

IMPERIAL

MSc Individual Project

Department of Computing

Imperial College London

A Semantics-Aware Sandbox for Efficient Isolated Execution

Author:

Yuzhe Shi

Supervisor:

Marios Kogias

05 September 2026

Submitted in partial fulfilment of the requirements for the
MSc Computing (Artificial Intelligence and Machine Learning) of Imperial College London

Abstract

Sandboxed code execution repeatedly pays two costs: independent Host reads block the program one by one, and each isolated run rebuilds substantially the same interpreter state. Pysolate addresses these costs with early Host requests and image-backed copy-on-write. Each run keeps its own control flow, permissions, mutable state, workspace and output.

Pysolate runs CPython/WASI Guests under a Go Host that owns external operations and credentials. Prepare-Linearize-Materialize (PLM) can start an eligible request before the program reaches it, but delivers its value or exception only at the original call. Prefix analysis applies this idea while source is arriving. For complete programs, an AST pass moves eligible Host requests earlier, while keeping them after their argument definitions and within the program's existing control flow. Copy-on-write shares a pre-initialised runtime image across Guests, avoiding repeated initialisation and reducing memory use.

The evaluation shows when each mechanism pays off. For complete programs, PLM saves 1.268 s with eight independent 200 ms calls. In one streamed-code example, it removes 1.405 s from a 2.9 s serial path. With eight instances, copy-on-write saves a median 48.434 s across corresponding runs and 534.0 MiB of retained process memory. In a prototype integration with Shimmy, Lambda Feedback's shim layer, image copy-on-write lowers median request latency in a ready worker pool from 154.5 to 1.22 ms and pool memory use from 2,049.6 to 237.7 MiB. PLM is useful for programs with long, independent Host calls. Copy-on-write is useful when many runs share the same runtime setup. Otherwise, ordinary execution remains preferable.

Acknowledgements

I would like to thank Marios Kogias for his supervision and technical guidance, and Robert Chatley for his feedback as second marker. I also thank the Lambda Feedback team for the inspiration and the opportunity to work with them. I am especially grateful to Marcus Messer for carefully reviewing my pull requests, identifying implementation problems and helping with testing.

Contents

1	Introduction	1
1.1	Work that can start earlier	1
1.2	Reusing runtime initialisation	2
1.3	Research questions	2
1.4	Contributions	3
2	Background	4
2.1	Sandboxed program execution	4
2.2	Containers and virtual machines	4
2.3	WebAssembly and WASI	4
2.4	Incremental code generation	5
2.5	External calls and side effects	5
2.6	Synchronous calls and asynchronous requests	6
2.7	Runtime initialisation and copy-on-write	6
3	Related Work	7
3.1	Executing generated programs and tool calls earlier	7
3.2	Speculation and visibility	8
3.3	Effect systems and access control	9
3.4	Reusing initialised execution state	9
4	Design	10
4.1	Sandbox design and interfaces	10
4.2	PLM: starting Host work early	11
4.3	Copy-on-write: reusing initialised memory	15
5	Implementation	17
5.1	Building the Python Guest	17
5.2	Runtime architecture and run lifecycle	19
5.3	PLM implementation	20
5.4	Copy-on-write implementation	23
6	Evaluation	26
6.1	What the timings include	26
6.2	Early Host preparation	27
6.3	Reused runtime state	31
7	Discussion	33
7.1	PLM design choices	33
7.2	Copy-on-write costs and reuse	34
7.3	Choosing and combining the mechanisms	34
8	Limitations	35

8.1 Supported PLM calls and Host rules	35
8.2 Supported packages and dataset selection	35
8.3 Correctness coverage	35
8.4 Copy-on-write workloads and memory measurement	36
8.5 Controlled benchmarks and deployment assumptions	36
9 Conclusion	37
References	38
Appendices	42
A. Whole-program PLM timing source template	42
B. Copy-on-write results	42
C. Controlled replay programs	42
D. Prefix-scaling programs	43
E. Comparison of related systems	44
F. Experimental details	45

List of Figures

Figure 1 Ordinary execution and an early quote request during code generation.	2
Figure 2 Python file operations use WASI, while external operations cross Pysolate's Tool ABI.	10
Figure 3 A synchronous stock-price call waits for the Host and external service before the program continues.	11
Figure 4 Prepare and Materialize overlap work but can return an old quote without a validity check.	12
Figure 5 Linearize either accepts the early quote or starts a new request. Materialize delivers the chosen result.	13
Figure 6 Serial and Pysolate timelines for one Host call that can start early in streamed source.	14
Figure 7 Native code is linked into the Guest, while Python files are packed into its embedded filesystem.	17
Figure 8 Pysolate program path, Host-operation path and sandbox boundaries.	19
Figure 9 Lifecycle of a streamed Pysolate run with prefix and whole-program preparation.	20
Figure 10 The Host initialises Python and finishes warm-up before capturing memory for later Guests.	23
Figure 11 An initialised Python image shared by two instances, with private copies for their writes.	24
Figure 12 Timing intervals for streaming PLM, whole-program PLM, COW fan-out and Shimmy reset. Segment lengths are illustrative.	26
Figure 13 Median end-to-end saving across independent calls and source windows. Positive values favour PLM.	28
Figure 14 Prefix PLM saving by Host-call count. Error bars show 95% bootstrap confidence intervals.	28
Figure 15 Whole-program PLM time difference. Negative = speed-up.	29
Figure 16 Copy-on-write lifecycle time and PSS by instance count.	31
Figure 17 Request latency for new interpreters, full-copy reset and COW reset. PSS compares the full-copy and COW worker pools.	32

List of Tables

Table 1 Streaming Overlap timing differences with reusable workers ready. 27

Table 2 Controlled replay results for five mechanism cases. 30

Table 3 Copy-on-write lifecycle totals and corresponding-run differences by instance fan-out. 42

Table 4 Work moved or reused, retained state and control boundaries in systems related to Pysolate. 44

Table 5 Repeated runs used by the experiments. 45

CHAPTER 1

Introduction

Many services run code supplied by users or generated by language models. Examples include online graders, notebooks and agents that use code to complete tasks. Running submitted code with the service's own permissions can expose its files and shared state. For example, if an online grader reuses one interpreter, one student's code can replace a library function and make another student's program fail. A sandbox restricts access to external resources and gives each run separate program state.

An agent can use external tools by generating a program that calls them as functions. The execution system provides these functions and runs the program in a sandbox. The code runs inside the sandbox, called the **Guest**, while the surrounding **Host** implements the external operations. For example, a stock-trading agent could call `market.recent_price("ACME")` to obtain a quote and `market.buy("ACME", 100)` to place an order. Looking up a quote reads market data. Placing an order changes the trading account.

With ordinary synchronous calls, the program stops at each external call and waits for its result. Can the system start some requests earlier to reduce this waiting, without requiring the agent to write asynchronous code?

1.1 Work that can start earlier

Consider this illustrative program for Acme, whose stock ticker is ACME:

```
price = market.recent_price("ACME")
if should_buy(price):
    market.buy("ACME", 100)
```

Normally, execution waits until the agent has generated the complete program, then waits again for the quote. Yet the price request is already known when the first line arrives. If the tool allows an early request, the Host can fetch the quote while the agent generates the remaining code, as shown in Figure 1. The tool provider must explicitly allow early requests and define when their results remain usable at the original call. Section 4.2.3 explains these rules. The final program still decides whether to buy.

Ordinary execution



Early request



Time →

Figure 1: Ordinary execution and an early quote request during code generation.

Starting the request is different from executing generated statements one by one. The external service starts handling the request immediately, but the program obtains the result at its original call. The quote may be out of date by then, so starting early also requires a check before using it.

Pysolate uses Prepare-Linearize-Materialize (PLM) to split a blocking external call into three steps: issue an eligible request early, check that it is still valid at the original call, then wait for and return its result or raise its error. The implementation applies PLM at two stages:

1. **While code is arriving:** analyse completed top-level calls in the received code and start eligible Host requests while the remaining code is generated. This is called prefix preparation. The complete program runs after source generation finishes.
2. **After the complete code arrives:** use an abstract syntax tree (AST) pass to move request starts earlier, after their argument definitions and within their controlling branches. This can overlap requests with independent program work or other requests.

Both stages can be enabled together. The transformation preserves the program's control flow and the point where values and errors are delivered.

1.2 Reusing runtime initialisation

Pysolate runs CPython compiled to WebAssembly in a new Guest for each run. Initialising the interpreter and importing modules takes time and creates a large amount of runtime state. A short program may modify only a small part of that memory, yet creating a new interpreter repeats the same setup and duplicates the unchanged data.

Copy-on-write (COW) addresses this separate cost. Pysolate initialises and warms up Python once, then saves its linear memory as a read-only image. New Guests share the unchanged pages and obtain private copies of pages they write. This avoids repeated initialisation and allows more isolated Guests to fit in memory.

1.3 Research questions

- **RQ1:** When can external requests start earlier while preserving program control flow and the delivery of values and errors?

- **RQ2:** How can isolated runs safely share initialised runtime state?
- **RQ3:** How do PLM and copy-on-write affect execution time and memory use?

1.4 Contributions

1. **PLM for streamed and complete programs.** The prototype implements early Host requests during code generation and an AST transformation for complete programs. Both can be enabled together to reduce waiting while retaining the synchronous calling interface.
2. **A copy-on-write Python sandbox.** The prototype creates CPython/WASI Guests from a shared pre-initialised memory image. The evaluation measures startup time and memory use, including an application to Shimmy, Lambda Feedback's shim layer.

CHAPTER 2

Background

2.1 Sandboxed program execution

A sandbox runs code with restricted access to files, networks and other resources. Online graders, notebook services, code-executing agents and serverless platforms use sandboxes to isolate runs. The surrounding service decides which resources each run can access.

This report calls the surrounding service the **Host** and the restricted program environment the **Guest**. A **run** is one invocation of a program. One Host can manage multiple Guests while keeping their program state separate.

2.2 Containers and virtual machines

Linux containers isolate processes with kernel namespaces and apply resource limits through control groups. Tools such as Docker add images, configuration and lifecycle management around this operating-system model [1]. Containers share the Host kernel, which makes them relatively lightweight while placing its system-call interface directly beneath the isolated process.

Virtual machines place a separate guest kernel behind hardware virtualisation. Firecracker reduces the device model and startup path for serverless microVMs [2], while gVisor takes a middle path by handling much of the Linux system-call interface in a user-space application kernel [3,4]. These systems differ in startup cost, memory footprint and the state that can be captured or restored.

2.3 WebAssembly and WASI

WebAssembly is a portable instruction format with isolated linear memory and an explicit import interface [5,6]. A module accesses Host resources through the functions made available to it. The WebAssembly System Interface, or WASI, supplies operating-system-like services such as file access [7]. The Host selects the interfaces and filesystem view available to the module.

2.3.1 Command and reactor modules

WASI Preview 1 supports two execution styles. A command runs through `_start` and normally exits when its task finishes. A reactor exposes functions that the Host can call after any `_initialize` entry point has performed module setup. Returning from an exported function leaves the instance available for another call [8].

2.3.2 Compiling native code for WASI

WASI SDK provides a Clang compiler, linker and libraries for building C and C++ code for WebAssembly [9]. This is a cross-compilation process: the tools run on the build machine, but

the code they produce runs in a different environment. A **sysroot** supplies the headers and libraries for that target. The compiler uses the headers and libraries from this sysroot. The build machine's Python interpreter runs build scripts during compilation. The target CPython is compiled for WebAssembly and later runs application code inside the Guest.

2.3.3 Python packages with native code

A Python package can contain both Python files and compiled C or C++ extensions. NumPy uses native extensions for operations on arrays. Running NumPy requires both its Python files and its compiled modules.

On Linux, a CPython extension is normally a shared library with a `.so` suffix [10]. During import, CPython uses `dlopen` to load that library and finds its module initialisation function [11]. A native Linux library contains machine code for its target processor. A Wasm Guest needs an extension compiled for the Wasm target.

Recompiling the extension is only part of the task. Dynamic loading also needs a loader that can connect the new module to the running interpreter. WebAssembly environments can provide this support: Emscripten, for example, implements dynamic linking and `dlopen` [12].

Static linking is an alternative. The extension code is included in the same executable as the interpreter, and its initialisation function is registered before Python starts. Import can then find the module without loading a separate shared library. The package's Python files are still needed. Adding a native package in this model requires rebuilding the executable.

2.4 Incremental code generation

A traditional grading system receives a complete program before execution begins. When a language model generates a program, the source can arrive a piece at a time: one statement may be complete while later statements are still being produced. Waiting for all source code before executing any of it makes generation and execution sequential.

Overlapping these activities requires identifying work that is ready to begin. An incomplete statement cannot be executed, and a complete statement may still depend on earlier results. Side effects and exceptions also matter when deciding whether work can start before the rest of the program is available.

2.5 External calls and side effects

External capabilities can be exposed to a sandboxed program as functions. Reading a stock price and placing a buy order may look like ordinary function calls, but their implementations interact with an external service.

Operations can change program memory, access workspace files or affect an external provider. A **Host read** obtains information without modifying provider state. A **Host mutation** may change that state. Reversibility depends on the operation.

A stock-price request may return a different answer later even though it changes nothing at the provider. Changing when a request is made can therefore change the value or error returned to the program.

2.6 Synchronous calls and asynchronous requests

A synchronous call blocks the program until it returns a value or raises an error. The work behind the call can still use asynchronous I/O: the runtime sends a request, waits for a completion event and then returns the result to the caller.

Request dispatch, completion and result delivery are separate events. If two independent requests are sent before either result is needed, their waits can overlap. If each request is sent only after the previous call returns, the program waits for them one after another.

