Computer Life
ISSN: 1819-4818 | Vol. 14, No. 1, 2026

Cold Start Latency Optimization Strategies for Function as a Service Platforms

Tingjie Chen[1, *] and Jianbo Ding[2]

[1] Xi'an Jiaotong University, China
[2] University of Shanghai for Science and Technology, China
*** Corresponding author:** tingjiec@gmail.com

Abstract: Function as a Service (FaaS) has established itself as the dominant delivery model within the serverless computing (SC) ecosystem, enabling developers to deploy stateless, event-driven workloads without provisioning or managing any underlying server infrastructure. Despite substantial operational advantages in cost granularity and scaling automation, FaaS platforms are subject to a persistent performance bottleneck known as cold start latency, which occurs whenever a new execution environment must be initialized from scratch before an incoming function invocation can be served. Cold start penalties range from tens of milliseconds for lightweight runtimes to several seconds for applications executing on the Java Virtual Machine (JVM), producing direct violations of service-level objectives (SLOs) in latency-sensitive production deployments. This paper reviews optimization strategies for cold start latency across four principal categories: pre-warming and keep-alive policies, snapshot-based checkpoint-restore techniques, lightweight virtualization and isolation mechanisms including WebAssembly (Wasm), and scheduling and resource management strategies. Machine learning (ML) is examined as a cross-cutting enabler for predictive and adaptive mitigation. This review synthesizes the current state of understanding, characterizes trade-offs among competing strategies, and identifies open challenges including snapshot staleness management, isolation-speed tensions, and edge deployment constraints.

Keywords: Function as a Service; Serverless computing; Cold start latency; Pre-warming; Checkpoint-restore; Lightweight virtualization; WebAssembly; Scheduling optimization.

1. Introduction

The serverless computing (SC) paradigm has fundamentally reshaped how cloud applications are designed, deployed, and operated. Under the Function as a Service (FaaS) model, developers upload discrete, stateless units of computation that execute on demand in isolated environments fully managed by the cloud provider, with resource consumption billed at per-invocation or per-millisecond granularity [1]. Commercial FaaS offerings including AWS Lambda, Google Cloud Functions, and Microsoft Azure Functions collectively support application workloads spanning web APIs, event-driven stream processing, machine learning (ML) inference, and scientific computing pipelines, and their adoption has grown substantially as engineering teams seek to eliminate the operational burden of server lifecycle management [2]. The FaaS execution model is particularly compelling for applications with variable, bursty, or unpredictable traffic patterns, as the platform's automatic scaling eliminates idle resource costs during low-demand periods and removes the need for manual capacity planning [3].

This on-demand execution model introduces a fundamental tension between resource efficiency and response latency consistency. When a function has been dormant long enough for the platform to reclaim its execution environment, or when a traffic surge exhausts the pool of available warm instances, the platform must initialize a fresh execution environment before the incoming request can be processed. This initialization sequence is universally referred to as a cold start, and it imposes a latency penalty directly observable by the requesting client or by dependent downstream services in multi-function invocation chains [4]. The severity of the cold start penalty is determined by three overlapping initialization phases: infrastructure provisioning, which encompasses container or micro-virtual machine (microVM) creation and function artifact retrieval; runtime initialization, which for the JVM includes class loading, bytecode verification, and just-in-time (JIT) compilation warm-up; and application startup, which encompasses dependency loading, connection establishment, and function-specific initialization logic [5].

Production measurements have confirmed that cold start latency constitutes a consequential operational challenge rather than a merely theoretical limitation. Cold start durations vary by more than two orders of magnitude across runtime environments and function profiles, with JVM-based functions regularly exceeding one second while Go and Python functions initialize in tens of milliseconds [6]. Production workload analysis has established that a substantial fraction of deployed functions are invoked infrequently enough that static platform-wide retention policies leave them persistently cold, and that invocation pattern heterogeneity makes any single timeout value simultaneously too aggressive for some functions and insufficiently aggressive for others [7]. In microservice architectures where FaaS functions compose request-handling pipelines, cold starts at individual stages cascade into amplified end-to-end latencies that violate SLOs even when individual function cold start durations appear tolerable in isolation [8].

Research addressing cold start latency has evolved along four complementary strategy families. Pre-warming and keep-alive strategies maintain warm function instances beyond their last served requests, providing pools of initialized environments ready to absorb subsequent invocations without initialization overhead [9]. Snapshot-

based checkpoint-restore techniques capture the memory state of a fully initialized execution environment as a persistent image and restore it rapidly upon cold invocations, decoupling initialization cost from request response latency [10]. Lightweight isolation mechanisms including Wasm sandboxes, purpose-built microVMs, and lean container designs reduce the inherent initialization overhead of the execution boundary itself, attacking cold start latency at the infrastructure layer rather than managing it through warm instance retention [11]. Scheduling and resource management strategies ensure invocations are routed toward nodes hosting warm instances and that warm pool resources are distributed intelligently across the function fleet. Recent advances in LLM-driven cloud resource scheduling further demonstrate that combining contextual reasoning over heterogeneous system states with optimization-based refinement enables more adaptive and efficient resource allocation under dynamic workload conditions, offering a promising direction for improving scheduling decisions in serverless platforms [12]. Machine learning (ML) has emerged as a cross-cutting enabler across all four directions, applied to predict invocation patterns and trigger proactive warm instance creation before demand materializes.

2. Literature Review

The cold start problem has attracted sustained research attention since the emergence of commercial FaaS platforms, and the body of work addressing it spans systems design, production workload characterization, performance benchmarking, and applied ML. Systematic reviews of the serverless computing field have consistently identified cold start optimization as the most actively studied challenge in the FaaS research community, receiving more dedicated research effort than any other operational concern [13]. Empirical analysis of real-world serverless application deployments has mapped workload usage characteristics and identified deployment patterns that correlate strongly with cold start exposure, providing an evidential foundation for understanding which application classes benefit most from optimization investment [14].

The most influential production workload characterization was conducted through analysis of two weeks of Microsoft Azure Functions traces spanning millions of function invocations across the deployed fleet. This study demonstrated that per-function invocation patterns exhibit extreme heterogeneity, that a large fraction of deployed functions are invoked infrequently enough to be persistently cold under fixed retention policies, and that a histogram-based adaptive policy conditioned on per-function inter-arrival time statistics substantially outperforms platform-wide fixed timeouts for the same memory budget [15]. A complementary mixed-method study involving developer interviews, code repository analysis, and deployment measurement established that cold start unpredictability—rather than mean cold start duration—is the most consistently reported practitioner concern, because variance in initialization time complicates SLO design and makes performance testing unreliable [16].

