

§ 14 PoC 4 — startup 償却の実測(クロスオーバー発見)+ cross-platform 実現性評価

項目	値
対象 spec	spec_section_14_draft.md v0.1, § 14.9 PoC 計画 PoC 4 系
実施日	2026-06-04
環境	WSL2 Ubuntu 24.04 / Rust 1.93.1 / CPython 3.12.3 / PyO3 0.22.6 / x86_64 / glibc 2.39
目標 (1)	PoC 3 の「~21% 速いのは起動コスト、常駐/バッチでは ~1.0× に漸近」という予測を、1 launch あたりの work 量(<code>AMORT_M</code>)を 1 → 10 ⁶ まで振って実測検証
目標 (2)	cross-platform(aarch64 / musl / WASI)のビルド実現性を評価
結果 (1)	△ 予測は半分外れ。短命実行では予測どおり Isolate が速い(~0.79×)が、work が増えると ~1.0× を通り越して Isolate が遅くなる(M=10 ⁶ で 1.375×)。原因は埋め込み CPython の定常 per-opcode ディスパッチが ~18-20% 重いという新発見(全 4 消去法で確定、bit-exact は終始維持)
結果 (2)	☒ ターゲットとも本環境ではビルド不可(クロス libpython / リンカ欠如)。本物のブロッカーエラーを採取し記録

0. 結論(正直版)

PoC 4 は「常駐/バッチでは速度差が ~1.0× に漸近する」を確認しに行き、それより重要な事実を見つけた:

- ・短命実行(per-call / per-request)では Isolate が ~21% 速い。これは PoC 3 の再現で、固定の launcher オーバーヘッド差(~4-6ms、1 回だけ)に由来。
- ・しかし sustained compute では Isolate が遅くなる。埋め込み CPython は `python3` CLI に対し定常 per-opcode 実行が ~18-20% 重い。work が起動コストを上回ると比率は 1.0 を越え、M=10⁶ で 1.375×(38% 遅い)。
- ・この per-opcode tax は (a) 計測アーティファクトではない(順序反転・交互でも persist)、(b) global/local スコープのせいではない(関数化しても persist)、(c) 動的機能固有ではない(純粋整数ループでも 1.18-1.20×)、(d) バイナリの置き場(drvfs/ext4)のせいでもない(差 2%)。 (e) `sys.flags` は完全一致。
- ・bit-exact はすべての M・全構成で維持(正しさは一切損なわれない)。

設計含意: Hybrid Isolate は「境界をまたいで短い動的コードを隔離実行する」 § 14 本来の用途(per-cell / per-call dispatch)に最適で、そこでは 速度ペナルティが無い(むしろ有利)。一方 hot な内側ループは Isolate

に丸投げせず、長命なホストインタプリタか **native tier** に置くべき、というティアリング指針が実測から導かれた。

1. startup 償却の実測(本命)

`bench/amort_workload.py` を **AMORT_M** (1 プロセスあたりの動的 work 反復数)を変えて **python3** と **Isolate** の両方で実行。warmup 2 + 計測 15 回の **中央値 wall time**。出力は 1 行(**AMORT_M** のみに依存)なので I/O ではなく **反復 dynamic-dispatch work** を測る。各 M で bit-exact も検証。

AMORT_M	python3(ms)	isolate(ms)	delta(ms)	iso/py	bitexact
-----	-----	-----	-----	-----	-----
1	27.58	21.69	5.89	0.787	PASS
10	27.83	22.10	5.73	0.794	PASS
100	27.66	21.97	5.69	0.794	PASS
1000	28.37	22.76	5.61	0.802	PASS
10000	34.58	31.39	3.19	0.908	PASS
100000	95.94	115.36	-19.42	1.202	PASS
1000000	695.44	956.40	-260.97	1.375	PASS

読み方:

- **delta(ms)**(= python3 - isolate)は $M \leq 1000$ で **+5.6~+5.9ms 一定** —— これが「launcher 経路差」の固定コスト。Isolate はこの分だけ常に得をする。
- M が 10^4 を越えると delta が **負に転じる**。work 時間(両者共通の CPython 実行)に Isolate 側だけ ~18-20% の上乗せが乗るため、work が増えるほど Isolate が損をする。
- **クロスオーバー**は $M \approx 10^4 \sim 10^5$ (iso/py が 1.0 を横切る)。この workload では「1 プロセスで ~30ms 以上の純 work」を越えると Isolate が不利。
- 全 M で **bitexact=PASS**。