In a basic Wasm Host call, the Guest waits for the imported function to return before continuing. Suspending the Guest while an external request is pending, and resuming it later, requires additional runtime or compiler support. For example, Emscripten's Asyncify transforms compiled code to support this behaviour [13].

2.7 Runtime initialisation and copy-on-write

Starting an interpreter and importing packages can take longer than executing a short request. NumPy, for example, creates module state and initialises native code during import. Repeating that work for every isolated run adds both startup time and memory use.

A runtime image saves the memory produced by initialisation so that later instances can start from that state. Copying the full image avoids repeating initialisation but still allocates and copies memory for each instance. With copy-on-write, separate mappings initially share the same physical pages. A write creates a private copy of the affected page. Unchanged pages remain shared. The saving depends on how much memory each instance modifies.

CHAPTER 3

Related Work

This chapter compares what each system starts early or reuses, how it controls execution, and when results or side effects become visible.

3.1 Executing generated programs and tool calls earlier

Executing As You Generate (EAGER) overlaps large language model (LLM) code generation with execution [14]. It parses a growing token stream into complete top-level Python chunks, uses lookahead where a definition may still continue, and sends eligible chunks to a persistent interpreter. Queued chunks may be batched before execution. A static gate defers low-yield declarations and returns external-side-effect, timing-sensitive and dynamic-execution chunks to whole-file execution. If an early chunk succeeds, the resulting imports, variable bindings and objects remain in the interpreter for later chunks. If it fails, EAGER can stop generation and expose the partial program for repair.

The unit moved by EAGER is therefore a Python chunk together with the interpreter state it creates. This can overlap ordinary Python computation as well as calls that wait on external services. Pysolate's prefix analyser starts eligible Host requests. A separate execution Guest runs the complete Python program, receiving each value or exception at its original call. Ordinary Python computation stays in that complete-program execution.

Conveyor supports tool-specific partial execution [15]. A tool-specific parser identifies when partial model output contains enough input to begin useful work. The serving system schedules this partial invocation while decoding continues, then incorporates later arguments or work as they become available. Its CodeGen application can execute completed Python lines incrementally, but does so through the tool's partial-execution interface. Each integration defines when its inputs are ready and which state it retains.

Unlike tool-defined partial execution, Pysolate binds preparation to a complete Python call and leaves the timing decision to the Host's operation rules.

AsyncFC represents unresolved function calls as futures [16]. Independent calls can continue, while a dependency on a future delays only the computation that needs its value. This exposes concurrency without changing the model, but it changes the value protocol presented to the generated program. Pysolate instead keeps the synchronous call interface and delivers the outcome at the original call.

LLMCompiler moves control above the program [17]. A planner constructs a task graph, and an executor schedules tasks whose dependencies are satisfied. This works well when the model is expected to decompose a workflow into calls. Pysolate instead places helpers after depen-

dencies and within the original Python branches, leaving reachability and exception context to CPython.

Cloudflare Code Mode and Vercel Run place model-written programs around registered Host functions inside isolated executors [18,19]. Their recorded calls support resumption after interrupted execution. Pysolate instead starts Host work earlier, then runs the complete program in a new Guest.

3.2 Speculation and visibility

Classical speculation allows execution to proceed before an assumption has been confirmed. Correctness then depends on controlling what provisional execution can expose and on recovering when the assumption fails.

Operating System I/O Speculation forks a process to discover future disk accesses [20]. The speculative process runs ahead to prefetch disk blocks. Mapped writes receive private copy-on-write pages, and operations that could affect the parent are suppressed.

Speculator lets a distributed file-system client continue after predicting the result of an operation [21]. Kernel support checkpoints the process, tracks dependencies and delays output. The file server validates the assumption. Success accepts the speculative execution, while failure restores the checkpoint and repeats affected work. Safe programmable speculative parallelism similarly combines dependence validation with restrictions that make failed speculation recoverable [22].

These systems continue application execution using an assumed result. They therefore need provisional state, validation and either rollback, suppression or a static restriction that makes re-execution safe. Pysolate starts only Host requests allowed by the operation's effect and timing rule. Its execution Guest waits at the original call until the chosen request supplies a value or error.

Ghost Tool Calls isolates a different hazard: an external read can disclose arguments, destination and timing when it is issued, even if its result is later abandoned [23]. Read-only status and later cleanup cannot undo that disclosure. This motivates Pysolate's distinction between provider issue time and Guest materialisation time.

Recent agent-serving systems predict likely actions, tools or retrieval steps while a model is still deciding what to do [24–29]. They execute predicted work and discard or replace it when the prediction is wrong. SpecBox applies the same idea to environment setup before an agent explicitly requests a sandbox or tool [30]. Their benefit depends on predicting a future action accurately enough to justify discarded work.

Pysolate starts from a concrete source call, its available arguments and the Host's decision to allow early issue.

3.3 Effect systems and access control

Effect systems describe which computations may read or modify state, allowing a compiler to constrain unsafe changes in execution order [31]. Such language-level information cannot establish whether a provider may observe a request earlier, which remains a Host policy decision in Pysolate.

Capability systems answer a related access question. CaMeL carries capability information through model-generated code to restrict sensitive values from reaching unauthorised tools [32]. Proof-carrying code instead requires a machine-checkable argument before untrusted code executes [33]. These systems govern whether an action is authorised. Pysolate also uses the current Host policy to decide when that work may begin.

3.4 Reusing initialised execution state

A separate line of work avoids repeated runtime construction by retaining initialised execution state. The systems differ in what they capture, how they restore it and whether isolation comes from a new instance or from resetting a reused one.

SEUSS creates unikernel snapshots after common initialisation paths [34]. New function contexts inherit the saved state, and page-level copy-on-write records later differences. A hierarchy of contexts can reuse operating-system, language-runtime and function-specific initialisation. Pysolate likewise reuses initialised state, but maps one CPython/WASI image into separate Wasm instances whose writes become private through `MAP_PRIVATE`.

Groundhog targets request isolation inside a reused Function-as-a-Service (FaaS) process [35]. It runs an initial request, captures process memory and register state, tracks later dirty pages and mapping changes, then restores the process after each real request. Groundhog restores one process sequentially, whereas Pysolate creates separate image-backed instances that can fan out concurrently.

REAP and FaaSnap focus on the restore path of Firecracker virtual machine (VM) snapshots. REAP records the stable working set used after restoration, packs those pages and prefetches them on later invocations [36]. Its main question is which saved pages should be loaded before demand faults stall execution. FaaSnap adds concurrent paging and region-aware working-set loading so snapshot restoration remains effective when different inputs touch different pages [37]. Both optimise how a VM snapshot reaches usable memory, whereas Pysolate maps a pre-initialised runtime image without predicting a working set or changing VM paging.

Faasm runs lightweight WebAssembly functions within shared processes [38]. Each function has private linear memory, while explicit shared-memory regions and a distributed state interface support application-visible sharing. Proto-Faaslets retain initialised runtime state to reduce startup work. Unlike Faasm's application-visible shared state, Pysolate shares only the physical storage of unchanged pages.

A summary comparison is given in [Appendix E](#).

CHAPTER 4

Design

Pysolate uses our CPython/WASI build as the Guest and a Go Host for external access. The sandbox supports two optimisations: PLM overlaps Host requests with other work, while copy-on-write avoids rebuilding the same runtime state for every run. The Host can call the Guest after Python setup and control its linear-memory mappings, allowing later instances to reuse the prepared state. This chapter describes the sandbox first, then each mechanism.

4.1 Sandbox design and interfaces

Each run executes in a new Guest. Three requirements guide the design:

1. **Separate runs.** Each Guest has its own mutable memory and workspace.
2. **Unchanged program logic.** The optimisations leave the program's control flow unchanged. Values and errors are delivered at the original calls.
3. **Transparent optimisation.** Program authors use ordinary synchronous calls. The runtime handles analysis, request scheduling and memory sharing without requiring an asynchronous API.

4.1.1 File access and Host tools

The Guest runs Python and holds its program state. Standard Python file operations use WASI. External tool calls cross a separate interface to the Go Host.

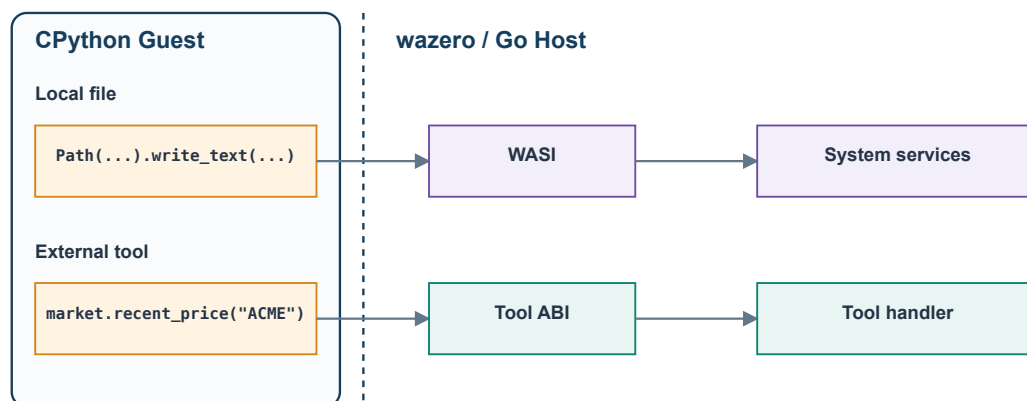


Figure 2: Python file operations use WASI, while external operations cross Pysolate's Tool ABI. For example, writing a report with Python's file API reaches `/workspace` through WASI. WASI also provides other system services, such as clocks and random bytes. A call such as `market.recent_price("ACME")` crosses Pysolate's Tool ABI, where the Go Host invokes the stock-price handler.

4.2 PLM: starting Host work early

4.2.1 Example: preparing a stock-price request

Consider a program that looks up a recent stock price for Acme (ticker ACME) before deciding whether to buy:

```
do_some_work() # Time-consuming computation
price = market.recent_price("ACME")
if should_buy(price):
    market.buy("ACME", 100)
```

The stock-price call is synchronous: the program cannot use `price` or enter the following branch until that call returns. The function returns the price. The Guest sends the operation and its arguments to the Go Host, which calls the external service and returns its response. As Figure 3 shows, the Guest waits during this exchange. The Host may use asynchronous I/O internally, but the Python function still blocks its caller.

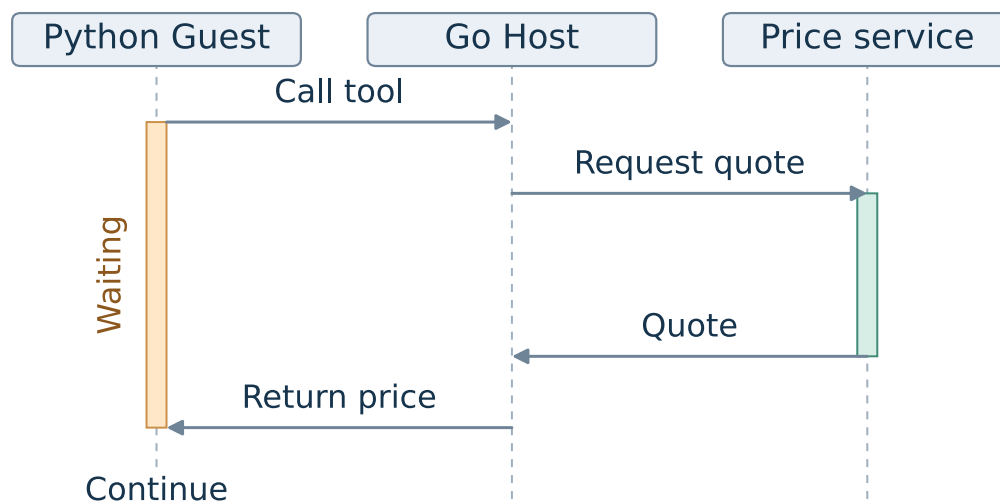


Figure 3: A synchronous stock-price call waits for the Host and external service before the program continues.

The example tool returns a quote observed no more than five minutes before the price call. Suppose the work before it is independent of the quote and the Host allows an early request. Starting that request before `do_some_work()` would overlap the provider wait with Python computation.

A first attempt separates the call into Prepare and Materialize. Prepare starts the request before the computation. Materialize returns the saved result at the original call.

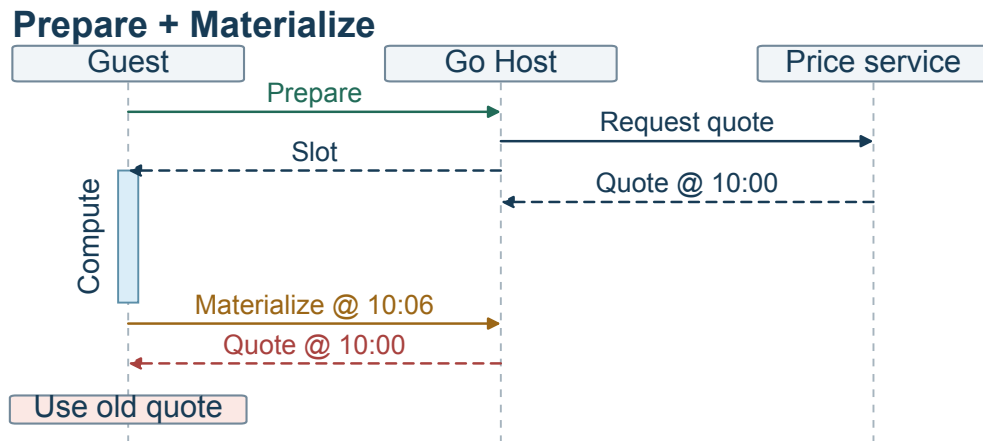


Figure 4: Prepare and Materialize overlap work but can return an old quote without a validity check.

At 10:06, the saved quote is too old when Python reaches the price call. The Host must check whether the result is still valid there and request another quote if necessary.

