WebAssembly, wasmCloud, and Containers: A Reproducible Benchmark Study

Davi Khvedelidze

January 9, 2026

Abstract

This report presents a reproducible benchmark study of lightweight runtime technologies for edge and serverless workloads. The focus is on cold-start latency, warm request latency, throughput, and resource footprint under realistic deployment constraints on a personal macOS laptop. The study compares native Rust binaries, Docker containers, wasmCloud (component + host model), Wasmtime (component server and direct WASI execution), and WasmEdge. The benchmarks include a minimal HTTP service (stateless and stateful endpoints), a CPU-bound hashing workload, and a tiny WASI module. We describe the complete experimental pipeline, the code written to automate and measure it, and the resulting performance trade-offs.

1 Motivation

Edge and serverless systems live or die by startup latency and resource footprint. Traditional containers are flexible but can be heavy for short-lived services. WebAssembly components promise fast instantiation and smaller memory use, while microservice frameworks such as wasmCloud provide capability-based composition. This project evaluates those claims on a realistic developer-class machine and produces a testbed that others can reproduce or extend.

2 Goals and Expectations

2.1 Goals

- Measure cold-start latency (process launch to first HTTP 200).
- Measure warm request latency and throughput for a minimal HTTP service.
- Compare CPU-bound execution time for a fixed SHA-256 workload.
- Compare minimal WASI execution overhead across runtimes.
- Produce a reproducible pipeline with clear logs and plots.

2.2 Expectations and Hypotheses

- Wasmtime cold-start should be fastest because it directly serves a component without a separate host+provider model.
- wasmCloud cold-start should be slower due to host and capability provider initialization.

- Warm HTTP latencies should be similar across runtimes for a trivial handler, with throughput bounded by the networking stack.
- Native binaries should be most memory efficient; wasmCloud should be heaviest because it includes a host, NATS, wadm, and the HTTP provider.
- WasmEdge may lag on this macOS setup for WASI workloads due to runtime maturity and platform-specific optimizations.

3 Experimental Design

3.1 Base Specifications (Project 9)

The project specification requires a reproducible testbed comparing lightweight runtimes for edge and serverless scenarios. The selected baselines are native execution, Docker containers, wasmCloud, and standalone WebAssembly runtimes (Wasmtime and WasmEdge). The benchmark workloads must include a minimal "hello world" invocation, a CPU-bound function, a networked microservice, and a simple stateful service. The key metrics are cold-start latency, warm-start latency, memory and CPU overhead per instance, throughput (requests/sec), scalability or density, and resource efficiency. In this implementation, the hello-wasm and CPU-hash workloads cover minimal invocation and CPU-bound behavior, while the HTTP service provides both stateless and stateful endpoints and exposes cold/warm latency and throughput under optional firewalling.

3.2 Scope and Platform Constraints

The benchmark runs on a MacBook Pro 13̈(M3 Pro, ARM64) with macOS (Darwin 25.2.0). Firecracker and unikernels were not included due to macOS constraints and the goal of using a single host. The chosen scope therefore includes:

- Native Rust binary
- Docker container
- wasmCloud (component + host model)
- Wasmtime (component server + WASI execution)
- WasmEdge (WASI execution)

3.3 Software Stack

Component	Version
Rust	1.91.1
Docker Desktop	25.0.3
wash	0.42.0 (wasmCloud 1.8.0, wadm 0.21.0)
Wasmtime	40.0.0
WasmEdge	0.15.0

Table 1: Runtime and toolchain versions from run logs.

3.4 Warm vs Cold Definition

Cold start is defined as the time from launching an instance to the first successful HTTP 200 response. **Warm start** means the runtime is already running and the component or module is already loaded and cached. For Wasmtime, the cache mode is controlled with `WASMTIME_CACHE_MODE`. For Docker, a cold run can remove images or disable cache to emulate a fresh start. wasmCloud cold runs fully stop hosts and purge JetStream state.

3.5 Firewall

macOS `pf` is used in some runs to reflect a common host firewall setup. The ruleset blocks inbound traffic, allows outbound traffic, and skips loopback. This provides a realistic security baseline while keeping loopback request paths functional. Firewall state is recorded in run logs.

