killed. On compute-bound benchmarks PyRR costs nothing measurable; on web servers recording costs 15–17%, and on almost purely I/O-bound key–value workloads it costs 1.3–2.8×, between 1.3× and 5× less than Mozilla rr on the same workloads. In failover experiments, followers promoted in all five attempts per workload. FastAPI and SSE each completed all five client workloads correctly on existing connections; the KV server completed two of five. The other KV attempts exposed reply mismatches or a timeout, identifying recovery of in-flight operations as the remaining challenge.

Contents

1 Introduction **1**

2 Problem **4**

3 Related Work **4**

4 Design Overview **5**

5 Intercepting a Function **5**
5.1 Declaring the function 6
5.2 Replacing the binding 6
5.3 Caches 7
5.4 The wrapper 7
5.5 Walking through one call 7
5.6 Calls that need extra handling 8

6 Record and Replay **8**
6.1 Recording 8
6.2 Replay 9
6.3 Seeding the random number generator 9

7 Multi-Version Execution **9**
7.1 The shared-memory ring 9
7.2 Passing file descriptors 9
7.3 Promotion 10
7.4 Asyncio 10

8 Evaluation **10**
8.1 FastAPI: compute-heavy server 11
8.2 Tornado HTTP: moderate I/O 11
8.3 KV store: read/write mix 11
8.4 pyperformance suite 12
8.5 Asyncio KV server 14
8.6 Failover 15
8.7 Replay fidelity 16

9 Conclusion **16**

1 Introduction

A program can fail even when its source code and initial arguments have not changed. A server may receive requests in a different order, a file may change between two reads, or a timeout may expire at a different point in a computation. Repeating the same command does not necessarily repeat these conditions. For a developer investigating an intermittent failure, the missing information is often the sequence of observations that led the program to that failure. For a service operator, the related problem is keeping a second process in a state from which it can continue the first process's work.

Record-replay (R/R) addresses the first problem by capturing the nondeterministic inputs observed during an execution. These include bytes read from files and sockets, clock readings, random inputs, and the outcomes of operations that may fail. During replay, intercepted operations return their recorded outcomes instead of obtaining new ones from the environment. If all relevant nondeterminism is captured and the program starts from compatible initial state, replay can reproduce the computation that followed those observations. This condition is essential: recording network traffic alone, for example, does not reproduce an execution whose decisions also depend on time or on the order in which asynchronous operations become ready.

Consider a Python service that occasionally returns the wrong reply after processing several requests. An application log may contain the request identifiers and the final error, yet omit the exact bytes returned by a read or the file metadata used in a branch. Sending the same requests again can produce different read boundaries or encounter a different file. A recording of the relevant operations preserves those observations so that the developer can revisit the failing execution and inspect the state preceding the error. The purpose is to make an observed failure repeatable, without requiring the developer to anticipate which values will matter and add logging for each of them in advance.

The distinction between an input and the outcome of an operation matters in this example. A socket read can return fewer bytes than requested, and a filesystem query can raise an exception instead of returning metadata. Application code may branch on either outcome. Replaying only the final request payload would omit those decisions. Similarly, restoring a saved application state identifies a point from which to restart, but does not determine the external observations that follow it. R/R needs an ordered account of the relevant operations and their outcomes. That account is useful only to the extent that the interception mechanism captures the sources of variation on which the execution depends.

Multi-version execution (MVX) addresses the second problem by running multiple executions, or variants, concurrently. A leader interacts with the environment and publishes the outcomes of its operations; followers consume those outcomes while executing their own program code. In this sense MVX is online R/R: the record stream is consumed while it is being produced, rather than saved only for a later run. The executions must agree on the operations represented in the stream. Followers can lag behind the leader, and any takeover must account for that lag and for operations already visible to clients.

A motivating use case is a Python key–value server with long-lived TCP connections. Restarting a crashed server does not by itself restore the contents of its in-memory store or the connections through which clients were using it. An MVX follower can reconstruct the store by executing the same request-handling code on the leader's recorded inputs. To continue serving existing clients after a crash, it also needs access to the live connections and a consistent point at which to resume external I/O. These are separate requirements: matching the leader's application state does not alone establish that a reply will be delivered exactly once. This report evaluates both the ability to retain connections and the correctness of the observed takeovers.

For example, a client can have received a reply even though the follower has not yet learned that the leader sent it. Repeating the send after promotion would produce a duplicate. Conversely, a request already consumed from a socket is no longer available for the follower to

read again. This motivates observing failover from the client's perspective: a process that remains alive after promotion is not sufficient evidence that the service continued correctly. The experiment must check which replies arrived and in what order.

The central design choice for R/R and MVX is where to capture nondeterminism. A native recording tool can observe a Python program by recording the interpreter process that executes it; Mozilla rr [3] provides the native R/R baseline in this report. An alternative is to intercept the operations exposed by the language runtime. JMVX [7] demonstrates this approach for Java by intercepting selected nondeterministic methods near the boundary between Java code and the Java virtual machine (JVM)'s native code. It uses the resulting stream for both recording and concurrent execution. This places the agreement between variants at the level of application-visible operations, allowing runtime activity below that boundary to proceed independently where it does not affect those observations.

This report asks how that design can be adapted to Python. Python is dynamically typed: the types of arguments and results are determined at runtime, and names refer to objects that can be rebound during execution. CPython, the reference implementation used here, executes Python bytecode but also exposes operations implemented directly in C. Python and CPython are therefore distinct in this discussion: the language determines the programming model, while this interpreter's object model and implementation determine where and how PyRR can intercept an operation. Supporting this boundary requires more than translating a Java wrapper into Python syntax.

First, an interception mechanism must reach the callable objects that the application actually uses. The same underlying operation can be available through several bindings, such as `os.stat` and `posix.stat`, and application code can retain its own reference to a callable. Replacing one binding need not replace every route to that operation. Python functions have signatures, but their type annotations are optional and do not provide the uniform, statically typed native-method declarations available in Java. Nor does Python bytecode describe the bodies of functions implemented in C. Some of these functions are methods of built-in types whose attributes cannot be replaced through ordinary Python assignment. Selecting an operation and installing an effective wrapper are thus distinct parts of the problem.