From a systems perspective, the Firecracker microVM represented a landmark contribution to lightweight isolation for serverless workloads by demonstrating that hardware-level virtualization isolation and sub-100-millisecond boot times are simultaneously achievable through a purpose-built minimal device model with a fixed, stripped-down device set optimized for serverless execution contexts [17]. Systematic benchmarking and optimization of serverless function snapshots identified memory page access patterns during restoration as the binding performance constraint limiting achievable restore speed, and proposed memory layout optimizations that improve restore throughput by approximately 3× on representative benchmarks [18]. Formal treatment of warm instance management as a cache replacement problem demonstrated that greedy-dual eviction policies conditioned on per-function memory footprint and invocation value outperform static timeout retention by over 30 percent in warm hit rate across diverse production-representative workload traces [19].

Wasm-based isolation was demonstrated as a viable alternative to container and microVM boundaries for FaaS workloads, with software fault isolation enabling microsecond-class sandbox instantiation latency for stateful serverless functions and achieving performance competitive with shared-memory programming models for data-parallel workloads [20]. Stateful FaaS execution was further addressed through a runtime built atop a distributed key-value store that extends the standard stateless execution model with shared mutable state, addressing the interaction between stateful execution patterns and snapshot-based cold start optimization strategies [21]. Application-level scheduling knowledge—specifically the precise timing of periodic cron-triggered invocations—was shown to enable effective cold start elimination for scheduled workloads without requiring any ML infrastructure, representing a lightweight practical approach applicable to the large fraction of FaaS workloads driven by scheduled triggers [22].

A taxonomy of cold start mitigation strategies organized by optimization target, prediction mechanism, and resource overhead profile was constructed to provide a reference classification framework, identifying predictive pre-warming as the most promising general direction for future work [23]. The distinct challenges facing SC deployments at the network edge—where resource constraints limit warm pool sizes, hardware heterogeneity limits technique portability, and intermittent connectivity complicates scheduling—were analyzed systematically, establishing the need for dedicated edge-aware optimization strategies rather than direct application of cloud-centric approaches [24]. Purpose-built edge FaaS platforms have been demonstrated to achieve cold start latencies compatible with edge application requirements by sacrificing generality in favor of streamlined initialization paths unavailable to general-purpose platforms [25]. Benchmarking of cold start mitigation mechanisms for AWS Lambda provided practitioner-oriented analysis bridging academic research and commercial platform behaviors, quantifying the cold start reduction achievable through deployment configuration choices within the Lambda execution model [26]. Multi-function pipeline-level cold start amplification was addressed through workflow orchestration that coordinates warm instance provisioning across pipeline stages, pre-warming downstream function instances before upstream functions complete and preventing the cascade of serial cold starts that amplifies end-to-end pipeline latency [27].

3. Anatomy of Cold Start Latency

Understanding the internal structure of cold start latency with precision is a prerequisite for selecting effective optimization strategies, since different strategies target

different initialization phases and may dramatically reduce latency for one class of functions while providing negligible benefit for another. A rigorous decomposition reveals that total cold start latency is the sum of infrastructure provisioning time, runtime initialization time, and application startup time, with the relative dominance of each phase varying substantially across deployment configurations [28]. As illustrated in Figure 1, this phase decomposition differs markedly across runtime environments, with scripted language runtimes dominated by infrastructure and application phases while JVM-based runtimes exhibit runtime initialization overhead that dwarfs all other phases combined. Profiling studies have confirmed that this variation spans two orders of magnitude across representative function types, making a single universal optimization strategy insufficient to address cold start latency across the full diversity of production FaaS workloads [29].

At the infrastructure layer, the platform must instantiate a sandboxed execution environment for each cold-started function. In container-based deployments this involves retrieving the function's container image from a registry if it is not present in the local cache on the worker node, creating Linux namespace and cgroup resource configurations, attaching virtual network interfaces, and mounting function code and configuration artifacts into the container filesystem [30]. Container image retrieval latency is among the most variable contributors to infrastructure initialization time: compact images for simple Python or Node.js functions may be retrieved quickly from local registry caches, while images for complex Java or .NET applications with large transitive dependency trees can exceed several hundred megabytes, introducing multi-second retrieval latencies when the image is absent from the worker node's local storage. Lazy image loading approaches that make only the image layers required for immediate function execution available at startup time, deferring retrieval of unreferenced layers, have been demonstrated to substantially reduce effective image availability time for large-image workloads by decoupling image completeness from execution readiness [31].

At the runtime layer, the programming language execution environment must reach a state capable of executing the function handler code. JVM-based languages require the JVM to perform class loading, bytecode verification, native library initialization, and initial JIT compilation before the function handler can be invoked, a process that regularly consumes several hundred milliseconds to over one second for production-grade applications with substantial framework dependencies [32]. Scripted languages such as Python and Node.js initialize their interpreters in tens of milliseconds, and natively compiled languages such as Go and Rust incur near-zero runtime startup overhead at the cost of a more complex build pipeline. The dominance of framework import time in ML inference FaaS workloads—where loading libraries such as TensorFlow or PyTorch can constitute the majority of total cold start latency regardless of infrastructure initialization speed—has been specifically characterized for production ML serving function profiles [33].

