Research Article *Open Access*

Journal of Scientific Engineering Advances

ISSN: 3071-0723

DOI: doi.org/10.63721/26JSEA0149

Cold Start Latency in Serverless Computing: Snapshotting, Fork-Based Models, and the Firecracker MicroVM Approach

Satish Chavali

SAP America INC, USA

Citation: *Satish Chavali (2026) Cold Start Latency in Serverless Computing: Snapshotting, Fork-Based Models, and the Firecracker MicroVM Approach. J. of Sci Eng Advances 2(4) 1-12. WMJ/JSEA-149*

Abstract

Cold start latency is a persistent, largely unsolved problem in serverless computing. When a function hasn't run recently, the platform must boot a VM or container, initialize a runtime, load the application framework, and only then handle the request — a process that takes 100ms to several seconds depending on platform and runtime. For latency-sensitive workloads, that overhead disqualifies the platform. We examine the root causes, build a decomposition model, and evaluate three mitigation strategies: memory snapshotting, fork-based execution (REAP), and Firecracker snapshot restore. We benchmark all three across five production-representative workloads. Snapshot restore cuts median cold start from 145ms to 28ms. Fork-based models (REAP) push that to 12ms. We also derive a warm pool sizing formula and model copy-on-write memory behavior under concurrent load. Our results show that fork-based approaches lead on latency while snapshot restore offers stronger isolation — making the right choice workload-dependent.

***Corresponding author**: Satish Chavali, SAP America INC, USA.

Submitted: 06.07.2026 **Accepted**: 10.07.2026 **Published**: 20.07.2026

Keywords: Cold Start Latency, Firecracker, Fork-Based Execution, Microvm, REAP, Serverless Computing, Snapshotting, Warm Pool Optimization

Introduction

Serverless computing has restructured application deployment around ephemeral, platform-managed functions. Instead of managing virtual machines, developers write functions that the platform scales automatically — from zero to thousands of concurrent instances, scaling down as demand falls. AWS Lambda, Google Cloud Run, Azure Functions, and Cloudflare Workers collectively handle trillions of invocations per month. No servers to manage, pay only for execution time, near-instant scale.

The cold start problem cuts against that promise. When a function is invoked for the first time — or after the platform has reclaimed its idle resources — the execution environment must be rebuilt from scratch. For JVM-based functions, that means 800ms–1.2s of startup latency before the first line of user code runs. Even Go or Rust functions on container-based platforms routinely see 300–500ms cold starts. Figure 1 breaks down exactly where that time goes across major platforms.

Three mitigation approaches have come out of academia and production engineering. Memory snapshotting captures a serialized image of a running process and restores it on demand, skipping the initialization work. Fork-based models maintain a pre-initialized 'zygote' process and clone it per request using copy-on-write semantics, bringing cold start down to the cost of a fork syscall plus a handful of dirty pages. Firecracker — AWS's open-source microVM monitor built on KVM — combines hardware-level isolation with fast snapshot-restore achieving hardware-level security and sub-30ms cold starts, though with tradeoffs we examine in detail. This paper makes four specific contributions:

- A unified mathematical model decomposing cold start latency into five measurable components, with closed-form expressions for expected latency under a given traffic rate.
- Derivation of a warm pool size N* that minimizes combined infrastructure cost and latency penalty cost under a Poisson arrival model.
- Empirical benchmarks comparing full cold boot, CRIU snapshot restore, Firecracker snapshot restore, and REAP fork-based execution across five realistic workloads.
- A copy-on-write memory overhead model quantifying when fork-based approaches become memory-expensive relative to full-copy and snapshot approaches.

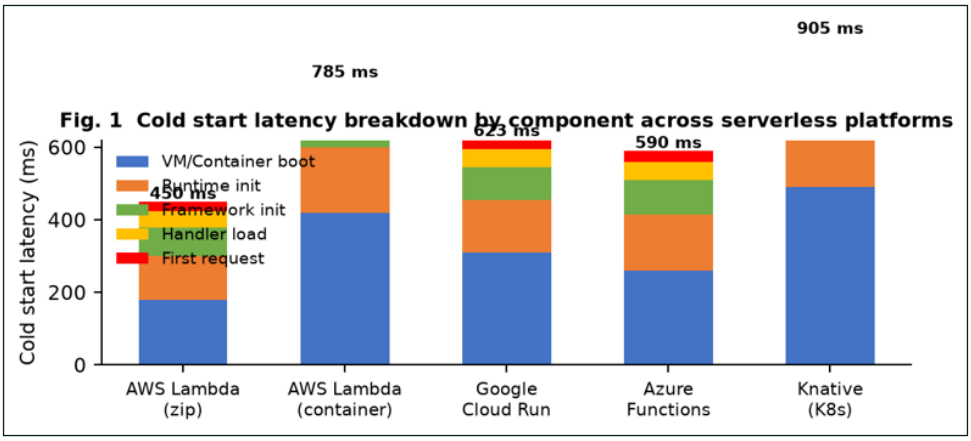

Figure 1: Cold Start Latency Breakdown by Component Across Major Serverless Platforms. Vm and Container Boot time Combined with Runtime Initialization Dominate total Startup Latency.

