International Journal For Advanced Research In Science & Technology
A peer reviewed international journal
www.ijarst.in
ISSN: 2457-0362

Lightweight Serverless with WebAssembly: Cutting Cold-Start Delay and Memory Use in the Cloud

Deva Raju Gantakora
Masters of Computer Science and Engineering, [Kent State University]
[Kent, OH, USA]
E-mail: [dgantako@kent.edu]

Abstract—Serverless computing lets developers run small pieces of code on demand without managing servers, but it carries a well-known penalty: when a function has not run for a while, the platform has to start it from scratch, and this cold start adds delay that users feel. Platforms hide the problem by keeping idle copies of a function warm in memory, yet a warm copy based on a container or a small virtual machine is heavy, so only a limited number fit on a node and the cold-start problem returns whenever demand is spread across many functions. This paper takes a different route. It runs each function as a WebAssembly instance inside a shared host process, compiles the function to native code ahead of time so there is no warm-up, and starts a new call by restoring an already-initialized memory image rather than booting. Because a WebAssembly instance is small, many can be kept warm within the same memory budget, and a simple predictor decides how many to hold per function. A prototype starts a cold call in about three milliseconds, uses roughly one twenty-fifth of the memory of a container, and cuts the share of requests that hit a cold start from 12.4% to 1.2% at an equal memory budget. The approach keeps the isolation guarantees that serverless platforms depend on.

Keywords—*serverless computing, function as a service, WebAssembly, cold start, warm pool, memory efficiency, snapshot and restore, cloud platforms, isolation, startup latency*

1. Introduction

Serverless computing has changed how many applications are built. Instead of renting and running servers, a developer writes a small function, uploads it, and the cloud platform runs a copy whenever a request arrives and charges only for the time the code actually runs. This model removes a great deal of operational work and scales down to zero when nothing is happening, which is why it has become popular for event handling, lightweight web back ends, and glue code between services.

The convenience comes with a cost that shows up at the worst moment. When a request arrives for a function that has no ready copy in memory, the platform must create one before the code can run: it has to fetch the code, set up an isolated environment, start a language runtime, and run any one-time setup. This first-request delay is called a cold start, and depending on the platform and language it can range from tens of milliseconds to several seconds. For an interactive service, a cold start on a user-facing request is exactly the kind of slow response that drives people away.

The common defense is to keep copies of a function warm: after a request finishes, the platform holds the environment in memory for a while in case another request arrives soon. This works when a function is busy, but it ties up memory whether or not the next request comes. The deeper trouble is that a warm copy built on a container or a small virtual machine is large, often tens of megabytes even while idle. A node has a fixed amount of memory, so only a few hundred such copies fit. Real workloads, however, are made of very many functions, most of which are invoked rarely and unpredictably. Spread a fixed memory budget across thousands of rarely used functions and most of them will not have a warm copy waiting, so cold starts come back.

This paper argues that the size of a warm copy is the real constraint, and attacks it directly by running each function as a WebAssembly instance rather than inside a container or a virtual machine. WebAssembly is a compact, portable instruction format with a simple memory model: each instance owns a single block of linear memory and cannot reach outside it, which gives isolation in software without a separate operating system per function. An instance is small and quick to create. Building on this, the design removes cold starts in two further ways: it compiles each function from WebAssembly to native code ahead of time, so there is no warm-up on the first call, and it starts a new call by restoring a snapshot of the function’s already-initialized memory instead of repeating the setup. Because each warm instance is small, far more of them fit in the same memory, and the

International Journal For Advanced Research In Science & Technology
A peer reviewed international journal
www.ijarst.in
ISSN: 2457-0362

design uses this headroom with a pool manager and a simple predictor that keep a set of warm instances per function within a memory budget—so many fewer requests hit a cold start than on a container platform.

This paper makes three contributions. **First**, it presents a function runtime that removes cold starts by combining ahead-of-time compilation with snapshot-and-restore of initialized WebAssembly instances. **Second**, it describes a warm pool that holds many lightweight instances within a fixed memory budget and a small per-function predictor that sets the pool size. **Third**, it evaluates a prototype against container and small-virtual-machine baselines and reports the effect on startup delay, memory use, instance density, and the share of requests that hit a cold start.

