

Pysolate

A Semantics-Aware Sandbox
for Efficient Isolated Execution

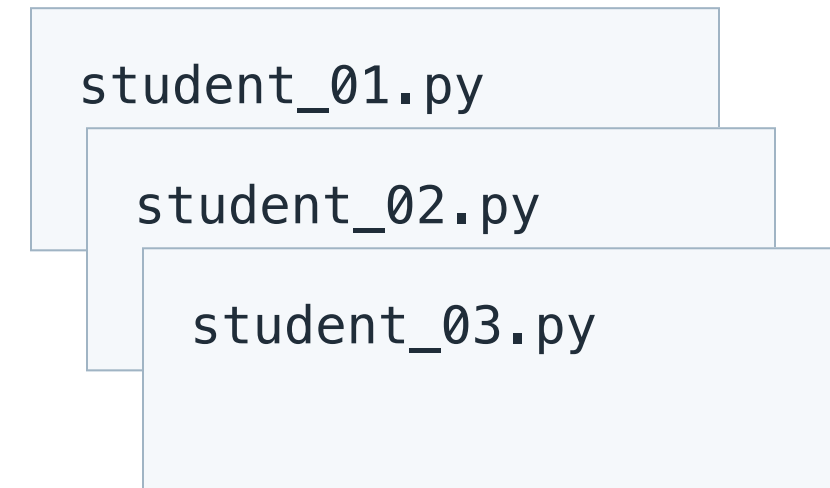
Yuzhe Shi

MSc Individual Project · Imperial College London

Running students' code

Imagine a data analysis class. Students need a platform to run their code.

We start with a Python server. It executes submissions in the same interpreter.



 **Python server**

```
exec(submission)
```

Read the scores, then take their mean

Normal run Mallory Bob

```
import numpy as np
scores = np.loadtxt("scores.csv")
result = scores.mean()
```

[1., 2., 3.]

2.0

Mallory prepares a replacement function

Normal run **Mallory** Bob

```
import numpy as np

def missing_file(path):
    raise FileNotFoundError(path)
```

Mallory changes the shared interpreter

Normal run **Mallory** Bob

```
import numpy as np

def missing_file(path):
    raise FileNotFoundError(path)

np.loadtxt = missing_file
```

Bob runs the original program

Normal run Mallory **Bob**

```
import numpy as np
scores = np.loadtxt("scores.csv")
result = scores.mean()
```

Bob had never met Mallory. His interpreter had.

Normal run Mallory **Bob**

```
import numpy as np
scores = np.loadtxt("scores.csv")
result = scores.mean()
```

FileNotFoundError

Bob had never met Mallory. His interpreter had.

Normal run Mallory **Bob**

```
import numpy as np
scores = np.loadtxt("scores.csv")
result = scores.mean()
```

FileNotFoundError

How can every run start clean?

Give each run its own interpreter

Mallory

Interpreter

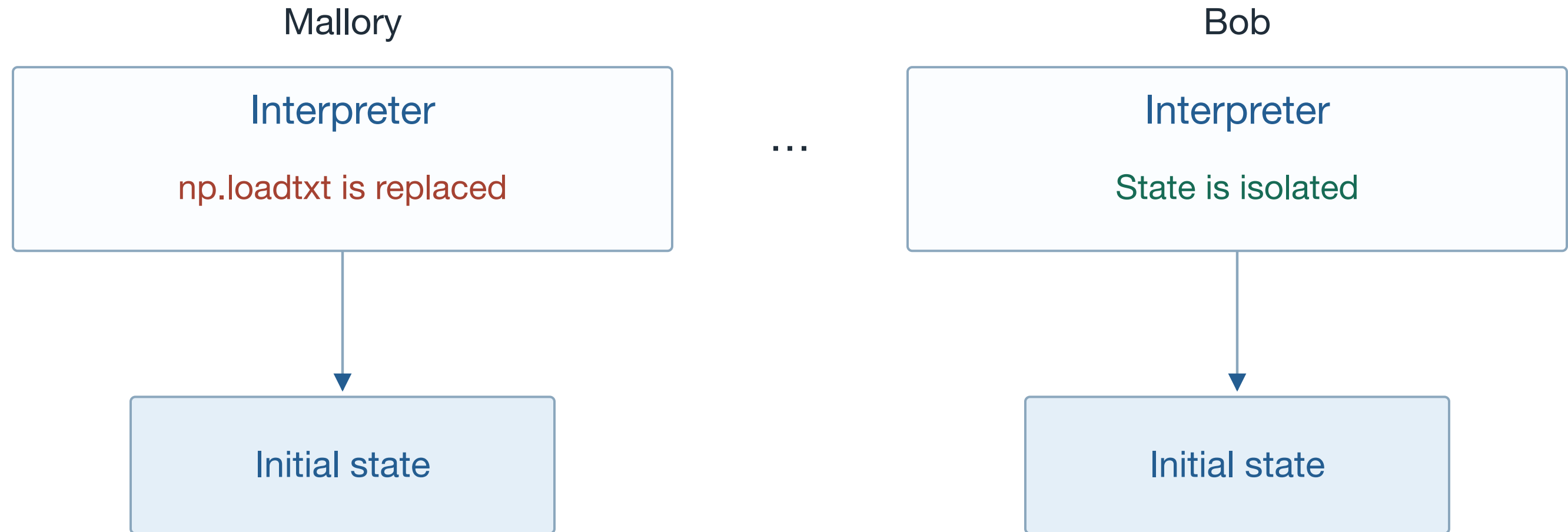
np.loadtxt is replaced

Bob

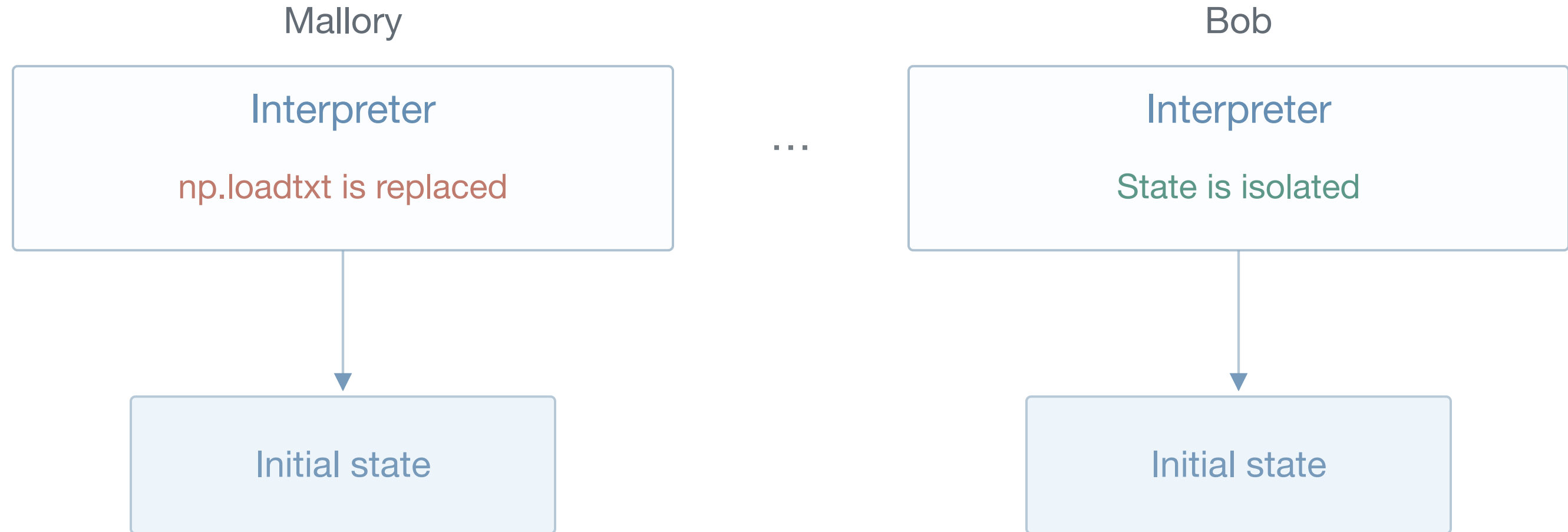
Interpreter

State is isolated

Each interpreter repeats the preparation



Each interpreter repeats the preparation

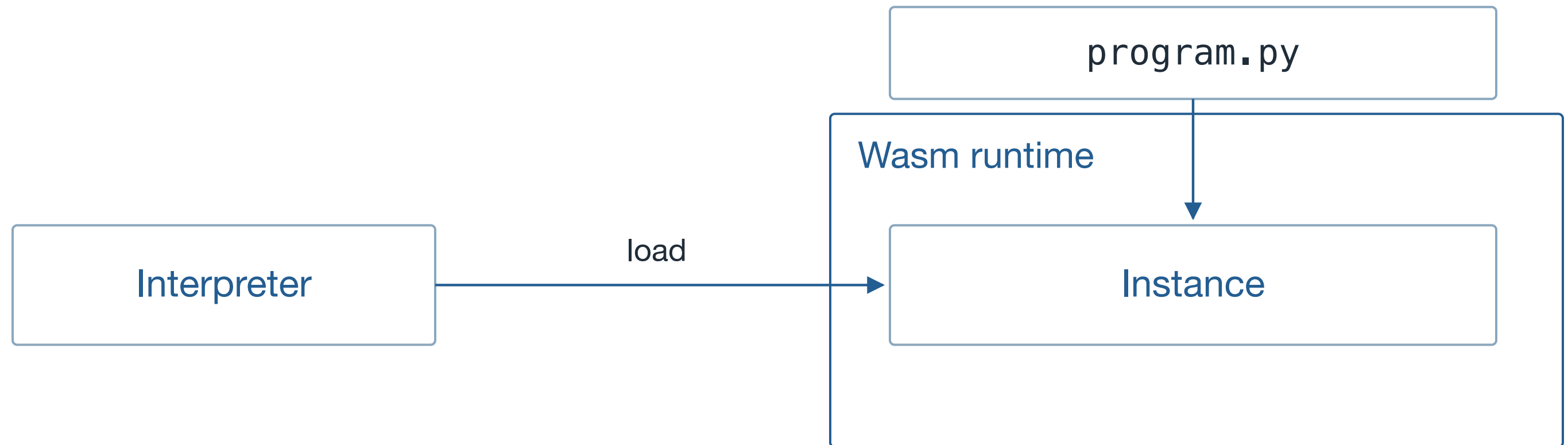


Can we keep runs independent and use fewer resources?

WebAssembly (Wasm)

Wasm is a portable, low-level instruction format.

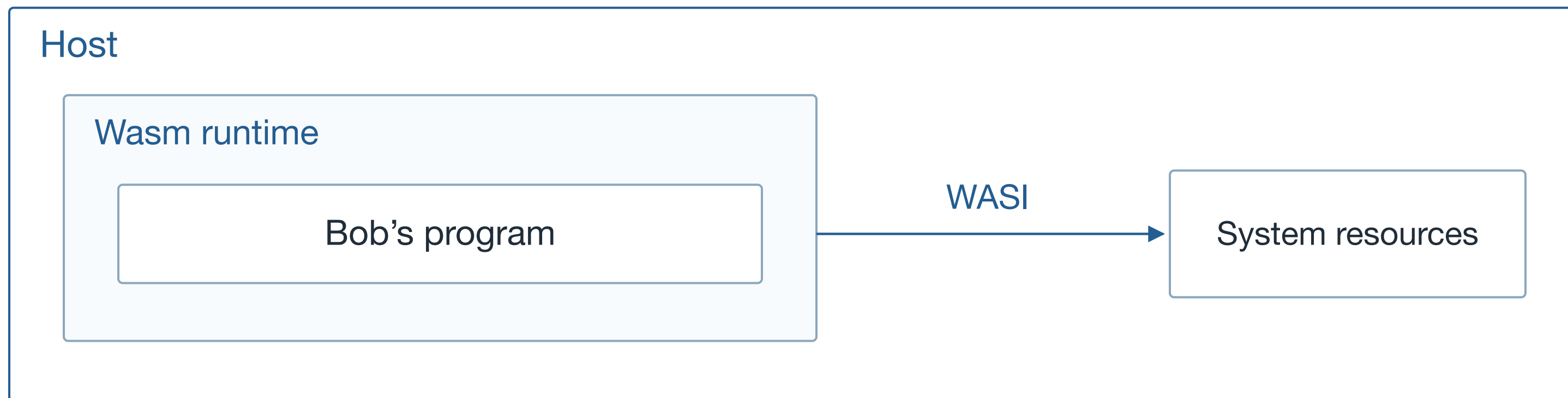
CPython can be compiled to Wasm.