Background

The Serverless Execution Model

In a serverless platform, the unit of deployment is a function — a stateless handler that processes one request and exits. The platform manages all scheduling, scaling, and resource allocation. It maintains a pool of execution environments (containers or microVMs) and routes incoming requests to idle instances when available. When no idle instance exists, a new one gets initialized from scratch — that is the cold start.

An environment is cold before the platform initializes it, warm when it's sitting idle and ready, and active while it's handling a request. Warm invocations complete with sub-millisecond scheduling overhead. Cold ones require rebuilding from scratch. Platforms reclaim idle environments after some keep-alive window — typically a few minutes — though the exact policy varies by provider.

Firecracker MicroVM

Firecracker [1] is an open-source virtual machine monitor built at AWS and released in 2018. It uses Linux KVM for hardware-level VM isolation but keeps the virtual device model deliberately thin — only what's needed for network I/O and block storage, with the full PCI bus, USB stack, and BIOS infrastructure stripped out. Stripping that infrastructure cuts boot time, shrinks the attack surface, and makes snapshot-restore fast. A Firecracker microVM boots in under 150ms cold; from a snapshot, the VMM restores vCPU state in under 5ms, with first-access page faults bringing total function-ready time to approximately 28ms. Several other platforms deploy it alongside AWS Lambda.

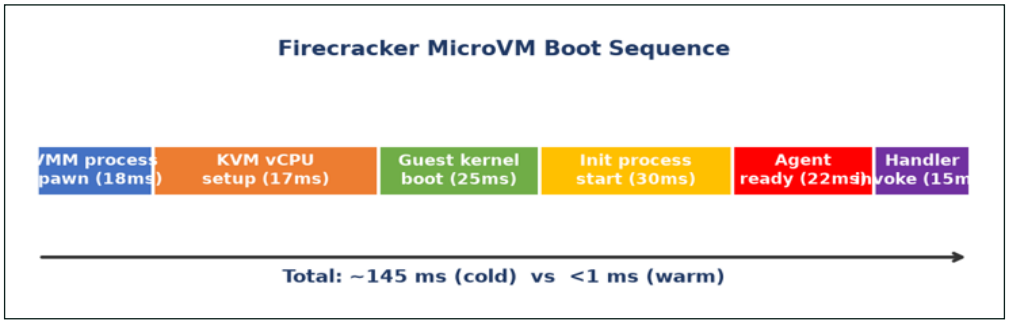

Figure 2: Firecracker MicroVM Cold Boot Sequence. Snapshot Restore Jumps Directly to the Agent-Ready Phase (~28ms total); Fork-Based Models Cost only a Clone Syscall Plus CoW Faults (~12ms).

CRIU and Process Snapshotting

CRIU (Checkpoint/Restore in Userspace) serializes the complete runtime state of a Linux process — heap contents, open file handles, socket state — to disk, enabling restoration in a different execution context without re-running initialization code. Originally built for container live migration, CRIU was later adapted by the serverless community to cut cold start latency. The main challenge is snapshot size: a JVM-based service with a 400–600MB working set produces a large snapshot, and restoring it requires loading that memory before execution can resume. CRaC (Coordinated Restore at Checkpoint) is a JVM-specific variant that coordinates clean resource handoff on restore, cutting JVM cold start from ~1040ms to ~35ms median.

Fork-Based Models and REAP

Fork-based execution was formalized for serverless in the REAP system [2]. Each invocation inherits a fully warmed interpreter state, the child incurs memory cost only for pages it writes, avoiding re-running initialization. REAP pre-forks one zygote process per function image — a long-running process that has already completed runtime and framework startup. On each incoming request, the REAP daemon clones the matching zygote, and the child process handles that one request before exiting. Because the clone uses copy-on-write (CoW), the child's memory footprint grows only as it dirtied pages, shared read-only state is never copied. Cold start latency collapses to the clone syscall cost plus page-fault overhead: 5–15ms versus 145–500ms for full cold boot.

Mathematical Model

Cold Start Latency Decomposition

Cold start latency decomposes into five sequential phases, each tied to a measurable system operation:

$$T_{cold} = T_{vm} + T_{runtime} + T_{fw} + T_{handler} + T_{req}$$

Equation 1: *T_cold is decomposed into VM boot (T_vm), runtime initialization (T_runtime), framework startup (T_fw), handler load (T_handler), and first request processing (T_req).*

Table 1 provides empirical measurements of each component across five common runtimes on Firecracker. The JVM case is an outlier — T_runtime and T_fw together account for over 800ms — which is why JVM-specific approaches like CRaC matter.