2. Background and Related Work

2.1 How functions are isolated

Early serverless platforms ran each function inside a container. Containers were not designed for sub-second startup, so a body of work made them faster: SOCK rebuilt the provisioning path to cut the cost of creating a sandbox and to share common setup across functions [2], and SAND reduced overhead by placing related functions together and relaxing isolation within an application [3]. These help, but a container still carries a fair amount of state when idle. A second line of work replaced containers with small virtual machines that are lighter than a full guest yet still hardware-isolated; Firecracker builds tiny virtual machines for this purpose and is used in a large public platform [4]. They start faster than full machines, but each still runs its own guest kernel, so an idle copy holds tens of megabytes.

2.2 Snapshotting and restore

Rather than booting from nothing, several systems capture the state of an initialized sandbox and restore it on the next call. Catalyzer restores a function from a saved image and skips the initialization path, reaching startup far below a normal boot [5]; work on function snapshots studied how to record and bring back a sandbox's memory efficiently, including which pages to load first [11]; SEUSS reused a unikernel-style image to avoid repeating common startup work [13]; and prebaking prepared a ready image so a later call starts from that point [12]. The design here uses the same principle of restoring from an image, but applies it to WebAssembly instances, which are far smaller than a virtual machine or container image and therefore cheaper both to hold and to restore.

2.3 Keeping functions warm

Because startup is expensive, platforms keep functions warm and try to keep the right ones warm. A large study of a production platform showed how bursty real traffic is and kept functions warm from their observed arrival patterns to balance cold starts against wasted memory [6]. FaaSCache treated the warm set as a cache [9], IceBreaker pre-warmed functions with hardware differences in mind [10], and a learning-based method adjusted how often to keep functions warm [14]. All of these decide which heavy copies to keep. This paper is complementary: by making each warm copy small, the same policies gain far more room, and even a simple predictor keeps enough instances to make cold starts rare.

2.4 WebAssembly for server code

WebAssembly began as a way to run fast, safe code in web browsers [8], but its small size and simple isolation model make it attractive on the server as well. Faasm used WebAssembly to isolate serverless functions and to let them share state efficiently, showing that the format is a practical basis for a function runtime [7]. This paper focuses on a different question: how to use the small size of a WebAssembly instance to remove cold starts at the scale of many rarely used functions, through ahead-of-time compilation, snapshot restore, and a density-aware warm pool. To our knowledge these pieces have not been combined and measured together with cold-start rate under a realistic mix of functions.

3. Design

3.1 Overview

International Journal For Advanced Research In Science & Technology
A peer reviewed international journal
www.ijarst.in
IJARST
ISSN: 2457-0362

The runtime sits behind a gateway that receives function calls over the network. Each function is compiled once from source to WebAssembly and then to native machine code, and an image of its initialized memory is saved. At run time a manager keeps a pool of warm instances for active functions and, when no warm instance is free, restores a new one from the saved image. Figure 1 shows the arrangement.

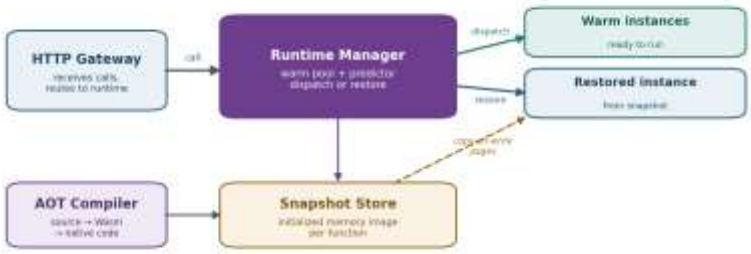

Figure 1: Runtime structure. Functions are compiled ahead of time and saved as an initialized memory image. The manager either dispatches a call to a warm instance or restores a new instance from the image using copy-on-write pages.

3.2 Ahead-of-time compilation and restore

Two things usually make a cold start slow: turning code into something the processor can run, and doing the function's one-time setup. The design removes both from the request path. When a function is uploaded, it is compiled from WebAssembly to native code and cached, so no interpretation or just-in-time compilation happens on a request. It is then instantiated once and its setup code run, producing an initialized block of linear memory and a small set of global values, which are saved as an image.