How does the program read a file?

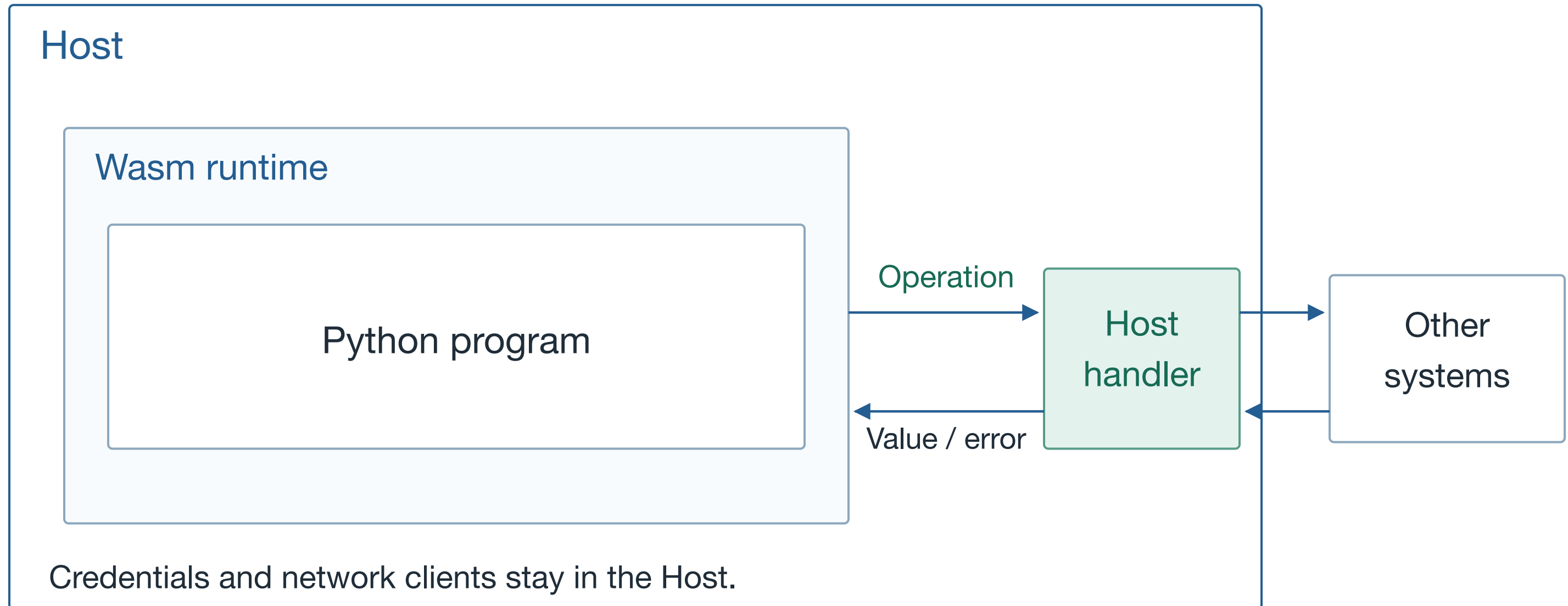
WASI: WebAssembly System Interface

Files and directories, standard input/output, clocks, random numbers, arguments and environment variables.



The Host chooses which resources are available.

Access other systems through named operations



Code mode: combine these operations in Python

```
acme = market.recent_price("ACME")  
beta = market.recent_price("BETA")  
portfolio_value = acme + beta
```

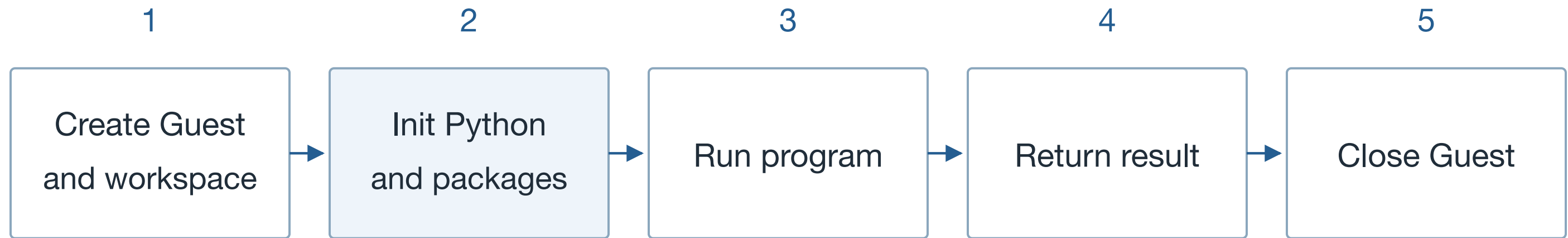
External call

External call

Local calculation

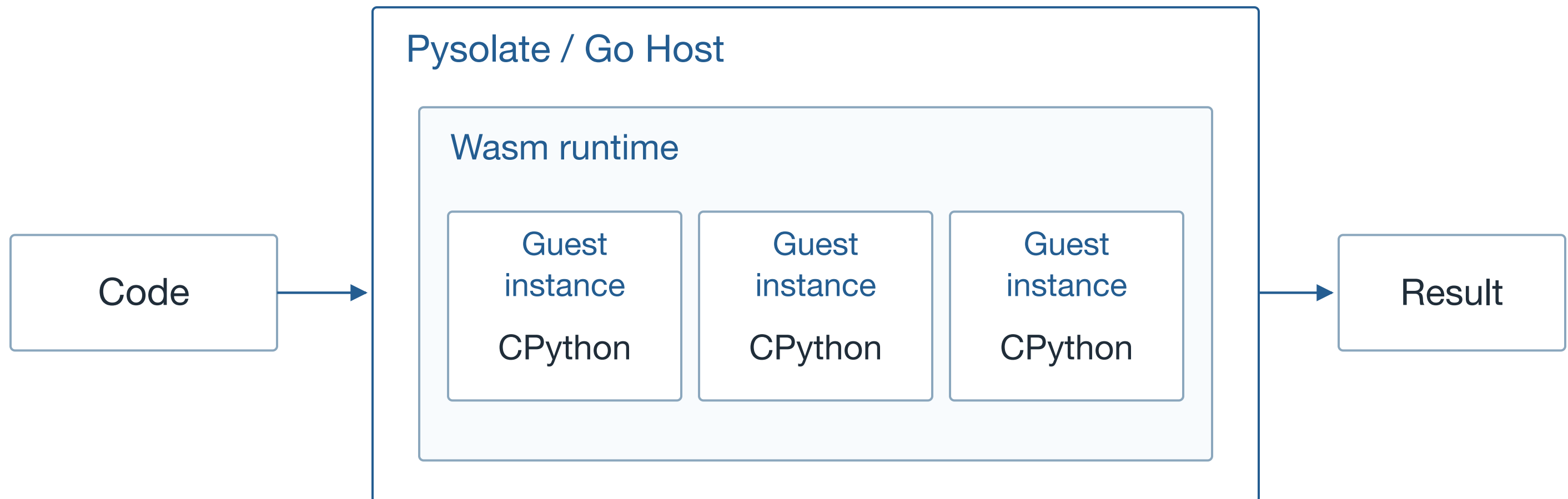
External operations are exposed as Python functions.

A basic execution model



Goal: reduce setup cost and waiting.

Pysolate: efficient, isolated Python execution



A natural fit for agent-generated code

Generated programs

Sandbox requirements

Short tasks



Fast startup

Frequent runs



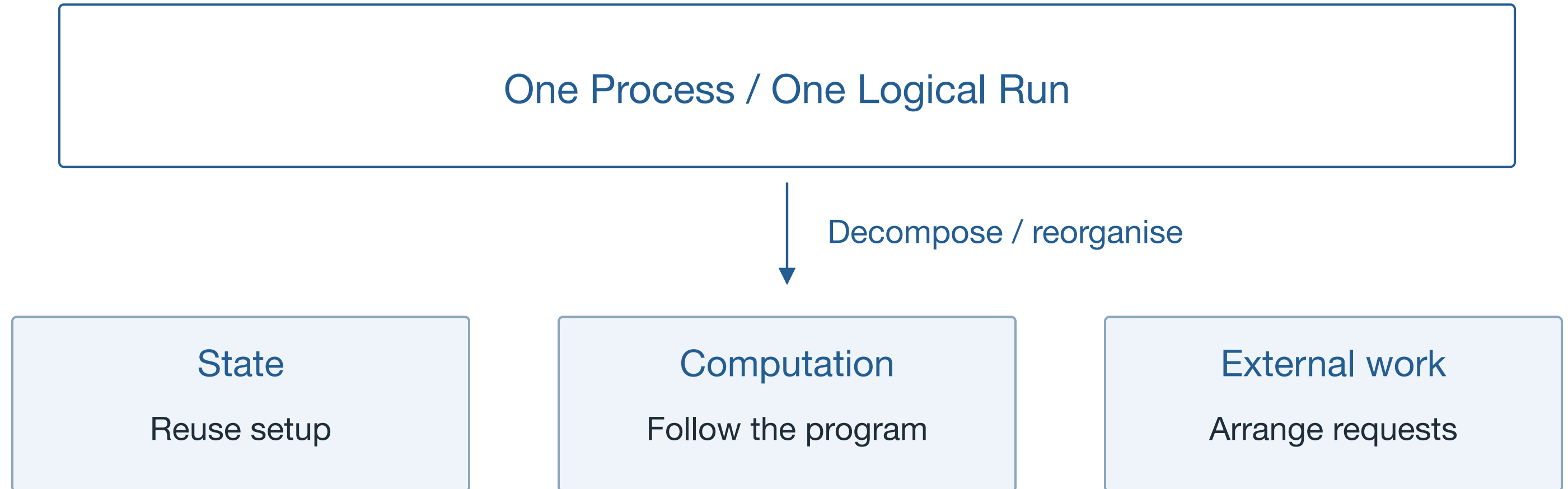
Low setup and memory cost

Untrusted code

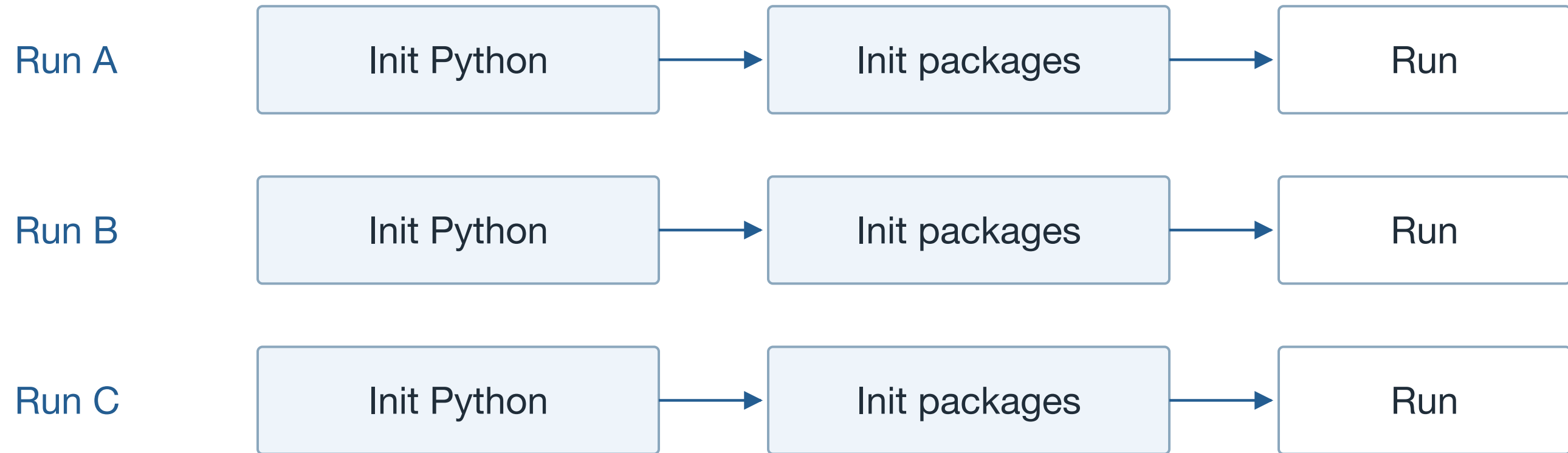


Isolated state and controlled access

Our idea: reorganise memory and external work



Avoid repeating interpreter setup



Prepare the starting state once?