Second, a replayed operation must return something that the Python program can use in the same way as the original result. A byte string can be copied into a log, but a directory iterator, a `stat_result`, and a socket object expose different behavior and state. Their replay representations require knowledge of the individual interfaces. File I/O adds another complication: text decoding and buffering occur above the raw operations that reach the kernel. The chosen interception point must preserve the observations made by these layers without recording the same effect repeatedly. These constraints make the boundary a collection of selected operations with explicit replay behavior, rather than a single generic hook for all Python calls.

Third, matching individual reads and writes is insufficient when the runtime makes other decisions that affect execution. Random-number initialization can change subsequent application choices. In an `asyncio` server, the order in which readiness events are processed can change which task runs next and which operation it performs. Two processes can receive the same request bytes yet issue their later calls in different orders. A Python adaptation must therefore consider these sources of nondeterminism alongside the ordinary file and socket interfaces. Section 2 develops the difficulties that motivate the design.

We present PyRR, a prototype that extends JMVX's boundary-interception design to R/R and MVX for CPython. Its central contribution is this adaptation to Python's dynamic bindings, C-implemented operations, and runtime-specific nondeterminism. The prototype installs Python wrappers around a selected set of operations and supplies the serialization and reconstruction behavior required for their results. It runs on the stock CPython 3.12 build used in the evaluation, without modifying the interpreter binary or rewriting application bytecode. The detailed mechanism for intercepting one function is presented in Section 5; the introduction

focuses on why that mechanism is needed and what it enables.

Figure 1 previews the relationship between the two modes. For standalone R/R, the recorder writes operation outcomes to a file and a later execution reads that file. For MVX, the leader publishes outcomes to a shared-memory ring that a follower consumes concurrently. The wrappers and replay behavior serve both modes; changing the transport changes when the recorded execution can be reproduced. The MVX design also passes live socket descriptors to the follower and provides a promotion mechanism, so that continuing the application's execution can be connected to continuing its external communication. The evaluation tests this mechanism with a killed leader and a client that keeps its connection open.

This design also suggests what to measure. Computation between intercepted calls can proceed without a record for each Python instruction, but each intercepted call adds wrapper, encoding, and transport work. The relative cost should therefore depend on how often the application crosses the selected boundary and how much data each call produces. Replay has a different balance: it avoids the original external operation but still decodes and reconstructs its outcome. A cheap, cached metadata query can consequently be less expensive to execute than to replay. The evaluation includes both computation-heavy and I/O-heavy workloads to expose these differences, rather than assuming replay will always be faster.

The evaluation covers key–value servers, FastAPI and Tornado web servers, and workloads drawn from the pyperformance suite [5]. These workloads exercise different balances between application computation and intercepted operations. On compute-bound benchmarks, measured overhead is within 3% of the uninstrumented baseline. Recording adds 15–17% on the web-server workloads. On the synchronous key–value workloads, recording takes 1.34–2.84× the uninstrumented time, depending on value size and write fraction. These results support the usefulness of the approach for the measured workloads, while also showing why a single overhead figure would be misleading.

Pathlib illustrates the cost when intercepted operations dominate execution. Its server measurements reach 6.53× for recording and 7.18× for replay. Repeated, inexpensive filesystem metadata operations leave little application work over which to spread wrapper and serialization costs. Separate local diagnostics identify result encoding and directory-list parsing as substantial contributors. Section 8 distinguishes those diagnostic measurements from the original server comparison. The performance benefit of intercepting at the language boundary therefore depends on both the workload and the cost of representing its observations.

The failover experiments demonstrate follower promotion in all five attempts for each workload. FastAPI and the streaming workload also completed all five attempts correctly without client reconnection. For the key–value server, two of five attempts completed all client rounds correctly, with reported client-side outage estimates of 0.04–0.22 ms. The other three promoted successfully but encountered two reply mismatches and one timeout, consistent with a gap between a socket operation and publication of its log record. These results demonstrate connection-preserving takeover and identify recovery of in-flight operations as the next step toward reliable failover. Section 8.6 distinguishes promotion from clean client completion and reports the evidence for both.

The scope of these results is the selected interfaces and workloads evaluated here. The experiments do not establish coverage of every Python extension or of arbitrary multithreaded executions. Within that scope, PyRR provides a concrete account of how language-boundary R/R and MVX can be implemented for CPython, what the adaptation costs, and where its current failover mechanism falls short.

The rest of the report is organised as follows. Section 2 describes the difficulties specific to CPython, Section 3 reviews related work, Section 4 gives an overview of the design, Section 5 explains how a single function is intercepted, Sections 6 and 7 describe the record-replay and multi-version execution modes, Section 8 presents the evaluation, and Section 9 concludes.

2 Problem

The problem is to reproduce the observations that drive a Python program's execution, either in a later replay or in a concurrent MVX follower, while using an unmodified CPython interpreter. Recording file and network contents alone is insufficient: the program also observes operation results, exceptions, changes to buffers, and the order in which events become available. The interception boundary must capture these observations while preserving the behavior of the Python objects through which they are exposed. CPython presents three related difficulties at this boundary.

1. C functions in immutable built-in types. CPython compiles Python source to bytecode, but operations such as `os.stat`, `socket.socket.recv`, and `io.FileIO.readinto` are implemented in C and exposed as Python callable objects. Their implementations have no Python bytecode to rewrite. Instrumenting Python call sites is a different possibility, but must account for dynamically resolved calls and aliases to the same callable.

The restrictions also depend on where a callable is exposed. A module's dictionary is writable, whereas built-in types such as `_io.FileIO` and `_socket.socket` reject ordinary assignment to their methods. For example, `_io.FileIO.readinto = f` raises `TypeError`. Thus, locating a C-implemented operation does not establish that ordinary Python-level replacement can intercept it. The challenge is to cover the required module functions and built-in methods without changing the interpreter binary.