4 Benchmarks and Workloads

4.1 HTTP Service

A minimal HTTP service is implemented in Rust and compiled for both native and WebAssembly runtimes. Two endpoints are exposed:

- `/`: returns `hello`
- `/state`: increments and returns a counter

The same logical behavior is used across native, Docker, wasmCloud, and Wasmtime component serving.

4.2 CPU-Bound Workload

A fixed SHA-256 loop (2,000,000 iterations) measures CPU-bound execution. This workload is run as a native binary, a Docker container, a Wasmtime WASI module, and a WasmEdge WASI module.

4.3 Minimal WASI Module

A tiny `hello-wasm` module measures runtime overhead for small WASI programs using Wasmtime and WasmEdge.

5 Measurement Methodology

5.1 Cold Start

Cold start is captured using nanosecond-resolution timestamps from `gdate` (`coreutils` on macOS). Each HTTP runtime script starts the server, polls for HTTP 200, and records the time delta.

5.2 Warm Latency and Throughput

Each HTTP run performs:

- 5 warm-up requests (not measured)
- 50 sequential requests for latency distribution
- 200 requests with concurrency 10 for throughput

These values are configurable with environment variables in the scripts.

5.3 Resource Footprint

RSS and CPU snapshots are recorded after warm-up. For wasmCloud, the snapshot is the sum of wasmCloud host + NATS + wadm + HTTP provider/listener processes.

6 Implementation and Automation (Code Written)

The benchmark suite is implemented as a set of Rust workloads and shell/Python scripts. The design goal is deterministic measurement and full reproducibility.

6.1 Workloads

- `workloads/http-hello`: native Rust HTTP service using `tiny_http`. The handler is intentionally small to make runtime overhead visible. The `/state` endpoint uses an atomic counter to represent a stateful microservice.
- `workloads/wasmcloud-http-hello`: wasmCloud component that implements `wasi:http/incoming-handle` It mirrors the same `/` and `/state` behavior.
- `workloads/cpu-hash`: fixed SHA-256 loop; used for native, Docker, Wasmtime, and WasmEdge.
- `workloads/hello-wasm`: minimal WASI module to expose runtime overhead on small programs.

6.2 Measurement Scripts

- `scripts/measure_native_http_hello.sh`
- `scripts/measure_docker_http_hello.sh`
- `scripts/measure_wasmcloud_http_hello.sh`
- `scripts/measure_wasmtime_http_hello.sh`
- `scripts/measure_native_cpu_hash.sh`
- `scripts/measure_docker_cpu_hash.sh`
- `scripts/measure_wasm_cpu_hash.sh`
- `scripts/measure_wasmedge_cpu_hash.sh`
- `scripts/measure_wasm_hello.sh`

- scripts/measure_wasmedge_hello.sh
- scripts/measure_cold_start_comparison.sh
- scripts/firewall/pf_enable.sh, pf_disable.sh, pf_status.sh

Each script writes a detailed run log to results/raw/<runtime>/<workload>/ containing timestamps, per-request latency lines, throughput summaries, and resource snapshots.

6.3 Representative Code Snippets

The HTTP workload is intentionally small to make runtime overhead visible. The native handler mirrors the wasmCloud component behavior:

```
static COUNTER: AtomicU64 = AtomicU64::new(0);
let path = request.url();
let body = if path.starts_with("/state") {
    let next = COUNTER.fetch_add(1, Ordering::Relaxed) + 1;
    next.to_string()
} else {
    "hello".to_string()
};
```

Cold-start measurement is implemented in the shell scripts by measuring the time from process launch to the first HTTP 200 response:

```
t0_ns=$(gdate +%s%N)
... start server ...
while (( retries < max_retries )); do
  http_code=$(curl -sS -o /dev/null -w "%{http_code}" "$URL" || echo "000")
  if [[ "$http_code" == "200" ]]; then
    t_ready_ns=$(gdate +%s%N)
    break
  fi
  sleep 0.01
done
cold_start_ms=$(awk "BEGIN { printf \"%.3f\", (t_ready_ns - t0_ns)/1000000 }")
```

6.4 Orchestration and Logging

scripts/run_all_benchmarks.sh is the central orchestrator. It:

1. Builds all workloads (native and WASI)
2. Runs the HTTP suite across cold and warm scenarios, firewall on/off, and both endpoints
3. Executes CPU-hash and hello-wasm suites
4. Runs the dedicated cold-start comparison
5. Generates plots with analysis scripts

The script writes a full raw log to results/processed/<timestamp>_run_all.log. For usability, a live ANSI dashboard summarizes progress, while full command output remains in the log for debugging and reproducibility.