Load the CPython Wasm module

Host actions

- 1 Load CPython module

Linear memory

+ Module data

Initialise Python inside the instance

Host actions

1 Load CPython module

2 Init Python

Linear memory

+ Module data

+ Python objects

Load NumPy into the starting state

Host actions

- 1 Load CPython module
- 2 Init Python
- 3 Init Packages

Linear memory

- + Module data
- + Python objects
- + NumPy objects

Save after setup, before user code runs

Host actions

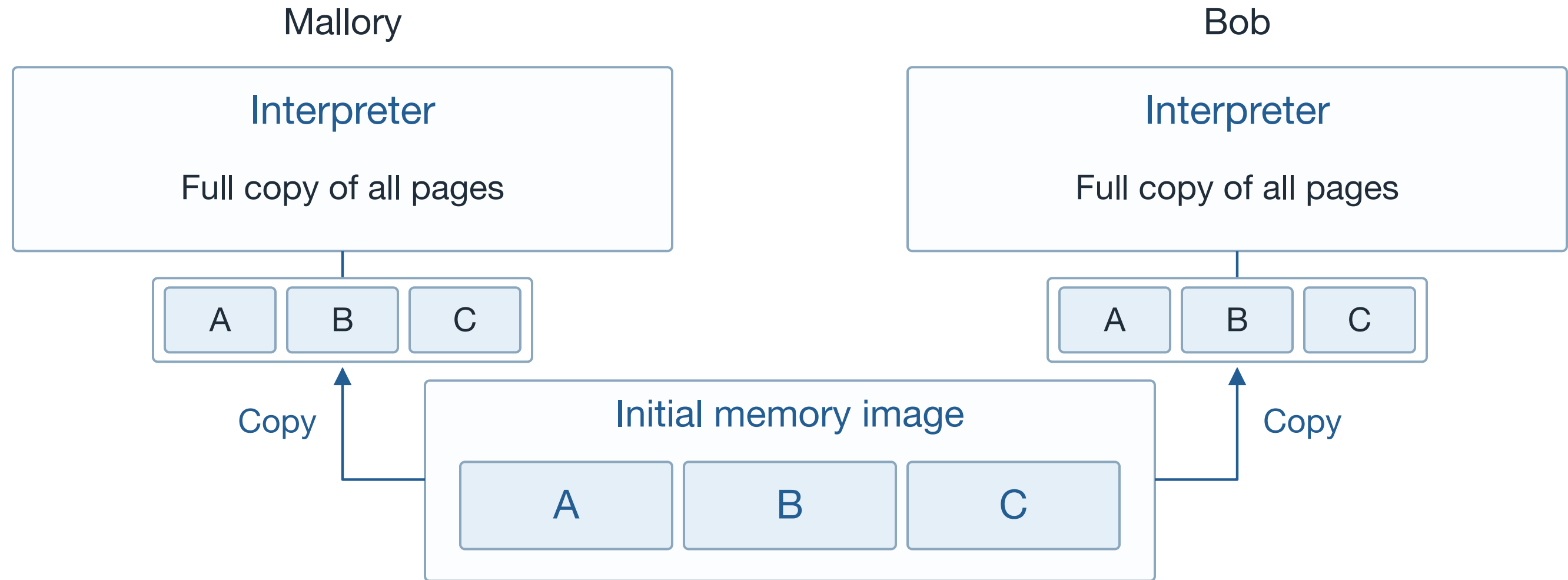
- 1 Load CPython module
- 2 Init Python
- 3 Init Packages
- 4 Snapshot

Linear memory

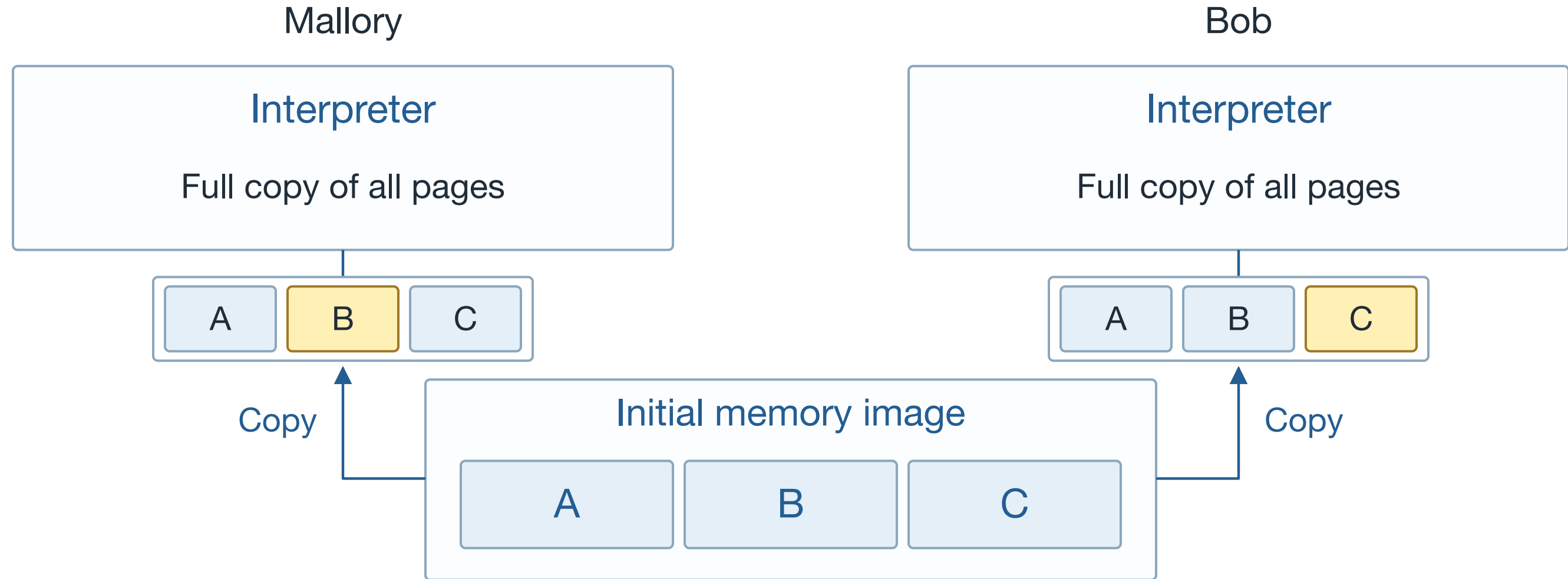
- + Module data
- + Python objects
- + NumPy objects

Saved linear memory

Full copy: one memory copy per run

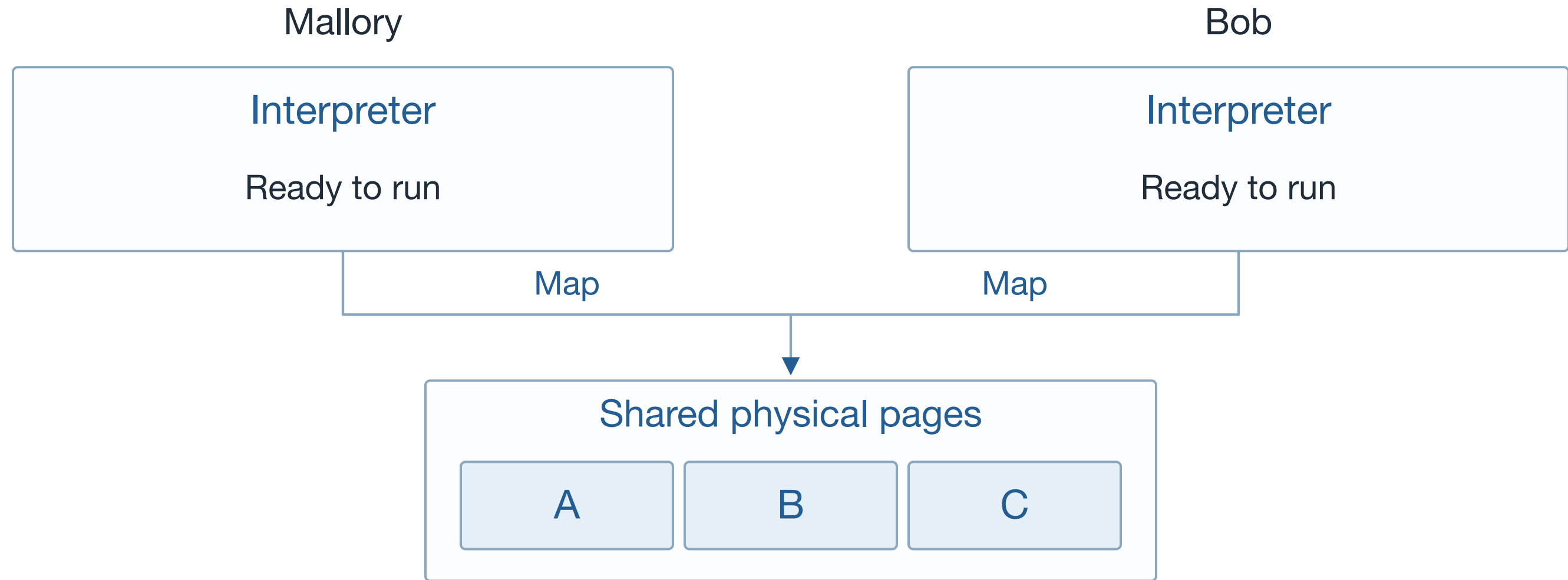


A short run may change only a few pages



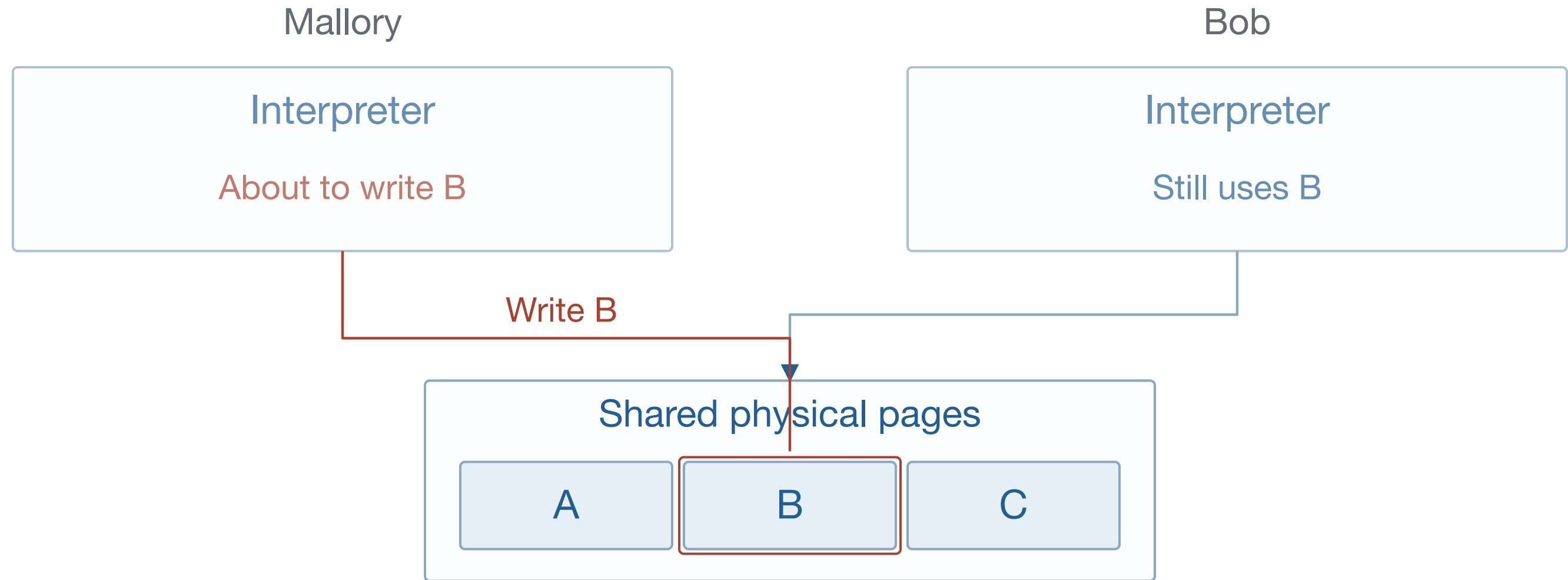
Can runs share the unchanged physical pages?