2. Layered I/O stack. A Python I/O call need not correspond to one system call. For instance, `f.read(n)` on a text file can involve text decoding, a buffered reader, and a raw `FileIO` read before reaching the `read(2)` system call. A later read may be satisfied entirely from buffered data. Knowing that an operation is implemented in C does not identify which layer produces the external observation or which state the layers above it retain.

This creates a choice of replay boundary. Reproducing a high-level result must also preserve the object behavior that later calls rely on, such as the file position and buffered contents. At a lower boundary, the replayed outcome may include both a return value and a change to a caller-supplied buffer, as with `readinto`. Recording at several nested layers introduces a further risk: the recorded sequence can contain inner operations that replay of an outer operation does not execute. The challenge is to select a consistent boundary that preserves observable state and operation ordering across recording and replay.

3. Internal nondeterminism. Some inputs that influence execution enter through runtime initialization or scheduling rather than through application file and socket reads. CPython randomises string hashes per process, and the `random` module's default seeding can obtain entropy through a C path without calling the Python-level `os.urandom` binding. In an `asyncio` server, readiness events influence which callbacks run next; clock readings also affect timer-driven execution.

Consequently, two executions can observe identical file and socket bytes yet make different application decisions or issue their next operations in different orders. Capturing the bytes alone does not ensure that replay follows the recorded sequence. The challenge is to identify the initialization values and scheduling observations relevant to the supported workloads, in addition to their direct I/O operations.

3 Related Work

Mozilla rr [3] records unmodified Linux processes using `ptrace`, hardware performance counters, and system-call buffering, while running one application thread at a time. Its paper focuses

on recording and replay for debugging, rather than concurrent MVX or connection-preserving takeover. PinPlay [4] uses dynamic binary instrumentation to capture executions of native programs for deterministic replay and reproducible analysis.

JMVX [7] is the closest architectural precedent for PyRR. It instruments selected Java APIs near the Java/native boundary and uses a shared-memory circular buffer to communicate between JVMs. For its evaluated workloads, the paper reports average recording overheads of 8% with multithreading support disabled and 25% with it enabled; the corresponding MVX overheads are 5% and 47%. Its Java 8 prototype instruments the class library using ASM and describes leader promotion and demotion. Its evaluation focuses on performance and replay correctness, rather than client-observed crash-takeover measurements.

Chronicler [1] captures nondeterministic inputs at the Java library level to reproduce field failures. Its evaluation reports overhead below 10% in most cases and a worst case of 86%. These measurements come from that paper's workloads and are not a direct comparison with the Python workloads evaluated here. VARAN [2] uses selective binary rewriting and event streaming for native N-version execution. It supports multithreaded applications and evaluates transparent failover using Redis and Lighttpd bugs. In its Redis leader-failure experiment, the triggering command's latency rises from 42.36 to 122.62 μs; this is a request-latency measurement for that experiment, not a universal takeover cost.

Sinatra [6] applies MVX to JavaScript programs to preserve page state across commercial-browser updates. It captures browser events and other nondeterministic inputs using JavaScript-level function replacement, then reconstructs application state in the updated browser through re-execution. It therefore provides a precedent for interception in a dynamic language on an unmodified runtime. PyRR applies this general idea to CPython's file and socket interfaces, where C-implemented methods and immutable built-in types require additional handling. Its evaluation distinguishes successful promotion from correct completion of a client workload after a crash.

4 Design Overview

PyRR follows the architecture of JMVX (Figure 1). It intercepts a fixed set of functions at the boundary between CPython and the kernel. In record mode each intercepted call runs for real and its result is appended to a log; in replay mode the call is not executed and its result is read from the log instead. The two modes use the same wrappers and differ only in the transport. For standalone R/R the log is a file, written during one run and read during a later one. For MVX the recorder is the leader and the replayer is the follower, both started by a launcher script, and the log is a single-producer single-consumer ring in shared memory that the follower reads while the leader is still writing.

5 Intercepting a Function

This section follows one function, `os.stat`, from its declaration to its replay.

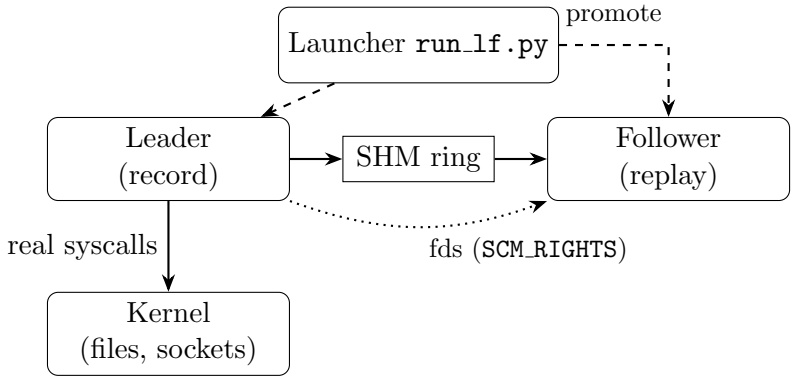

Figure 1: PyRR in MVX mode. The leader executes intercepted calls for real and publishes their results to the shared-memory ring; the follower consumes them instead of executing the calls. Socket descriptors are passed to the follower so that it can take over the leader's connections. In standalone R/R the ring is replaced by a log file.

5.1 Declaring the function

Every intercepted function has one entry in a table, `SYSCALL_REGISTRY`, keyed by (module, function). Figure 2 shows the entry for `stat`.

```
("posix", "stat"): {
    "target":         "posix",       # where the binding lives
    "function_attr":  "stat",        # which binding to replace
    "serializer":     serialize_stat_result,
    "replay_decoder": StatResultReplayDecoder(),
    "buffer_capture": None,
    "active":         True,
},
```

Figure 2: Registry entry specifying the patch target, serializer, and replay decoder for `posix.stat`.