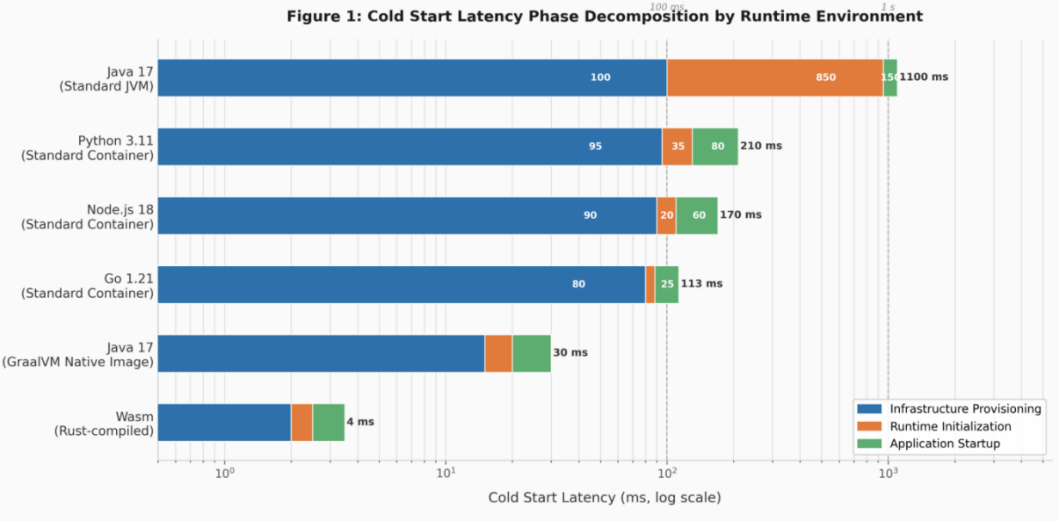

Figure 1. Cold Start Latency Phase Decomposition by Runtime Environment

At the application layer, function-specific initialization logic contributes latency overhead entirely independent of the platform or runtime environment. Functions that load large dependency trees, establish database connection pools, or retrieve ML model weights from remote object storage during initialization can exhibit application-layer startup times that exceed all other phases combined [34]. Developer-adopted patterns such as lazy module loading and connection reuse across warm invocations amortize application-layer startup costs but require manual implementation discipline and cannot be universally applied across all function types and dependency structures. Automated function partitioning techniques that restructure function initialization logic to separate lightweight startup paths from heavier deferred initialization have been demonstrated to reduce effective cold start latency without requiring developers to manually refactor their application code [35].

The security and isolation requirements of multi-tenant FaaS platforms impose binding constraints on how aggressively cold start latency can be reduced at the infrastructure layer, as recent work on responsible generative AI governance highlights the need for transparency, accountability, and robust data and system control mechanisms when deploying intelligent services in multi-tenant cloud environments [36]. Any optimization that relaxes the isolation boundary between co-located tenant workloads must be evaluated carefully against the security requirements of the deployment context, as the commercial viability of public multi-tenant FaaS platforms depends on preventing cross-tenant information leakage and resource interference. The tension between isolation strength and initialization speed is a defining characteristic of the cold start optimization

landscape: stronger isolation mechanisms generally impose higher initialization costs, and the strategies achieving the lowest initialization latency—most notably Wasm sandboxes—adopt isolation models with different security properties from hardware-level virtualization that may not be appropriate for all deployment contexts.

4. Pre-warming and Keep-alive Strategies

Pre-warming and keep-alive strategies are the most immediately deployable class of cold start optimizations because they operate within existing container and runtime infrastructure without requiring changes to isolation mechanisms, language runtimes, or function code. Keep-alive strategies extend the lifetime of warm function instances beyond the completion of their most recently served request, maintaining pools of initialized environments available to absorb future invocations without initialization overhead [37]. Pre-warming strategies take a more proactive stance, creating warm instances in advance of anticipated demand based on schedule knowledge, historical invocation patterns, or predictive models [38]. Together these strategies represent the primary cold start mitigation toolkit available to FaaS platform operators without requiring modifications to the underlying execution stack.

The simplest keep-alive implementation is a fixed-duration timeout that retains function instances for a platform-defined period—commonly between 5 and 30 minutes in commercial offerings—before reclaiming their resources. This approach is operationally simple but poorly suited to the diversity of invocation patterns observed in production workloads [39]. Functions whose typical inter-arrival times exceed the timeout remain persistently cold regardless of the timeout value, while functions invoked continuously derive no benefit from retention policies they never exercise. The aggregated inefficiency of applying a single timeout to a heterogeneous function fleet—in which the optimal keep-alive duration varies by orders of magnitude across functions—has motivated adaptive policies that condition retention decisions on per-function behavioral statistics rather than a global platform constant. A histogram-based adaptive policy that records the empirical inter-arrival time distribution for each function and dynamically sets the keep-alive window to cover a high percentile of that distribution was demonstrated to substantially reduce cold start rates for the same memory budget compared to fixed-timeout baselines, and this policy has been validated in production at Microsoft Azure Functions [40].

Table 1 below summarizes representative keep-alive and pre-warming strategies, comparing their decision signals, reported cold start reductions, and applicable workload classes. As shown in Table 1, the progression from fixed to adaptive to ML-driven strategies delivers increasingly consistent cold start reduction across heterogeneous workload mixes, at the cost of increasing policy complexity and computational overhead for the decision mechanism. The greedy-dual caching formulation of warm instance retention, which defines a per-function cost metric balancing memory retention cost against the expected invocation value of serving a future request from the warm instance, was demonstrated to achieve warm hit rate improvements exceeding 30 percent over fixed-timeout baselines across diverse production-representative workload traces [41].

Table 1. Comparison of Keep-alive and Pre-warming Strategies

Table 1: Comparison of Keep-alive and Pre-warming Strategies for Cold Start Latency Mitigation in FaaS Platforms

Strategy	Strategy Type	Key Decision Signal	Reported Cold Start Reduction vs. Baseline	Memory Overhead	Primary Workload Class
Fixed-Duration Timeout	Keep-alive (Reactive)	Global constant timeout value	Minimal; still cold for low-frequency functions	Low–Medium (proportional to pool)	High-frequency, regular invocations
Histogram-based Adaptive Keep-alive	Keep-alive (Adaptive)	Per-function inter-arrival time distribution	Substantial reduction; validated in Azure prod.	Moderate (per-function tuned)	Mixed heterogeneous invocation rates
FaasCache Greedy-Dual Retention	Keep-alive (Cost-aware)	Memory cost × invocation value	>30% warm hit rate improvement reported	Optimized (budget-constrained)	Memory-diverse function fleets
Schedule-aware Pre-warming	Pre-warming (Rule-based)	Application-level schedule knowledge	Near-100% elimination for periodic workloads	Low (targeted instances)	Cron-triggered, periodic workloads
Time-Series Predictive Pre-warming	Pre-warming (ML-based)	Historical invocation time-series forecast	Consistent outperformance of reactive baselines	Medium (predictive pool)	Bursty with temporal structure
Reinforcement Learning Lifecycle Management	Keep-alive + Pre-warming (RL)	Learned invocation pattern model	Outperforms rule-based on non-stationary loads	Medium–High (inference overhead)	Non-stationary, complex mixed loads