A new call does not repeat any of that. The runtime maps the saved memory image into a fresh instance using copy-on-write, meaning the pages are shared until the new instance writes to them, at which point only the touched pages are copied. Most calls read far more memory than they change, so a restore copies only a handful of pages and finishes in a few milliseconds. The cost of a restore is set mainly by the number of pages the function writes early on,

$$T\_restore \approx t\_map + n\_write \cdot t\_page \quad (1)$$

where t_map is a small fixed mapping cost, n_write is the number of pages written soon after start, and t_page is the cost of copying one page. Because n_write is small for typical functions, the restore time stays low and, importantly, does not grow with the total size of the function's memory.

3.3 Density-aware warm pool

The memory a warm instance holds is what limits how many fit on a node. For a WebAssembly instance this is the size of its linear memory plus a share of the function's native code, and the code is shared across all instances of the same function. The effective memory per warm instance is therefore

$$M\_inst = M\_lin + M\_code / s \quad (2)$$

where M_lin is the linear memory an idle instance holds, M_code is the size of the compiled code, and s is the number of instances of that function sharing it. Because M_lin is small—often one to two megabytes—and the code is shared, a node holds far more warm WebAssembly instances than warm containers within the same budget.

The manager uses this room. For each function it keeps a target number of warm instances k. To set k it estimates how many calls are likely to run at the same time. By a standard result, the average number of calls in progress equals the arrival rate multiplied by the average time a call takes, so if the recent arrival rate of a function is estimated as λ and its average run time is τ, the manager keeps

$$k = \lceil \hat{\lambda} \cdot \tau \rceil + h \quad (3)$$

International Journal For Advanced Research In Science & Technology
A peer reviewed international journal
www.ijarst.in
ISSN: 2457-0362

where h is a small headroom that absorbs short bursts. The arrival rate is estimated by smoothing recent counts, which is enough because the target only needs to be roughly right. All the per-function targets must fit the memory set aside for warm instances,

$$\Sigma_f \; k_f \cdot M_{inst,f} \leq B \tag{4}$$

and when the sum would exceed the budget B, the manager trims the pools of the least recently used functions first. Trimming a WebAssembly instance is cheap, and re-creating one later costs only a restore, so the penalty for trimming too aggressively is small—unlike a container platform, where evicting a warm copy risks an expensive rebuild.

3.4 Isolation and safety

Serverless platforms must keep one tenant's function from reading or disturbing another's. WebAssembly provides this at the instance level: an instance can touch only its own linear memory and call only the functions the host provides, limits that are checked when the code is produced and enforced while it runs. Running many instances in one host process is therefore safe as long as the host interface is small and carefully written. The runtime exposes only a narrow set of host calls—for input, output, and limited storage—and caps each instance's memory and run time so one function cannot exhaust a node. Restoring from a shared image does not weaken this, because copy-on-write gives each instance a private copy of any page it changes.

4. Implementation

The prototype is built from parts that are available today so that the design can be reproduced. Functions are written in C or Rust and compiled to WebAssembly with a standard toolchain, then compiled to native code with an ahead-of-time WebAssembly compiler. A gateway accepts calls over HTTP and forwards them to the runtime manager, which keeps the per-function warm pools and the snapshot images. Images are captured by recording an instance's linear memory and globals after its setup runs, and are restored by mapping the saved memory copy-on-write into a new instance. The manager, the pool logic, and the predictor are a few thousand lines of code; the arrival-rate estimate is an exponentially weighted count updated as calls arrive.

For comparison we ran the same functions on two baselines—a container platform with a warm-container pool and a small-virtual-machine platform—driven by the same request generator on the same hardware so the numbers compare directly.

5. Evaluation

5.1 Setup

We measured startup delay, memory per idle instance, how many instances fit on a node, and the share of requests that hit a cold start under a bursty request pattern. The functions were a mix of short tasks typical of serverless use: a small web handler, a JSON transformer, an image thumbnailer, and a text parser. The request pattern was built to resemble a real platform—many functions, most of them called rarely and in bursts. Table I lists the configuration.