The serializer turns the returned `os.stat_result` into something that can be written to the log, and the replay decoder rebuilds an `os.stat_result` from it. A second, otherwise identical entry with target `os` exists because the `os` module copies the binding when it is imported, so a program can reach the same C function through either name. Functions that fill a caller-supplied buffer, such as `readinto`, also declare a `buffer_capture` that copies the filled bytes after the call. Adding a function to PyRR means adding one entry here.

5.2 Replacing the binding

PyRR intercepts selected operations by replacing their entries in module and type dictionaries, without rewriting their implementations. Sinatra uses a related technique in JavaScript: it replaces functions such as `addEventListener` and `setInterval` to intercept events and timer callbacks [6, Sections 3.3 and 3.5]. The shared idea is replacing a callable binding with an interception wrapper. PyRR additionally needs access to the dictionaries behind CPython's immutable built-in types, as described below.

At startup, `apply_patch` looks up the original function, builds a wrapper around it (Section 5.4), and installs the wrapper under the same name. For a module such as `posix` the module's dictionary is an ordinary Python `dict`, so installing the wrapper is a plain assignment. Direct assignment to a built-in type is rejected (Figure 3).

```
>>> _io.FileIO.readinto = wrapper
TypeError: cannot set 'readinto' attribute of immutable type ...
>>> vars(_io.FileIO)["readinto"] = wrapper
TypeError: 'mappingproxy' object does not support item assignment
```

Figure 3: Direct assignment to `FileIO.readinto` and to the type's dictionary view is rejected.

`vars(_io.FileIO)` returns a read-only view (`mappingproxy`) of the type's internal dictionary. The view refuses assignment, but it does not hide the dictionary completely. When the view is compared with `==`, it passes the comparison on to the dictionary it wraps, and if the dictionary does not know how to compare itself with the other operand, Python calls the other operand's `__eq__` with the dictionary as its argument. An object whose `__eq__` writes into its argument therefore receives the real, writable dictionary, as shown in Figure 4.

```
class Leaker:
    def __eq__(self, other):      # 'other' is the type's real dict
        other["readinto"] = wrapper
        return False

vars(_io.FileIO) == Leaker()      # step 1: view forwards to the dict
                                  # step 2: dict defers to Leaker.__eq__
                                  # step 3: Leaker writes the wrapper
```

Figure 4: The comparison callback installs a wrapper in the type's underlying dictionary.

After this comparison, `_io.FileIO.__dict__["readinto"]` is the wrapper. PyRR uses the same code for modules and types, so every patch goes through this path.

5.3 Caches

Writing the wrapper into the dictionary is not quite enough, because CPython caches lookups. For modules, a write to the dictionary changes the dictionary's version number, which invalidates the interpreter's inline caches by itself. Types have a second, per-type method cache that is not tied to the dictionary version, so after patching a type `apply_patch` calls `ctypes.pythonapi.PyType_Modified()` on it. Without that call, code that had already looked up `FileIO.readinto` keeps getting the original function.

5.4 The wrapper

Every patched function is replaced by a closure, `wrap_f`, built by `create_wrapper`. Figure 5 shows its main paths, leaving out the special cases described below.

The first check prevents PyRR from recording its own I/O. The second, `skiprr`, walks at most ten frames up the stack to recognise calls issued by the import machinery (`importlib`, `tokenize`, `linecache`) and by installed packages' metadata lookups, and skips writes to standard output and to PyRR's log files. Everything that depends only on (module, function) is computed once, when the wrapper is built, because `wrap_f` is on the path of every intercepted call. The mode is the exception: it is read on each call from a shared `role` object, so that promoting a follower (Section 7.3) can switch every wrapper at once with a single assignment.

5.5 Walking through one call

With the entry declared and the wrapper installed, a call to `os.stat("data.txt")` proceeds as follows.

```
def wrap_f(*args, **kwargs):
    # 1. Calls made by PyRR itself go straight through.
    if replayer._in_replay or recorder._in_record:
        return func(*args, **kwargs)
    # 2. Calls the program did not make (imports, PyRR's own
    #    log files, stdout/stderr) are not recorded either.
    if skiprr(args, function_name, func):
        return func(*args, **kwargs)
    # 3. Replay: return the logged value, never call func.
    if role.is_replay:
        return replayer.replay_call_v2(*args,
                   module_name=module_name,
                   function_name=function_name, **kwargs)
    # 4. Record: call func, then log what it returned.
    ret = func(*args, **kwargs)
    recorder.record_call(*args, ret=ret,
                         module_name=module_name,
                         function_name=function_name, **kwargs)
    return ret
```

Figure 5: Simplified wrapper dispatch: bypass internal and excluded calls, replay recorded results, or execute and record a call.

1. On the leader, `wrap_f` passes both checks, calls the real `stat`, and gets an `os.stat_result`.
2. `serialize_stat_result` turns it into a plain record, and `record_call` appends the record, tagged with (`posix`, `stat`), to the file or ring.
3. On the follower, the same line of the program calls the same wrapper. `replay_call_v2` takes the next record, checks that it is also tagged (`posix`, `stat`), and reports a divergence if it is not.
4. `StatResultReplayDecoder` rebuilds an `os.stat_result` from the record and returns it. The real `stat` is never called on the follower, so the follower sees the leader's file size and timestamps even if its own copy of the file differs.

5.6 Calls that need extra handling

A few functions do not fit this pattern. `io.open` builds a text wrapper around a buffered writer around a `FileIO`, and calls `FileIO.tell` along the way. If both the inner `tell` and the outer `open` were recorded, the leader would log two entries where the follower, whose `open` is replayed as a whole, consumes one. PyRR marks `io.open` as atomic: inner calls are not recorded while it runs, and it produces one entry. `tempfile.mkdtemp` and `mkstemp` are instead executed for real on both processes with interception disabled, so that leader and follower each get their own temporary directory rather than racing on one path.

6 Record and Replay

6.1 Recording

A record holds a global sequence number, the module and function names, a short preview of the arguments, the serialized return value, any captured buffer bytes, and, for a call that raised, the exception type, message, and errno. Arguments longer than 128 bytes are truncated, since they are kept only for debugging.