4.2.2 Prepare, Linearize and Materialize

PLM separates the call into three stages:

1. **Prepare** may start the provider request and returns a slot identifying the request in the Host's request table, without waiting for the result.
2. **Linearize** checks that the earlier request has the right arguments and still meets the tool's validity rule.
3. **Materialize** waits if necessary, then returns the value to the program or raises the provider error.

The resulting schedule is shown below as pseudocode. Here Linearize starts a new request if the quote is too old.

```

slot = Prepare(market.recent_price, ["ACME"])
do_some_work() # Time-consuming computation
slot = Linearize(slot)
price = Materialize(slot)
if should_buy(price):
    market.buy("ACME", 100)
  
```

A quote observed at 10:00 is valid at 10:03 and has expired by 10:06. Materialize waits for whichever request Linearize chooses. The buy operation stays in its original branch and runs only if Python reaches it. An early request is called a **candidate** until the Host validates it for the original call.

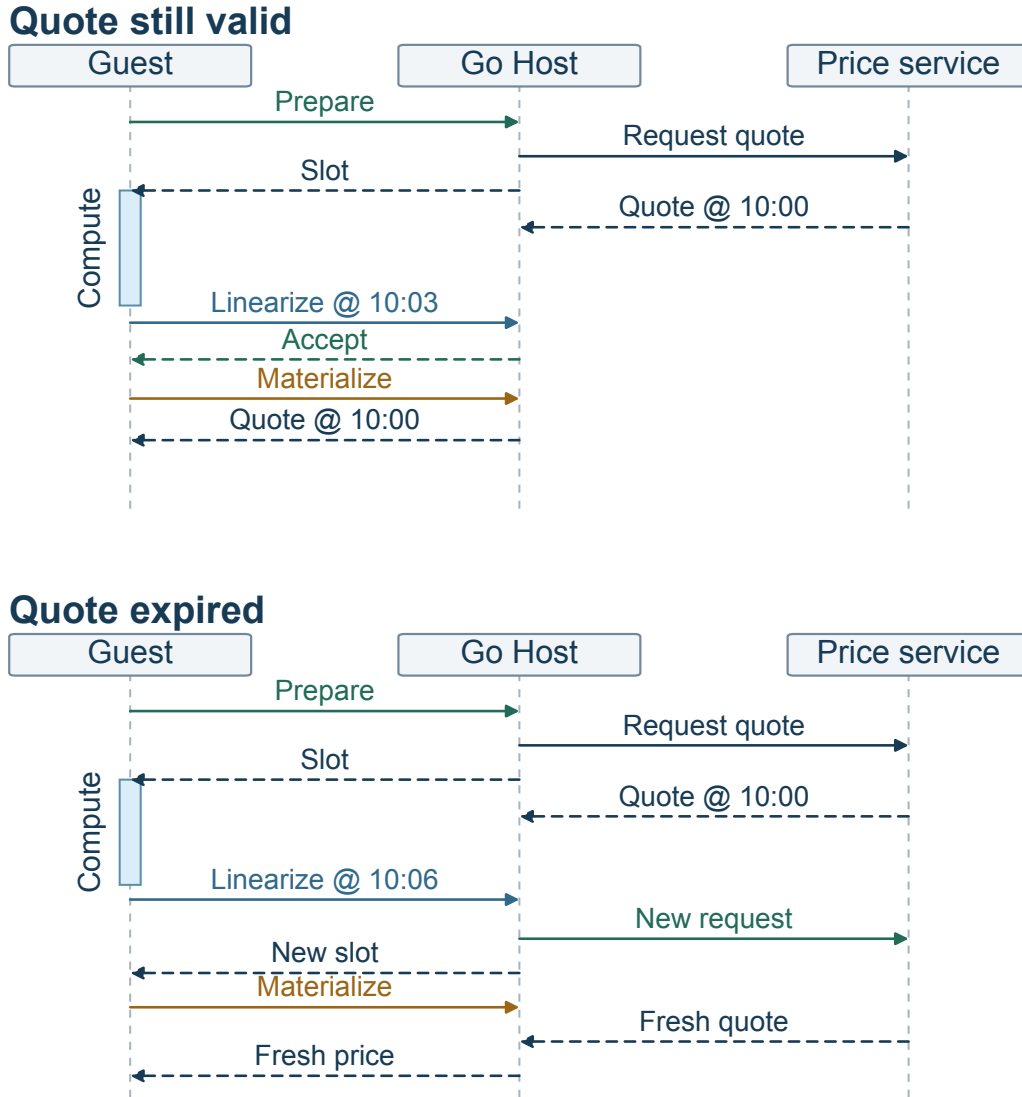


Figure 5: Linearize either accepts the early quote or starts a new request. Materialize delivers the chosen result.

In the sketch, `do_some_work()` stands for independent computation. Unknown function calls act as barriers in the AST placement checks.

Here, **logical execution** means the program's control flow and its delivered values or errors. **Physical work** includes the provider requests, computation and memory setup that support that execution.

4.2.3 Host rules

The tool's Host rule sets the five-minute allowance. Source analysis cannot infer it from the function name. We represent an external operation as $R(a, h, t)$: a contains its arguments, h is provider state outside the program, and t is the time at which the provider observes the request. Starting earlier can change the returned value or error by changing h , t or both.

Each operation therefore has a Host-authored rule describing its effects, whether a request may start early and what makes it valid at the original call. The Host applies the rule before starting an early request and again at the original call.

4.2.4 Prefix preparation

Suppose the first source chunk contains:

```
listing = tools.listing("Acme")
```

The call is complete, its argument is known and later source is still arriving. Prefix analysis reports it to the Host. If the operation rule allows preparation, the Host starts the listing request and stores it in its request table, called the PLM table. When the final program later reaches the listing call, Linearize and Materialize reuse that request.

This path overlaps Host waiting with source generation. It handles complete top-level calls with statically known arguments.

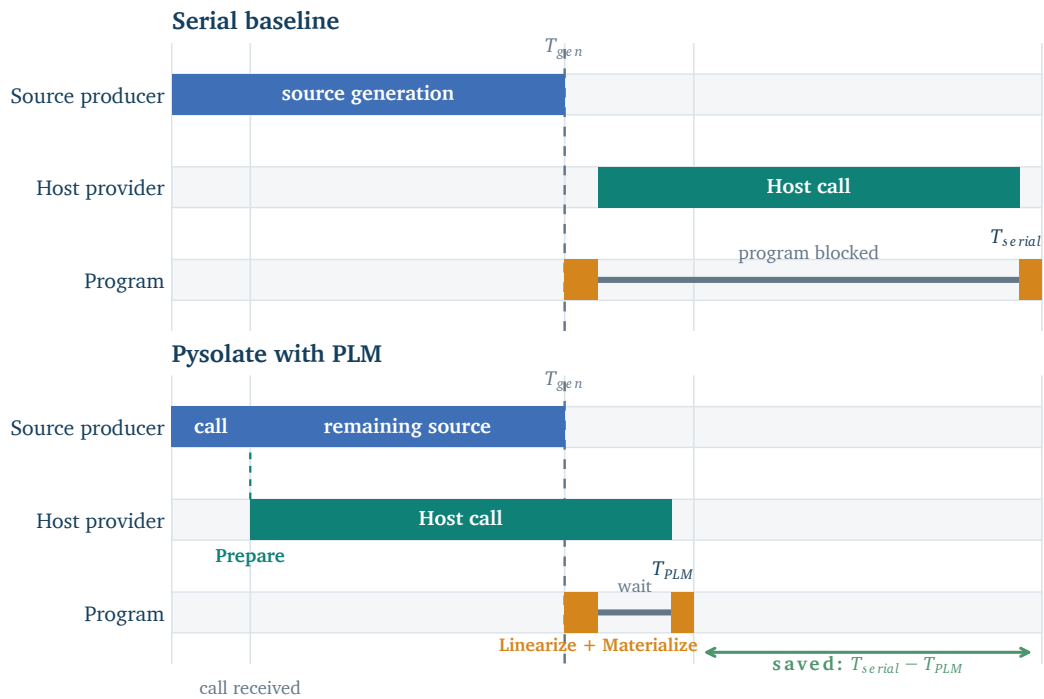


Figure 6: Serial and Pysolate timelines for one Host call that can start early in streamed source. In Figure 6, the provider request overlaps the remaining source generation. The complete program still executes from the beginning in a new Guest. Several independent calls can use the same overlap window when their arguments and Host rules allow it.

4.2.5 Whole-program preparation

The complete program may contain a second call whose arguments and reachability depend on the first:

```
listing = tools.listing("Acme")
if listing["market_open"]:
    ticker = listing["ticker"]
```

```
label = "Current price"
price = tools.price(ticker)
```

After parsing the full source, the whole-program pass places the price Prepare inside the reached branch and after the ticker assignment. It may place Prepare before the independent label assignment, while Linearize and Materialize stay at the original price call. If the market is closed, Python never reaches Prepare and no price request starts.

The general ordering rule follows from this example: Prepare may move only after its data dependencies and inside the same controlling path as the original call. It never moves above an earlier Host mutation, unknown call or other non-transparent statement. Unsupported structures retain their synchronous call. The Host receives only the operation and concrete arguments. Python still decides the branch.

4.2.6 Results and failure handling

The Host stores each candidate until the program reaches its call. Materialize returns the value or raises a provider error there. If the arguments no longer match or the result is no longer valid, the Host makes the synchronous call instead. The tool's failure rule determines whether an earlier error can be reused. Unused requests are cancelled or discarded when the run ends. In short, Prepare starts a request as early as its dependencies and the Host rule allow. Linearize decides whether that work can serve the original call or a new request is needed. Materialize waits for and delivers the chosen result. It could move closer to the value's first use, but only if that move also preserves exception order. The prototype keeps Linearize and Materialize together at the original call, and focuses on the overlap from earlier Prepare. Section 7 considers the requirements for moving Materialize closer to the value's first use.

4.3 Copy-on-write: reusing initialised memory

4.3.1 One shared starting state

The compiled Wasm module contains executable code and packaged Python modules. The memory image is different: it contains the state created when Python starts and imports those modules. Reusing that image avoids repeating the same setup for every short program.

Pysolate first creates a reference Guest, starts Python and imports any packages needed by later runs. It captures the linear memory after this setup returns, before any request-specific program runs. This captured image becomes the common starting state. It remains read-only while later Guests execute.

The Host chooses the setup code and captures the resulting Python heap and imported module objects before user execution.

4.3.2 Separate writes for each run

Each new Guest receives its own memory view of the image. At first, those views refer to the same physical pages. When a Guest writes to a page, the operating system gives it a private copy. Other Guests continue to see the original page. Wasm instance metadata, such as globals

and tables, and the workspace are created separately for each run. These are distinct from the Python-module objects stored in the image.

This is useful when the interpreter occupies many pages but each program changes only a small part of it. A run may update Python objects and allocate temporary arrays without copying every unchanged module and runtime page. If it writes to most of the image, the memory saving becomes smaller.

When a run ends, its modified pages are released while the shared image remains available. Section 5.4 describes the mappings and the private-copy alternative.

CHAPTER 5

Implementation

The prototype is implemented as a Go Host around CPython compiled for the WebAssembly System Interface and executed by wazero.

5.1 Building the Python Guest

The Guest needs the interpreter, standard library and application packages. We also include support code that runs submitted programs and connects their tool calls to the Host. NumPy adds another requirement: its array operations depend on compiled extensions as well as Python files.

The NumPy build uses CPython 3.14.0, WASI SDK 33 and a selected NumPy 1.26.0b1 source revision. The scripts in `guest/build` build these parts and combine them into one Wasm reactor [39]. Figure 7 separates the native code from the Python files. Both are needed for the final Guest.

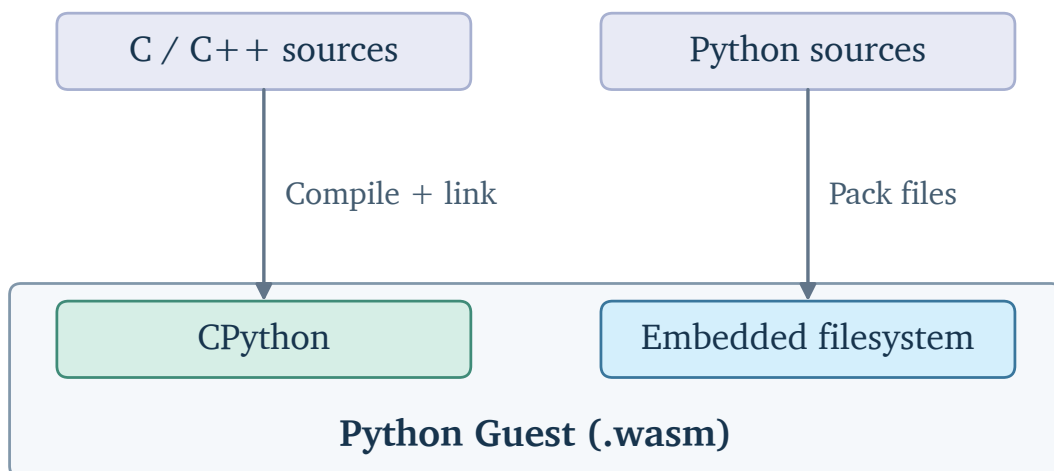


Figure 7: Native code is linked into the Guest, while Python files are packed into its embedded filesystem.

5.1.1 Cross-compiling CPython

The build starts with CPython's WASI build tools. They first provide a Python executable for the build machine and then build CPython for `wasm32-wasi`. The first interpreter runs build scripts. The second becomes part of the Guest.

The WASI build produces `libpython3.14.a`, a static library containing the interpreter. It also produces target headers and Python build settings. NumPy's build uses those target settings. The compiler and system libraries come from WASI SDK.