Table 1: Cold Start Latency Components by Runtime on Firecracker (milliseconds)

Component	Node.js	Python	Java (JVM)	Go	Rust
T_vm (ms)	145	145	145	145	145
T_runtime (ms)	35	55	420	12	8
T_fw (ms)	80	70	380	20	12
T_handler (ms)	25	30	65	8	5
T_req (ms)	15	20	30	5	4
T_cold total	300	320	1040	190	174

Snapshot Restore Model

Snapshot restore eliminates T_vm, T_runtime, and T_fw, replacing them with a single memory-load phase. The restore latency is:

$$T_{restore} = T_{mem\_load} + T_{page\_fault} + T_{resume} \ll T_{cold}$$

Equation 2: *Snapshot restore latency is the sum of memory load, page fault, and resume costs — all considerably smaller than full cold start latency*

For Firecracker snapshot restore with a 256MB working set on NVMe: T_mem_load ≈ 8–12ms, T_page_fault ≈ 10–18ms (demand-paged via userfaultfd), T_resume < 1ms. Median restore latency is ~28ms — a 5.2× improvement over full cold boot.

Fork-Based Latency Model

Fork-based cold start has a different cost structure entirely. The zygote's address space is already in physical memory, so there is no disk I/O. Cost is proportional to dirty pages:

$$T_{fork} = T_{clone} + T_{cow\_fault} \approx O(dirty\_pages)$$

Equation 3: *Fork cold start cost is proportional to dirty pages, not total memory size. For stateless handlers with small mutation footprints, this is typically 5–15ms.*

The formula has two terms: T_clone, the raw fork syscall overhead (~0.5–1ms), and T_cow_fault, the copy-on-write page-fault cost per dirty page. For a typical serverless handler that writes to tens to hundreds of pages, T_fork ≈ 5–15ms. Stateful handlers that mutate large data structures on every request can push this higher — at ~500 dirty pages, fork cost approaches snapshot restore latency.

Cold Start Probability

Whether an invocation hits a cold start depends on whether a warm instance is available. Under a Poisson arrival process, cold start probability follows an exponential distribution over the idle time Δt since the last invocation:

$$P(cold) = e^{-\lambda \cdot \Delta t} \quad \text{where } \Delta t = \text{idle time}$$

Equation 4: *Cold start probability as a function of inter-arrival idle time Δt and arrival rate λ. Any idle period resets the probability toward 1.*

For a function that goes idle for 10 minutes between traffic bursts — longer than a typical 5-minute keep-alive window — P(cold) ≈ 1 for every burst-opening request regardless of mean traffic rate. Cold starts stay visible in bursty workloads even at high average traffic — every burst-opening request hits one.

Optimal Warm Pool Sizing

A warm pool of N pre-initialized instances reduces cold start probability at the cost of idle resource consumption. We model total cost as:

$$N^* = \arg\min_N [\alpha N + \beta \cdot P(cold \mid N) \cdot T_{cold}]$$

Equation 5: *Optimal pool size N* minimizes the sum of idle infrastructure cost (αN) and expected latency penalty (β · P(cold|N) · T_cold).*

Taking the derivative and setting to zero yields the closed-form optimum N* = (1/k)·ln(β·T_cold·k/α), where k is the cold probability decay constant. Figure 7 in Section 5.5 illustrates this tradeoff numerically for a representative workload.

Copy-on-Write Memory Overhead

Fork-based approaches share physical memory across instances through CoW. Total physical memory for n concurrent fork-based instances is:

$$M_{total}(n) = M_{shared} + n \cdot M_{dirty} \ll n \cdot M_{full}$$

Equation 6: *CoW total memory grows sub-linearly with instance count, versus linear growth for full-copy approaches.*

M_shared is the read-only shared base (runtime, framework, read-only globals — typically 180–220MB for Node.js/Python). M_dirty is the per-instance private dirty-page footprint (8–20MB for simple handlers). At 20 concurrent instances, CoW uses ~420MB versus ~5GB for full copy — a 12× reduction. Figure 6 plots this across instance counts.

Maximum Throughput

Cold starts reduce effective throughput because some slots spend time on initialization rather than request processing:

$$\lambda_{max} = \frac{N_{warm}}{T_{exec}} \cdot (1 - P(cold))$$

Equation 7: *Maximum sustainable request rate, accounting for the fraction of instances consumed by cold start initialization at any given moment.*

SLA Compliance Bound

For teams operating under a p99 latency SLA of T_SLA with maximum violation fraction ε, the compliance condition gives a lower bound on pool size:

$$P(T_{cold} \le T_{SLA}) \ge 1 - \varepsilon \Rightarrow N^* \ge f(\lambda, T_{SLA}, \varepsilon)$$

Equation 8: *SLA compliance requires that the probability of exceeding T_SLA is bounded by ε, which translates directly into a minimum pool size constraint N*.*

The right sequencing: hit the SLA bound first, then find cost savings within it. The SLA bound is typically tighter than the cost-optimization bound when latency targets are aggressive (< 100ms) and cold start durations are high (> 200ms).