Predictive pre-warming applies time-series forecasting or ML models to function invocation histories to anticipate future demand and create warm instances before requests materialize. A reinforcement learning-based warm instance management approach that trains against simulated FaaS environments parameterized by real production workload traces was demonstrated to adapt to workload distribution shifts and outperform rule-based policies on non-stationary invocation patterns without requiring manual policy reconfiguration [42]. Time-series predictive pre-warming applied to function invocation histories has been shown to consistently outperform reactive initialization policies on workloads with exploitable temporal structure, and resource management frameworks embodying this approach demonstrate substantially reduced cold start rates for non-stationary workload patterns [43]. The key calibration challenge for predictive pre-warming is balancing prediction lead time against resource cost: launching instances too early relative to the actual invocation wastes memory and CPU on instances that expire before being used, while insufficient lead time fails to eliminate the cold start penalty for requests that arrive before initialization completes [44].

Intra-server function instance scheduling that coordinates the placement of warm instances from different function deployments on individual worker nodes to maximize warm reuse within local resource limits has been shown to reduce cold starts arising from suboptimal within-node instance placement patterns, complementing fleet-level warm pool management policies with node-level placement intelligence [45]. Pool-based pre-provisioning approaches that allocate warm instances across the function fleet proportionally to predicted invocation frequency demonstrate that even simple

frequency-weighted allocation consistently outperforms purely reactive initialization when workload statistics are reasonably stationary over the allocation planning horizon [46]. Memory overcommitment and idle page reclamation techniques that progressively recover physical memory from paused containers reduce the effective cost of maintaining warm instances, enabling larger effective warm pools within a fixed physical memory capacity at the cost of additional wake-up latency when reclaimed pages must be restored to physical memory before a new request can be served [47].

5. Snapshot-based and Checkpoint-Restore Approaches

Snapshot-based and checkpoint-restore approaches represent a fundamentally different cold start mitigation paradigm. Rather than preventing initialization by retaining live instances, these techniques capture the complete memory state of a fully initialized function execution environment as a persistent image, then restore that image upon subsequent cold invocations to bypass the full initialization sequence and reduce effective cold start latency to restoration time alone [48]. This approach decouples initialization cost from request response latency by performing initialization once per snapshot and amortizing it across many rapid restore operations. For heavyweight runtimes—particularly JVM-based languages whose startup regularly exceeds one second—snapshot-based approaches can reduce effective cold start latency by one to two orders of magnitude without requiring any changes to function code or language runtimes.

The Checkpoint/Restore In Userspace (CRIU) framework provides the foundational Linux mechanism for container-level snapshotting, enabling serialization of a running process's memory pages, file descriptors, network connection state, and kernel-maintained process metadata into a checkpoint archive that can be stored on disk and subsequently restored as a new process instance [49]. Applied to FaaS containers, CRIU-based checkpointing allows the platform to capture a fully initialized function container after runtime startup and application-level initialization but before the first request is served, then restore that checkpoint for subsequent cold invocations. Research on CRIU-based restoration for JVM-based FaaS deployments confirmed that checkpoint restoration reduces effective cold start latency from hundreds of milliseconds to tens of milliseconds for representative Java function workloads, with the reduction magnitude increasing with the complexity of JVM initialization captured in the checkpoint [50].

As illustrated in Figure 2, snapshot-based approaches deliver the most substantial cold start reduction for heavyweight JVM-based functions, while providing more modest but still meaningful improvements for lightweight runtime functions where infrastructure provisioning rather than runtime initialization dominates cold start time. Memory access patterns during snapshot restoration were identified as the critical performance bottleneck limiting achievable restore speed: when a snapshot is restored, subsequent memory accesses trigger page faults that must be serviced by loading pages from the snapshot archive, and if the post-restore access pattern is not aligned with archive page ordering, the resulting random-access fault pattern degrades restoration throughput significantly [18]. Memory layout optimization algorithms that reorder snapshot pages to match the empirically observed post-restore access sequence reduce page fault overhead and improve restore throughput by approximately 3× on representative serverless function benchmarks.

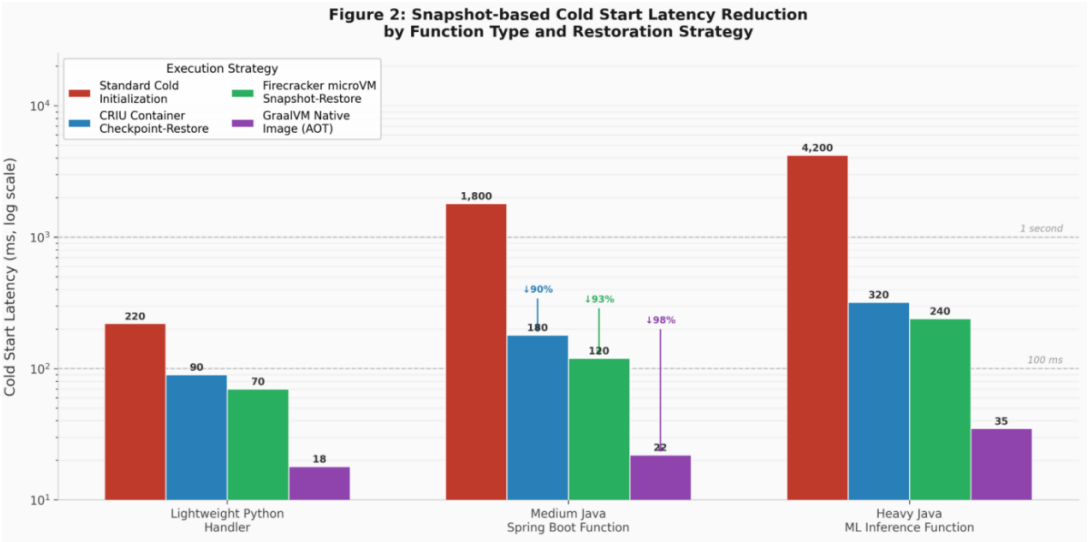

Figure 2. Snapshot-based Cold Start Latency Reduction by Function Type and Restoration Strategy

The Firecracker microVM snapshotting mechanism enables capture of entire microVM states—including virtualized CPU registers, guest memory, and emulated device configurations—as snapshot files that can be restored rapidly, providing hardware-level isolation in the restored instance without repeating the microVM boot sequence. GraalVM Native Image addresses the JVM cold start problem at the language level through ahead-of-time (AOT) compilation that produces standalone native executables requiring no JVM startup, class loading, or bytecode interpreter at runtime, achieving cold start latencies of 5 to 20 milliseconds for Java applications that would otherwise require hundreds of milliseconds to seconds of JVM initialization [51]. The primary trade-off for AOT-compiled executables is reduced peak throughput for compute-intensive workloads due to the absence of JIT-based runtime optimization; for FaaS functions that serve small numbers of requests before reclamation, this throughput trade-off is frequently acceptable given the order-of-magnitude cold start improvement.