6.5 Pipeline Walk-through

The full run follows a deterministic sequence designed to separate build costs, runtime startup costs, and steady-state performance:

1. Clean build artifacts (optional) to make cold builds explicit.
2. Compile native and WASI workloads to eliminate compilation variability.
3. Execute HTTP benchmarks across cold and warm scenarios, for both `/` and `/state`, and optionally with the firewall on/off.
4. Reset runtime state between runs (e.g., stop wasmCloud hosts and purge JetStream for cold starts; optionally prune Docker images).
5. Execute CPU-hash runs to measure sustained CPU execution.
6. Execute hello-wasm runs to isolate runtime overhead on tiny programs.
7. Run the cold-start comparison that splits build time from runtime start.
8. Generate plots and copy them to a timestamped folder for this run.

6.6 Plot Generation

The analysis scripts parse raw logs, compute summary statistics, and generate plots in `results/processed/`:

- `scripts/analyze_http_hello_all.py`
- `scripts/analyze_cpu_hash_comparison.py`
- `scripts/analyze_wasm_hello_comparison.py`
- `scripts/analyze_cold_start_comparison.py`

6.7 Developments Achieved and Challenges

The main development outcome is a fully automated benchmarking pipeline that builds workloads, runs multi-scenario experiments, and generates plots with timestamped outputs. The biggest technical challenge was achieving reliable wasmCloud HTTP readiness on macOS, especially with the firewall enabled. This was resolved by switching to a `wash up` deployment flow, binding the HTTP provider to loopback, and improving reset logic to avoid stale host state. Another challenge was ensuring consistent cold versus warm semantics across runtimes; this required explicit cache controls for Wasmtime, clean teardown for wasmCloud, and optional Docker image pruning. The output logging and plot pipeline were then refined to preserve raw data for debugging while presenting clean progress feedback during long runs.

7 Results

All results below aggregate all available run logs in `results/raw`. Counts are the number of valid run logs (HTTP) or the number of recorded samples (CPU-hash and hello-wasm). The graphs referenced are generated automatically by the analysis scripts.

Runtime	Runs	Cold mean (ms)	p50 Lat (ms)	Throughput (rps)	RSS (MB)	rps/MB
Native	55	177.74	11.27	684.1	1.82	375.31
Docker	49	183.89	12.98	576.2	N/A	N/A
wasmCloud	45	5103.16	12.45	612.3	219.33	2.79
Wasmtime	38	54.81	11.27	693.3	24.20	28.65

Table 2: HTTP summary across all available runs. RSS is mean resident set size. Docker RSS is not captured in logs.