We link this interpreter library with the small C interface in `guest/src/runtime.c`. That interface lets the Go Host initialise Python, pass source and inputs into it, and read the response. The final link uses `-mexec-model=reactor`, exposing functions that the Host can call repeatedly. The initialisation calls and their role in COW are described in Section 5.4.

5.1.2 Statically linking NumPy

A normal NumPy build creates separate libraries that Python loads during import. Our build puts the compiled NumPy code inside the same Wasm file as CPython. A small wrapper collects that code and passes it to the final link step. This includes NumPy's array operations, FFT, linear algebra and random-number modules.

The build scripts still expect a `.so` file for each extension. The wrapper creates an empty file with that name so the scripts can finish. The useful compiled code has already been saved for the final link. These empty placeholders are removed before the Python package is packed.

The recipe disables external BLAS and LAPACK linkage, SVML and NumPy's optional build optimisations. This avoids requiring external numerical libraries. WebAssembly itself supports SIMD vector instructions [5], but desktop libraries still need to be built for the Wasm target. The final compilation and link use `-O2`.

The port adjusts platform-dependent calls through configuration changes and small source patches. For example, the timer patch replaces `clock_nanosleep` with relative `nanosleep` calls supported by wazero.

Each extension exports a `PyInit_...` function. The final link keeps these functions, and a generated header registers each module name with its function before `Py_InitializeFromConfig` starts the interpreter. One entry is:

```
PyImport_AppendInittab(  
    "numpy.core._multiarray_umath",  
    PyInit__multiarray_umath  
);
```

This registers the extension as a built-in Python module. Import finds its code inside the Guest.

5.1.3 Packaging files and loading modules

Static linking supplies the native code, but Python still needs the package files around it. We use `wasi-vfs` to pack the standard library under `/usr/lib/python3.14`. The NumPy and `agent_runtime` packages are placed in its `site-packages` directory. The packed files become part of the final Wasm module.

Import must connect these two parts. For `import numpy`, Python first loads the package's `__init__.py`. When that file imports a native submodule, the registered built-in supplies its implementation. A small import finder in `guest/include/wasm_extension_finder.h` checks

registered built-ins and locates package files in the embedded filesystem. It uses Python's existing built-in and source-file loaders.

The `agent_runtime` package is our Python support code. It runs submitted programs and sets up their tool interfaces. An ordinary tool function sends its name and arguments through a common `host_call` entry in the C bridge. The Go Host selects the registered handler and returns its response. This keeps the Python tool wrappers small. The Host implements the external operation.

5.1.4 Checking the resulting Guest

The build validates the Wasm module, then runs import and operation checks through the real Guest runner. Checking only `import numpy` would miss problems in the native operations. The NumPy check therefore creates arrays, performs arithmetic, converts array data to bytes and back, and checks matrix multiplication. For example, its integer-array check is equivalent to:

```
import numpy as np

values = np.arange(6, dtype=np.int64).reshape((2, 3))
assert (values * 2 + 1).tolist() == [[1, 3, 5], [7, 9, 11]]
```

The passing checks show that these array operations work in the built Guest. The evaluation then uses it for the selected NumPy programs and the COW array workload.

5.2 Runtime architecture and run lifecycle

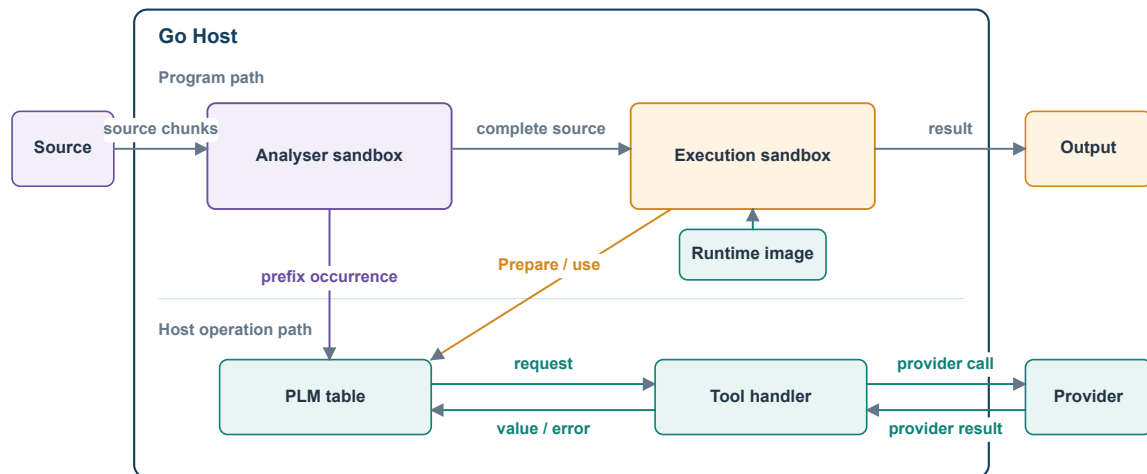


Figure 8: Pysolate program path, Host-operation path and sandbox boundaries.

Source chunks enter the analyser sandbox, which reports eligible calls to the Host for early preparation. The complete source is validated and transformed inside the execution Guest before Python runs it. Both preparation paths use the run's PLM table. The Host tool handler connects that table to the external provider, while a runtime image can supply the execution Guest's initial memory.

5.2.1 A Guest for each run

Each run uses a new Guest with per-run `/workspace` and `/tmp` directories and its own table of pending PLM requests. The Host supplies the allowed tool handlers. Compiled code and the initial memory image can be reused.

5.2.2 Python interfaces

Python uses two interfaces:

- **WASI:** ordinary file APIs access the Guest's workspace and other filesystem resources.
- **Host tools:** module methods encode an operation and its arguments, then call Pysolate's Tool ABI. The Go Host decodes the request and invokes the handler. Provider credentials stay in the Host.

Generated Python support code can use the synchronous Host call, Prepare, and a combined Linearize-and-Materialize operation. The synchronous path remains available for unsupported operations or calls that cannot start early.

5.2.3 Passing byte buffers

The interface exchanges JSON-encoded byte buffers.

Running a program (Host → Guest). The Host calls `alloc(n)`, copies the JSON bytes into Guest memory and calls `execute(offset, n)`. The offset and length are two 32-bit integer arguments. Together they describe a byte buffer: where it starts and how many bytes it contains. The offset is counted from the start of Guest linear memory.

`execute` returns the offset of a response frame. Its first four bytes hold a little-endian payload length. The JSON response follows them. The Host copies out the response and frees the input allocation with `dealloc`. The response frame is a reusable buffer owned by the Guest:

```
Input:  (offset, length) -> JSON bytes
Output: offset -> [4-byte length][JSON bytes]
```

Calling a tool (Guest → Host). The Guest passes request offset and length, plus reply offset and capacity. The Host writes its reply into that buffer and returns the number of bytes written. This keeps Python objects inside CPython while the two sides exchange a simple byte representation.

5.2.4 Run lifecycle

Figure 9 follows source arrival, final execution and cleanup. Prefix requests enter the PLM table before the complete source arrives. The execution Guest later uses those requests at the original calls.

5.3 PLM implementation

PLM starts requests early in two places. During source generation, prefix analysis finds top-level calls that can already be issued. After the complete source arrives, an AST pass moves later calls earlier while keeping their data dependencies and controlling branches.

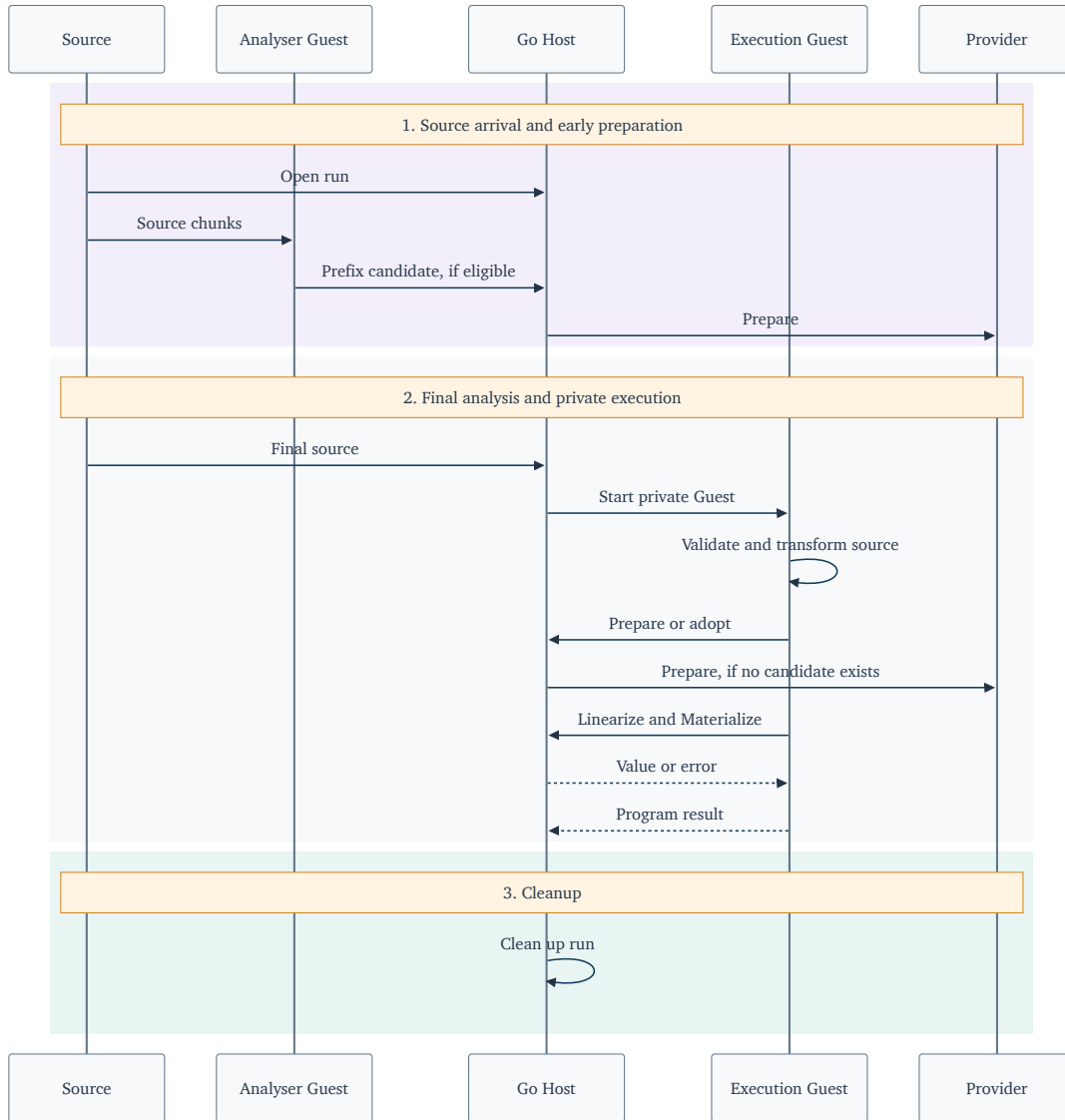


Figure 9: Lifecycle of a streamed Pysolate run with prefix and whole-program preparation.

5.3.1 Prefix preparation

During source generation, the analyser finds complete top-level tool calls with statically known arguments. Calls inside branches or loops, and calls that need earlier computed values, are left for the whole-program pass.

For each call it finds, the analyser reports the operation, source call and normalised arguments. The Host applies the operation's PLM rule before starting the request and stores the candidate in the PLM table.

After the final source arrives, the whole-program pass generates the PLM helpers. At the Prepare position, a helper uses matching prefix work or starts a new request. At the original call, the combined Linearize and Materialize helper validates that request and uses the synchronous path if validation fails. Prefix analysis therefore stays shallow, while the full pass owns Python dependency and control-flow placement.

5.3.2 Whole-program transformation

After final source arrival, the execution Guest parses the program into a Python abstract syntax tree (AST). The pass visits supported Host-call assignments, inserts Prepare after their dependencies, and replaces each original call with the combined Linearize and Materialize helper. The resulting tree is compiled and executed in that Guest.

The pass handles direct assignments to one name and calls to known Host module methods. Call arguments are limited to names and scalar constants, with a small set of simple expressions used to follow dependencies. The traversal covers module blocks, `if`, `for` and `async for` bodies. It rejects dynamic source observation or execution and leaves unsupported code unchanged. Within the supported subset, Prepare is inserted after the definitions and barriers on which the call depends. An earlier Host mutation, unknown call or other non-transparent statement is a barrier. For a call controlled by a branch, the helper remains inside that branch.

5.3.2.1 Transformation example

The example contains two Host calls. The first returns listing information and decides whether the second call is reached. The pseudocode shows Linearize and Materialize on separate lines to make their roles clear. The implementation combines them in one call.

Original program

```
listing = tools.listing("Acme")
if listing["market_open"]:
    ticker = listing["ticker"]
    label = "Current price"
    price = tools.price(ticker)
    result = [label, price]
else:
    result = ["Market closed"]
```

Generated PLM schedule

```
listing_slot = Prepare(
    tools.listing, "Acme")
listing_slot = Linearize(listing_slot)
listing = Materialize(listing_slot)
if listing["market_open"]:
    ticker = listing["ticker"]
    price_slot = Prepare(
        tools.price, ticker)
    label = "Current price"
    price_slot = Linearize(price_slot)
    price = Materialize(price_slot)
    result = [label, price]
else:
    result = ["Market closed"]
```

The listing Prepare can run before its original call because its argument is already available. The price Prepare remains inside the branch because both reachability and the ticker depend on listing. An independent assignment can execute while the price request is in flight. Linearize and Materialize remain at the original calls, where the program receives each value or exception.

Each loop iteration gets its own slot, so Prepare and Linearize use the same request for that iteration.