二つの相反する定数的効果の重ね合わせ:

$\text{total_iso} \approx \text{total_py} - (\text{launch_saving} \sim 5\text{ms}) + 0.18 \times (\text{work_time})$ 。work_time が ~28ms($\equiv \text{launch_saving} / 0.18$)を越えた点でクロスオーバー。

2. per-opcode tax の切り分け(4 段階消去法)

「同一 CPython ・ 同一バイトコードなのに定常実行が遅い」という反直観的結果を、捏造せず順に潰した。

#	仮説	検証	結果	結論
(a)	直列実行の熱/順序 アーティファクト	$M=10^6$ で rep 毎交互 / isolate-first	交互 1.256 / iso-first 1.372(消えない)	アーティファクトではない
(b)	module(global)スコープの LOAD_GLOBAL 差	loop を def run(M) に入れて LOAD_FAST 化	関数版でも $M=10^6$ 1.346	スコープ起因ではない
(c)				

#	仮説	検証	結果	結論
	動的機能 (descriptor/ dispatch)固有	純粋整数ループ <code>acc+=i*i-(i&7)</code>	M=10 ⁶ 1.201 / M=5×10 ⁶ 1.175	全 opcode で ~18-20%、動的固有ではない
(d)	バイナリ置き場 (WSL2 drvfs / mnt/d)の I/O	binary を ext4 / tmp にコピーして 同測	drvfs 1.204 vs ext4 1.179(差 2.1%)	ファイルシステム起因ではない

加えて `sys.flags` を両者でダンプ —— `optimize=0 / isolated=0 / no_site=0 / hash_randomization=0 / recursionlimit=1000 / int_max_str=4300 / gc=(700,10,10)` が完全一致。Python から見える設定差は無い。

2.1 残る原因(仮説、over-claim 回避)

hot な eval ループ(`_PyEval_EvalFrameDefault`)は共有 `libpython3.12.so` 内にあり、`python3` と `Isolate` の両プロセスが同一 .so をロードする —— つまり `opcode` を回す機械語はバイト同一のはず。にもかかわらず ~18% 遅い。`sys.flags` も同一。したがって原因は バイトコード/インタプリタ設定ではなく、プロセス/ランタイム環境側にあると考えられる。候補:

1. `pymalloc` アリーナ配置 / メモリレイアウト差: PyO3 経由の埋め込みプロセスは Rust ランタイム + 自前 .text を抱えた address space を持ち、`pymalloc` アリーナの配置が `python3` のクリーンなマップと異なり、tight loop の D-cache/TLB 挙動が悪化する。
2. `Py_InitializeFromConfig` (PyO3 auto-initialize) vs `Py_RunMain` (`python3 CLI`)の初期化差: `sys.flags` に出ない内部 `PyConfig` /runtime 状態の違い。
3. CPU 周波数/スケジューリング(WSL2): 交互計測で persist したため主因ではないが、寄与の可能性は残る。

根本原因の確定には `perf stat` (instructions / IPC 比較)が要るが、本 WSL2 環境では `perf` が利用不可 —— PoC 5+ に正直に延期。なお「effect が ~18-20% 実在する」ことは §2 の 4 消去法で固く測定済みで、原因未確定でも結論(クロスオーバーの存在と設計含意)は変わらない。

3. cross-platform 実現性(目標 2)

現行 `Isolate runner` は `PyO3 auto-initialize` でホストの動的 `libpython` を埋め込む構造。host バイナリの実依存:

```
poc3_isolate: ELF 64-bit LSB pie, x86-64, dynamically linked,
interpreter /lib64/ld-linux-x86-64.so.2, stripped
=> libpython3.12.so.1.0 (/lib/x86_64-linux-gnu, glibc)
=> libc.so.6 / libm.so.6 / libz.so.1 / libexpat.so.1 / libgcc_s.so.1
```