Records are encoded in a struct-framed binary format: a fixed header holding the sequence number, the function's index in `SYSCALL_REGISTRY`, and flags, followed by length-prefixed fields. Using the index instead of the names works because leader and follower load the same registry. Byte payloads are stored raw. A newline-delimited JSON encoding is also available for debugging; it was the original format, and profiling showed JSON encoding and hex conversion of buffers taking about 20% of record-mode run time, which is why the binary format was added.

In standalone R/R the frames are written to a file with one vectored `os.writev` per 256 KB or 1024 frames, whichever comes first.

6.2 Replay

`replay_call_v2` takes the next record and compares its (module, function) with the call being made. A mismatch means the replayed run has diverged from the recorded one, and PyRR stops with a report showing the requested call, the recorded call, and the previous ten records. If the record holds an exception, PyRR rebuilds it by class name, including `OSError` subclasses and `socket.gaierror`, restores the errno, and raises it. Otherwise the registry's decoder rebuilds the return value.

Replay substitutes data, not the filesystem. Creating and deleting files and directories are not intercepted; both runs perform them for real, each in its own working directory. What is replayed is the data the program observes: bytes read, `stat` results, directory listings, socket payloads, and random seeds. This split is forced by the `tp_init` limitation from Section 2: the file object itself cannot be faked, so it has to exist.

6.3 Seeding the random number generator

Importing `tempfile` seeds Python's global Mersenne Twister. PyRR therefore applies its patches in two phases. Modules whose import does not touch `random` (`posix`, `_io`, `_random`, `os`) are patched first; then recording or replay is switched on; then modules like `tempfile` are patched. In this order the seeding happens after recording has started, so the seed is recorded on the leader and replayed on the follower, and both processes end up with the same generator state.

7 Multi-Version Execution

7.1 The shared-memory ring

In MVX mode the log is a single `mmap`ed file in `/dev/shm` (16 MiB by default) shared by the leader and the follower. The first 64 bytes hold the head (the leader's write position), the tail (the follower's read position), and a done flag; the rest holds length-prefixed records. A record that does not fit before the end of the buffer is preceded by a padding marker, and both sides wrap to the start.

The ring has no locks. The leader writes a record's bytes first and updates the head afterwards, so whenever the follower sees a new head value, the record behind it is complete. On x86, aligned 8-byte stores are atomic and stores become visible in program order, so no explicit fence is needed; the ring is x86-only as implemented. The same ordering matters for failover: a record interrupted by the leader's death is never visible to the follower. When the ring is full, the leader polls the tail every 100 μs; the follower never blocks the leader otherwise.

7.2 Passing file descriptors

To take over the leader's connections, the follower needs the real sockets, not their recorded results. The launcher creates a Unix socket pair before starting the two processes. Whenever

the leader creates a socket, in `socket.socket()` or `accept`, it sends the new descriptor to the follower over this pair with `SCM_RIGHTS`, and the follower's decoder for the same call receives it and wraps it in a socket object. Both processes then refer to the same kernel socket. The leader numbers each descriptor it sends and writes the number into the ring entry, so the follower can check that it pairs the right descriptor with the right call.

The follower holds these sockets but never uses them while the leader is alive: a `recv` on the follower would take bytes the leader needs, and a `shutdown` would close the shared connection. Since every socket method on the follower is replayed from the log, this holds without additional checks.

7.3 Promotion

The launcher waits on the leader with `waitpid`. If the leader dies abnormally while the follower is still running, the launcher writes one byte into a control pipe that the follower watches. The follower keeps replaying until it has consumed every record the leader published, since those calls really happened. At the first call for which there is no record and the promotion byte is set, `replay_call_v2` switches the shared role to record mode and returns a marker that tells `wrap_f` to re-dispatch the call. The call then runs for real on the passed socket, and from then on the follower serves the leader's connections itself.

This design leaves one window open. If the leader completes a system call and dies before the record for it reaches the ring, the follower does not know the call happened. For a write, the follower performs it again and the client receives the reply twice; for a read, the bytes have already been taken from the socket by the leader, and the follower waits for data that will not arrive. Section 8.6 measures how often this happens.

7.4 Asyncio

The asyncio event loop decides which task runs next from the result of `selector.select()`, which reports which sockets are ready. PyRR intercepts `select` on `selectors.EpollSelector`: the leader performs the real `epoll` wait and records the ready list, and the follower replays it. Given the same ready lists and the same socket results, the follower's loop schedules tasks in the same order as the leader's. Registration calls (`register`, `unregister`, `modify`) update the follower's selector bookkeeping, and when descriptors are passed they also update the follower's real `epoll` set, so that after promotion the first real `select()` already watches the right sockets. Timer-driven loops are not yet supported, because `loop.time()` reads the real clock and a follower that replays `select()` instantly can run more loop iterations than were recorded.

8 Evaluation

Except for the separately identified local Pathlib diagnostics in Table 6, all experiments ran on an Intel Xeon Gold 5318H server (four NUMA nodes, 72 physical cores) with the `powersave` governor on every core. Leader and follower are pinned to separate cores on NUMA node 3 and the client runs on node 2. All runs use CPython 3.12 and the binary log format. In the server evaluation, each configuration runs five times after a discarded warm-up, and we report medians. Every benchmark checksums its output, and a result is accepted only if the checksum matches the uninstrumented run.

Each workload is measured in the following modes. *Off* runs the program without PyRR and is the baseline for all ratios. *Record* runs it under PyRR with the log written to a file. *Replay* runs it again from that file without performing the recorded calls, which is why it is often faster than the baseline. *MVX-L* and *MVX-F* are the leader and follower of an MVX pair connected by the ring; the follower is the replay measurement for asyncio workloads, which have no standalone replay mode. *rr-Rec* and *rr-Rep* are Mozilla rr 5.7.0 recording and replaying

the same workload on the same cores, with ratios taken against rr's own uninstrumented run. rr virtualises the clock during replay, so its replay time is measured externally and includes startup.