Copy-on-write: map shared physical pages



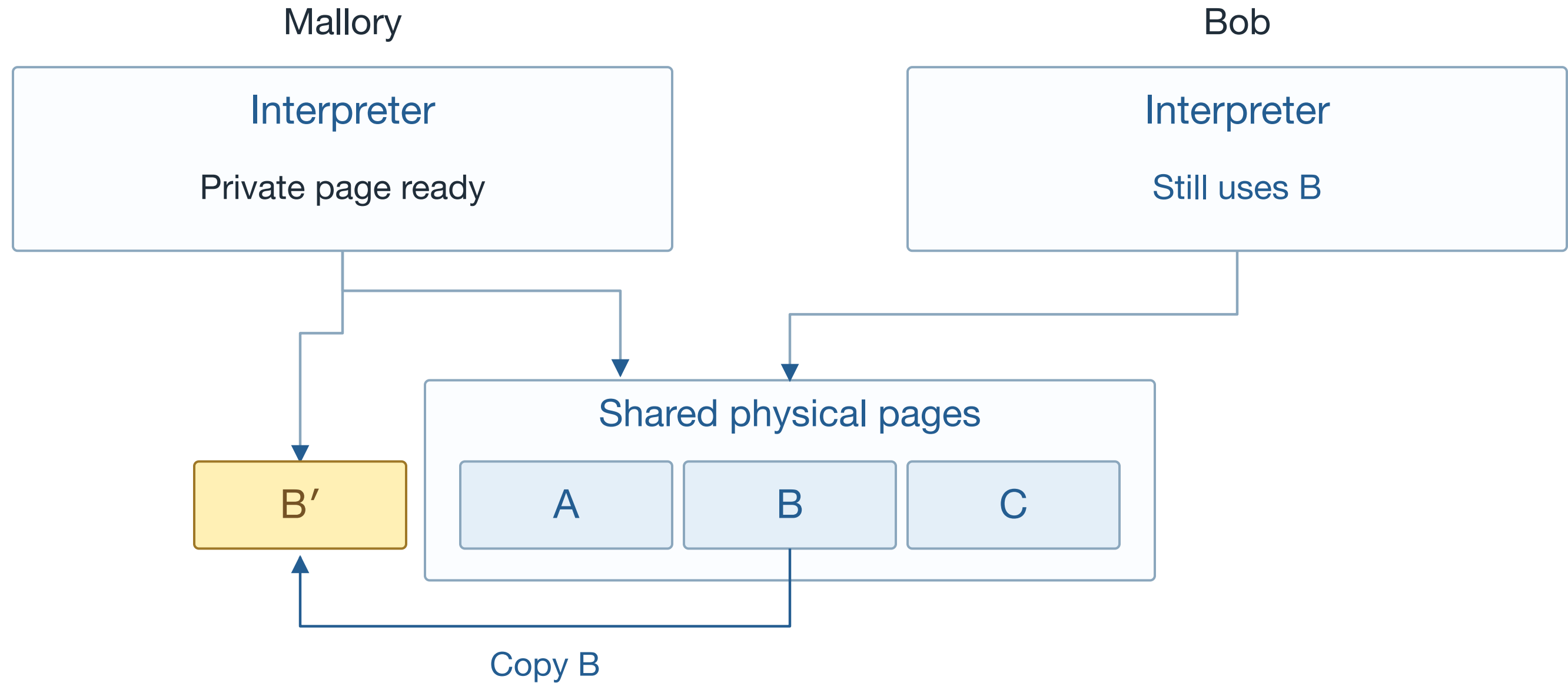
Two instances. One shared set of physical pages.

Mallory is about to write page B

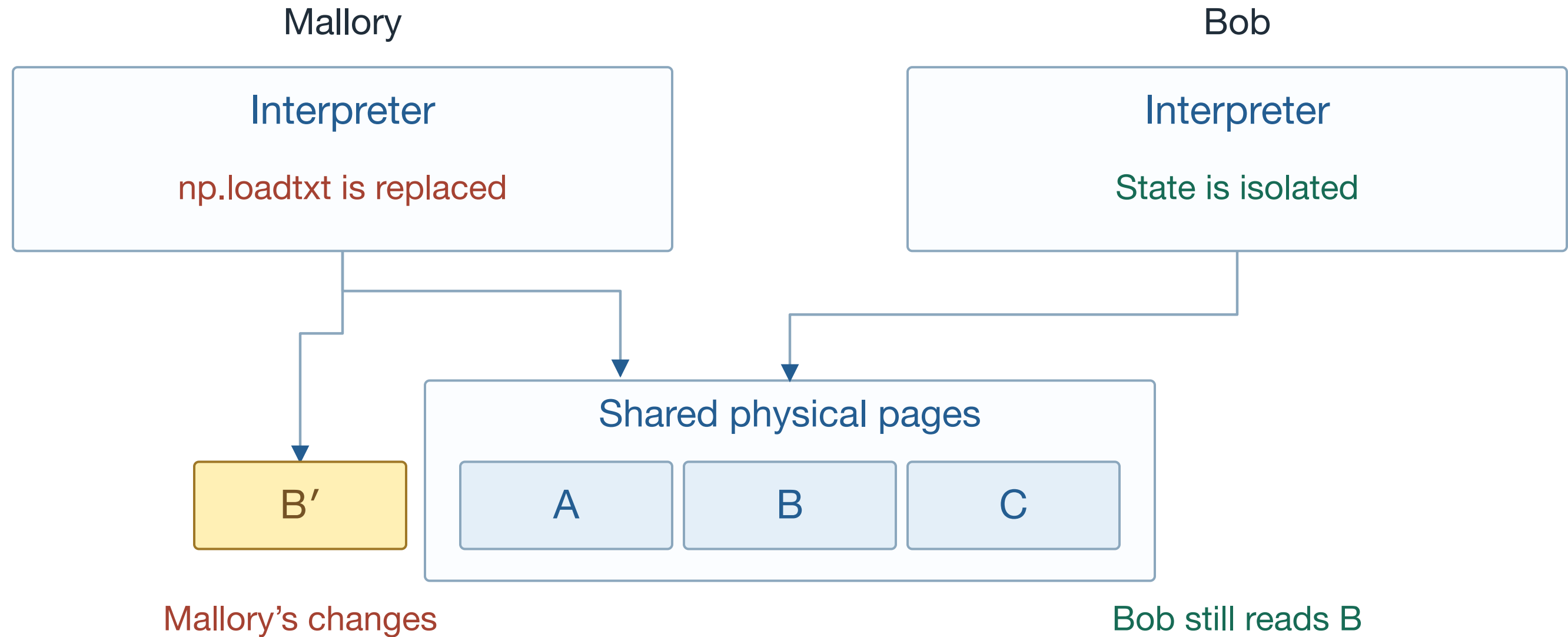


What happens when Mallory writes?