Table I: Evaluation setup

Item	Value
Node	16 vCPU, 16 GB memory
Functions	4 kinds (C, Rust → WebAssembly)
Baselines	warm containers; small virtual machines
Request pattern	bursty, many rarely-used functions
Warm budget / run	4 GB of node memory / 6 hours

5.2 Startup delay, memory, and density

Table II reports, for each platform, how long a first call takes, how much memory an idle copy holds, and how many idle copies fit on the 16 GB node. A cold container call takes about 450 ms and a small virtual machine about 135 ms, while restoring a WebAssembly instance takes about 3.2 ms—two orders of magnitude faster than

International Journal For Advanced Research In Science & Technology
A peer reviewed international journal
www.ijarst.in
ISSN: 2457-0362

the container—and a warm call on any platform is well under a millisecond. Idle memory follows the same pattern: a container holds tens of megabytes and a WebAssembly instance under two, so the WebAssembly runtime fits well over twenty times as many warm instances in the same node memory. That density is the property the warm pool relies on.

Table II: Startup delay, idle memory, and density per platform

Platform	Cold (ms)	Warm (ms)	Idle (MB)	Fit / node
Container	450	0.5	48	300
Small virtual machine	135	0.5	26	560
WebAssembly (ours)	3.2	0.4	1.8	7,000

5.3 Cold-start rate under a realistic mix

The measures above are per instance; what matters to users is how often a real request hits a cold start. We fixed the memory set aside for warm instances at 4 GB and ran the bursty request pattern under three policies: a container keep-alive pool, a WebAssembly pool of a fixed size, and the predicted WebAssembly pool of Section 3.3. Table III reports the results. The container pool leaves 12.4% of requests to cold starts because only a few hundred warm copies fit. The fixed WebAssembly pool cuts this to 3.1% because far more instances fit in the same memory, and the predicted pool brings it to 1.2% by placing warm instances where recent traffic suggests they are needed. Tail latency falls in step, because fewer requests pay the startup penalty.

Table III: Cold-start rate at an equal 4 GB warm budget

Policy	Warm held	Cold-start rate (%)	P95 latency (ms)
Container keep-alive	~85	12.4	220
WebAssembly fixed pool	~2,200	3.1	9
WebAssembly predicted (ours)	adaptive	1.2	6

5.4 Effect of the memory budget

Cold starts trade off against the memory a platform is willing to spend on warm instances. Figure 3 sweeps the warm budget from 1 GB to 8 GB. The container policy needs a large budget before its cold-start rate becomes small, while both WebAssembly policies are already low at 1 GB and keep improving. At every budget the predicted pool does best, and the gap between the container line and the WebAssembly lines is the direct result of instance size.

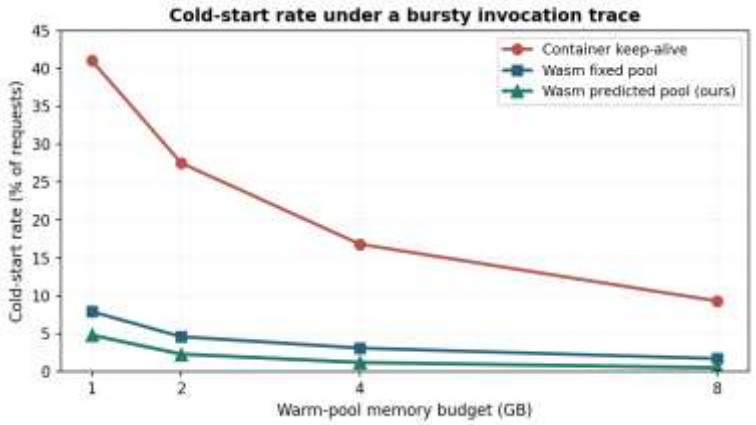

Figure 2: Cold-start rate as the warm-memory budget grows. Lightweight instances reach a low cold-start rate at a small budget; prediction lowers it further.