5.3.3 Host calls and cleanup

The benchmarks use a synthetic `sources.read` provider. At the original call, the Host checks whether it can reuse the prepared request. It waits if needed, or makes the ordinary call if the prepared request cannot be used.

PLM delivers values and provider errors at the original call, where the program's `try/except` handles them.

An early request can go unused if an earlier statement raises an exception or the run is cancelled before reaching the call. The PLM table then cancels or discards that work.

5.4 Copy-on-write implementation

5.4.1 Python reactor and warm-up

The Guest is linked with `-mexec-model=reactor`. Its C interface exposes `runtime_init` to initialise Python, `runtime_prepare` to run setup code, and `execute` to process a request [39].

The Host first calls `_initialize` for module setup, then `runtime_init`. The latter invokes `Py_InitializeFromConfig` and loads the `agent_runtime` support module. Before capture, the Host can use `runtime_prepare` for trusted imports and other setup shared by later runs.

The Host captures memory after these calls return, as shown in Figure 10. The Wasm call stack is empty, while Python's heap and imported modules remain in linear memory.

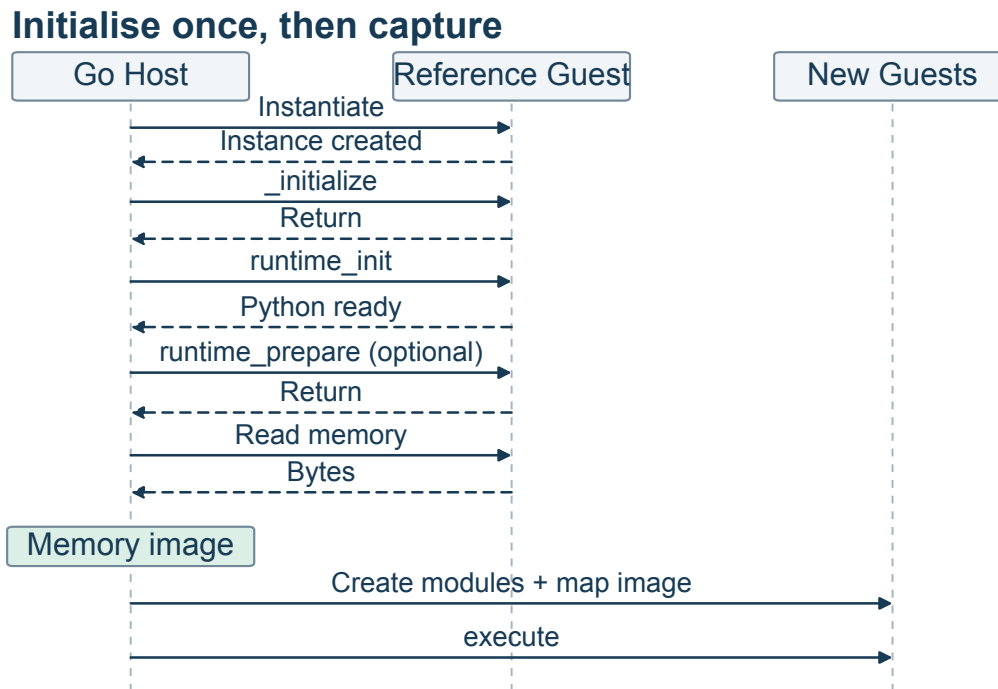


Figure 10: The Host initialises Python and finishes warm-up before capturing memory for later Guests.

5.4.2 wazero state and Wasm linear memory

Within a wazero engine, compiled code is separate from each module instance [40]. Instance data includes globals, tables, a WASI context and an internal `MemoryInstance`. Its `Buffer` field is a Go `[]byte` view of linear memory. Guest pointers are offsets within this memory, so they remain valid when the buffer is mapped at a different Host address.

`Buffer` provides a Go view of the mapped memory. The operating system decides which physical pages serve each mapping.

The request and response buffers described in Section 5.2.3 are regions of this same linear memory. Python's heap also lives there. The wazero module metadata is held separately in Go objects.

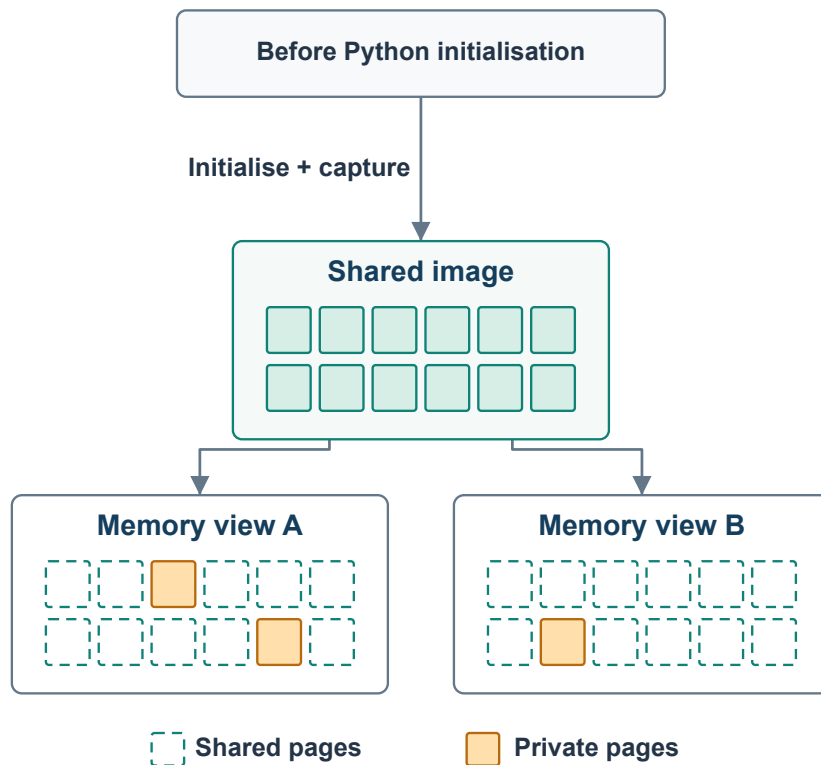


Figure 11: An initialised Python image shared by two instances, with private copies for their writes.

The image stores the memory produced by Python initialisation. The COW allocator connects each instance's byte view to this image through wazero's `MemoryAllocator` interface. `MAP_PRIVATE` provides a writable mapping for each instance while sharing the unchanged image pages. The module's globals, tables, WASI resources and workspace are created separately.

5.4.3 Capturing and mapping the image

Pysolate currently implements COW on Linux only. It creates a Guest from the shared memory image in five steps:

1. Create a reference Guest, call `_initialize` and `runtime_init`, and complete any trusted warm-up.
2. Read its linear memory after the calls return. Store the bytes in a `memfd` and seal it against writes and resizing.
3. Instantiate a new module from the same compiled Guest, with its own globals and tables. The COW allocator supplies a `MAP_PRIVATE` linear-memory buffer, with its visible size grown to the captured baseline size.

4. Call `_initialize` to complete module setup. Instantiation and module setup may overwrite memory. `MADV_DONTNEED` discards these private changes and restores the captured Python state.
5. Set up the Host tool interfaces and execute the program. Closing the Guest releases its private mapping.

5.4.4 Shimmy integration

Shimmy executes evaluation functions supplied to Imperial College London's Lambda Feedback platform [41,42]. It installs an evaluator's Python code and dependencies before requests arrive.

Pysolate creates and closes an execution Guest and workspace for each run. Shimmy keeps an installed Wasm instance in each worker slot. After an evaluation, it copies the response into Host memory, restores the instance's linear memory and returns the slot to the pool. A failed Wasm call or restore causes slot replacement.

Full-copy reset restores the evaluator by copying its initial memory back after a request. COW reset discards the pages changed by the request and returns to the shared image. Full-copy and COW reset both keep the installed evaluator while clearing request-specific changes [43].

CHAPTER 6

Evaluation

The evaluation refines RQ3 into two empirical questions:

1. **EQ1:** How much waiting can PLM remove, and when does that saving offset analysis?
2. **EQ2:** When does copy-on-write offset image setup, and how does image reset affect Shimmy?

The PLM studies measure overlap and its limits. The COW studies measure the instance-count crossover and then test image reset in Shimmy.

6.1 What the timings include

The experiments run on Linux/x86-64. Figure 12 shows which parts of each run are timed. The Wasm module is compiled before timing starts. Machine configurations, repeated-run counts and statistical methods are collected in [Appendix F](#).

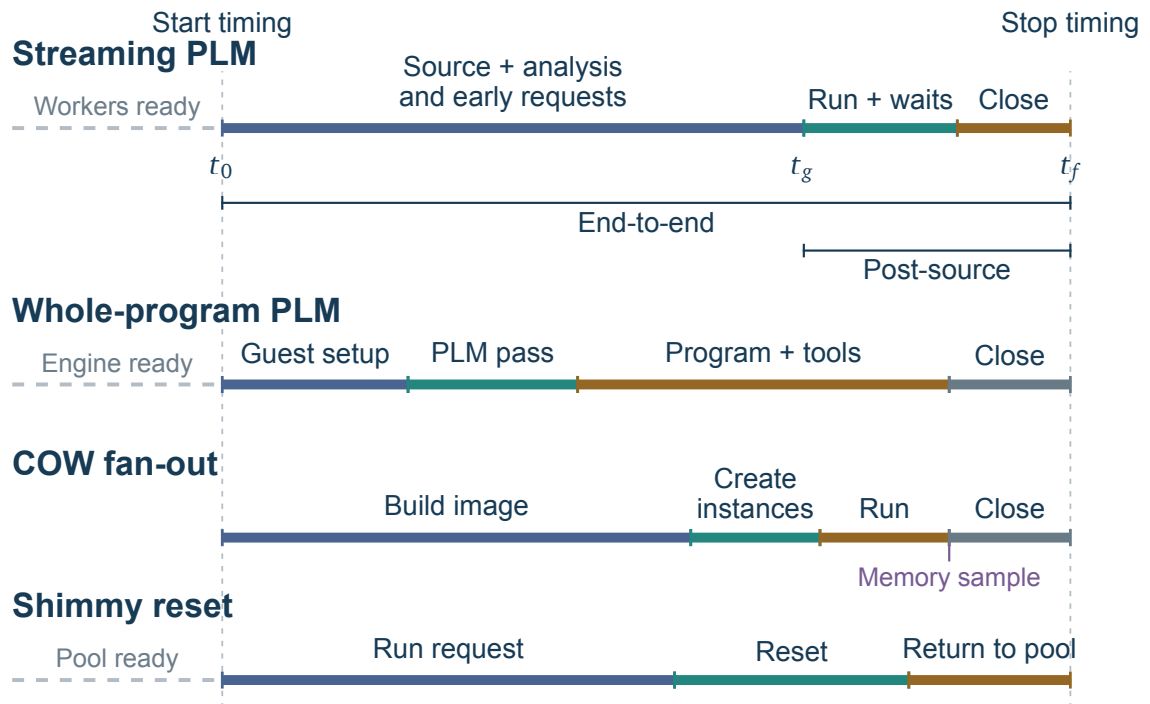


Figure 12: Timing intervals for streaming PLM, whole-program PLM, COW fan-out and Shimmy reset. Segment lengths are illustrative.

For streamed code, end-to-end time begins with the first source chunk and ends after execution and close. Post-source time begins when the complete source has arrived. It shows how much work remains after generation:

$$T_{\text{end-to-end}} = t_f - t_0 \quad T_{\text{post-source}} = t_f - t_g$$

Here t_0 , t_g and t_f mark first source arrival, complete source arrival and completion respectively. The main comparison uses end-to-end time. A PLM-minus-serial difference is $T_{\text{PLM}} - T_{\text{serial}}$, so a negative value means PLM is faster. Figures that show **time saved** use the opposite sign.

6.2 Early Host preparation

For independent Host-call delays d_i , serial waiting approaches $\sum_i d_i$, while concurrent preparation approaches $\max_i d_i$. Let H be the usable overlap time and T_{analysis} the analysis time that adds to the run. The approximate saving is

$$S_{\text{PLM}} \approx \sum_i d_i - \max(0, \max_i d_i - H) - T_{\text{analysis}}$$

For prefix preparation, H mainly comes from source that arrives after the call is found. For whole-program preparation, it comes from independent program work between Prepare and the original call.

6.2.1 Prefix preparation during source arrival

Prefix preparation can start a Host call before the complete program arrives. Streaming Overlap releases source chunks at 0, 700 and 1,400 ms around one 1,500 ms Host read of test data. Whole-file serial execution waits for the complete source. An immediate-dispatch reference starts the read as soon as the complete call appears. Pysolate starts the request early and uses its result at the original call. The timing follows the streaming row in Figure 12.

Measure	Median result
Pysolate minus serial	−1.405 s
Immediate dispatch minus serial	−1.390 s

Table 1: Streaming Overlap timing differences with reusable workers ready.

Pysolate saves 1.405 seconds. For corresponding runs, the median Pysolate-minus-immediate-dispatch difference is −9.6 ms. Most of the 1.5-second request runs while later code is arriving. Prefix analysis takes a median 89.6 ms. Some of it runs while code is arriving. Only the extra time on the execution path is counted as T_{analysis} .

6.2.1.1 Concurrency and saturation

The heatmap in Figure 13 varies the number of independent calls and the time available before the complete source arrives. The first source chunk contains one, two, four or eight independent Host calls. The final result statement arrives 0, 100, 200 or 400 ms later. This gap is the source window. Every provider call takes 200 ms. Serial and PLM use the same ready runtime capacity, program result and call accounting.

The heatmap reports time saved as serial minus PLM, so positive values favour PLM. With one call and no source window, the median is −4.3 ms and the 95% interval, −27.1 to +16.8 ms, crosses zero. There is no independent wait to hide. At the same zero window, two, four and eight calls save 193, 589 and 1,370 ms because PLM starts the independent provider requests together.