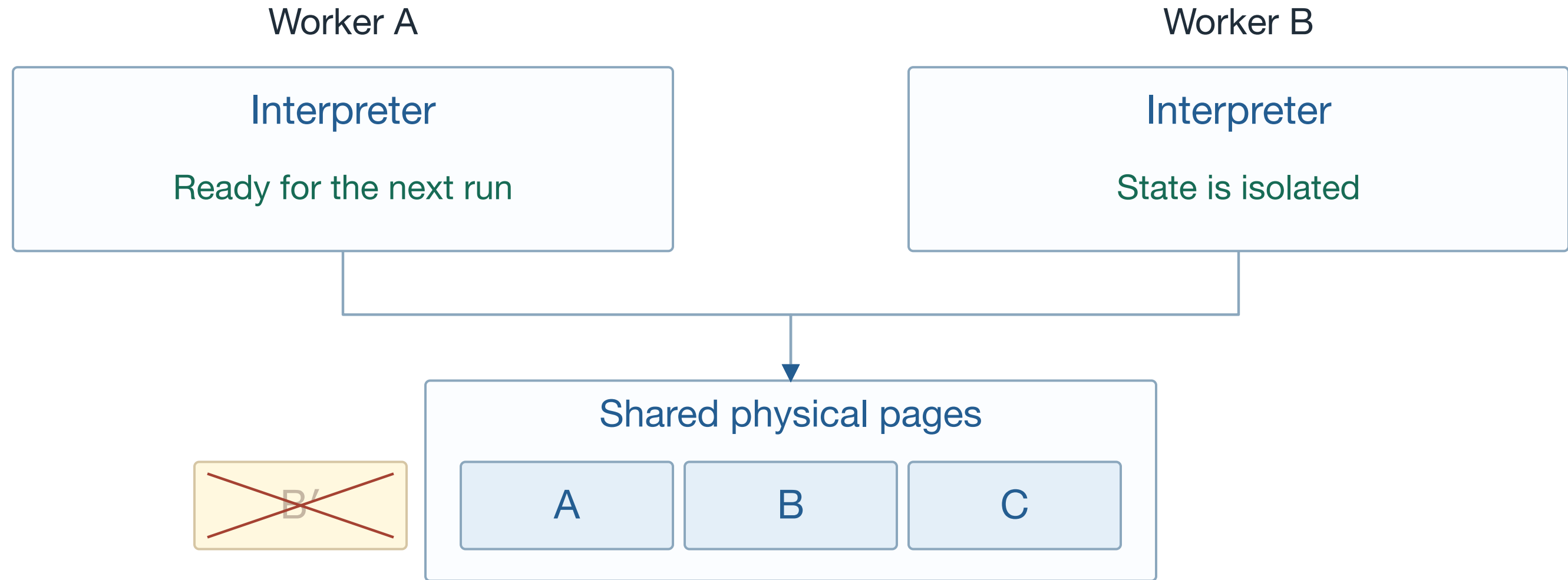
A write creates a private page



The interpreters keep their own changes

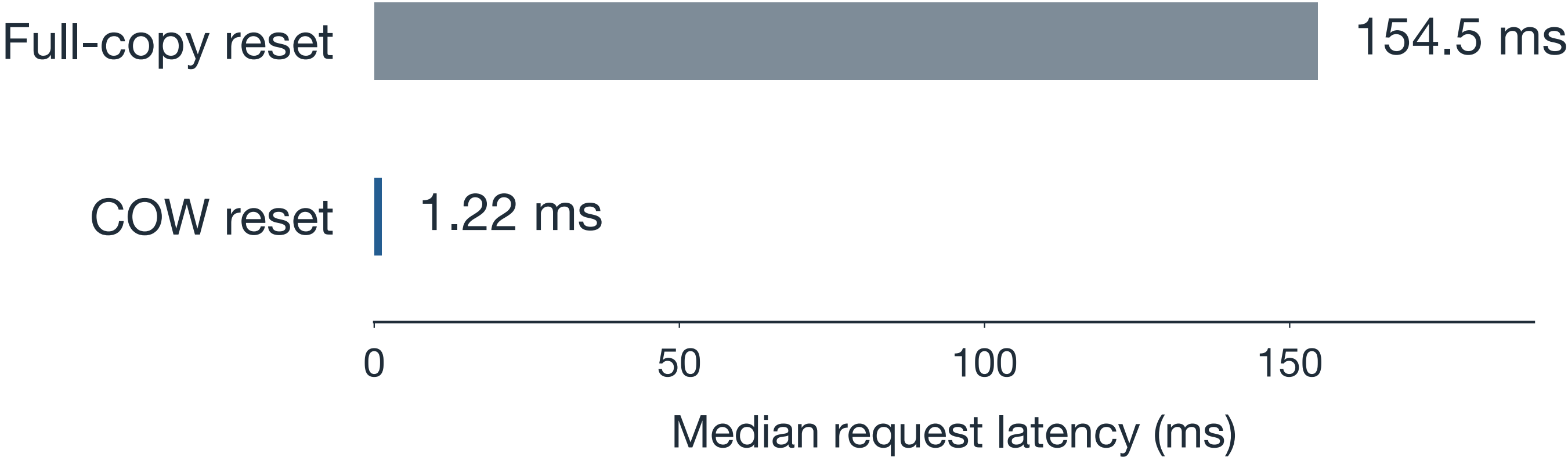


Reset an evaluator between requests

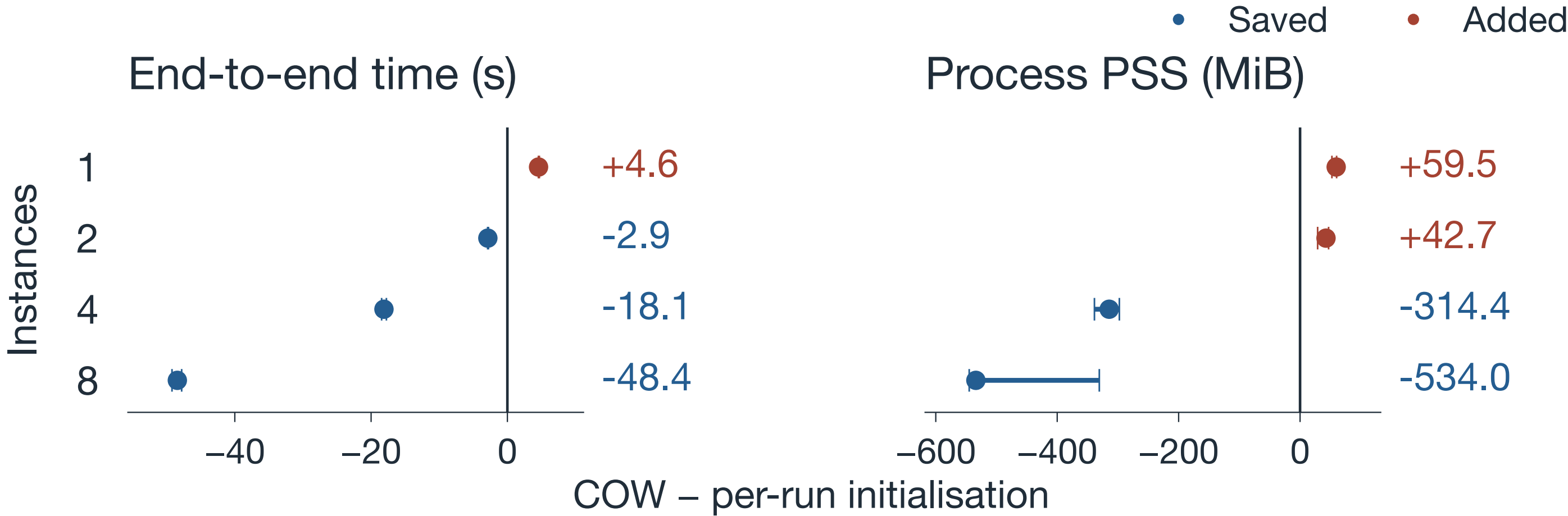


Discard private changes. Reuse the starting state.

Lower request latency with COW reset



More instances, greater savings

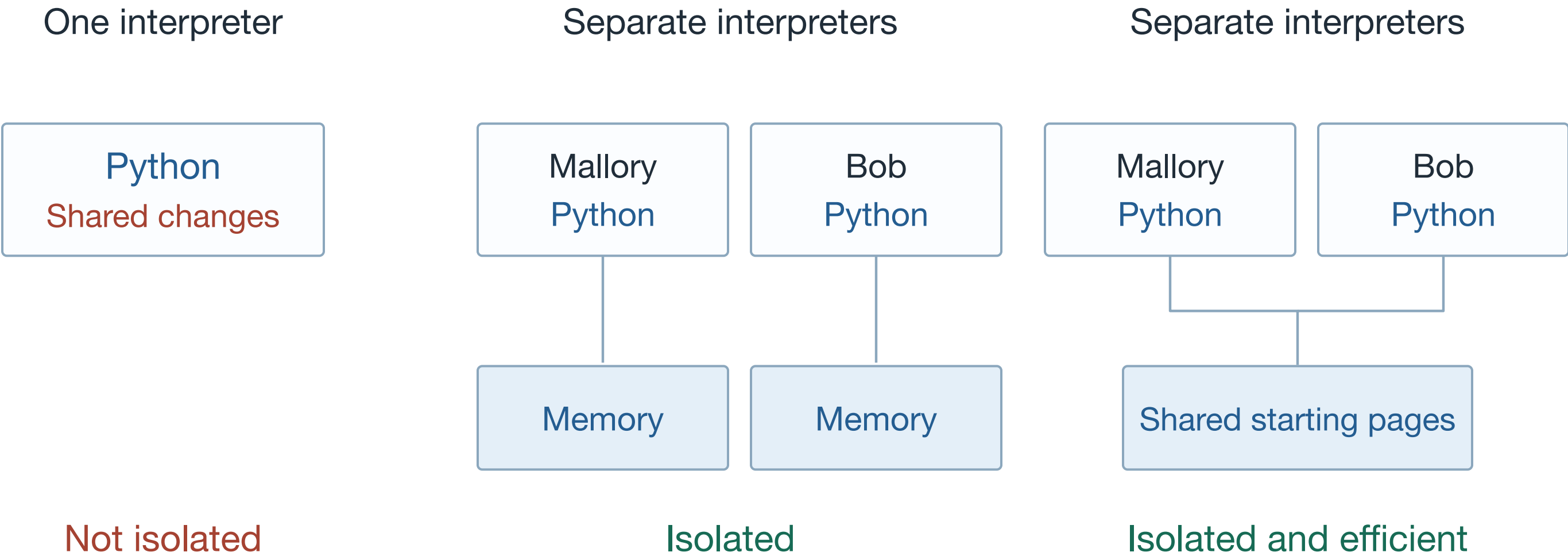


Dirty pages reduce how many instances fit

With 28 GiB memory available

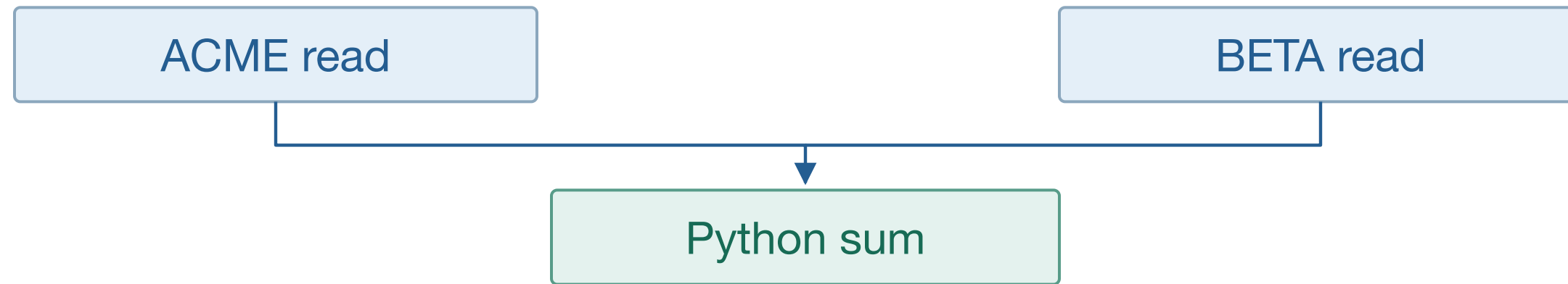


COW summary: isolated runs with shared memory



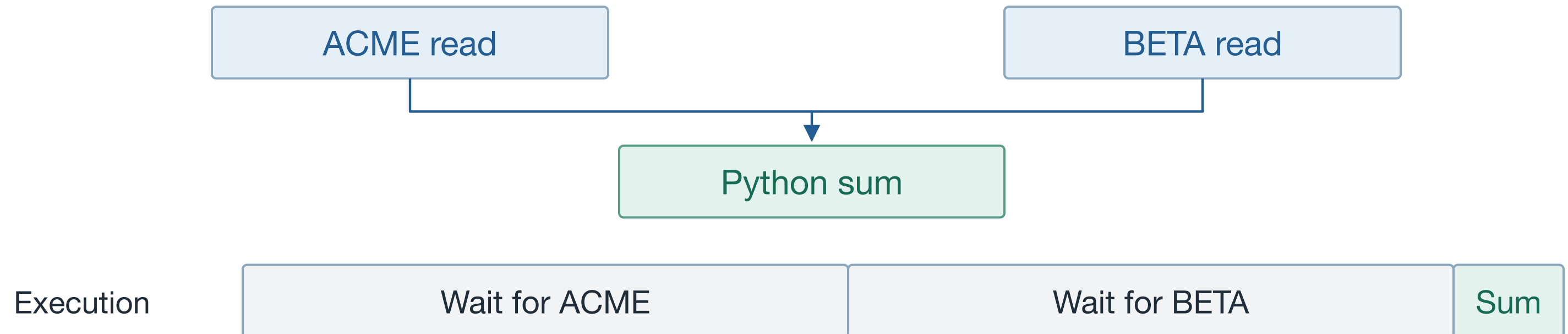
Now reduce waiting in the price program

```
acme = market.recent_price("ACME")  
beta = market.recent_price("BETA")  
portfolio_value = acme + beta
```



Ordinary execution waits at each call

```
acme = market.recent_price("ACME")  
beta = market.recent_price("BETA")  
portfolio_value = acme + beta
```



Ordinary execution waits at each call

```
acme = market.recent_price("ACME")  
beta = market.recent_price("BETA")  
portfolio_value = acme + beta
```



Both tickers are known. Why wait?

Start both price reads early

```
q_acme = prepare(market.recent_price, "ACME")  
q_beta = prepare(market.recent_price, "BETA")
```

These price reads leave market data unchanged. **Prepare** starts a request and returns a handle. It does not wait for the result.

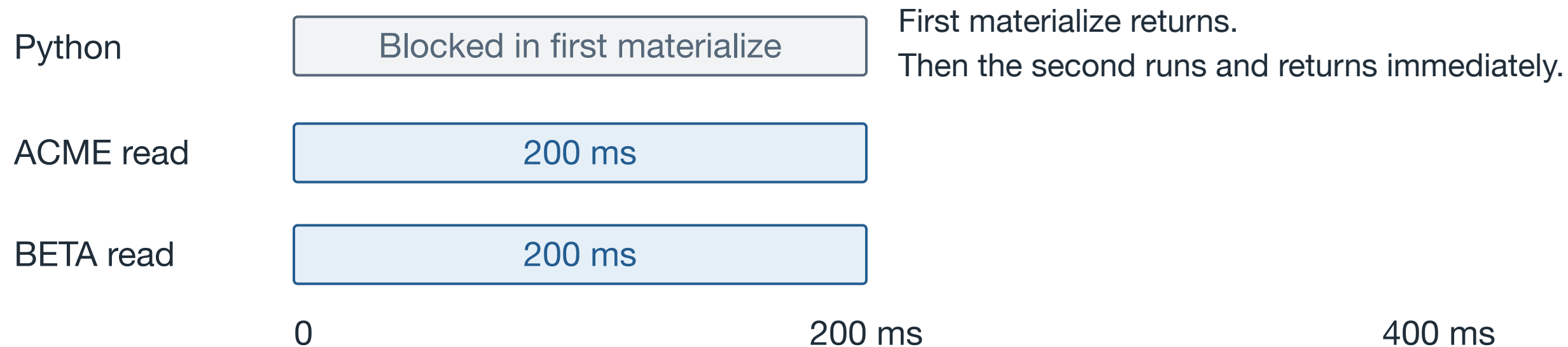


Materialize delivers the result at the original call

```
q_acme = prepare(market.recent_price, "ACME")  
q_beta = prepare(market.recent_price, "BETA")
```

```
acme = materialize(q_acme)  
beta = materialize(q_beta)  
portfolio_value = acme + beta
```

Materialize waits here if the result is not ready. Python then uses the result and continues.