8.1 FastAPI: compute-heavy server

The pyperformance `fastapi` benchmark is a single-threaded FastAPI/uvicorn server answering `GET /items/{id}`, where the per-request work is validation, Pydantic serialisation, and JSON encoding rather than socket I/O. Eight client processes with 15 connections each send 2 400 requests per repetition.

Table 1: FastAPI, 2 400 requests per repetition, 8×15 client connections.

Mode	Median (s)	Overhead	Req/s
Off	2.956	1.00×	812
Record	3.405	1.15×	705
MVX-L	3.498	1.18×	686
MVX-F	3.498	1.18×	686
rr-Rec	4.438	1.50×	541
rr-Rep	5.309	1.79×	452

Recording adds 15% and MVX 18% (Table 1); the interception cost is small next to the framework's work per request.

8.2 Tornado HTTP: moderate I/O

The pyperformance Tornado benchmark is an asyncio HTTP server returning bodies of about 60 KB with several `send` calls per request, driven over 150 concurrent connections for 3 000 requests per run.

Table 2: Tornado HTTP, 3 000 requests per repetition, 60 KB response bodies.

Mode	Median (s)	Overhead	Req/s
Off	6.018	1.00×	499
Record	7.016	1.17×	428
Replay	5.204	0.86×	577
MVX-L	7.052	1.17×	425
MVX-F	7.050	1.17×	426
rr-Rec	35.841	5.99×	84
rr-Rep	6.156	1.03×	487

Recording costs 17% and MVX the same (Table 2). Replay is faster than the baseline because replayed socket calls return from the log without a network round trip. rr's recording cost is 6× here, because each of the many `send` calls pays a `ptrace` stop.

8.3 KV store: read/write mix

A single-threaded key–value store serves GET and SET requests over TCP, and the fraction of SETs is varied from 1% to 99%. A SET carries its value into the log, so with 64 KiB values PyRR's record overhead grows from 1.67× at 1% SET to 2.84× at 99% (Table 3). rr's overhead stays roughly flat across the mix because it pays the same `ptrace` stop for every system call regardless of its size. The rr sweep did not include the 2% point.

Table 3: KV store, 64 KiB values, overhead relative to the uninstrumented run.

	PyRR				Mozilla rr	
SET%	Record	Replay	MVX-L	MVX-F	rr-Rec	rr-Rep
99	2.84×	1.19×	3.11×	2.83×	3.69×	1.62×
90	2.68×	1.09×	2.88×	2.61×	3.55×	1.56×
70	2.25×	0.85×	2.39×	2.12×	3.13×	1.41×
50	1.97×	0.62×	1.93×	1.68×	2.74×	1.25×
30	1.77×	0.54×	1.70×	1.39×	2.89×	1.40×
10	1.65×	0.36×	1.42×	1.03×	3.03×	1.56×
5	1.61×	0.30×	1.39×	0.98×	3.04×	1.56×
2	1.64×	0.26×	1.37×	0.94×	—	—
1	1.67×	0.24×	1.37×	0.93×	3.06×	1.63×

With 4 KiB values PyRR's overhead falls, since each record carries fewer bytes, while rr's rises to about 7×, since its fixed cost per system call is now spread over less work (Table 4).

Table 4: KV store, 4 KiB values, overhead relative to the uninstrumented run.

	PyRR				Mozilla rr	
SET%	Record	Replay	MVX-L	MVX-F	rr-Rec	rr-Rep
99	1.63×	0.62×	1.89×	1.02×	6.98×	3.91×
90	1.62×	0.61×	1.87×	1.00×	7.08×	3.90×
70	1.59×	0.58×	1.79×	0.94×	7.18×	3.94×
50	1.54×	0.55×	1.73×	0.90×	7.29×	4.05×
30	1.46×	0.50×	1.66×	0.84×	7.28×	4.01×
10	1.39×	0.47×	1.62×	0.78×	7.37×	4.06×
5	1.36×	0.46×	1.56×	0.74×	7.37×	4.07×
2	1.36×	0.45×	1.57×	0.74×	—	—
1	1.34×	0.45×	1.56×	0.74×	7.21×	4.14×

8.4 pyperformance suite

Table 5 lists eight benchmarks from the pyperformance suite, ordered from pure computation to heavy filesystem metadata access; Whoosh and TinyDB have separate phases and are reported in Tables 7 and 8. On the compute-bound benchmarks every mode is within 3% of the baseline, and overhead appears only as the number of intercepted calls grows.

Pathlib is the outlier, at 6.53× for record and 7.18× for replay. Its repeated metadata operations leave little application work over which to spread the interception cost. With 2 000 files, each iteration issues 5 002 `stat` calls, two `listdir` calls, and two `scandir` calls: 750 900 intercepted calls over the 150 iterations used in the server evaluation. Although the outer record format is binary, `stat` results are serialised as JSON and directory listings as Python literal strings. Replay parses those strings with `ast.literal_eval` before reconstructing the directory entries.

Pathlib has no reported MVX measurement.

To investigate the Pathlib result, we ran a separate local diagnostic on October 30, 2026, using CPython 3.12.3, logical CPU 8, and the `powersave` governor. These runs used 2 000 files and 20 iterations. Table 6 reports medians of three unprofiled repetitions after one discarded

Table 5: pyperformance benchmarks, ordered by increasing I/O density. Checksums are identical across all modes, including rr replay.

		PyRR				Mozilla rr	
Benchmark	Off (s)	Record	Replay	MVX-L	MVX-F	rr-Rec	rr-Rep
raytrace	11.55	1.00×	1.00×	1.00×	0.99×	0.99×	1.00×
nbody	16.59	0.98×	0.99×	0.97×	0.98×	1.01×	1.01×
json_loads	14.47	1.02×	1.01×	1.01×	1.01×	1.01×	1.01×
pyflakes	13.97	1.01×	0.99×	1.03×	1.01×	1.08×	1.04×
pyflate	5.05	1.00×	1.00×	1.01×	1.00×	1.04×	1.03×
docutils	28.61	1.01×	1.01×	1.02×	1.02×	1.05×	1.01×
dulwich_log	3.61	3.38×	2.01×	2.96×	2.90×	5.54×	2.28×
pathlib	12.87	6.53×	7.18×	—	—	1.39×	0.85×

warm-up, with variant order rotated between rounds. Ratios use the local uninstrumented median of 0.3119 s. They are distinct from the Xeon server measurements in Table 5.