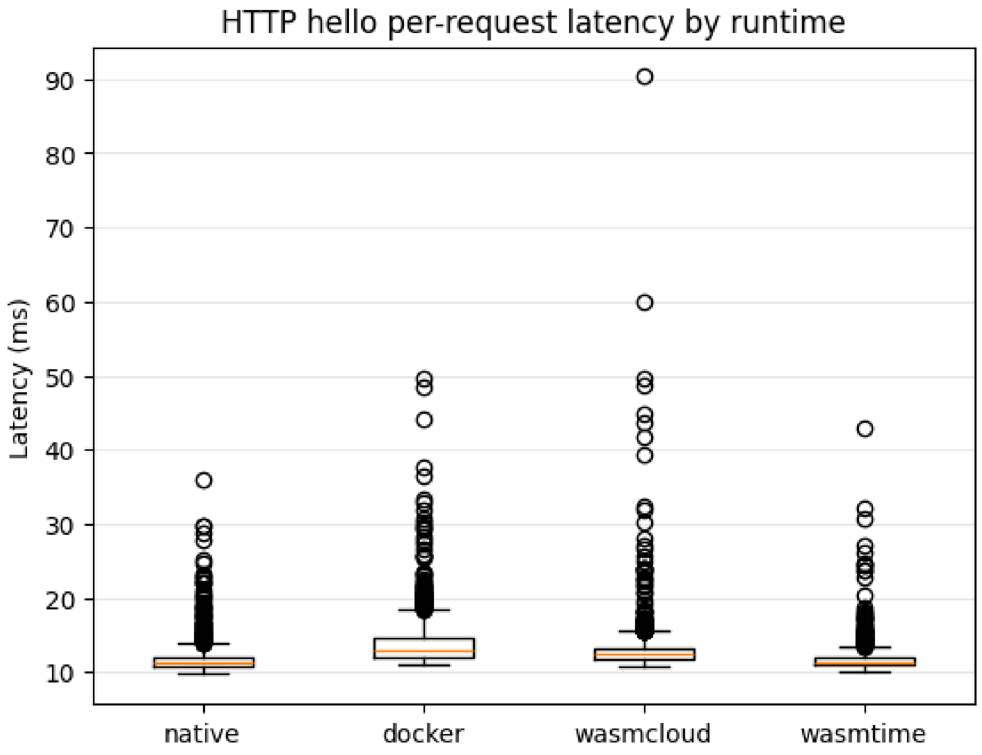

Figure 1: HTTP latency distribution by runtime.

7.1 HTTP Service Summary

Interpretation and trade-offs. The boxplots show that all runtimes cluster around similar median latency for this trivial handler, which indicates that once warm, network stack and HTTP server overhead dominate more than runtime choice. wasmCloud exhibits a wider tail, consistent with additional scheduling and provider hops in its host model. Wasmtime tracks native closely in the median, suggesting its component server adds little steady-state overhead for this workload. The trade-off is clear: wasmCloud buys modularity and capability isolation at the cost of more variance, while native and Wasmtime deliver tighter latency distributions but offer fewer built-in deployment features.

Interpretation and trade-offs. Throughput differences are modest, which is expected for a tiny handler where the bottleneck is often the HTTP stack and kernel scheduling. Wasmtime and native are slightly higher, reflecting lower per-request overhead. Docker shows some penalty, likely

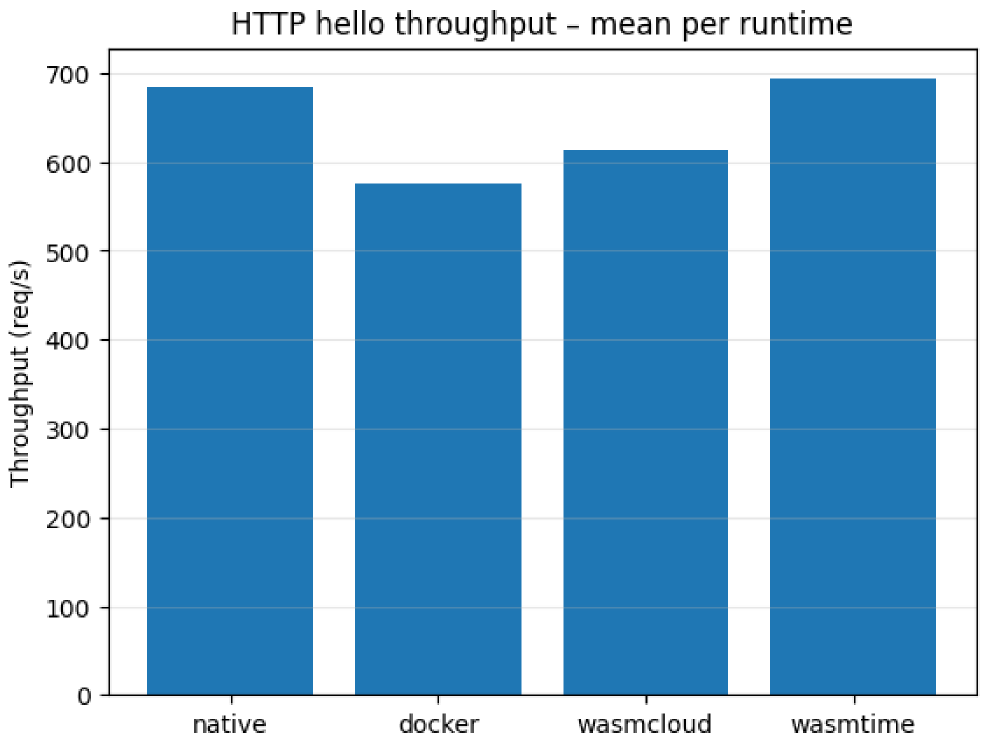

Figure 2: HTTP throughput (mean requests/sec).