Tool work can also overlap Python computation

Original order

```
[ independent computation ]  
price = market.recent_price("ACME")
```

Pysolate steps (pseudocode)

```
q = prepare(market.recent_price, "ACME")  
  
[ independent computation ]  
  
price = materialize(q)
```

The quote arrives while Python is still computing

```
q_acme = prepare(market.recent_price, "ACME")
q_beta = prepare(market.recent_price, "BETA")
# ... independent computation ...
acme = materialize(q_acme)
beta = materialize(q_beta)
portfolio_value = acme + beta
```

Suppose independent work runs before the price call. The quote arrives while Python is still busy.

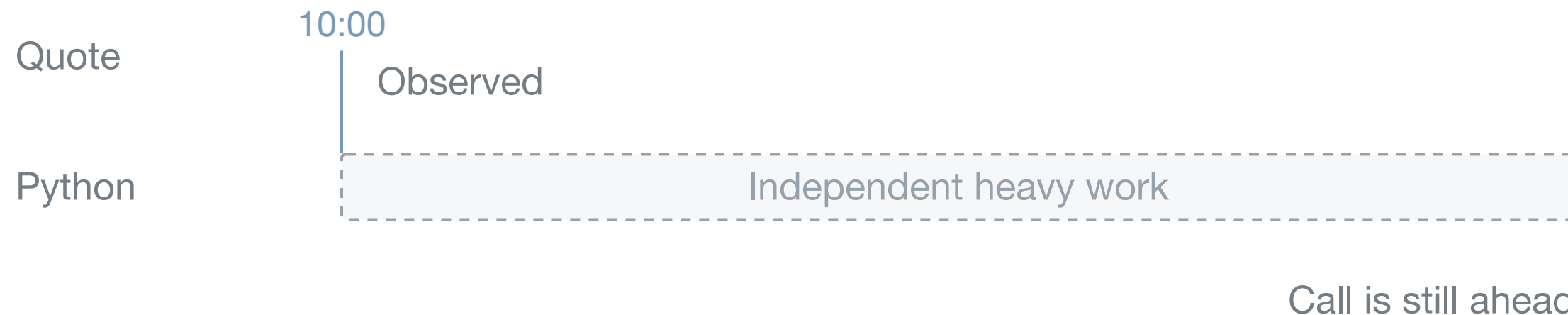


Call is still ahead

The quote arrives while Python is still computing

```
q_acme = prepare(market.recent_price, "ACME")
q_beta = prepare(market.recent_price, "BETA")
# ... independent computation ...
acme = materialize(q_acme)
beta = materialize(q_beta)
portfolio_value = acme + beta
```

Suppose independent work runs before the price call. The quote arrives while Python is still busy.

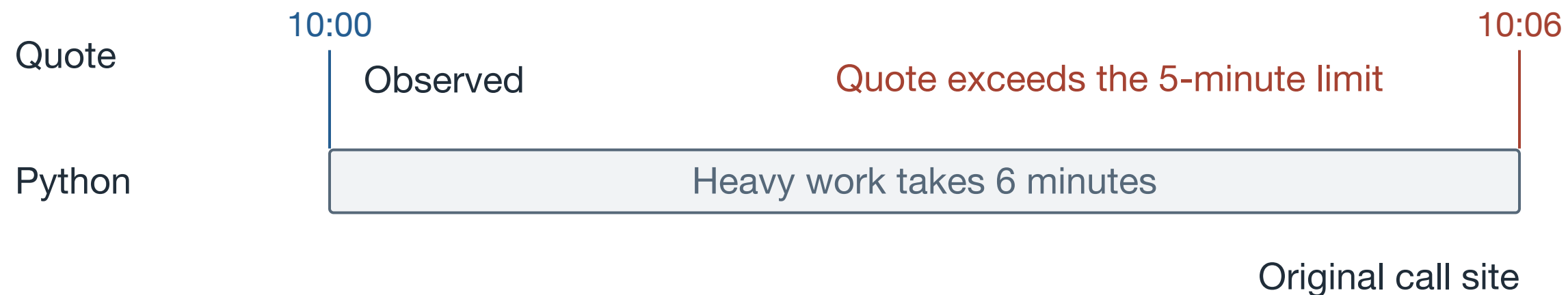


Can we still use the quote at the call?

Heavy work can make the prepared quote too old

```
q_acme = prepare(market.recent_price, "ACME")
q_beta = prepare(market.recent_price, "BETA")
# ... independent computation ...
acme = materialize(q_acme)
beta = materialize(q_beta)
portfolio_value = acme + beta
```

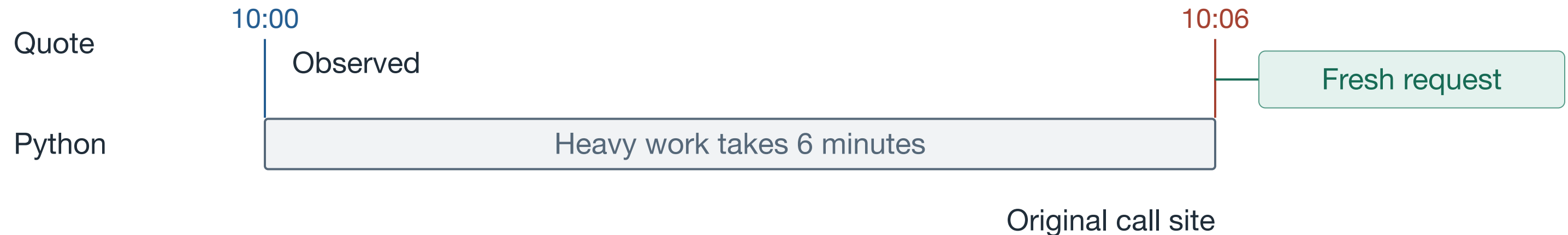
The tool allows quotes up to five minutes old. By the time this call runs, our saved quote is too old.



Check the quote when Python reaches the call

```
q_acme = prepare(market.recent_price, "ACME")
q_beta = prepare(market.recent_price, "BETA")
# ... independent computation ...
acme = materialize(linearize(q_acme))
beta = materialize(linearize(q_beta))
portfolio_value = acme + beta
```

Linearize checks the saved result at the **original call site**. The ticker must match and the quote must be at most five minutes old. Otherwise, request a new quote.

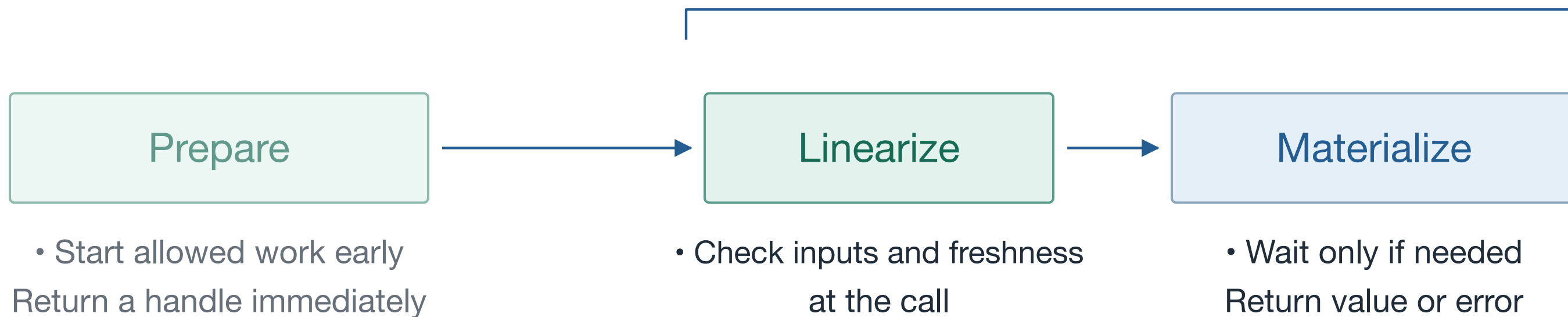


Prepare, Linearize, Materialize

```
price = market.recent_price("ACME")
```

Earlier

When Python reaches the call



Pysolate adds the steps automatically

Original Python

```
acme = market.recent_price("ACME")  
beta = market.recent_price("BETA")  
portfolio_value = acme + beta
```

Pysolate adds the steps

```
q_acme = prepare(  
    market.recent_price, "ACME")  
q_beta = prepare(  
    market.recent_price, "BETA")  
acme = materialize(linearize(q_acme))  
beta = materialize(linearize(q_beta))  
portfolio_value = acme + beta
```

The Host tracks these requests and checks them at the original calls.

Rewriting takes time too

The full program is ready. Each synthetic read takes 200 ms.

Ordinary execution

Run the program

With PLM

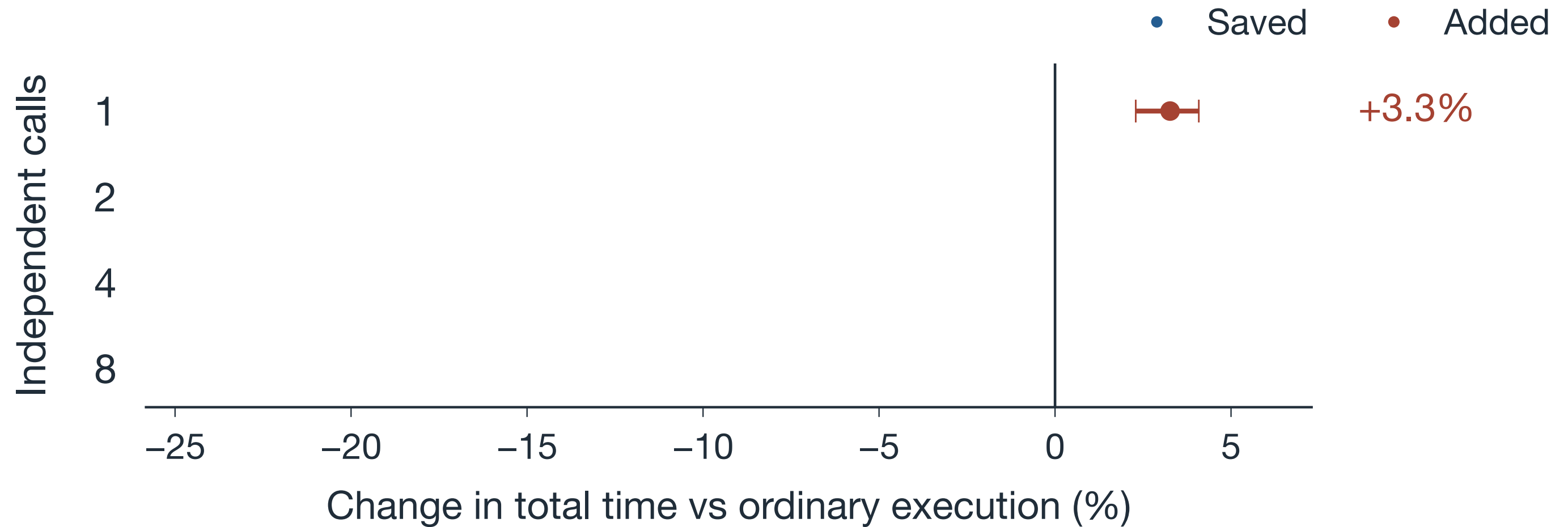
Rewrite the calls

Then run the program

Does the optimization save time overall?