Table 6: Local Pathlib serialization diagnostics, 2 000 files and 20 iterations. Changes were applied only in the diagnostic processes; the runtime source was unchanged.

Configuration	Record (s)	Ratio	Replay (s)	Ratio
Current implementation	1.5900	5.10×	1.9842	6.36×
JSON directory listings	1.5896	5.10×	1.2059	3.87×
Binary stat payloads	1.3288	4.26×	1.5599	5.00×
Both changes	1.3129	4.21×	0.8707	2.79×

Replacing only the directory-list serializers and decoders with `json.dumps` and `json.loads` reduced replay time by 39%, from 1.9842 s to 1.2059 s, while leaving recording time approximately unchanged. This made replay faster than recording, which identifies literal parsing as a substantial cause of the additional replay cost. A separate diagnostic replaced JSON stat payloads with `marshal`-encoded stat tuples and timestamp fields; combining the two changes reduced replay time to 0.8707 s. Every configuration produced the same workload checksum and consumed or produced 100 120 records in the timed loop.

Separate ten-iteration profiles support this attribution. Literal parsing accounted for approximately 21% of profiled replay time, and `skiprr`'s checks accounted for approximately 25% of both record and replay time. The path checks issued two real `getcwd` calls per intercepted operation in this workload. These profile shares are diagnostic, since profiling perturbs timing; the table uses unprofiled runs. Setup and cleanup were outside the timed interval. The diagnostic harness bypassed interception during cleanup after a separate `rmtree` failure compared recorded `lstat` metadata with the recreated directory's real `fstat` metadata. The measurements therefore isolate the timed workload and do not establish end-to-end cleanup correctness, general codec correctness, or MVX performance.

Whoosh is a full-text search engine written in Python, and TinyDB is a document database stored in a single JSON file. Under record, Whoosh's indexing phase costs 1.19× and its search phase 1.38×; TinyDB's load costs 1.36× and its queries and updates about 1.20×. On TinyDB PyRR and rr cost the same.

Table 7: Whoosh, indexing and search phases. rr replay is omitted because its virtualised clock gives no per-phase timing.

Mode	Index (s)	Ratio	Search (s)	Ratio
Off	6.259	1.00×	9.287	1.00×
Record	7.430	1.19×	12.799	1.38×
Replay	6.884	1.10×	11.331	1.22×
MVX-L	7.663	1.22×	13.123	1.41×
MVX-F	7.455	1.19×	13.123	1.41×
rr-Rec	9.156	1.46×	13.159	1.42×

Table 8: TinyDB, load, query, and update phases.

Mode	Load (s)	Ratio	Query (s)	Ratio	Update (s)	Ratio
Off	1.086	1.00×	3.463	1.00×	0.760	1.00×
Record	1.473	1.36×	4.161	1.20×	0.921	1.21×
Replay	1.191	1.10×	3.824	1.10×	0.796	1.05×
MVX-L	1.508	1.39×	4.179	1.21×	0.902	1.19×
MVX-F	1.294	1.19×	4.178	1.21×	0.902	1.19×
rr-Rec	1.463	1.36×	4.097	1.18×	0.891	1.18×

8.5 Asyncio KV server

The asyncio KV server handles all connections in one event loop. The client varies the number of connections from 1 to 1000, with 20 000 operations per run and 64-byte values. PyRR's overhead stays between 2.2× and 2.9× across the range (Table 9), while rr's grows with the number of connections, since every readiness check is a separate system call that pays a `ptrace` stop. At 1000 connections rr completed only two of six recording attempts.

Table 9: Asyncio KV server, 1 to 1000 concurrent connections. †Median of the two rr recordings that completed; the other four ended in client timeouts. PyRR completed every repetition.

		PyRR			Mozilla rr	
Conns	Off (s)	Record	MVX-L	MVX-F	rr-Rec	rr-Rep
1	3.628	2.26×	2.55×	2.51×	5.90×	5.12×
8	2.020	2.34×	2.55×	2.49×	6.75×	6.31×
64	1.864	2.20×	2.36×	2.29×	7.08×	6.55×
256	1.816	2.31×	2.48×	2.42×	7.21×	6.41×
1000	1.582	2.59×	2.87×	2.80×	26.0×†	5.92×

Varying the write fraction on the same server (64 connections, 1000 operations, 4 KiB values) gives an almost flat curve (Table 10). Record stays between 2.22× and 2.34× and the leader between 2.37× and 2.55×, because every operation goes through the intercepted `select()` whether it reads or writes. The follower costs about 1.6×, since it reads ready lists from the ring instead of waiting on `epoll`.

Table 10: Asyncio KV server, varying SET% with 64 connections, 1000 operations, and 4 KiB values.

		PyRR		
SET%	Off (s)	Record	MVX-L	MVX-F
90	0.117	2.22×	2.40×	1.62×
70	0.118	2.22×	2.37×	1.57×
50	0.114	2.28×	2.48×	1.63×
30	0.114	2.27×	2.46×	1.60×
20	0.111	2.32×	2.51×	1.63×
10	0.110	2.32×	2.51×	1.63×
5	0.109	2.33×	2.53×	1.63×
2	0.108	2.34×	2.55×	1.65×

8.6 Failover

The experiments demonstrate follower promotion in all five attempts for each workload, and complete client-visible continuity in all five FastAPI and all five SSE attempts. The KV workload distinguishes these two outcomes: all five followers promoted, and two attempts completed all 20 000 SET/GET rounds correctly on the existing connection (Table 11).