A critical engineering challenge specific to snapshot-based approaches is snapshot staleness. Certain categories of

application state captured at checkpoint time become invalid after a dormancy period: database connections may be closed by remote servers, authentication tokens may expire, cached data may diverge from external system state, and network socket state recorded in the snapshot may no longer be valid in the current network environment [52]. Plug-in snapshotting frameworks that address staleness through configurable post-restore hook mechanisms enable functions to selectively re-execute targeted initialization steps whose state cannot be safely restored from a snapshot while recovering the bulk of initialization from the checkpoint, providing a practical trade-off between restoration speed and state correctness that makes snapshot-based cold start reduction applicable to a broader class of functions [53]. Co-locating snapshot archives with compute nodes using local high-speed storage and applying compression to snapshot images substantially reduces effective restoration latency by minimizing the storage I/O time on the critical restoration path [54].

6. Lightweight Runtime and Isolation Techniques

The isolation mechanism underlying a FaaS execution environment is a fundamental determinant of cold start latency because it establishes the minimum initialization overhead below which no amount of lifecycle management or pre-warming can reduce the cold start penalty for new instances. Standard Linux container isolation using namespaces and cgroup resource groups imposes initialization overhead inherent to namespace creation, network interface attachment, and cgroup configuration that is structurally irreducible within the standard container model [55]. A family of alternative isolation mechanisms has been developed that reduces this structural initialization overhead by one to three orders of magnitude while providing security properties ranging from comparable to substantially stronger than standard container isolation.

Wasm provides the most aggressive reduction in initialization latency among current isolation technologies. Wasm defines a portable, compact binary instruction format with formally specified deterministic execution semantics enabling sandboxed execution of arbitrary code within a host runtime process, and Wasm modules are instantiated by runtimes such as Wasmtime, Wasmer, and WasmEdge in microseconds, representing a reduction of three to four orders of magnitude compared to container or microVM initialization [56]. Wasm-based software fault isolation achieves microsecond-class instantiation latency for stateful FaaS workloads by enabling function instances to access shared memory regions through a software-enforced isolation boundary rather than hardware-level virtual machine boundaries, demonstrating that Wasm-based isolation can support the stateful execution patterns required by data-intensive FaaS applications without sacrificing initialization speed. The compactness of Wasm binaries—typically an order of magnitude smaller than equivalent container images—reduces both storage footprint and artifact retrieval latency on resource-constrained edge nodes, making Wasm particularly well-suited to edge FaaS contexts where storage and bandwidth constraints preclude large container image caches [57].

Firecracker achieves hardware-level virtualization isolation at sub-100-millisecond boot times by implementing a minimal device model exposing only the devices necessary for serverless function execution—a network interface, a block device, and a serial console—reducing both the attack surface and the initialization overhead compared to general-purpose hypervisors that emulate broad hardware ecosystems. A comparative evaluation of Firecracker and gVisor against standard Docker containers for serverless workloads found that Firecracker provides hardware-level virtualization isolation at initialization latencies competitive with container startup, while gVisor's user-space kernel interposition introduces per-system-call overhead that accumulates significantly for function workloads with high system call rates. Both represent meaningful improvements over standard Docker for cold-start-sensitive deployments, and the migration of AWS Lambda to Firecracker-based execution demonstrated the production readiness of microVM-based isolation at hyperscale.

A two-phase initialization architecture for Python FaaS functions that separates common interpreter and library initialization from function-specific module loading, maintaining a pool of pre-initialized Python interpreter instances with widely-used libraries already loaded and completing per-function initialization by cloning an available pre-initialized instance, was demonstrated to reduce cold start times for Python functions by over 18× compared to standard Docker-based deployments [58]. This approach exploits the observation that a large fraction of Python FaaS functions share common library dependencies, enabling common initialization work to be amortized across function instances rather than repeated independently for each cold start. Cross-instance code deduplication approaches that identify and eliminate redundant code loading across co-located container instances through sharing of common code pages reduce both initialization time and memory footprint simultaneously, addressing two distinct resource costs of container-based FaaS isolation through a unified mechanism [59].

Lean container implementations that strip unnecessary device support, unmounted network namespace variants, and irrelevant cgroup subsystems from the standard container initialization sequence reduce cold start latency while preserving the namespace-based isolation model familiar to existing FaaS platform operators [60]. AOT compilation of Java applications through GraalVM Native Image produces self-contained native executables that require no JVM at runtime, achieving startup times of 5 to 20 milliseconds for representative Java functions that otherwise require hundreds of milliseconds of JVM initialization, with the principal trade-off being reduced peak throughput for compute-intensive workloads due to the absence of JIT optimization. The edge deployment context has been shown to demand purpose-built lightweight isolation approaches, as techniques designed for hyperscale cloud environments require adaptation to heterogeneous edge hardware and the constraints of resource-limited nodes that cannot support the isolation mechanism diversity available on cloud worker nodes [61].

7. Scheduling and Resource Management

Cold start optimization at the platform level extends beyond individual function instance lifecycle management to encompass the scheduling and resource management decisions that govern how invocations are assigned to worker nodes, how warm instances are distributed across the cluster, and how memory resources are allocated among competing

function deployments. Even with well-designed per-function keep-alive policies, poor scheduling that routes invocations to worker nodes lacking warm instances of the requested function unnecessarily triggers cold starts by bypassing available warm instances on other nodes [62]. The interplay between scheduling policy, warm pool management, and resource allocation jointly determines the aggregate cold start rate experienced by platform users at fleet scale.