One read leaves little work to overlap

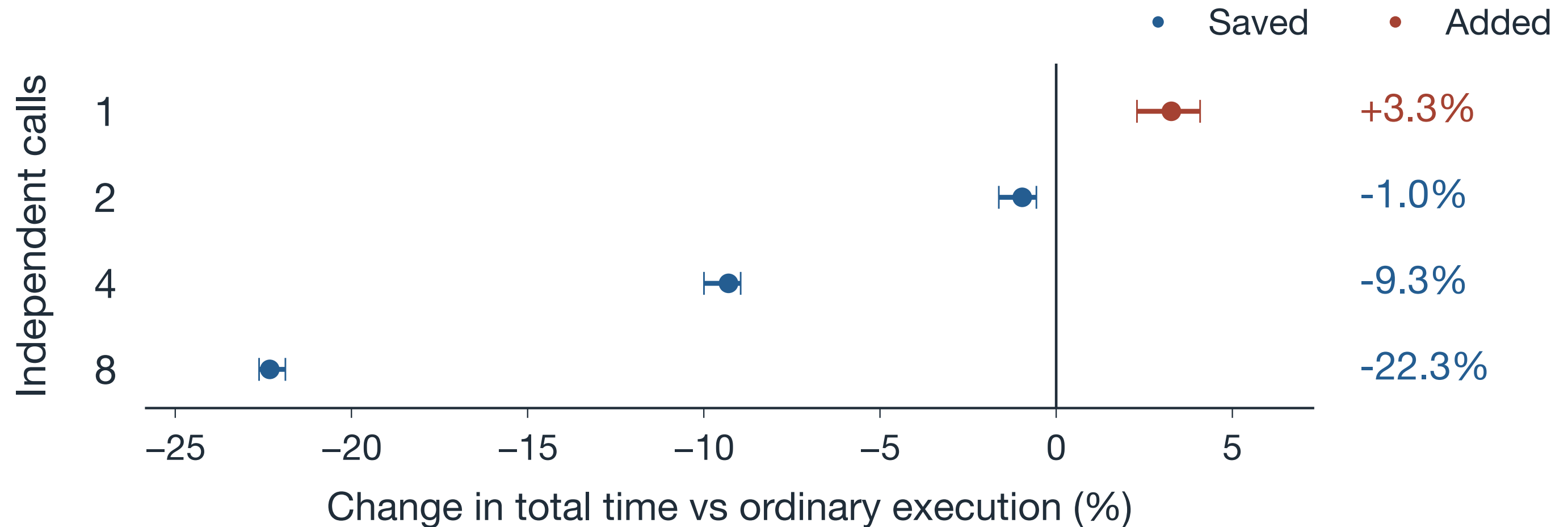
The full program is ready. Each synthetic read takes 200 ms.
End-to-end time



More independent reads save more time

The full program is ready. Each synthetic read takes 200 ms.

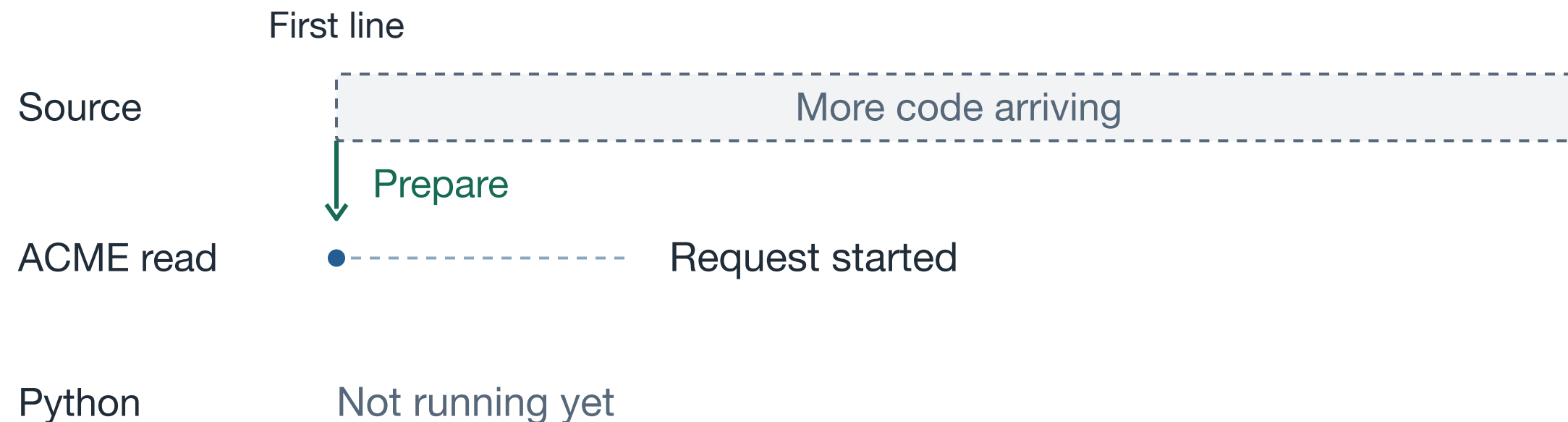
End-to-end time



Prepare while code is still arriving

```
acme = market.recent_price("ACME")  
...  
...
```

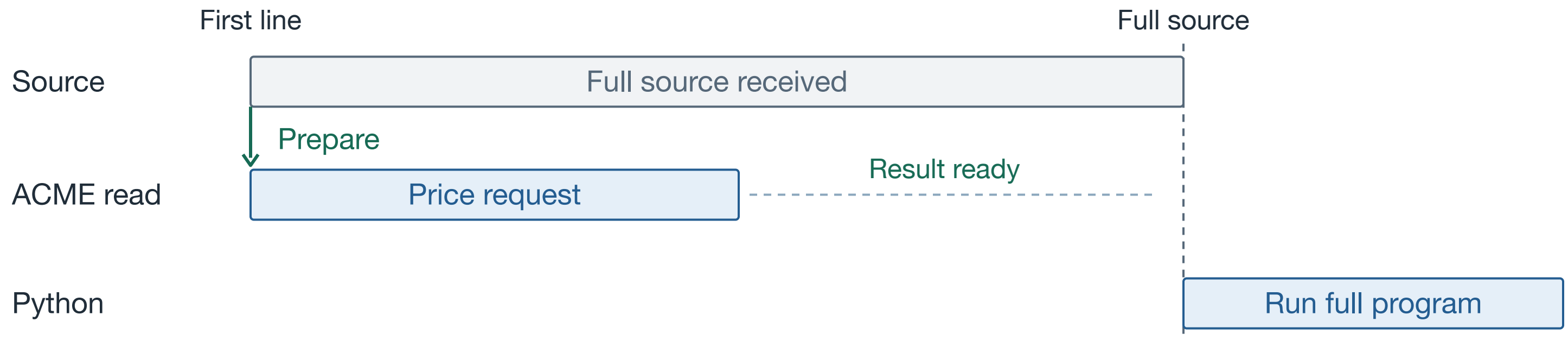
The first complete line gives us the call and its input. The Host can prepare the ACME read while more code arrives.



Prepare while code is still arriving

```
acme = market.recent_price("ACME")  
beta = market.recent_price("BETA")  
portfolio_value = acme + beta
```

With the full source, we validate and run the program. At the ACME call, we check and reuse the early request.



Requests can overlap code generation too

Preparing requests during code generation reduced end-to-end time by about **22%**, compared with waiting for the full source and then running serially.

Pysolate: a Wasm sandbox with COW and PLM

A Wasm sandbox

Experimental CPython + NumPy build for WASI Preview 1.

High-density execution with COW

Share starting memory and keep each run's changes private.

Tool semantics and PLM

The tool interface exposes operations and inputs to the Host.

PLM overlaps allowed calls towards the program's critical path.

Questions

What is shared, rebuilt and restored?

Saved linear memory

Python heap
Imported module objects
Prepared data

Built for each instance

An instance of the same module
Fresh globals and tables
Host resources and workspace

Map the sealed image

`memfd / MAP_PRIVATE`

Set up the module

`_initialize`

Restore the image bytes

`MADV_DONTNEED`

Guest pointers are offsets. The Host mapping can move.

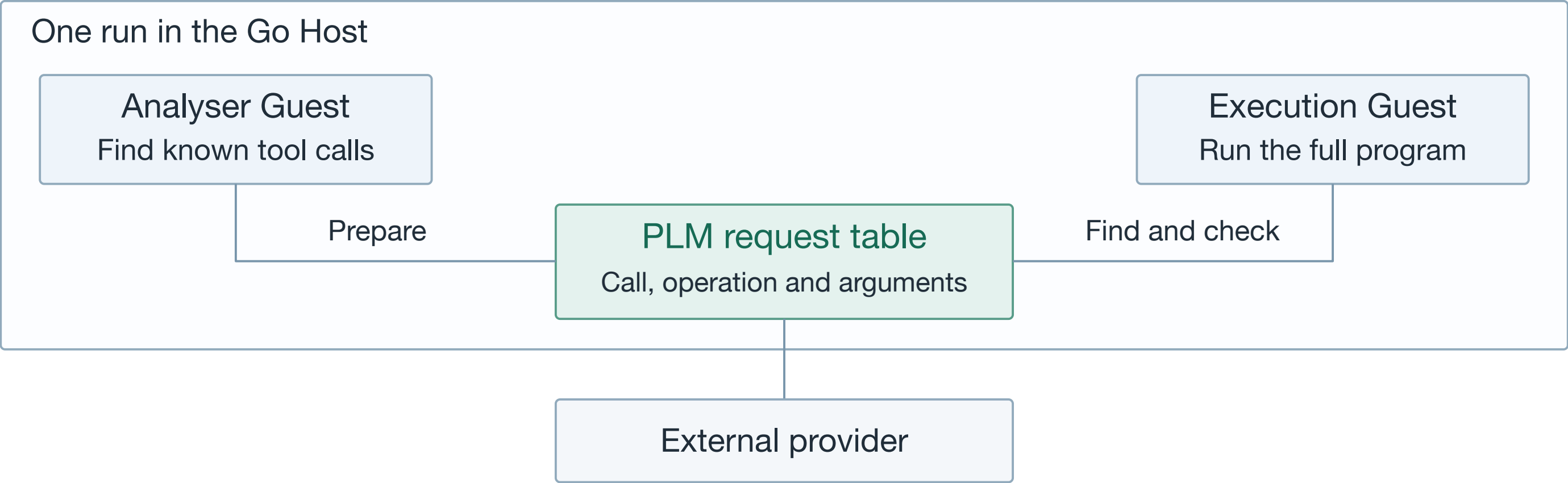
How do Host and Guest exchange data?

```
alloc(n) → request_offset  
write JSON bytes at request_offset  
execute(request_offset, n)  
response_offset → [u32 length][JSON]  
dealloc(request_offset)
```

Two call directions

- **Host → Guest:** source and inputs enter as JSON bytes.
- **Guest → Host:** tool requests use request offset/length and reply offset/capacity.
- The Host writes reply bytes and returns their length.

How does execution reuse an early request?



Pending

Wait at the call

Ready

Check and return

Cannot reuse

Call the tool now

Where can the source pass place Prepare?

User code

```
ticker = "ACME"  
unknown_call()  
label = "Latest quote"  
price = market.recent_price(ticker)  
result = [label, price]
```

Rewritten Python

```
ticker = "ACME"  
unknown_call()  
q = prepare(market.recent_price, ticker)  
label = "Latest quote"  
price = materialize(linearize(q))  
result = [label, price]
```

`unknown_call()` has unknown effects. Prepare stays after it.