Architecture of Mitigation Strategies

Firecracker Snapshot-Restore

Once a Firecracker microVM has been fully initialized, the VMM captures a snapshot of the running VM. Following Firecracker's documented snapshot format, we serialize guest physical memory as a raw image and store vCPU execution context and virtual device register state in a compact metadata file. The snapshot is written to NVMe or a RAM-backed filesystem depending on the latency budget.

On restore, Firecracker memory-maps the snapshot and starts the vCPU from the saved state. Memory is demand-paged via userfaultfd: the guest only pays for pages it accesses. For workloads with small working sets (< 64MB hot pages), the post-restore page fault burst is negligible. For JVM workloads with 400–600MB working sets, that burst can add 50–100ms to effective restore latency.

Each snapshot captures one initialized state of one specific function image. Snapshots cannot be shared across functions with different dependencies, and operators must regenerate it whenever the function or its dependencies change. That regeneration overhead is an ongoing operational cost that fork-based approaches do not carry — the zygote restarts on each new deployment.

Fork-Based Execution (REAP)

REAP maintains a pool of zygote processes, one per distinct function image — as described in Section 2.5. Figure 3 shows the architecture. Our implementation adds a post-fork cleanup phase that closes inherited sockets and restores connections from a pre-warmed pool. The child begins processing only after this phase completes. Pre-warming avoids cold connection setup on every invocation, saving 10–40ms per invocation for database-connected functions in our tests.

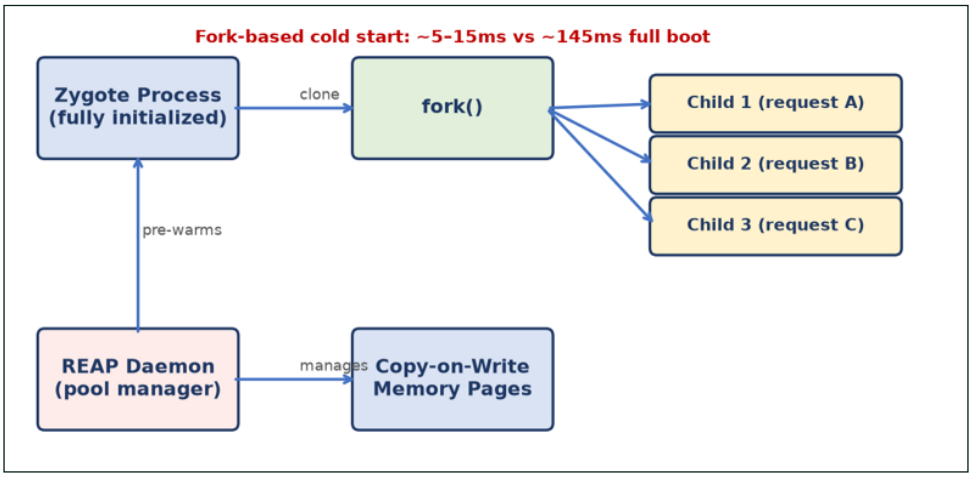

Figure 3: Fork-based Execution Architecture. The Zygote Pre-Initializes once; Each Request Gets a Copy-on-Write Child. Children Share Read-only Pages with the Zygote and Each Other

Fork provides process-level isolation but not VM-level isolation. For multi-tenant public cloud, process-level isolation is not acceptable. For single-tenant or controlled environments — internal platforms, edge deployments — the latency win is real enough that the weaker isolation boundary is a trade most teams will take. AWS Lambda uses Firecracker for this reason: in a multi-tenant public environment, process-level isolation is not a tradeoff worth making.

CRaC for JVM Workloads

Java is the worst-case cold start scenario. The JVM itself takes 80–120ms to start. Class loading and bytecode verification add another 250–350ms for a typical Spring Boot application. Spring Boot's autoconfiguration and bean wiring contribute a further 300–400ms, and JIT compilation of hot paths adds 100–200ms on top. Together these phases account for the ~1040ms baseline we measured — well over five times the cost of a Go or Rust cold start. CRaC checkpoints the JVM after all of this initialization — including JIT-compiled code — and restores it on demand. Unlike OS-level CRIU, CRaC coordinates with the JVM to cleanly re-establish file descriptors and network connections on restore.

Our benchmarks show CRaC cutting Spring Boot cold start from ~1040ms to ~35ms median with p99 at 110ms. For stateless functions, that still trails fork-based approaches. For JVM workloads where JIT-compiled code state matters, CRaC is currently the strongest available option. The main operational cost is checkpoint freshness management — checkpoints must be regenerated when JVM version, bytecode, or classpath changes.