Locality-aware scheduling that prioritizes routing function invocations to worker nodes already hosting warm instances of the requested function exploits the spatial locality of warm instance deployment to serve invocations without initialization overhead, and has been demonstrated to substantially reduce aggregate cold start rates compared to policies optimizing exclusively for load balance. The tension between warm-instance-first routing and load balancing can be managed through hybrid policies that apply warm-instance routing within a configurable load imbalance threshold and fall back to load-balanced routing when candidate warm-instance nodes are too heavily loaded to accept additional requests without service degradation. Quality-of-service-aware scheduling that explicitly models the latency-throughput trade-off and provides differentiated cold start rate targets for different function classes enables platform operators to concentrate warm pool resources on functions where cold start mitigation delivers the greatest value, achieving target cold start rates across diverse workload mixes with lower total resource overhead than uniform cold start minimization policies.

Bin-packing and consolidation strategies that co-locate function instances with complementary memory footprints and temporal activity patterns maximize warm instance density within a given memory budget, enabling a larger effective warm pool without increasing total hardware capacity [63]. Workflow orchestration that coordinates warm instance provisioning across multi-function pipelines by pre-warming downstream function instances before upstream functions complete prevents the cascade of serial cold starts that amplifies end-to-end pipeline latency to multiples of the per-function cold start penalty. Analytical performance models for SC platforms that characterize the relationship between warm pool size, function invocation rate, and cold start probability under various arrival process assumptions enable warm pool resources to be dimensioned analytically to meet SLO-specified cold start rate targets rather than through empirical trial-and-error capacity planning [64].

Storage performance on the critical artifact retrieval path has been identified as a frequently overlooked bottleneck in FaaS cold start optimization, as research attention has historically focused on compute and runtime dimensions while storage access patterns—which can dominate cold start duration for functions with large dependency trees or model weight artifacts—have received less systematic treatment. Local artifact caching strategies that pre-stage function code and dependency artifacts on worker nodes likely to receive future invocations decouple artifact retrieval latency from cold start latency by overlapping retrieval with the interval preceding request arrival, and content-addressed deduplication across shared dependency artifacts reduces the storage footprint of artifact caches and improves local cache hit rates across the function fleet. Network-aware management systems for edge FaaS deployments that incorporate network topology into scheduling decisions have been demonstrated to substantially reduce end-to-end invocation latency for edge applications compared to topology-agnostic scheduling, as network transfer time from edge nodes to function execution sites is non-negligible at edge network speeds [65].

Dynamic warm pool sizing policies that adjust the number of retained instances in response to observed invocation rates and platform load target a configurable warm hit rate rather than a fixed pool size, navigating the trade-off between cold start minimization and idle resource consumption more effectively than static pool size policies under variable workload conditions. The deployment of FaaS platforms across geographically distributed edge nodes introduces additional scheduling complexity absent in centralized cloud environments, where resource constraints limit warm pool sizes and function deployment diversity that can be simultaneously supported on individual edge nodes. A real-time data analytics SC architecture designed specifically for edge deployments demonstrated that edge-specific scheduling policies that account for the interaction between warm pool limitations and strict latency requirements of time-critical IoT analytics workloads substantially outperform general-purpose policies adapted from cloud FaaS platforms.

8. Conclusion

Cold start latency in FaaS platforms is a multi-layered challenge requiring coordinated optimization across the execution stack, from the isolation mechanism and language runtime through application design patterns to warm instance lifecycle management and cluster scheduling. This review has examined the four principal strategy families developed to address this challenge, characterizing their mechanisms, demonstrated effectiveness, inherent trade-offs, and the workload contexts in which each delivers the greatest benefit.

Pre-warming and keep-alive strategies are the most immediately deployable class of optimizations, requiring no changes to isolation mechanisms or language runtimes. The evolution from fixed-duration to adaptive histogram-based to ML-driven predictive policies has delivered meaningful cold start rate reductions in both research settings and production deployments at major commercial FaaS providers, establishing that per-function data-driven lifecycle management is both practical and impactful at cloud scale. Snapshot-based checkpoint-restore techniques offer the largest absolute reductions in effective cold start latency for heavyweight runtimes, reducing JVM-based cold start from seconds to tens of milliseconds by amortizing one-time initialization across many rapid restore operations. Snapshot staleness management and network state coordination remain the primary engineering challenges limiting universal applicability of snapshot-based approaches, and hybrid post-restore re-initialization mechanisms represent the most practical path to broad deployment.

Lightweight isolation technologies including Wasm sandboxes, Firecracker microVMs, and lean containers address cold start latency at the infrastructure root cause by reducing the structural initialization overhead of the execution boundary to microseconds or low milliseconds. These technologies are increasingly adopted in production FaaS platforms and are particularly compelling for edge deployments where warm pool sizes are constrained by hardware limitations that preclude aggressive warm instance retention policies. Scheduling and resource management strategies complete the optimization landscape by ensuring warm instances are distributed intelligently, invocations are

routed toward nodes with warm capacity wherever possible, and pool sizing decisions account for per-function invocation value rather than applying uniform global policies that misallocate retention resources across heterogeneous function fleets.

Several important research challenges remain open. The fundamental tension between isolation strength and initialization speed has not been fully resolved at the systems level, and achieving hardware-level multi-tenant isolation at sub-millisecond timescales remains a frontier systems problem without a generally accepted solution. Predictive and adaptive ML-based lifecycle management shows consistent promise but must handle the high burstiness and non-stationarity of production serverless workloads while operating within tight inference latency budgets imposed by real-time scheduling requirements. The interactions among simultaneously deployed optimization strategies—where aggressive snapshot-based mitigation, adaptive keep-alive, and locality-aware scheduling all operate concurrently on the same function fleet—have received limited systematic treatment and may exhibit non-obvious interactions warranting dedicated investigation. Finally, the growing diversity of FaaS deployment environments from hyperscale cloud platforms to resource-constrained edge nodes demands optimization strategies that generalize across this heterogeneity, and edge-specific cold start optimization represents a particularly fertile direction for future research given the strict latency requirements and severe resource constraints characteristic of edge application contexts.

References

[1] Leitner, P., Wittern, E., Spillner, J., & Hummer, W. (2019). A mixed-method empirical study of function-as-a-service software development in industrial practice. Journal of Systems and Software, 149, 340-359.