from container runtime indirection and networking layers. wasmCloud remains competitive but slightly lower because requests pass through the provider and host layers. The throughput results reinforce the idea that startup and memory footprint matter more than steady-state throughput for very small services.

Interpretation and trade-offs. Native has the smallest RSS because it is a single binary with no external control plane. Wasmtime is larger because it keeps runtime state, compiled code cache, and the component server alive; this is the price of fast instantiation and isolation without a full container. wasmCloud is substantially higher because the measurement sums multiple processes: the host, NATS broker, wadm, and the HTTP capability provider. In other words, a wasmCloud "instance" includes a control plane and messaging layer, which is useful for modular services and dynamic linking of capabilities, but it reduces density on a memory-constrained node. The key trade-off is operational flexibility and capability isolation versus per-instance memory overhead. Docker RSS is not shown here because it was not captured in the logs; including it would likely place Docker between Wasmtime and wasmCloud on this machine.

7.2 CPU-Bound Workload

Interpretation and trade-offs. This figure isolates raw compute performance for a fixed hashing loop. Native is fastest because it runs directly on the host with no runtime indirection. Wasmtime is close to native, which indicates its JIT and WASI layer add only a small constant overhead relative to the work per iteration. Docker is slower on this macOS host, consistent with the extra layers of container startup, virtualized networking, and CPU scheduling under Docker Desktop.

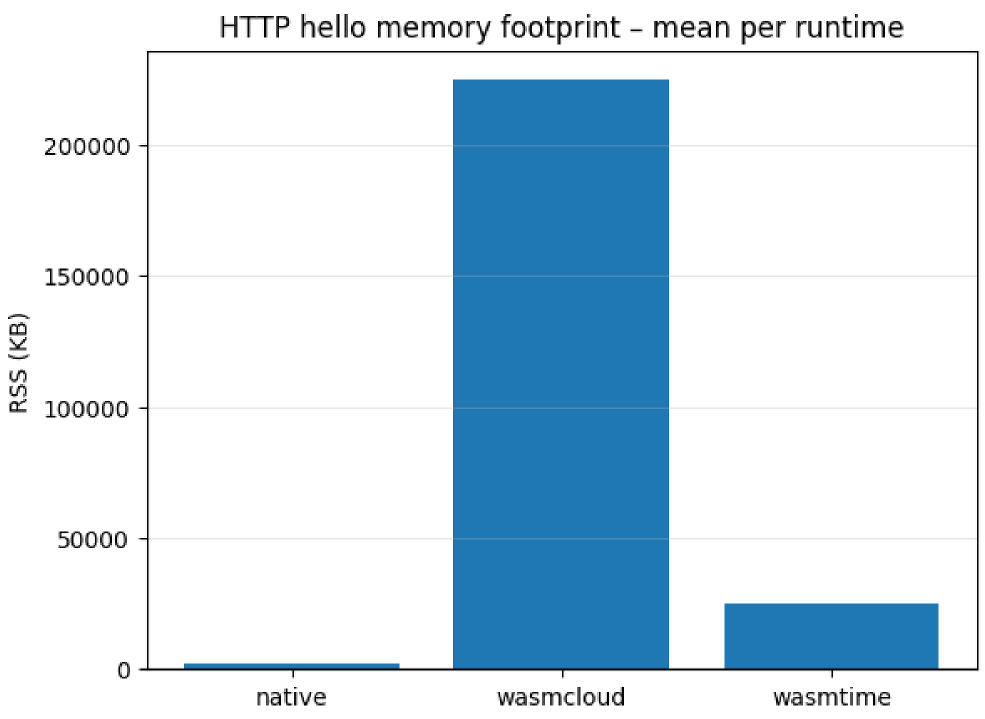

Figure 3: HTTP memory footprint (mean RSS).