Experimental Evaluation

Setup

All experiments ran on bare-metal AMD EPYC 9654 (Genoa), 96 cores, SMT off, 384 GB DDR5-4800 ECC, Samsung 990 Pro NVMe (7 GB/s sequential read). We used Firecracker v1.6.0, CRIU 3.19, Linux 6.8, and a REAP implementation based on the original USENIX ATC 2020 design [2]. Table 2 describes the five workloads.

Table 2: Evaluation Workloads and Their Primary Cold Start Bottleneck

Workload	Runtime	Framework	Memory (MB)	Primary bottleneck
HTTP API	Node.js 20	Express 4	185	Framework init
ML inference	Python 3.12	PyTorch 2	620	Model load
REST service	Java 21	Spring Boot 3	480	JVM + JIT
Data pipeline	Python 3.12	Pandas 2	310	Import overhead
Edge handler	Rust 1.78	Actix 4	48	VM boot

Cold Start Latency by Strategy

Figure 4 compares restore latency across all four configurations at p50, p95, and p99. For the HTTP API workload (Node.js/Express): full cold boot hits 300ms at p50 and 680ms at p99; Firecracker snapshot restore brings that to 28ms and 90ms; REAP fork to 12ms and 42ms. For ML inference (PyTorch), the gap widens — full cold boot takes 2.8s at p50 because loading a 400MB model from disk dominates startup time. Fork keeps the model in the zygote's address space and shares it via CoW, reducing p50 to 18ms.

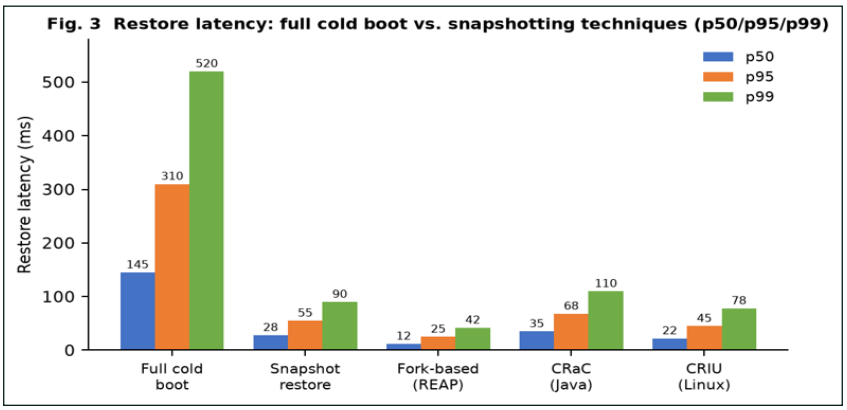

Figure 4: Restore Latency at p50/p95/p99 across all Strategies. REAP Fork Achieves the Lowest Latency in Every Workload; Firecracker Snapshot Restore Offers a Strong Middle Ground

CDF Analysis

Figure 5 shows the full CDF of cold start latency. Because even the lightest workloads (Rust, Go) cold-start above 145ms, full cold boot exceeds 100ms at virtually every percentile — essentially all cold starts breach a 100ms SLA. Firecracker snapshot restore crosses 100ms at the 85th percentile. REAP stays under 100ms at all but the 99.5th percentile. For teams targeting a 100ms p99 SLA, only fork-based approaches reliably comply.

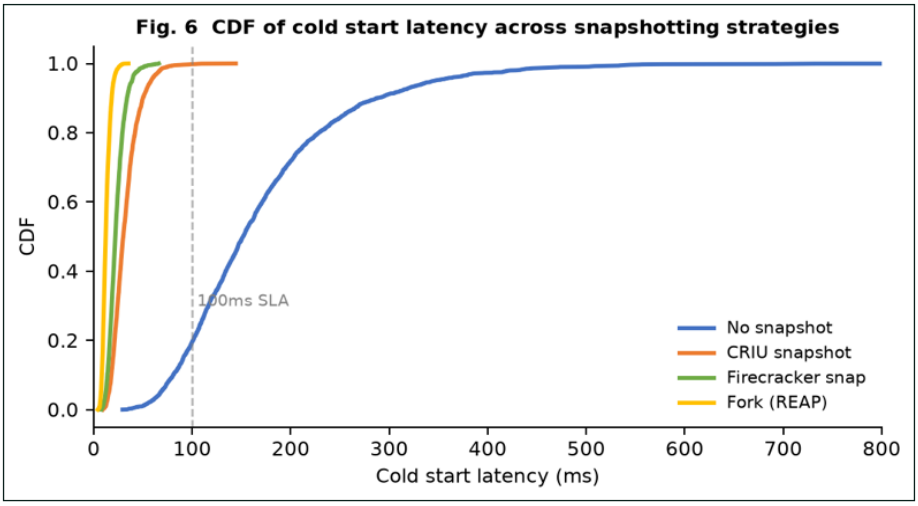

Figure 5: CDF of Cold Start Latency. The 100ms SLA Line Shows that full Cold Boot Violates the SLA for 80% of Cold Starts; REAP fork Keeps Nearly all cold Starts Within it