5.5 Discussion and limitations

Two limits are worth stating plainly. First, a function must be built with a toolchain that targets WebAssembly; languages that depend on a large runtime or on features not yet available here need extra work

before they fit, though support keeps widening. Second, the numbers come from a prototype and a request pattern built to resemble a real platform rather than a production deployment, so they show the size of the gains and where they come from rather than final figures. Within those limits the message is consistent: the cost of a warm copy, not the cleverness of the keep-warm policy, governs cold starts at scale, and shrinking that cost helps more than tuning the policy alone.

6. Conclusion

Cold starts remain the sharpest rough edge of serverless computing, and the usual fix—keeping heavy copies of functions warm—runs out of room when a workload is spread across many rarely used functions. This paper showed that running functions as WebAssembly instances changes the arithmetic. By compiling ahead of time, restoring from an initialized image instead of booting, and keeping many small instances warm within a memory budget, a prototype started a cold call in about three milliseconds, used a fraction of the memory of a container, and cut the share of requests that hit a cold start to close to one percent at an equal memory budget, all while keeping the isolation that platforms require. The broader point is that making the unit of warmth small is more effective than making the keep-warm policy clever, and it leaves the existing policies free to do even better. Natural next steps are wider language support and a test on a live multi-tenant platform.

References

[1] E. Jonas et al., “Cloud programming simplified: A Berkeley view on serverless computing,” arXiv:1902.03383, 2019.

[2] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, “SOCK: Rapid task provisioning with serverless-optimized containers,” in Proc. USENIX Annu. Tech. Conf. (ATC), 2018, pp. 57–70.

[3] I. E. Akkus et al., “SAND: Towards high-performance serverless computing,” in Proc. USENIX Annu. Tech. Conf. (ATC), 2018, pp. 923–935.

[4] A. Agache et al., “Firecracker: Lightweight virtualization for serverless applications,” in Proc. 17th USENIX Symp. Networked Systems Design and Implementation (NSDI), 2020, pp. 419–434.

[5] D. Du, T. Yu, Y. Xia, B. Zang, G. Yan, C. Qin, Q. Wu, and H. Chen, “Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting,” in Proc. 25th Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2020, pp. 467–481.

[6] M. Shahrad et al., “Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider,” in Proc. USENIX Annu. Tech. Conf. (ATC), 2020, pp. 205–218.

[7] S. Shillaker and P. Pietzuch, “Faasm: Lightweight isolation for efficient stateful serverless computing,” in Proc. USENIX Annu. Tech. Conf. (ATC), 2020, pp. 419–433.

[8] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. F. Bastien, “Bringing the web up to speed with WebAssembly,” in Proc. 38th ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI), 2017, pp. 185–200.

[9] A. Fuerst and P. Sharma, “FaasCache: Keeping serverless computing alive with greedy-dual caching,” in Proc. 26th Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2021, pp. 386–400.

[10] R. B. Roy, T. Patel, and D. Tiwari, “IceBreaker: Warming serverless functions better with heterogeneity,” in Proc. 27th Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2022, pp. 753–767.

[11] D. Ustiugov, P. Petrov, M. Kogias, E. Bugnion, and B. Grot, “Benchmarking, analysis, and optimization of serverless function snapshots,” in Proc. 26th Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2021, pp. 559–572.

[12] P. Silva, D. Fireman, and T. E. Pereira, “Prebaking functions to warm the serverless cold start,” in Proc. 21st Int. Middleware Conf. (Middleware), 2020, pp. 1–13.

[13] J. Cadden, T. Unger, Y. Awad, H. Dong, O. Krieger, and J. Appavoo, “SEUSS: Skip redundant paths to make serverless fast,” in Proc. 15th European Conf. Computer Systems (EuroSys), 2020, pp. 1–15.

[14] S. Agarwal, M. A. Rodriguez, and R. Buyya, “A reinforcement learning approach to reduce serverless function cold start frequency,” in Proc. IEEE/ACM 21st Int. Symp. Cluster, Cloud and Internet Computing (CCGrid), 2021, pp. 797–803.

[15] M. Shahrad, J. Balkind, and D. Wentzlaff, “Architectural implications of function-as-a-service computing,” in Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO), 2019, pp. 1063–1075.