Runtime	Samples	Mean (ms)	p50 (ms)	Min–Max (ms)
Native	60	160.67	142.47	140.18–548.99
Docker	60	386.87	371.64	337.65–533.16
Wasmtime	80	190.85	161.15	149.40–362.34
WasmEdge	60	19288.11	18548.95	18158.54–31221.51

Table 3: CPU-hash execution time (2,000,000 iterations).

WasmEdge is significantly slower here, which suggests that its macOS execution path is not as optimized for this workload or that its JIT/AOT settings are less tuned for this CPU. The practical trade-off is that Wasmtime provides near-native CPU performance with sandboxing and portability, Docker offers mature packaging and deployment at a CPU cost, and WasmEdge may be better suited for Linux edge deployments where it is more aggressively optimized.

7.3 Minimal WASI Module (hello-wasm)

This subsection isolates runtime overhead by executing a tiny WASI program. The goal is to compare baseline invocation cost without network or provider infrastructure in the loop.

Interpretation and trade-offs. The hello-wasm benchmark is dominated by runtime initialization and dispatch, not by application logic. Wasmtime is faster, which is consistent with a highly optimized startup path and efficient WASI integration. WasmEdge is slower on this machine, implying higher fixed overhead per invocation. This distinction matters for serverless "microfunctions" that execute for only a few milliseconds: a small fixed cost can dominate total latency. In

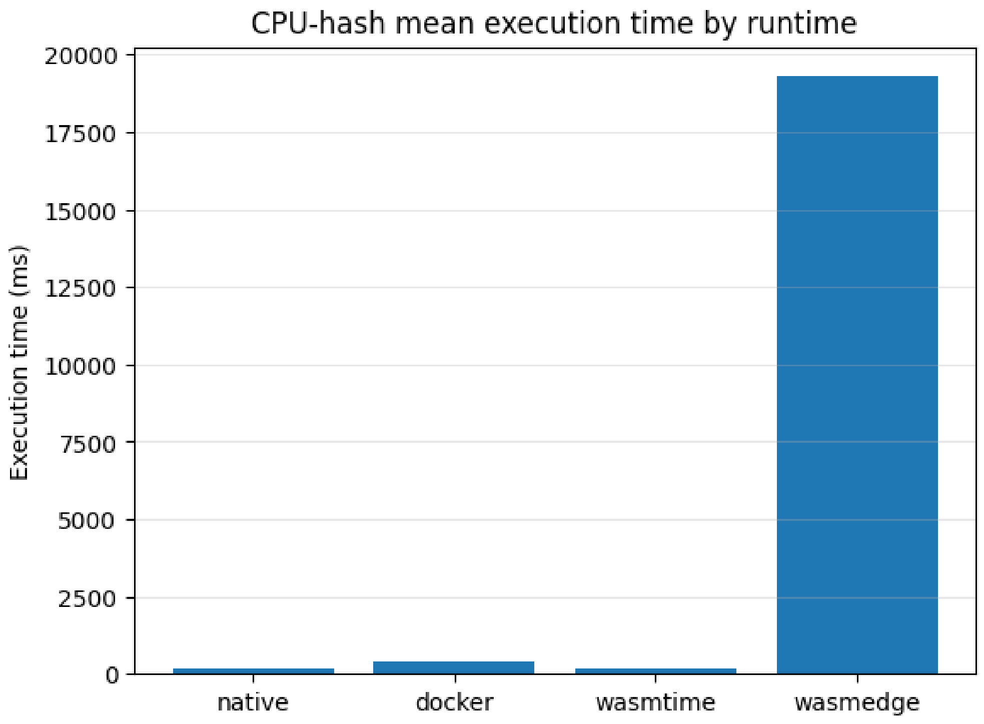

Figure 4: CPU-hash mean execution time by runtime.