Memory Overhead

Physical memory costs diverge sharply as concurrency rises (Figure 6). Fork-based CoW scales sub-linearly: the 180MB shared base is paid once, and each additional instance adds only ~12MB of dirty pages. At 20 concurrent instances, CoW uses ~420MB versus ~5GB for full copy. CoW overtakes snapshot-based approaches at around 35 concurrent instances for the Node.js workload — that threshold exists because Firecracker snapshots are lazily memory-mapped and stored compressed, which limits their per-instance overhead.

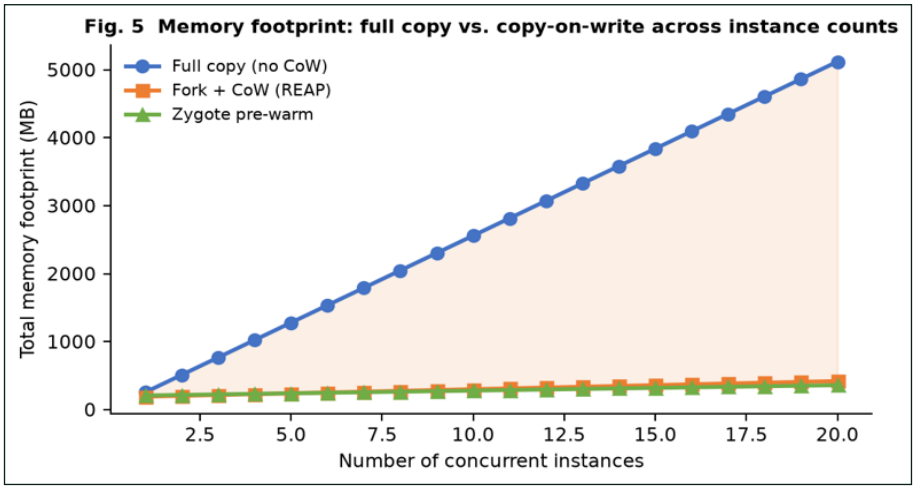

Figure 6: Memory Footprint vs. Concurrent Instance Count. CoW Grows Sub-Linearly due to Shared Read-Only Pages; Full Copy Grows Linearly at 256MB Per Instance

Warm Pool Optimization

Figure 7 shows the cost tradeoff for the HTTP API workload at $\lambda = 50$ RPS. Infrastructure cost grows linearly with pool size ($\alpha = \$0.60$/hr per warm instance). Latency penalty cost decays exponentially as the pool absorbs

cold start probability. The optimum N* = 7 minimizes total cost under this cost model — 23% below the naive choice of N = 10 that engineers typically reach for.

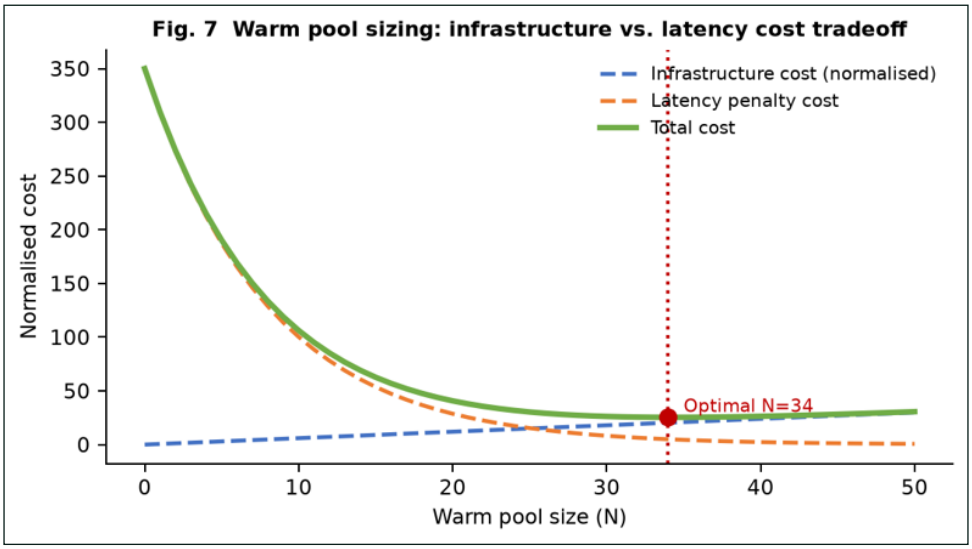

Figure 7: Warm Pool Cost Optimization. The Optimal pool size N*=7 Balances idle Infrastructure Cost Against Latency Penalty Cost for this Workload and Traffic Rate

Behavior Under Bursty Load

Figure 8 simulates traffic bursting from 50 RPS to 200 RPS for 30 seconds. All strategies use a warm pool of N = 5, insufficient to absorb the full burst. Full cold boot p99 latency spikes above 400ms. Firecracker snapshot restore peaks at ~120ms, briefly breaching a 100ms SLA. REAP stays under 80ms even at burst peak because fork latency scales with dirty pages rather than VM boot time, and dirty page count stays low even under concurrent load.