このため別 ABI/arch にはターゲット用 `libpython` が必須。本環境での実測:

target	ブロッカー(実採取)	状態
aarch64-linux-gnu		☒不可

target	ブロッカー(実採取)	状態
	aarch64-linux-gnu-gcc リンカ無し / /usr/lib/aarch64-linux-gnu/libpython* 無し	
x86_64-unknown-linux-musl	musl-gcc 無し / libpython は glibc 専用(musl 静的リンク不可)	☒不可
wasm32-wasip1	cargo build --target wasm32-wasip1 実行 → pyo3-ffi build script が error: PYO3_CROSS_PYTHON_VERSION or an abi3-py3* feature must be specified when cross-compiling and PYO3_CROSS_LIB_DIR is not set. で停止(exit 101)	☒不可

wasm32-wasip1 ターゲットと wasmtime は導入済みだが、PyO3 はターゲット版 CPython の sysroot(PYO3_CROSS_LIB_DIR)を要求する。WASI 上で CPython を動かすには「WASI ビルドの CPython」を別途用意して PYO3_CROSS_* を指す必要があり、本環境には無い。

結論: cross-platform は「現行構成では不可」を捏造せず実エラーで確定。実現には各ターゲットの libpython 供給(クロス sysroot / WASI CPython ビルド)という environment provisioning が前提—— PoC 4.5 / PoC 5 の独立タスクとして切り出すのが妥当。

4. 重要な観察

- PoC 3 の楽観は短命実行限定だった: 「Isolate は ~21% 速い」は per-call の話。sustained compute では逆に ~18-38% 遅い。用途で符号が変わることを実測で確定——これが PoC 4 最大の収穫。
 - 正しさは速度と独立: per-opcode tax があっても bit-exact は全 M・全構成で PASS。「Isolate は同じ結果を出す」という § 14 の中核は速度特性と切り離して成立。
 - § 14 ティアリングへの実測根拠: 「短い動的コードは Isolate、hot loop は host/native」という分担が、感覚論ではなくクロスオーバー点(この workload で ~30ms work / launch)という数値で裏付けられた。
 - cross-platform は「コードの問題」ではなく「provisioning の問題」: runner ソースは arch 非依存。足りないのは各ターゲットの libpython。ここを分離して認識できたのは前進。
 - honest finding の型どおり: 予測 → 反証データ → 4 段階消去法 → 原因仮説 + 未確定部分の明示(perf 延期)。Phase B 系で確立した「正直な経験的 finding」の作法を § 14 PoC にも適用。
-

5. 配布物(PoC 3 クレートに追加)

```
engine/poc3_realworld/
├─ bench/amort_workload.py          AMORT_M で反復数可変の動的
workload(descriptor+dispatch+getattr)
├─ run_amortize.sh                  M=1..106 で python3 vs Isolate 中央値 + delta +
bitexact
└─ (既存) src/main.rs / samples/ / run_bit_exact.sh / run_speed.sh
```

再現コマンド:

```
cd engine/poc3_realworld
bash run_amortize.sh                # 償却テーブル(本書 §1)
# 切り分け($2)は /tmp に展開した pure.py / amort_fn.py で実施(本書に手順記載)
```

6. § 14.9 PoC ロードマップ 進捗

Phase	内容	状態
☒PoC 0	最小実証	完了
☒PoC 1	Any 5 sample	完了
☒PoC 2	*args/**kwargs + 動的 getattr/setattr + monkey 15 sample	完了
☒PoC 3	巷の実コード動的 20 sample + 速 度比較	完了
☒PoC 4	startup 償却実測(クロスオーバー 発見)+ cross-platform 実現性評価	完了(本書)
PoC 4.5	per-opcode tax 根本原因(perf stat の使える環境で IPC 比較 / Py_RunMain 相当に runner を寄 せて差分計測 / pymalloc アリーナ 検証)	次
PoC 5	cross sysroot / WASI CPython を provisioning して aarch64・ WASI で同一 sample 実行、+ RustPython(Light tier)差分計測	

PoC 4 完了。startup 償却は予測の「~1.0× 漸近」では終わらず、埋め込み CPython の定常 per-opcode が~18-20% 重いため、work が起動コストを越えると Isolate が遅くなる(M=10⁶ で 1.375×)というクロスオーバーを実測発見。4 段階消去法でアーティファクト/スコープ/動的固有/FS を排除、sys.flags 一致を確認、bit-exact は全構成維持。根本原因(共有 libpython なのに遅い)は perf 不可のため PoC 4.5 に延期。cross-platform は aarch64/musl/WASI と本環境ではビルド不可を実エラーで確定(libpython

provisioning 前提)。設計含意: Isolate は per-call 用途に最適、hot loop は host/native tier —— という § 14 ティアリングが数値で裏付いた。