[2] Eismann, S., Scheuner, J., Van Eyk, E., Schwinger, M., Grohmann, J., Herbst, N., ... & Iosup, A. (2021). The state of serverless applications: Collection, characterization, and community consensus. IEEE Transactions on Software Engineering, 48(10), 4152-4166.

[3] Cordingly, R., Yu, H., Hoang, V., Perez, D., Foster, D., Sadeghi, Z., ... & Lloyd, W. J. (2020, August). Implications of programming language selection for serverless data processing pipelines. In 2020 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech) (pp. 704-711). IEEE.

[4] Jonas, E., Schleier-Smith, J., Sreekanti, V., Tsai, C. C., Khandelwal, A., Pu, Q., ... & Patterson, D. A. (2019). Cloud programming simplified: A berkeley view on serverless computing. arXiv preprint arXiv:1902.03383.

[5] Mohan, A., Sane, H., Doshi, K., Edupuganti, S., Nayak, N., & Sukhomlinov, V. (2019). Agile cold starts for scalable serverless. In 11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19).

[6] Varshney, R. P., & Sharma, D. K. (2020, October). Cold start in function as a service: a systematic study, analysis and evaluation. In International Conference on Futuristic Trends in Networks and Computing Technologies (pp. 337-349). Singapore: Springer Singapore.

[7] Shahrad, M., Balkind, J., & Wentzlaff, D. (2019, October). Architectural implications of function-as-a-service computing. In Proceedings of the 52nd annual IEEE/ACM international symposium on microarchitecture (pp. 1063-1075).

[8] Wen, J., Chen, Z., Jin, X., & Liu, X. (2023). Rise of the planet of serverless computing: A systematic review. ACM Transactions on Software Engineering and Methodology, 32(5), 1-61.

[9] Lin, P. M., & Glikson, A. (2019). Mitigating cold starts in serverless platforms: A pool-based approach. arXiv preprint arXiv:1903.12221.

[10] Tariq, A., Pahl, A., Nimmagadda, S., Rozner, E., & Lanka, S. (2020, October). Sequoia: Enabling quality-of-service in serverless computing. In Proceedings of the 11th ACM symposium on cloud computing (pp. 311-327).

[11] Mendki, P. (2020, October). Evaluating webassembly enabled serverless approach for edge computing. In 2020 IEEE Cloud Summit (pp. 161-166). IEEE.

[12] Ding, G., Yang, S., Lin, H., Chen, Z., & Yang, J. S. (2026). LLM-Driven Adaptive Cloud Resource Scheduling: Bridging Reasoning Intelligence with Optimization Guarantees. IEEE Open Journal of the Computer Society.

[13] Eismann, S., Scheuner, J., Van Eyk, E., Schwinger, M., Grohmann, J., Herbst, N., ... & Iosup, A. (2020). A review of serverless use cases and their characteristics. arXiv preprint arXiv:2008.11110.

[14] Li, Y., Lin, Y., Wang, Y., Ye, K., & Xu, C. (2022). Serverless computing: state-of-the-art, challenges and opportunities. IEEE Transactions on Services Computing, 16(2), 1522-1539.

[15] Shahrad, M., Fonseca, R., Goiri, I., Chaudhry, G., Batum, P., Cooke, J., ... & Bianchini, R. (2020). Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In 2020 USENIX annual technical conference (USENIX ATC 20) (pp. 205-218).

[16] Kulkarni, V., Reddy, N., Khare, T., Mohan, H., Murali, J., Balajee, S., ... & Simmhan, Y. (2024, May). XFBench: a cross-cloud benchmark suite for evaluating faas workflow platforms. In 2024 IEEE 24th International Symposium on Cluster, Cloud and Internet Computing (CCGrid) (pp. 543-556). IEEE.

[17] Agache, A., Brooker, M., Iordache, A., Liguori, A., Neugebauer, R., Piwonka, P., & Popa, D. M. (2020). Firecracker: Lightweight virtualization for serverless applications. In 17th USENIX symposium on networked systems design and implementation (NSDI 20) (pp. 419-434).

[18] Zhao, T., Hall, M., Johansen, H., & Williams, S. (2021, February). Improving communication by optimizing on-node data movement with data layout. In Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (pp. 304-317).

[19] Fuerst, A., & Sharma, P. (2021, April). Faascache: keeping serverless computing alive with greedy-dual caching. In Proceedings of the 26th ACM international conference on architectural support for programming languages and operating systems (pp. 386-400).

[20] Shillaker, S., & Pietzuch, P. (2020). Faasm: Lightweight isolation for efficient stateful serverless computing. In 2020 USENIX Annual Technical Conference (USENIX ATC 20) (pp. 419-433).

[21] Sreekanti, V., Wu, C., Lin, X. C., Schleier-Smith, J., Faleiro, J. M., Gonzalez, J. E., ... & Tumanov, A. (2020). Cloudburst: Stateful functions-as-a-service. arXiv preprint arXiv:2001.04592.

[22] Bermbach, D., Karakaya, A. S., & Buchholz, S. (2020, March). Using application knowledge to reduce cold starts in FaaS services. In Proceedings of the 35th annual ACM symposium on applied computing (pp. 134-143).

[23] Sui, Y., Yu, H., Hu, Y., Li, J., & Wang, H. (2024, November). Pre-warming is not enough: Accelerating serverless inference

with opportunistic pre-loading. In Proceedings of the 2024 ACM Symposium on Cloud Computing (pp. 178-195).

[24] Aslanpour, M. S., Toosi, A. N., Cicconetti, C., Javadi, B., Sbarski, P., Taibi, D., ... & Dustdar, S. (2021, February). Serverless edge computing: vision and challenges. In Proceedings of the 2021 Australasian computer science week multiconference (pp. 1-10).

[25] Pfandzelter, T., & Bermbach, D. (2020, April). tinyfaas: A lightweight faas platform for edge environments. In 2020 IEEE International Conference on Fog Computing (ICFC) (pp. 17-24). IEEE.

[26] Golec, M., Walia, G. K., Kumar, M., Cuadrado, F., Gill, S. S., & Uhlig, S. (2024). Cold start latency in serverless computing: A systematic review, taxonomy, and future directions. ACM Computing Surveys, 57(3), 1-36.