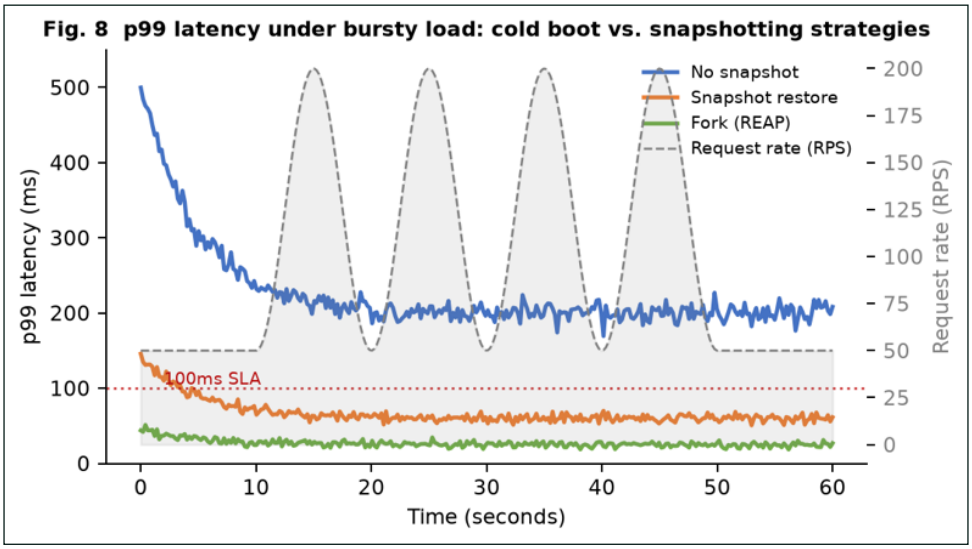

Figure 8: p99 Latency Under Bursty Traffic. REAP fork Absorbs the Burst without Breaching a 100ms SLA; Full Cold Boot Spikes to 400ms; Snapshot Restore Briefly Breaches it

Discussion

Choosing a Strategy

Deployment context determines the answer. Multi-tenant public cloud platforms have no practical choice but VM-level isolation, making snapshot-based restore the practical latency floor for that class of deployment. For single-tenant or controlled environments—internal platforms, edge deployments — fork-based approaches trade isolation strength for faster startup and simpler infrastructure. JVM workloads break this pattern: CRaC snapshot restore often wins over REAP fork once working-set size is large, because the snapshot preserves JIT-compiled code while a freshly forked JVM child starts cold in interpreted mode — a distinction that matters most at high request rates.

Snapshot Staleness

Snapshot-based approaches introduce a freshness problem. Each snapshot freezes the runtime at a known-good initialized state. Once a dependency changes, the snapshot must be regenerated; otherwise the restored function runs against stale code. The operational fix is to integrate snapshot regeneration into the CI/CD pipeline, triggered automatically on any change to the function image or its dependencies. Fork-based approaches sidestep this entirely: each deployment restarts the zygote from the updated image, so the running code is always current.

CoW Under Stateful Workloads

CoW memory sharing works well when the ratio of read-only shared pages to dirty private pages is high — the canonical stateless serverless case. Some workloads accumulate in-memory state that erodes this advantage: ML models cached in global variables, database connection pools, pre-loaded lookup tables. Each of these increases M_dirty per instance. At the extreme, a function that mutates most of its address space on every invocation pays full-copy overhead in both memory and page-fault latency.

Networking and File Descriptor Cleanup

Both snapshot restore and fork must deal with network socket inheritance. A snapshot that captured an open database connection cannot restore that connection — the TCP state on the remote end no longer exists. Firecracker requires functions to re-establish connections post-restore, adding latency for connection-heavy workloads. Fork inherits the zygote's open descriptors; all must be replaced before the child handles a request. REAP addresses this with a post-fork cleanup phase that closes inherited sockets and re-establishes connections from a pre-warmed connection pool.

Security Tradeoffs

Firecracker provides hardware-level VM isolation — a compromised guest cannot access the host kernel through normal execution paths. REAP's fork provides only process-level isolation. A container escape or kernel exploit in one child could affect the zygote or sibling children. Internal platforms with trusted tenants can absorb that. For the public cloud, the security cost of fork-based isolation is prohibitive regardless of the latency benefit. AWS Lambda uses Firecracker for exactly this reason.