Pysolate applies the optimization automatically

Original Python

```
listing = tools.listing("ACME")
if listing["market_open"]:
    ticker = listing["ticker"]
    label = "Current price"
    price = market.recent_price(ticker)
```

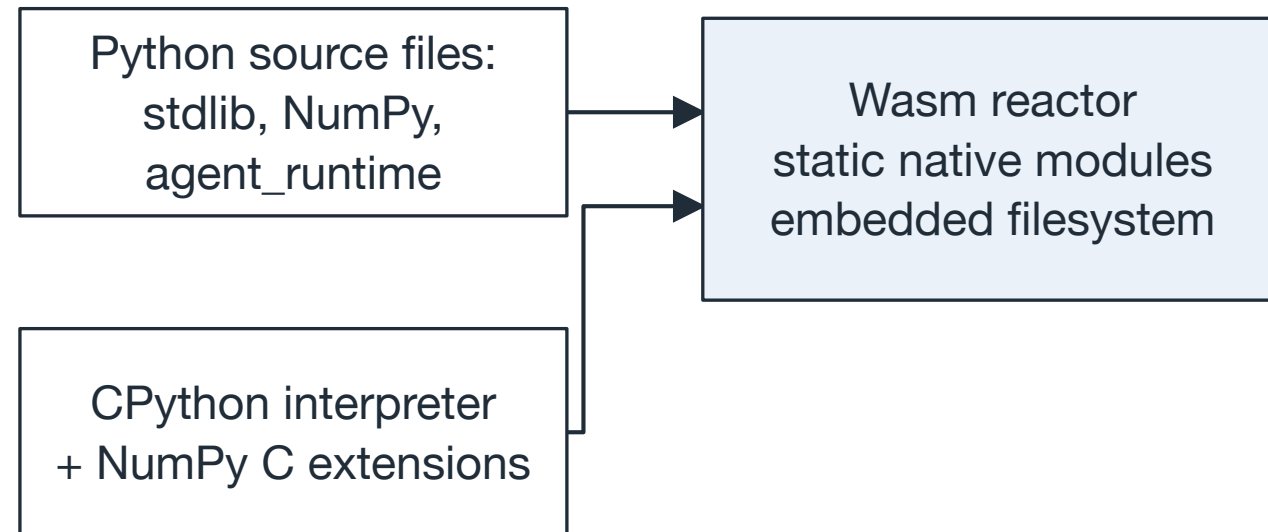
Rewritten Python

```
listing = tools.listing("ACME")
if listing["market_open"]:
    ticker = listing["ticker"]
     q = prepare(market.recent_price, ticker)
    label = "Current price"
    price = materialize(linearize(q))
```

What does each timer include?

COW	Build state	Create instances	Run	Close
Shimmy	Install evaluator	Handle request	Copy response	Reset
Source-stream PLM	Prepare workers	Source arrives	Final execution	Close
Whole-program PLM	Prepare engine	Guest setup	Rewrite and run	Decode and close

How does NumPy run inside the Guest?

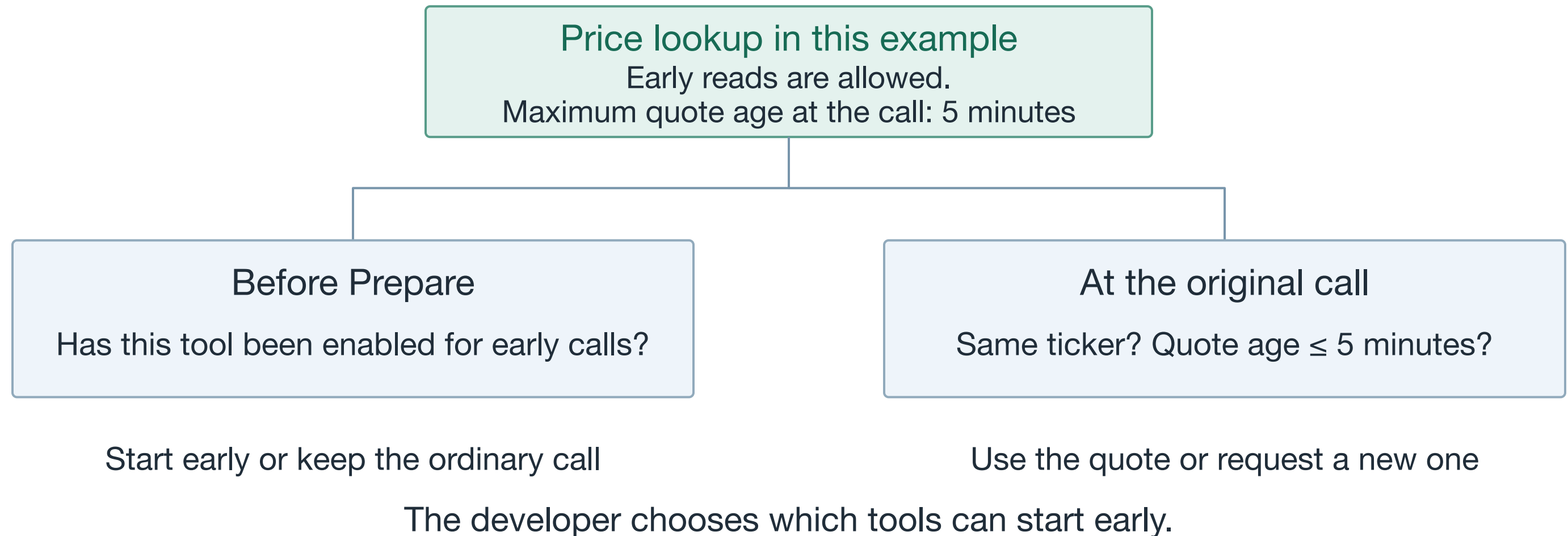


How import finds the native code

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

- Register native initialisers before starting CPython.
- Pack Python files with wasi-vfs.
- Import uses registered built-ins and packaged source.

Which tools can start early?



Why preserve syntax and exception order?

Check the full source

```
open("out.txt", "w").write("done")  
if (
```

A late syntax error stops Python before the file write.

Complete source **SyntaxError**

Deliver errors at the call

```
price = market.recent_price("ACME")  
write_receipt()
```

An error at the original call stops the following write, preserving the program's behaviour.

At the call **Error** Stop

Our function model

$$F(a, t, h)$$

a Arguments

t Call time

h Hidden state

Here, h includes the market's current data, which may change before the call.

Prepare

Start the request early.

Linearize

Check the inputs and the saved result at the call.

Materialize

Return the value or error at that call.

Python can wait for approval and continue

Python local state stays with this run.



What do the paging tests show?

Memory mapping test



RAM occupied by the mapped pages

Python execution test

Objects kept their values.

Python continued after the wait.

The next run started clean.

Moving pages while a run waits

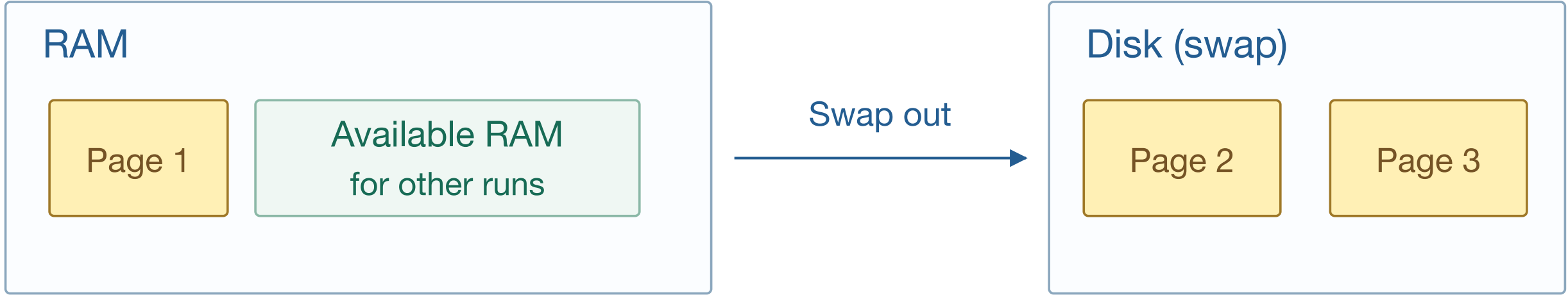
Run B

Waiting for a tool

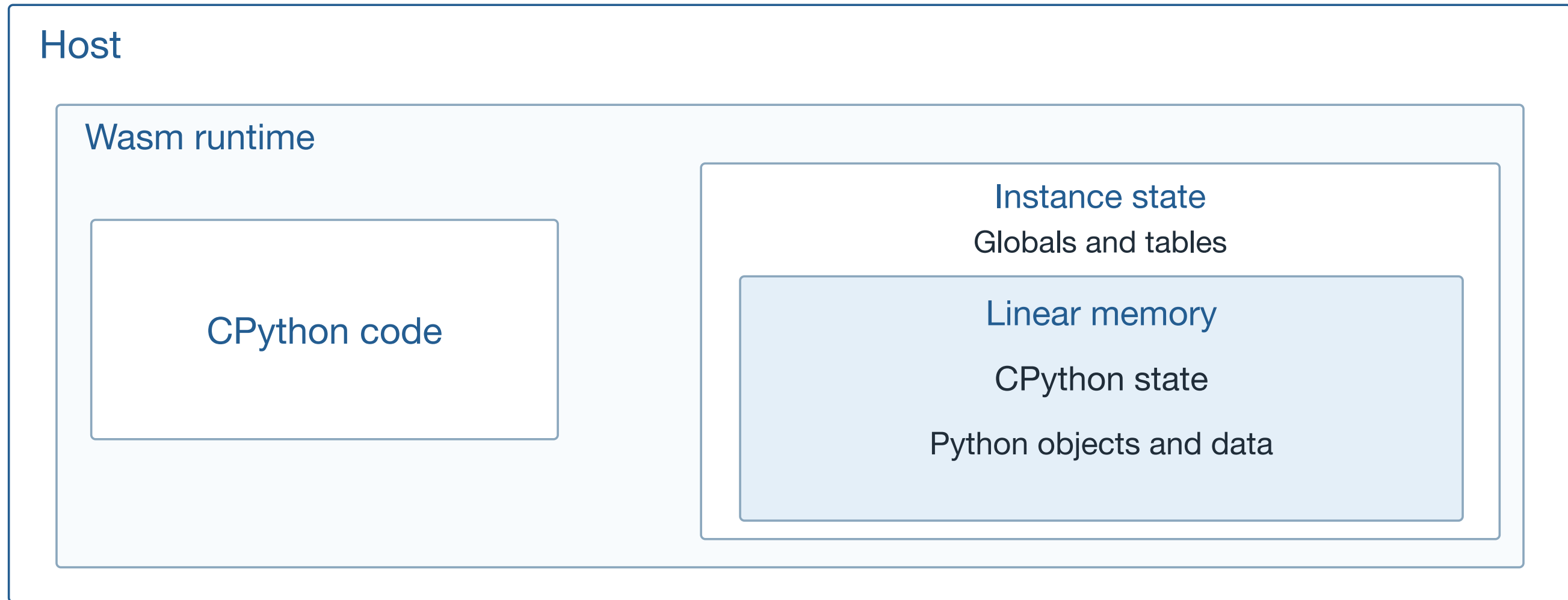
Its Python state is kept.

Host request pending

Run B's private pages



Code and state in a Wasm instance



What does Guest memory COW capture?

Inside linear memory

CPython data

Python objects, modules and bytecode

Guest request and reply buffers

Outside linear memory

Compiled CPython code

Wasm call stack, globals and tables

Host resources and tool requests

Unix fork()

Process address space is inherited.

File descriptors are inherited too.

Writes create private pages.

Pysolate COW

Linear-memory bytes form the image.

Globals and tables are built per instance.

The Host manages files and tools.

Capture after setup returns → store in a sealed memfd

Evaluation data and reproducibility

- Workloads and frozen data
- Experiment scripts and parameters
- Analysis, figures and plotting sources

[github.com/bkmashiro/pysolate-thesis
evaluation/README.md](https://github.com/bkmashiro/pysolate-thesis-evaluation/README.md)