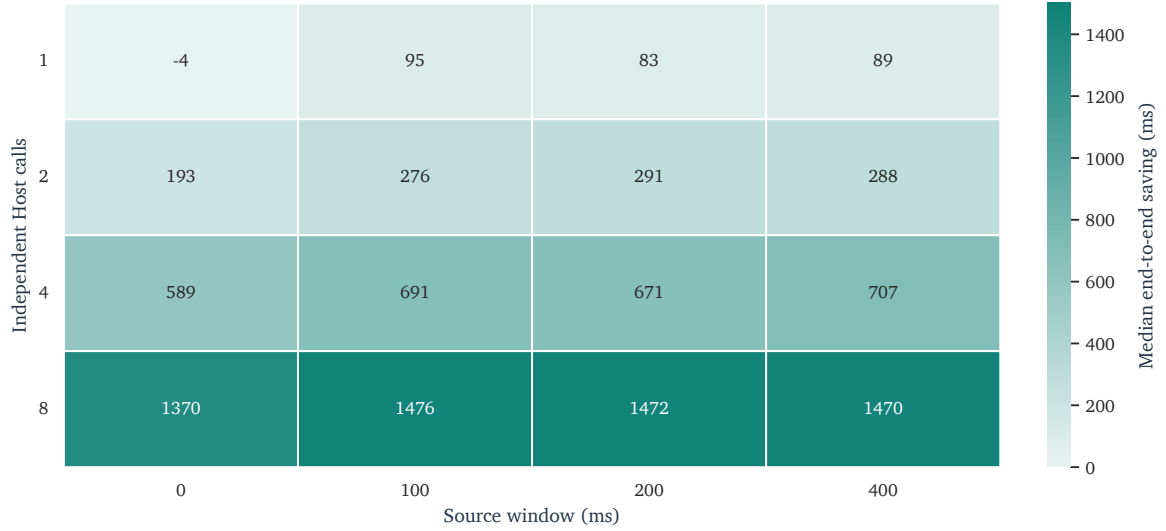


Figure 13: Median end-to-end saving across independent calls and source windows. Positive values favour PLM.

A source window adds another opportunity. With one call, windows from 100 to 400 ms save between 83 and 95 ms. With two or more calls, the same window adds roughly another 80 to 100 ms to the saving already created by concurrent provider work. The 200 ms request duration limits the available overlap. Analysis and final execution also contribute to elapsed time. At 400 ms, every early call is complete before the final source arrives.

6.2.2 Prefix preparation on NumPy programs

To test less repetitive program bodies, this experiment uses 15 NumPy programs from GitChameleon 2.0 [44] and replaces literal arrays with values from one or two Host calls. The calls appear in the first source chunk and each takes 200 ms. The remaining source arrives at 20, 50, 100 or 200 tokens/s. Every run checks the original NumPy assertion, call count and materialisation path. The package and input-selection limits are described in Section 8.2.

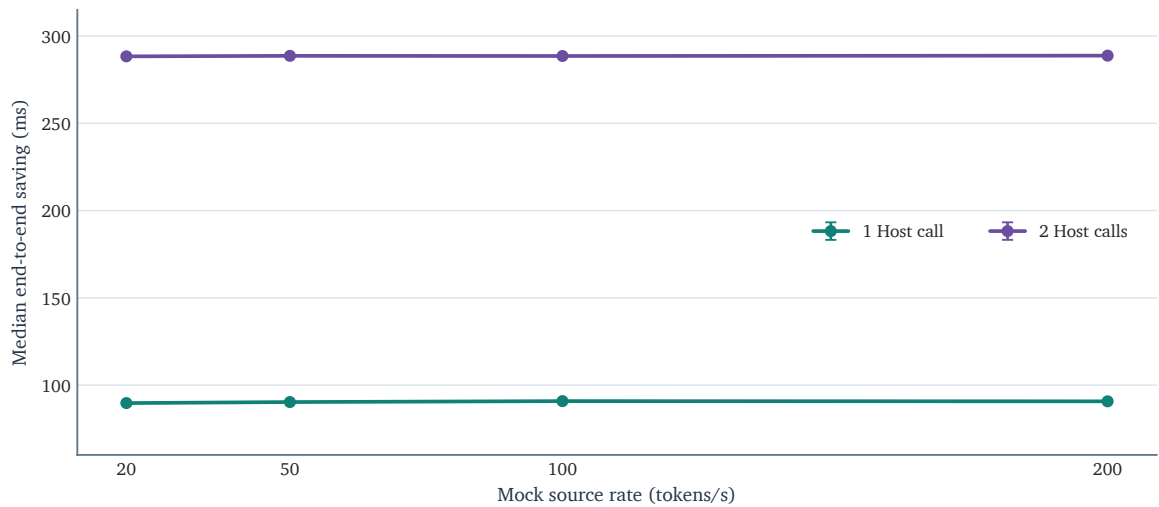


Figure 14: Prefix PLM saving by Host-call count. Error bars show 95% bootstrap confidence intervals.

Median saving is about 90 ms with one Host call and 289 ms with two. It changes little with source rate because every tested source window is longer than the 200 ms call. The experiment is therefore already in the saturated part of the model.

6.2.3 Whole-program preparation and crossover

This experiment asks when early requests save more time than the PLM pass costs. It uses the source template in [Appendix A](#): 750 chained increments followed by 1, 2, 4 or 8 independent reads. Each read takes 25, 75 or 200 ms. The complete source is available before the run. PLM overlaps requests during program execution.

The two-read case below shows the change. Prepare starts both requests before the increments. The Guest then computes while the Host waits for the responses. Linearize and Materialize remain at the original calls. The omitted assignments are the same in both versions.

Serial

```
x0 = 0
# ... 750 increments ...
a = sources.read("key-0")
b = sources.read("key-1")
result = [a, b, x750]
```

PLM schedule

```
sa = Prepare(sources.read,
             ["key-0"])
sb = Prepare(sources.read,
             ["key-1"])
x0 = 0
# ... 750 increments ...
sa = Linearize(sa)
a = Materialize(sa)
sb = Linearize(sb)
b = Materialize(sb)
result = [a, b, x750]
```

The whole-program row in Figure 12 includes Guest setup, source parsing and transformation, execution, result decoding and close. Serial execution omits the PLM transformation but uses the same timing boundaries. The reusable engine is ready before either run begins.

Longer calls leave more time to recover the analysis cost. With eight 200 ms reads, PLM saves 1.268 seconds. With delays of only 25 ms, it does not establish a reliable speed-up even at eight reads. A zero-read control put pass overhead at about 200 ms, which helps explain why short calls are difficult to accelerate.

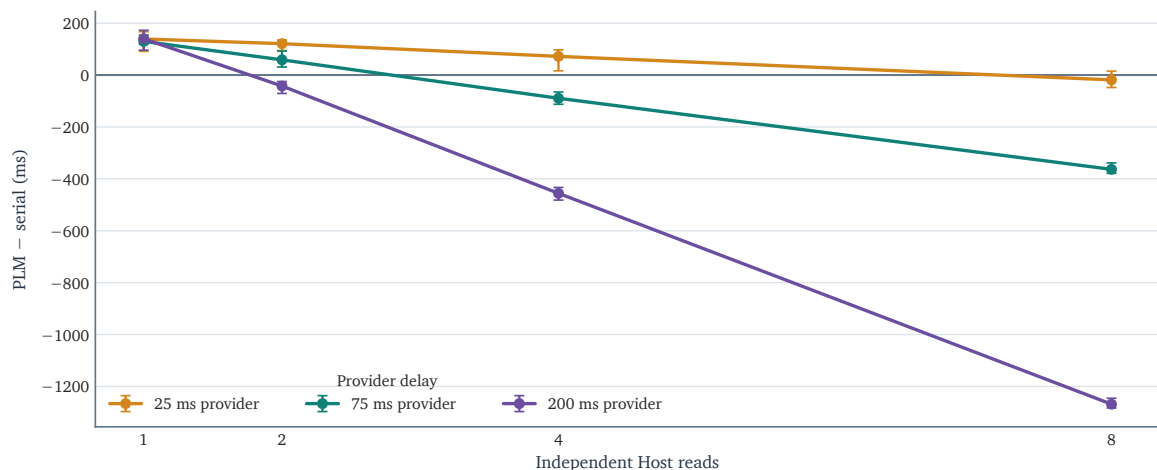


Figure 15: Whole-program PLM time difference. Negative = speed-up.

Eight serial 200 ms reads spend about 1.6 seconds waiting in total. Starting them together reduces their combined wait towards 200 ms, and the increments can run during that wait. Analysis and runtime overheads consume part of this gain, leaving the measured 1.268-second saving.

6.2.4 Controlled replay: Host calls and Python computation

A final comparison asks when EAGER and Pysolate recover the same overlap and when their different scheduling rules matter. EAGER replays complete Python chunks at chosen token rates and is compared with its own whole-file serial mode [14]. Pysolate uses its own serial baseline. Since the runtimes differ, only each system's gain over serial is compared.

Pysolate measures from first source arrival to the checked result. Its analyser pool is warmed up before timing starts. Analysis sessions reuse the initialised Python memory through COW, avoiding repeated Python startup. Each program uses a new analysis session and execution Guest.

EAGER replays eligible complete Python chunks, while Pysolate moves eligible Host requests. A local model of EAGER's paper gate¹ confirms that a name-based gate may route the same call differently from Pysolate's Host rule. The two timing columns contain the provided EAGER and Pysolate runs.

Case	Δ EAGER	Δ Pysolate
One eligible Host call	−1.401 s	−1.387 s
Two independent Host calls	−1.402 s	−2.888 s
Dependent second Host call	−1.401 s	−1.349 s
Short 25 ms Host call	−0.025 s	+0.078 s
CPU-only Python	−0.768 s	+0.277 s

Table 2: Controlled replay results for five mechanism cases.

Each delta compares the optimised and serial modes on the same execution platform. Negative values are faster. [Appendix C](#) gives the programs and schedules used by the five cases.

What moves. When both mechanisms start one Host call, each removes about 1.4 seconds relative to its own serial path. Two independent calls increase Pysolate's gain to 2.888 seconds, while a dependent second call limits it to 1.349 seconds. EAGER gains from CPU-only Python because it executes the complete chunk early. Pysolate leaves that computation in the final Guest, so analysis adds 0.277 seconds without moving the dominant work.

When it pays. At 25 ms of provider delay, Pysolate is 77.9 ms slower: analysing the source costs more than the small amount of waiting it removes. The whole-program experiment shows the same problem. Even with eight 25 ms calls, the gain is too small to establish a reliable speed-up.

¹At the time of writing this report, the paper's routing gate was not included in the released code, so the rule was implemented locally.

6.3 Reused runtime state

PLM addresses waiting within one run. Copy-on-write addresses repeated runtime setup across runs.

Copy-on-write follows a separate amortisation model. Let I be the one-off cost of building the reference image, C the cost of constructing initialised state for one instance, M the cost of mapping one instance and N the number of instances:

$$T_{\text{construct}(N)} \approx NC \quad T_{\text{COW}(N)} \approx I + NM.$$

Copy-on-write pays off when the repeated construction it removes exceeds the image cost, or $N(C - M) > I$. The fan-out experiment finds that crossover. Shimmy then tests image-based reset in an evaluator service with a retained worker pool.

6.3.1 When copy-on-write pays off

COW Fan-Out compares two ways to create 1, 2, 4 or 8 instances:

1. **Per-instance initialisation.** Each instance builds its own Python/NumPy state and an 8 MiB array. This setup is repeated for every instance.
2. **Shared-image COW.** Python/NumPy and the array are initialised once in a reference Guest. Each instance maps that memory image and gets private copies of the pages it writes.

Each program increments the first array element and returns the sum. The instances remain open and run one after another. Time covers setup, instance creation, execution and close. Memory is measured after the runs, before closing the instances, as marked in Figure 12.

The memory metric is proportional set size (PSS), read from `/proc/self/smaps_rollup` for the process containing the Host and its open Guest instances. PSS counts private pages in full and assigns a proportional share of each shared page.

Copy-on-write is faster from two instances and uses less memory from four. At eight instances, the median savings are 48.434 seconds and 534.0 MiB.

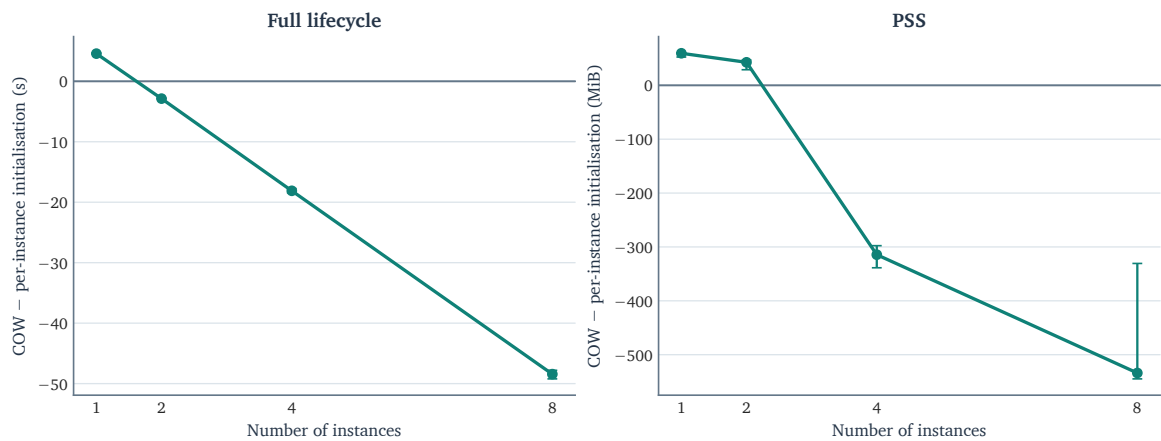


Figure 16: Copy-on-write lifecycle time and PSS by instance count.