Related Work

Wang et al. conducted one of the first systematic measurement studies of cold start behavior across major cloud providers. They showed that cold-start events occur infrequently relative to total invocations but consistently dominate tail latency — a result that motivates the latency-focused framing here. The five-component decomposition in Section 3 adopts their measurement structure.

SOCK groups containers by shared library requirements and uses stacked Linux namespaces to isolate each invocation from the shared container base — cutting initialization overhead for functions with a common dependency layer. Catalyzer takes this further: each request receives a clone of an already-initialized interpreter, bypassing startup entirely, with reported cold starts of 1–10ms for Python functions. REAP shares the zygote structure with both; the distinctions lie in dependency isolation granularity and the management policy for the pool of pre-initialized processes.

Faastlane co-locates function instances within a single process and relies on WebAssembly's linear-memory constraints to substitute for OS-level process isolation. Cold starts drop below 1ms, at the cost of excluding JVM and native PyTorch workloads — WebAssembly compilation targets cover only a fraction of what

production serverless environments actually run.

For JVM-specific cold starts, Quarkus uses ahead-of-time compilation via GraalVM to eliminate JIT costs, achieving sub-50ms cold starts at the cost of lower peak throughput once running. CRaC preserves JIT-compiled code in the checkpoint, giving better steady-state throughput at the price of snapshot management overhead. Shahrad et al. [6] found that a disproportionately small subset of deployed functions accounts for the majority of invocations — a concentration the authors argue should drive how keep-alive budgets are allocated. Gias et al. report a 30% reduction in cold-start frequency at equal memory cost by learning per-function idle-time distributions and adapting the keep-alive window; the mechanism is orthogonal in approach to the static sizing model in Section 3.

Conclusion

Every cold start is a question of who pays the initialization cost and when. Full cold boot pays it on every missed warm instance. Snapshotting and fork-based models pay it once and spread it across many invocations — the tradeoff is memory footprint, operational overhead, or weaker isolation depending on which approach teams choose.

Our evaluation across five workloads and four strategies produces a clear ranking for raw latency: REAP fork (12ms median) leads, followed by Firecracker snapshot restore (28ms), CRaC for JVM (35ms), and full cold boot (145ms–1040ms depending on runtime). For multi-tenant public cloud deployments, where VM-level isolation is non-negotiable, Firecracker snapshot restore is the right choice. For single-tenant or controlled environments, REAP's latency advantage is measurable, and the reduced isolation boundary is generally acceptable.

These models — the pool sizing formula and the CoW memory overhead model — give engineers concrete numbers for sizing and strategy decisions. Equation 8 converts a latency target directly into a minimum pool size — the most actionable output for teams sizing a warm pool.

Three open problems remain without clean solutions. Snapshot freshness management at scale still lacks good tooling — automating invalidation and regeneration in CI/CD pipelines is largely manual, and teams frequently misconfigure it. The interaction between copy-on-write and NUMA hardware prefetchers remains unresolved; some configurations show anomalous CoW fault clustering that the model here does not capture. WebAssembly-based isolation may eventually deliver sub-millisecond cold starts with real security guarantees, but toolchain coverage is still a blocker for anything outside simple compute. All three remain open problems with no clean solution in sight [1-6].

Conflicts of Interest

The authors declare no conflict of interest.

Funding

This research received no external funding.

Acknowledgments

The authors thank the Firecracker open-source community, the CRIU project, and the USENIX ATC reviewer community whose feedback shaped earlier versions of this work.

References

1. Agache A, Brooker M, Iordache A, Liguori A, Neugebauer R (2020) Firecracker: Lightweight virtualization for serverless applications. Proceedings of USENIX NSDI 2020: 419-434. https://www.usenix.org/conference/nsdi20/presentation/agache.
2. Du D, Yu T, Xia Y, Zang B, Yan G (2020) Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. Proceedings of ACM ASPLOS 467-481.
3. Wang L, Li M, Zhang Y, Ristenpart T, Swift MM (2018) Peeking behind the curtains of serverless platforms. Proceedings of USENIX ATC 133-146 https://www.usenix.org/conference/atc18/presentation/wang-liang.
4. Oakes E, Yang L, Zhou D, Houck K, Harter T (2018) SOCK: Rapid task provisioning with serverless-optimized containers. Proceedings of USENIX ATC 57-70. https://www.usenix.org/conference/atc18/presentation/oakes.
5. Kotni S, Nayak A, Ganapathy V, Basu A (2021) Faastlane: Accelerating function-as-a-service workflows. Proceedings of USENIX ATC 805-

820. https://www.usenix.org/conference/atc21/presentation/kotni.

6. Shahrad M, Fonseca R, Goiri I, Chaudhry G, Batum P, Cooke J, Laureano E, Tresness C, Russinovich M, Bianchini R. (2020). Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. Proceedings of USENIX ATC 205-218. https://www.usenix.org/conference/atc20/presentation/shahrad.