In each experiment we send `SIGKILL` to the leader during back-to-back requests or an active token stream. The client keeps its existing connection. We count *promotion* when the launcher logs the leader's death and the follower logs its transition to real execution. A *clean completion* additionally requires the full client workload to finish with the expected replies or tokens, in order and without detected duplicates, gaps, or protocol errors. Every attempt remains in the denominator, including failed attempts.

Timing summaries use clean completions only. The reported *outage* is the interval from the driver's kill timestamp to the next client-observed reply. The *excess* is the round-trip time of the request spanning that timestamp minus the median round-trip time of the others. Failover executes on the server, but these timings reflect delays observed by the client, including server processing, network delay, and scheduling. They do not directly measure the server's internal promotion duration. For the streaming (SSE) workload, the *stall* is the gap between the two tokens on either side of the kill minus the steady 30.9 ms token interval. The steady-state reference values are a 0.20 ms KV round-trip time, a 1.47 ms FastAPI round-trip time, and the 30.9 ms SSE token interval.

Table 11: Failover outcomes, five attempts per workload. *Promoted* means the follower switched from replay to real execution and took the leader's role. *Clean completion* means the entire client workload finished correctly on the same connection, with all expected replies in order and no detected duplicates, missing replies, or timeouts. *Outage* is the time from the driver's kill timestamp to the next client-observed reply. *Excess* is the latency of the request spanning that timestamp minus the median latency of the other requests. *Stall* is the gap between streaming tokens around that timestamp minus the normal token interval. Failover executes on the server; these timings reflect delays observed by the client, rather than the server's internal promotion duration. Ranges cover clean completions only.

Workload	Promoted	Clean completions	Outage	Excess / stall
KV store (20 000 rounds)	5/5	2/5	0.04–0.22 ms	excess 0.06–0.08 ms
FastAPI (5000 POSTs)	5/5	5/5	—	excess 6.65–6.94 ms
SSE stream (120 tokens)	5/5	5/5	—	stall 0.7–18.8 ms

The FastAPI and SSE results demonstrate that the follower can continue application execution and external communication without the client reconnecting. Every measured SSE stall remained below one steady-state token interval. In the KV experiments, repetitions 3 and 5 completed all rounds correctly. Repetitions 1 and 4 received an unexpected second `OK` where the client expected `VALUE <sequence>`; repetition 2 timed out. Promotion was confirmed in all three of these attempts, so their unsuccessful completion concerns the continuity of in-flight operations.

These symptoms are consistent with the operation-to-log publication window described in Section 7.3: a reply sent before its record is published can be repeated after promotion, while a request consumed before its read is recorded can leave the follower waiting for unavailable bytes. The archived logs establish the client outcomes and promotion events; they do not directly trace the precise syscall interruption point. The results identify recovery of in-flight operations as the next implementation task.

Together, the experiments demonstrate connection-preserving takeover and delimit its current reliability: promotion succeeded throughout the measured runs, while clean completion depended on the workload and interruption point. Five repetitions per workload establish these observed outcomes, not a general success probability. Guaranteeing correct completion across arbitrary interruption points remains future work. rr has no live standby and cannot take over a connection, so Table 11 has no rr column.

8.7 Replay fidelity

For the successful non-failover performance runs reported in the tables, output checksums matched the uninstrumented baseline. This validates the recorded observations for those completed workloads. Crash-injection experiments assess a separate property: continuity when the leader stops during execution. Their promotion and clean completion counts are reported in Section 8.6.

9 Conclusion

PyRR demonstrates record-replay and multi-version execution for the evaluated Python workloads on an unmodified CPython interpreter. Its measured overhead is within 3% on compute-bound benchmarks, and recording adds 15–17% on the web servers. The cost depends on the frequency and representation of intercepted operations: the synchronous KV workloads take 1.34–2.84× the uninstrumented time to record, while Pathlib reaches 6.53× for recording and 7.18× for replay in the server measurements.

The failover experiments demonstrate promotion in five of five attempts per workload and continued service over existing client connections. FastAPI and SSE each completed five of five attempts cleanly; the KV workload completed two of five, with two reply mismatches and one timeout in the remaining attempts. The distinction between promotion and clean completion makes both the demonstrated capability and its present limit explicit. Recovery of in-flight I/O whose effects are not yet represented in the log is the main next step toward reliable takeover across arbitrary interruption points.

References

[1] J. Bell, N. Sarda, and G. Kaiser. Chronicler: Lightweight recording to reproduce field failures. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE)*, pages 362–371. IEEE, 2013. https://doi.org/10.1109/ICSE.2013.6606582.

[2] P. Hosek and C. Cadar. VARAN the unbelievable: An efficient N-version execution framework. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 339–353. ACM, 2015. https://doi.org/10.1145/2694344.2694390.

[3] R. O'Callahan, C. Jones, N. Froyd, K. Huey, A. Noll, and N. Partush. Engineering record and replay for deployability. In *2017 USENIX Annual Technical Conference (USENIX ATC)*, pages 377–389. USENIX Association, 2017. https://www.usenix.org/conference/atc17/technical-sessions/presentation/ocallahan.

[4] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie. PinPlay: A framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–11. ACM, 2010. https://doi.org/10.1145/1772954.1772958.

[5] The pyperformance contributors. Python Performance Benchmark Suite. https://github.com/python/pyperformance. Accessed October 31, 2026.

[6] U. Rumsevicius, S. Venkateshwaran, E. Kidane, and L. Pina. Sinatra: Stateful Instantaneous Updates for Commercial Browsers Through Multi-Version eXecution. In *37th European Conference on Object-Oriented Programming (ECOOP)*, volume 263 of *LIPIcs*, pages 26:1–26:29. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2023. https://doi.org/10.4230/LIPIcs.ECOOP.2023.26.

[7] D. Schwartz, A. Kowshik, and L. Pina. Jmvx: Fast multi-threaded multi-version execution and record-replay for managed languages. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA2), Article 329, pages 1–29, October 2024. https://doi.org/10.1145/3689769.