Runtime	Samples	Mean (ms)	p50 (ms)	Min–Max (ms)
Wasmtime	200	13.03	12.55	10.53–42.46
WasmEdge	150	21.93	21.16	18.70–32.23

Table 4: hello-wasm execution time (standalone WASI).

longer-running tasks, the gap should shrink because startup cost amortizes. The trade-off is therefore between Wasmtime's low-latency startup for tiny jobs and WasmEdge's potential advantages in other ecosystems or deployment footprints.

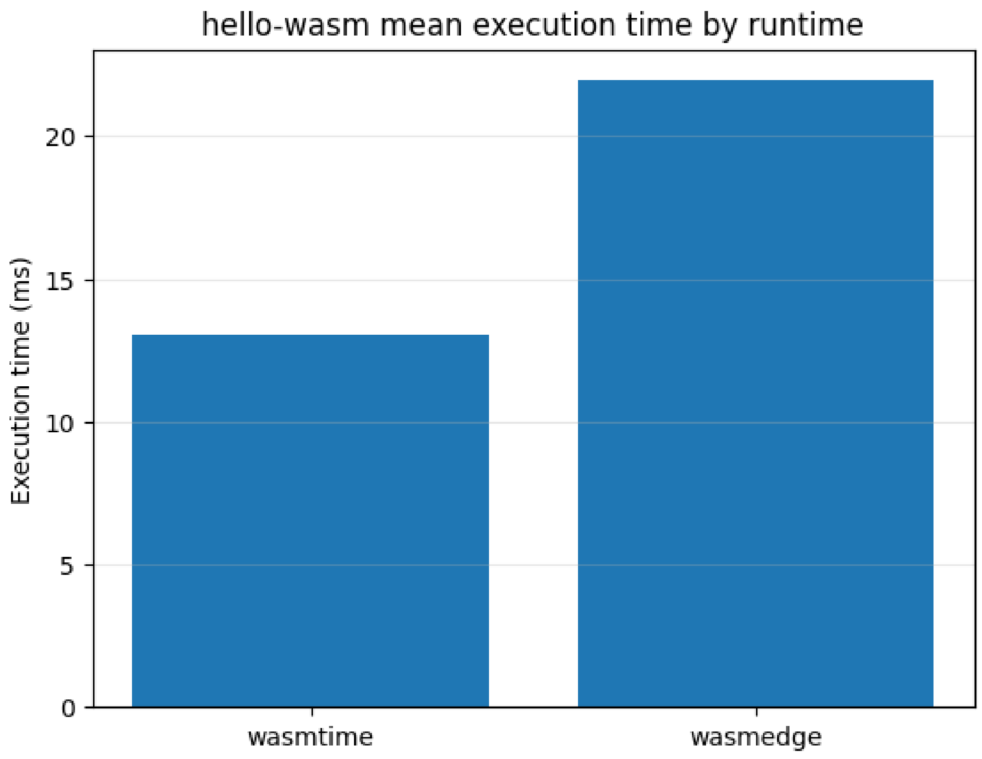

Figure 5: hello-wasm mean execution time by runtime.

7.4 Cold-Start Comparison (Build + Run)

This comparison separates full cold (build + start) from runtime cold (start only) to show how build cost dominates for containers and language toolchains, while runtime cold highlights pure startup behavior.

Scenario	Runs	Build mean (ms)	Start mean (ms)	Total mean (ms)
Docker full cold	3	3138.31	165.21	3303.52
Docker runtime cold	3	N/A	N/A	199.37
Wasmtime full cold	3	9178.64	116.36	9294.99
Wasmtime runtime cold	3	N/A	N/A	71.63

Table 5: Cold-start comparison dataset. Full cold includes a build from source. Runtime cold uses prebuilt artifacts.