6.3.2 Shimmy evaluator application

The Shimmy experiment measures how quickly an installed evaluator can serve new requests. Lambda Feedback checks submitted answers in a similar way to an online judge: an evaluation

function receives an answer and returns feedback. The test evaluator represents this with a JSON input and a reference answer.

Request latency is compared for three approaches:

1. **New interpreter for each request.** Start Python and install the evaluator for every request.
2. **Full-copy reset.** Keep workers ready and copy their initial linear memory back after each request.
3. **COW reset.** Keep workers ready and restore the shared image by discarding pages changed by each request.

The full-copy and COW worker pools install the evaluator before timing starts. The new-interpreter baseline includes startup in each request's time. For the full-copy and COW approaches, we sample PSS for the process holding the four-worker pool before shutdown. Sampling details are in [Appendix F](#).

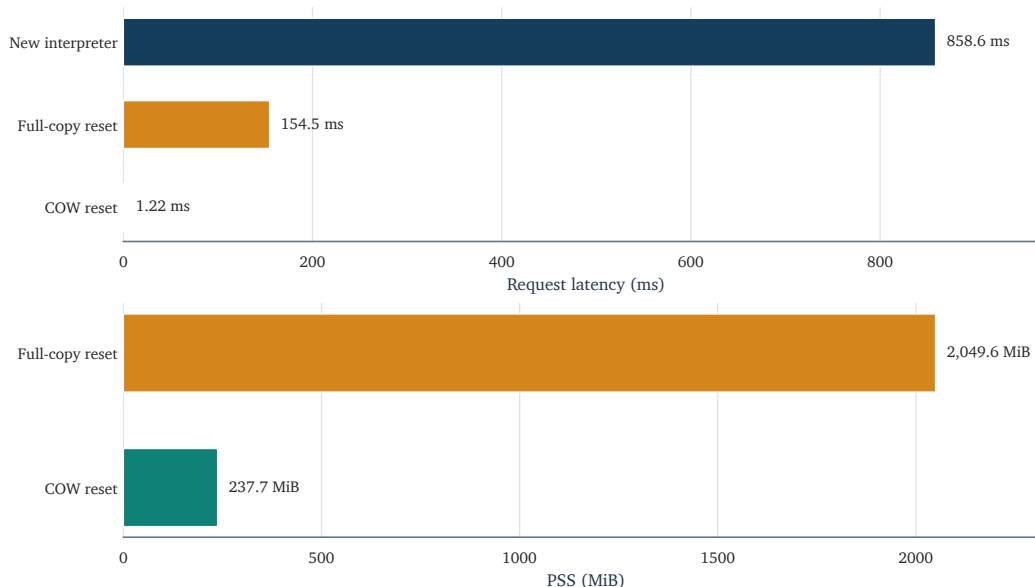


Figure 17: Request latency for new interpreters, full-copy reset and COW reset. PSS compares the full-copy and COW worker pools.

For this workload, median request latency falls from 154.5 ms with full-copy reset to 1.22 ms with copy-on-write. Starting a new interpreter takes 858.6 ms. Retained process PSS falls from 2,049.6 MiB to 237.7 MiB.

CHAPTER 7

Discussion

7.1 PLM design choices

A Host rule decides whether a request may start early. The available overlap decides whether it is worth doing: long, independent Host calls offer more waiting to hide. Short calls leave less waiting to offset the analysis cost.

7.1.1 Placement of early work

An AST describes program structure. Safe placement also depends on how imports, object creation, mutations and exceptions affect later execution. Moving a pure calculation is a simpler case than moving code that changes interpreter or provider state.

Pysolate uses the AST to place request preparation. In the CPU-only case in Section 6.2.4, PLM adds analysis to the same Python computation.

7.1.2 Execution state across source chunks

Python checks a complete source unit for syntax errors before executing its statements. Suppose an early chunk writes a file and a later chunk ends with an incomplete `if` (. Executing the chunks separately could write the file before discovering the error. Executing the complete source would raise a syntax error before the file write.

Pysolate parses the complete program before executing its Python statements in a new Guest. During source arrival, prefix analysis can start Host requests allowed by the operation rules. EAGER retains interpreter state across chunks and can overlap Python computation with generation. Pysolate keeps that computation in the final execution to preserve the complete program's execution order.

7.1.3 Where to start requests

Prefix analysis can start complete top-level calls while source is arriving. Once the full program is available, the AST pass places Prepare after the definitions and barriers it depends on, inside the controlling block. Independent requests can then overlap. A dependent request must wait until the program has produced its arguments.

The program's values, branches and loops already determine which calls are reached. A Guest-side dispatcher or a Host-side model would have to maintain some of this state separately. Pysolate leaves these decisions in the program. The Host stores requests, checks their validity and delivers their results.

7.1.4 Could Materialize move later?

If Linearize starts another request, delaying Materialize could overlap the new wait with work before the value's first use.

However, a failing call should prevent the work after it from running. Delaying Materialize could allow a file write to happen before that failure is raised. The prototype keeps Linearize and Materialize together to preserve this order. Moving Materialize later would need checks for exception order as well as value use.

7.2 Copy-on-write costs and reuse

7.2.1 Why copy-on-write fits this runtime

The reactor can return after Python initialisation while retaining its heap, and the engine separates compiled code from instance state. Pysolate can therefore map the initial memory into new Guests while creating their globals, tables and Host resources separately. Control over linear-memory allocation allows page sharing without a Go process snapshot. Section 5.4 gives the construction sequence.

7.2.2 Copy-on-write economics

Let B be the size of the resident initial memory image and D the average amount privately copied after writes by one instance. For these image pages, N instances use roughly NB bytes with separate copies, or $B + ND$ with copy-on-write.

The saving is largest when the interpreter and imported modules occupy many pages but each run writes to only a few. Short agent-generated programs can then share most of the runtime image while paying for their private writes.

For execution time, let M be the full initialised memory size. Excluding one-time image creation, the recurring costs can be approximated as

$$\begin{aligned} T_{\text{new}} &= T_{\text{startup}} + T_{\text{run}}, \\ T_{\text{copy}} &= T_{\text{copy}(M)} + T_{\text{run}}, \\ T_{\text{COW}} &= T_{\text{map}} + T_{\text{fault}(D)} + T_{\text{run}}, \end{aligned}$$

where $T_{\text{copy}(M)}$ is the cost of copying the full memory and $T_{\text{fault}(D)}$ is the cost of creating private pages after writes. Avoiding repeated initialisation can save time even when a large dirty set reduces memory savings. Image creation must also be repaid across the later runs, so time and memory can cross over at different instance counts.

7.3 Choosing and combining the mechanisms

PLM targets waiting inside a run. Copy-on-write targets repeated setup across runs. A service can use either or both, depending on whether Host waits, repeated initialisation or memory use dominate its workload.

CHAPTER 8

Limitations

8.1 Supported PLM calls and Host rules

The whole-program PLM pass accepts a limited subset. It places direct Host calls in module, `if`, `for` and `async for` blocks when their arguments and dependencies can be derived from the AST. Calls hidden behind helper functions, dynamic features or unresolved control flow remain unchanged.

The prototype keeps `Materialize` at the original call.

Safety depends on correct Host declarations. Unknown or mutating operations remain in program order. If a Host rule wrongly allows an earlier request, program behaviour can change. The prototype starts only read-like operations early. Retrying a request whose external outcome is unknown can repeat its effects.

`Prepare` can also change what an external provider observes. A request may disclose its arguments before the program reaches the source call, even if the candidate is later abandoned. `Pysolate` therefore allows early issue only when the Host rule allows the earlier provider-visible event. The present implementation uses read-like operations with stable benchmark data. Services with changing data, rate limits or access policies require their own rules.

Python makes prefix analysis practical because indentation and line breaks mark statement structure. Another language frontend would need to recover call sites, arguments and dependencies, while the Host would still provide effect and timing rules.

8.2 Supported packages and dataset selection

The current Guest includes NumPy. Extending dataset coverage to libraries such as pandas requires building their native extensions and dependencies for WASI, then linking and packaging them as described in Section 5.1.

Within the NumPy-only tasks, the experiment uses 15 programs whose literal numeric arrays can be replaced with one or two Host reads. It retains their original assertions and runs them with the same NumPy build [39]. Other input forms would need changes to this workload adapter.

8.3 Correctness coverage

The implementation passed the correctness checks for the experimental workloads. Adversarial security evaluation remains future work.

Adding Python packages with native shared libraries can require changes to their loading code, dependencies and system calls. Each port needs further tests. The passing NumPy checks

cover the operations used in this report. Behaviour for other functions, package versions and platforms remains untested.

8.4 Copy-on-write workloads and memory measurement

COW Fan-Out uses one 8 MiB array and fan-out up to eight. Image setup, instance creation and execution are measured separately. Pysolate's COW implementation is Linux-only. Other kernels and WebAssembly engines remain untested.

Memory use also depends on the allocator's layout. Spreading frequently modified objects across many pages causes more pages to be copied on writes. This can increase memory use even when the objects themselves are small. Different allocator layouts remain untested. Programs that grow linear memory aggressively or retain large native resources may also gain less.

The PSS samples describe memory held after execution on the tested machines. Peak usage, other input sizes, concurrent request arrival and other sandbox backends remain evaluation targets.

The Shimmy prototype installs one evaluation function and serves three requests. Larger functions, different imports, sustained queues and mixed request sizes may change both reset cost and memory sharing.

8.5 Controlled benchmarks and deployment assumptions

Whole-program PLM uses one source template, four read counts and three provider delays. Deterministic delays make the crossover interpretable, but they omit provider jitter, throttling and failures. The crossover points describe where overlap exceeds analysis cost for this program and runtime revision.

Streaming Overlap covers one 1.5-second operation and a 1.4-second source tail. Prefix scaling varies calls and source windows with deterministic 200 ms provider delays. The NumPy-derived study uses mock token rates and deterministic provider delay. All three studies assume reusable analyser and execution workers are already available. Worker startup adds a deployment cost that can outweigh the waiting saved.

Controlled replay covers five cases with three runs per case. The local EAGER-derived rule models routing decisions. Real-LLM streaming, natural provider latency, larger programs and sustained concurrency remain evaluation targets.

CHAPTER 9

Conclusion

This report investigated how to reduce Host-call waiting and repeated Python setup in an isolated runtime.

For RQ1, Pysolate applies PLM during source generation and to complete programs. Prefix analysis can start an eligible Host call once a complete call and its arguments are available in received source. The whole-program pass places Prepare after dependencies and inside the program's control flow. Linearize and Materialize remain at the original call in the prototype. The program therefore still decides which path runs and where a value or error appears. Python evaluates branches and dependencies. The Host manages provider access and timing rules.

For RQ2, Pysolate reuses an immutable image of initialised WebAssembly linear memory. Pre-initialised images provide copy-on-write Guests. Each run has its own module, permissions and workspace, and writes create private pages. Each run starts from the image captured after trusted setup.

For RQ3, prefix PLM removes a median 1.405 seconds from the streamed example, while whole-program PLM saves 1.268 seconds with eight independent 200 ms calls. In the measured COW workload, copy-on-write becomes faster at two instances and uses less memory at four. In Shimmy, image reset lowers request latency from 154.5 to 1.22 ms and pool memory use from 2,049.6 to 237.7 MiB.

References

- [1] Docker. What is Docker? 2026. <https://docs.docker.com/get-started/docker-overview/> (accessed September 2, 2026).
- [2] Agache A, Brooker M, Florescu A, Iordache A, Liguori A, Neugebauer R, et al. Firecracker: Lightweight Virtualization for Serverless Applications. In: 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20), USENIX Association; 2020, pp. 419–34.
- [3] Google. What is gVisor? 2018. <https://gvisor.dev/docs/>.
- [4] Anjali, Caraza-Harter T, Swift MM. Blending containers and virtual machines: A study of Firecracker and gVisor. In: Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '20), ACM; 2020, pp. 101–13. <https://doi.org/10.1145/3381052.3381315>.
- [5] W3C WebAssembly Working Group. WebAssembly Core Specification. 2026.
- [6] Haas A, Rossberg A, Schuff DL, Titzer BL, Holman M, Gohman D, et al. Bringing the web up to speed with WebAssembly. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '17), ACM; 2017, pp. 185–200. <https://doi.org/10.1145/3062341.3062363>.
- [7] Bytecode Alliance. WASI 0.1 (Preview 1) 2026. <https://wasi.dev/releases/wasi-p1> (accessed August 26, 2026).
- [8] WebAssembly Community Group. WASI Application ABI 2020. <https://github.com/WebAssembly/WASI/blob/snapshot-01/design/application-abi.md>.
- [9] WASI SDK n.d.. <https://github.com/WebAssembly/wasi-sdk/blob/wasi-sdk-33/README.md> (accessed September 7, 2026).
- [10] Building C and C++ Extensions n.d.. <https://docs.python.org/3.14/extending/building.html> (accessed September 7, 2026).
- [11] CPython 3.14.0: Unix Dynamic Extension Loader n.d.. https://github.com/python/cpython/blob/v3.14.0/Python/dynload_shlib.c (accessed September 7, 2026).
- [12] Dynamic Linking n.d.. <https://emscripten.org/docs/compiling/Dynamic-Linking.html> (accessed September 7, 2026).
- [13] Asynchronous Code n.d.. <https://emscripten.org/docs/porting/asyncify.html> (accessed September 7, 2026).
- [14] Sun Z, Lin Z, Chen Z, Yang C, Zhou M, Li L, et al. Executing as You Generate: Hiding Execution Latency in LLM Code Interpreters 2026. <https://arxiv.org/abs/2604.00491>.
- [15] Xu Y, Kong X, Chen T, Zhuo D. Conveyor: Efficient Tool-aware LLM Serving with Tool Partial Execution 2024. <https://arxiv.org/abs/2406.00059>.
- [16] Feng G, Mao H, Dutta B, Gonzalez JE. Concurrency without Model Changes: Future-based Asynchronous Function Calling for LLMs 2026. <https://arxiv.org/abs/2605.15077>.
- [17] Kim S, Moon S, Tabrizi R, Lee N, Mahoney MW, Keutzer K, et al. An LLM Compiler for Parallel Function Calling. In: Salakhutdinov R, Kolter Z, Heller K, Weller A, Oliver N, Scarlett J, et al., editors. Proceedings of the 41st International Conference on Machine Learning, vol. 235, PMLR; 2024, pp. 24370–91.
- [18] Cloudflare. How Code Mode Works 2026. <https://developers.cloudflare.com>.

- com/agents/tools/codemode/how-it-works/.
- [19] Vercel. Introducing Run SDK: Secure Eval for Your Agents 2026. <https://vercel.com/blog/introducing-run> (accessed August 26, 2026).
 - [20] Fraser K, Chang F. Operating System I/O Speculation: How Two Invocations Are Faster Than One. In: 2003 USENIX Annual Technical Conference (USENIX ATC 03), USENIX Association; 2003, pp. 325–38.
 - [21] Nightingale EB, Chen PM, Flinn J. Speculative Execution in a Distributed File System. In: Proceedings of the Twentieth ACM Symposium on Operating Systems Principles, Association for Computing Machinery; 2005, pp. 191–205. <https://doi.org/10.1145/1095810.1095829>.
 - [22] Prabhu P, Ramalingam G, Vaswani K. Safe Programmable Speculative Parallelism. In: Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation, Association for Computing Machinery; 2010, pp. 50–61. <https://doi.org/10.1145/1806596.1806603>.
 - [23] Mohammadi B, Klein L, Arora A, Bind-schaedler L. Ghost Tool Calls: Issue-Time Privacy for Speculative Agent Tools 2026. <https://arxiv.org/abs/2606.02483>.
 - [24] Ye N, Ahuja A, Liargkovas G, Lu Y, Kaffes K, Peng T. Speculative Actions: A Lossless Framework for Faster Agentic Systems 2026. <https://arxiv.org/abs/2510.04371>.
 - [25] Nichols D, Singhanian P, Jekel C, Bhatele A, Menon H. Optimizing Agentic Language Model Inference via Speculative Tool Calls 2025. <https://arxiv.org/abs/2512.15834>.
 - [26] Sui Y, Zhao H, Ma R, He Z, Wang H, Li J, et al. Parallelizing Tool Execution and LLM Generation for Low-Latency Agent Serving 2026. <https://arxiv.org/abs/2603.18897>.
 - [27] Bai H, Lv W, Zheng H, Lu Y, Shu J. SPORK: Self-Speculative Forking to Accelerate Agentic LLM Inference 2026. <https://arxiv.org/abs/2607.03333>.
 - [28] Saberi M, Rezaei K, Feizi S. SpecHop: Continuous Speculation for Accelerating Multi-Hop Retrieval Agents 2026. <https://arxiv.org/abs/2605.21965>.
 - [29] Chen HM, Guo J, Luk W, Fan H. AOSpec: Action and Observation Co-Speculation for Low-Latency Agent Serving 2026. <https://arxiv.org/abs/2608.00881>.
 - [30] Zhang Y, Wo T, Wang J, Sun X, Zhang M, Yuan C, et al. SpecBox: Speculative Sandbox Scheduling for Efficient LLM Agent Serving 2026. <https://arxiv.org/abs/2607.23933>.
 - [31] Lucassen JM, Gifford DK. Polymorphic Effect Systems. In: Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Association for Computing Machinery; 1988, pp. 47–57. <https://doi.org/10.1145/73560.73564>.
 - [32] Debenedetti E, Shumailov I, Fan T, Hayes J, Carlini N, Fabian D, et al. Defeating Prompt Injections by Design 2025. <https://arxiv.org/abs/2503.18813>.
 - [33] Necula GC. Proof-Carrying Code. In: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Association for Computing Machinery; 1997, pp. 106–19. <https://doi.org/10.1145/263699.263712>.
 - [34] Cadden J, Unger T, Awad Y, Dong H, Krieger O, Appavoo J. SEUSS: Skip Redundant Paths to Make Serverless Fast. In: Proceedings of the Fifteenth European Conference on Computer Systems, New York, NY, USA: Association for Computing Machinery; 2020. <https://doi.org/10.1145/3342195.3392698>.
 - [35] Alzayat M, Mace J, Druschel P, Garg D. Groundhog: Efficient Request Isolation

- in FaaS. In: Proceedings of the Eighteenth European Conference on Computer Systems, New York, NY, USA: Association for Computing Machinery; 2023, pp. 398–415. <https://doi.org/10.1145/3552326.3567503>.
- [36] Ustiugov D, Petrov P, Kogias M, Bugnion E, Grot B. Benchmarking, Analysis, and Optimization of Serverless Function Snapshots. In: Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, New York, NY, USA: Association for Computing Machinery; 2021, pp. 559–72. <https://doi.org/10.1145/3445814.3446714>.
- [37] Ao L, Porter G, Voelker GM. FaaSnap: FaaS Made Fast Using Snapshot-Based VMs. In: Proceedings of the Seventeenth European Conference on Computer Systems, New York, NY, USA: Association for Computing Machinery; 2022, pp. 730–46. <https://doi.org/10.1145/3492321.3524270>.
- [38] Shillaker S, Pietzuch P. Faasm: Lightweight isolation for efficient stateful serverless computing. In: 2020 USENIX Annual Technical Conference (USENIX ATC 20), USENIX Association; 2020, pp. 419–33.
- [39] Shi Y. Agent Python Runtime 2026. <https://github.com/bkmashiro/agent-python-runtime/tree/3a0e32cad2a67967a73292eb4f04faaa4b5793e3> (accessed September 2, 2026).
- [40] wazero contributors. wazero: The Zero Dependency WebAssembly Runtime for Go Developers 2020. <https://github.com/wazero/wazero>.
- [41] Lambda Feedback. Evaluation Functions 2026. https://docs.lambdafeedback.com/teacher/reference/evaluation_functions/.
- [42] Pfurtscheller A. Shimmy 2026. <https://github.com/lambda-feedback/shimmy/releases/tag/v0.0.6> (accessed August 26, 2026).
- [43] Shi Y. Shimmy WASI Evaluator Runtime Prototype 2026. <https://github.com/bkmashiro/shimmy/tree/7a5993fa3ffb80cd72b39c4b0ddb5fa8c9bf8744> (accessed August 29, 2026).
- [44] Misra D, Islah N, May V, Rauby B, Wang Z, Gehring J, et al. GitChameleon 2.0: Evaluating AI Code Generation Against Python Library Version Incompatibilities 2025. <https://arxiv.org/abs/2507.12367>.

Declaration

Generative AI. OpenAI Codex was used as a collaborative tool for programming and textual review.

Ethics. Only publicly available datasets were used. The project did not involve human participants or personal data.

Sustainability. The project used workstations provided by Imperial College London for experiments.

Access to code and materials. The Pysolate runtime and experiment materials are available at github.com/bkmashiro/agent-python-runtime. The evaluated Shimmy revision is available at github.com/bkmashiro/shimmy. Its upstream repository is github.com/lambda-feedback/shimmy.

Appendices

A. Whole-program PLM timing source template

Every timing case uses the same Python body: 750 chained increments followed by zero, one, two, four or eight independent reads. Only the read count and provider delay change. The same Python work gives PLM something to overlap with the reads.

```
x0 = 0
x1 = x0 + 1
# ... x2 through x749 ...
x750 = x749 + 1
v0 = sources.read("key-0")
# ... zero, one, two, four or eight reads ...
result = [v0, ..., x750]
```

The non-zero read counts are 1, 2, 4 and 8, and the provider delays are 25, 75 and 200 ms. A separate zero-read control retains the assignments and result check but issues no provider request.

B. Copy-on-write results

Each delta is calculated between corresponding runs before taking the median. The total-time columns report their own medians. Negative differences favour copy-on-write.

Instances	Per-instance total	COW total	Time difference	PSS difference
1	12.105 s	16.738 s	+4.573 s	+59.5 MiB
2	24.071 s	21.179 s	−2.867 s	+42.7 MiB
4	48.426 s	30.176 s	−18.113 s	−314.4 MiB
8	96.636 s	47.869 s	−48.434 s	−534.0 MiB

Table 3: Copy-on-write lifecycle totals and corresponding-run differences by instance fan-out.

C. Controlled replay programs

Table 2 uses four program bodies. Unless stated otherwise, source chunks arrive at 0, 0.7 and 1.4 seconds and each Host call takes 1.5 seconds. The short-call case changes the provider delay to 25 ms. The CPU-only case has chunks at 0 and 1.4 seconds. Each listing below names the row that uses it.

One eligible Host call. Used by “One eligible Host call” with the default schedule and by “Short 25 ms Host call” with the shorter provider delay:

```
# chunk 1: t = 0 s
value = sources.read("alpha")
# chunk 2: t = 0.7 s
label = value.upper()
# chunk 3: t = 1.4 s
result = [label, 12]
print(result)
```

Two independent Host calls. Used by the row with the same name:

```
# chunk 1: t = 0 s
left = sources.read("alpha")
right = sources.read("beta")
# chunk 2: t = 0.7 s
left_label = left.upper()
right_label = right.upper()
# chunk 3: t = 1.4 s
result = [left_label, right_label, 12]
print(result)
```

CPU-only Python. Used by the row with the same name:

```
# chunk 1: t = 0 s
acc = sum((i * 17) % 97 for i in range(20000000)) % 1000000007
# chunk 2: t = 1.4 s
result = acc
print(result)
```

Dependent second Host call. Used by the row with the same name. The second key is available only after the first result:

```
# chunk 1: t = 0 s
left = sources.read("alpha")
right = sources.read(left)
# chunk 2: t = 0.7 s
left_label = left.upper()
right_label = right.upper()
# chunk 3: t = 1.4 s
result = [left_label, right_label, 12]
print(result)
```

D. Prefix-scaling programs

The controlled matrix places all independent calls in the first chunk and the checked result in the second. The second chunk arrives 0, 100, 200 or 400 ms later. The template below expands to one, two, four or eight calls, each with a 200 ms simulated provider delay.

```
value0 = sources.read("input-0")
# ... up to value7 for the eight-call case ...

# second source chunk
result = [value0, ..., "done"]
print(result)
```

The NumPy-derived study uses the same two-chunk shape. The runtime image has NumPy imported as np. Host calls return the literal arrays used by the original program. This two-call convolution case is representative:

```
_host_arr1 = sources.read("gitchameleon/66/arr1")
_host_arr2 = sources.read("gitchameleon/66/arr2")

# streamed suffix
arr1 = np.array([int(x) for x in _host_arr1.split(",")])
arr2 = np.array([float(x) for x in _host_arr2.split(",")])
def apply_convolution_full(arr1, arr2):
    return np.convolve(arr1, arr2, mode="full")
```

```
assert apply_convolution_full(arr1, arr2).all() == False
result = {"example_id": "66", "oracle": "passed"}
```

The study uses 15 NumPy-derived programs with literal-array inputs replaced by Host calls. The experiment records include the complete cell summaries and source identities.

E. Comparison of related systems

System	Work moved or reused	Control and visible state
EAGER [14]	Complete Python chunk	Persistent interpreter retains early Python state
Conveyor [15]	Partial tool work	Tool integration defines readiness
AsyncFC [16]	Future-backed call	Program exposes unresolved values
LLMCompiler [17]	Planner task	Planner graph controls readiness
Speculator [21]	Checkpointed execution	Validation accepts or restores process state
Groundhog [35]	Process snapshot	The same process is restored between requests
SEUSS [34]	Snapshot-backed unikernel context	New, restored or idle contexts retain initialised state
REAP [36]	Recorded VM working-set pages	A separate microVM prefetches likely pages
FaaSnap [37]	VM snapshot regions	A separate microVM overlaps restore and paging
Faasm [38]	Proto-Faaslet and explicit shared state	Private memory resets, while declared shared state remains visible
Pysolate PLM	Host request	The Guest materialises the outcome at its original call
Pysolate COW	Pre-initialised runtime image	Each run receives a new Guest instance

Table 4: Work moved or reused, retained state and control boundaries in systems related to Pysolate.

F. Experimental details

All experiments use Linux/x86-64. Streaming Overlap, whole-program PLM and COW Fan-Out use four workstations with 16 logical CPUs and Linux 6.17. The other experiments use one workstation. Comparisons pair runs on the same workstation and rotate the execution order.

Experiment	Repeated runs
Streaming Overlap	20 corresponding pairs
Prefix scaling	Ten pairs for each of 16 heatmap cells
NumPy-derived programs	600 pairs across 15 programs and four source rates
Whole-program PLM	24 corresponding pairs for each setting
COW Fan-Out	12 corresponding pairs for each instance count
Controlled replay	Three runs per case
Shimmy	Three latency runs and four separate memory measurements

Table 5: Repeated runs used by the experiments.

Plots show medians and interquartile ranges unless their captions state otherwise. Differences are calculated within corresponding runs before taking the median. For multi-machine comparisons, 95% confidence intervals use 10,000 hierarchical bootstrap resamples: machines are sampled first, then pairs within each sampled machine. For single-machine comparisons, uncertainty is estimated over runs. The interquartile bars describe run variation. The reported confidence intervals describe uncertainty in the median.

In Shimmy, each latency run sends three requests in sequence. The second and third requests contribute to the latency result. Full-copy reset and COW reset each use a pool of four workers. Memory is the median process PSS from four separate measurements taken before pool shutdown.