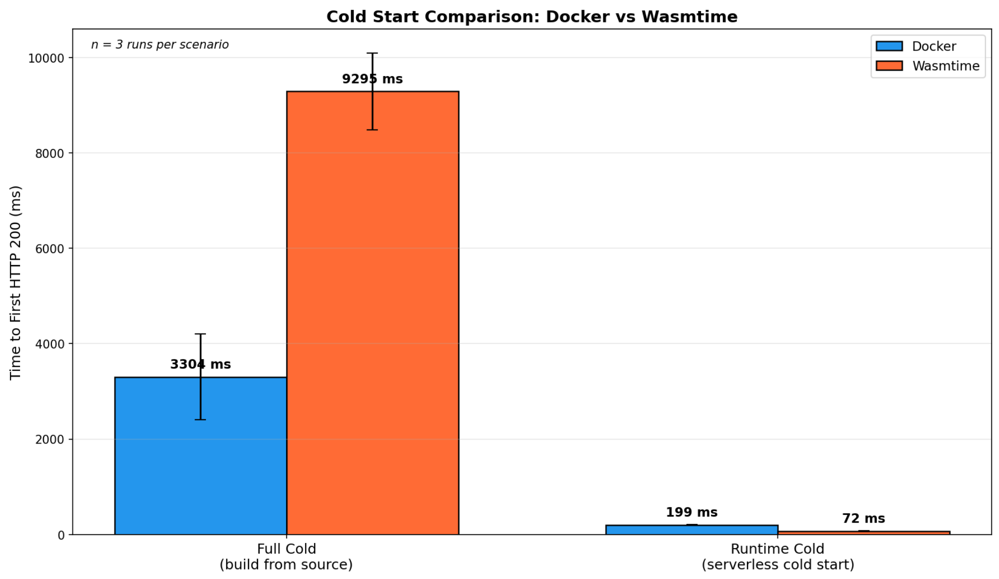

Figure 6: Cold-start comparison (mean time to first HTTP 200).

Interpretation and trade-offs. Full cold runs are dominated by build time, which explains why both Docker and Wasmtime are much slower in that mode. In runtime cold, Wasmtime is faster because it launches a component server with cached artifacts, while Docker must initialize a container and networking. The takeaway is that deployment pipelines that avoid rebuilds (runtime cold) drastically reduce time-to-first-response. Docker remains operationally convenient and widely supported, but it pays a startup cost that WebAssembly runtimes can avoid for latency-sensitive services.

8 Discussion

Cold-start latency. Wasmtime has the fastest runtime cold start, consistent with its direct component server design. wasmCloud exhibits the largest cold-start overhead due to its multi-process host + provider architecture. Docker sits between native and wasmCloud for cold start.

Warm latency and throughput. Warm latencies are tightly clustered around 11–13 ms across runtimes for this minimal handler. Throughput is also close, which suggests the network stack and HTTP server design dominate steady state behavior more than runtime choice for trivial logic.

Resource footprint. Native uses minimal memory, while wasmCloud is substantially heavier because it includes wasmCloud host, NATS, wadm, and the HTTP provider process. Wasmtime is moderate, reflecting a single runtime server plus caching.

CPU-bound performance. Wasmtime is close to native on this workload. Docker is slower (container overhead and limited CPU scheduling). WasmEdge is significantly slower on this macOS setup; this warrants follow-up on platform optimization and runtime configuration.

9 Limitations and Validity Threats

- Results are from a single macOS laptop; performance may differ on Linux servers or cloud hardware.
- Aggregated datasets include both `/` and `/state`, and firewall on/off runs; separate analysis by condition is future work.
- Docker RSS was not captured in logs, so memory efficiency comparisons exclude Docker.
- Cold-start times can be influenced by OS caching, background activity, and filesystem state despite attempts to reset caches.

10 Future Work

- Add Firecracker or unikernel baselines on a Linux host.
- Separate analysis by firewall state and endpoint (`/` vs `/state`).
- Add more realistic microservice logic (JSON parsing, database calls).
- Automate collection of Docker RSS for parity with other runtimes.

11 Conclusion

This project delivers a reproducible benchmark suite for WebAssembly and container runtimes on macOS. The results show that Wasmtime provides the fastest cold starts and competitive throughput, while wasmCloud trades higher startup and memory cost for composability and capability-based design. Docker remains a solid baseline but shows higher latency for CPU-bound workloads. Overall, the data supports the idea that WebAssembly components can outperform containers for short-lived, latency-sensitive services, especially when runtime cold start is the dominant cost.

References

- wasmCloud: `https://wasmcloud.com/docs`
- Wasmtime: `https://github.com/bytecodealliance/wasmtime`
- WasmEdge: `https://github.com/WasmEdge/WasmEdge`
- Docker: `https://www.docker.com`