[27] Daw, N., Bellur, U., & Kulkarni, P. (2021, November). Speedo: Fast dispatch and orchestration of serverless workflows. In Proceedings of the ACM Symposium on Cloud Computing (pp. 585-599).

[28] Anjali, Caraza-Harter, T., & Swift, M. M. (2020, March). Blending containers and virtual machines: a study of firecracker and gVisor. In Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (pp. 101-113).

[29] Das, A., Imai, S., Patterson, S., & Wittie, M. P. (2020, May). Performance optimization for edge-cloud serverless platforms via dynamic task placement. In 2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID) (pp. 41-50). IEEE.

[30] Li, H., Yuan, Y., Du, R., Ma, K., Liu, L., & Hsu, W. (2020). {DADI}:{Block-Level} Image Service for Agile and Elastic Application Deployment. In 2020 USENIX Annual Technical Conference (USENIX ATC 20) (pp. 727-740).

[31] Vahidinia, P., Farahani, B., & Aliee, F. S. (2020, August). Cold start in serverless computing: Current trends and mitigation strategies. In 2020 International Conference on Omni-layer Intelligent Systems (COINS) (pp. 1-7). IEEE.

[32] Ménétrey, J., Pasin, M., Felber, P., & Schiavoni, V. (2022, July). Webassembly as a common layer for the cloud-edge continuum. In Proceedings of the 2nd Workshop on Flexible Resource and Application Management on the Edge (pp. 3-8).

[33] Barrak, A., Petrillo, F., & Jaafar, F. (2022). Serverless on machine learning: A systematic mapping study. IEEE Access, 10, 99337-99352.

[34] Toutain, L., Elloumi, O., Liang, S., van der Schaaf, H., De Lathouwer, B., Minaburo, A., & Raggett, D. (2023). Application Layer Protocols. In Springer Handbook of Internet of Things (pp. 481-507). Cham: Springer International Publishing.

[35] Boza, E. F., Andrade, X., Cedeno, J., Murillo, J., Aragon, H., Abad, C. L., & Abad, A. G. (2020). On implementing autonomic systems with a serverless computing approach: The case of self-partitioning cloud caches. Computers, 9(1), 14.

[36] Chen, Z., Wang, Y., & Zhao, X. (2025). Responsible Generative AI: Governance Challenges and Solutions in Enterprise Data Clouds. Journal of Computing and Electronic Information Management, 18(3), 59-65.

[37] Rausch, T., Hummer, W., Muthusamy, V., Rashed, A., & Dustdar, S. (2019). Towards a serverless platform for edge {AI}. In 2nd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 19).

[38] Mampage, A., Karunasekera, S., & Buyya, R. (2022). A holistic view on resource management in serverless computing environments: Taxonomy and future directions. ACM Computing Surveys (CSUR), 54(11s), 1-36.

[39] Eismann, S., Grohmann, J., Van Eyk, E., Herbst, N., & Kounev, S. (2020, April). Predicting the costs of serverless workflows. In Proceedings of the ACM/SPEC international conference on performance engineering (pp. 265-276).

[40] Besozzi, V., Della Bartola, M., Dazzi, P., & Danelutto, M. (2025). High-Performance Serverless Computing: A Systematic Literature Review on Serverless for HPC, AI, and Big Data. IEEE Access, 13, 195611-195656.

[41] Mahmoudi, N., & Khazaei, H. (2020). Performance modeling of serverless computing platforms. IEEE Transactions on Cloud Computing, 10(4), 2834-2847.

[42] Vahidinia, P., Farahani, B., & Aliee, F. S. (2022). Mitigating cold start problem in serverless computing: A reinforcement learning approach. IEEE Internet of Things Journal, 10(5), 3917-3927.

[43] Gunasekaran, J. R., Thinakaran, P., Nachiappan, N. C., Kandemir, M. T., & Das, C. R. (2020, December). Fifer: Tackling resource underutilization in the serverless era. In Proceedings of the 21st International Middleware Conference (pp. 280-295).

[44] Ghorbian, M., & Ghobaei-Arani, M. (2024). A survey on the cold start latency approaches in serverless computing: an optimization-based perspective. Computing, 106(11), 3755-3809.

[45] Schleier-Smith, J. M. (2022). Understanding and Exploring Serverless Cloud Computing. University of California, Berkeley.

[46] Gias, A. U., Casale, G., & Woodside, M. (2019, July). ATOM: Model-driven autoscaling for microservices. In 2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS) (pp. 1994-2004). IEEE.

[47] Wang, R., Yu, G., Casale, G., Chen, P., & Filieri, A. (2026). Fine-grained Tracing for Performance Anomaly Diagnosis of Serverless Functions. ACM Transactions on Autonomous and Adaptive Systems.

[48] Cadden, J., Unger, T., Awad, Y., Dong, H., Krieger, O., & Appavoo, J. (2020, April). SEUSS: skip redundant paths to make serverless fast. In Proceedings of the Fifteenth European Conference on Computer Systems (pp. 1-15).

[49] Chen, Y., Liu, B., Lin, W., Guo, Y., & Peng, Z. (2025). CASR: Optimizing cold start and resources utilization in serverless computing. Future Generation Computer Systems, 170, 107851.

[50] Holmes, B., Dinis, B., Honcharuk, L., Fried, J., & Belay, A. (2025). Taming Serverless Cold Starts Through OS Co-Design. arXiv preprint arXiv:2509.14292.

[51] Wang, S. (2021). Thin serverless functions with graalvm native image.

[52] Agarwal, S., Rodriguez, M. A., & Buyya, R. (2025). Learning in Serverless Computing. Computational Intelligence and Data Analytics: Proceedings of ICCIDA 2024, 1.

[53] Kim, S. J., You, M., Kim, B. J., & Shin, S. (2023, October). Cryonics: Trustworthy Function-as-a-Service using Snapshot-based Enclaves. In Proceedings of the 2023 ACM Symposium on Cloud Computing (pp. 528-543).

[54] Ustiugov, D., Petrov, P., Kogias, M., Bugnion, E., & Grot, B. (2021, April). Benchmarking, analysis, and optimization of serverless function snapshots. In Proceedings of the 26th ACM international conference on architectural support for programming languages and operating systems (pp. 559-572).

[55] Jain, S. M. (2020). Linux Containers and Virtualization. A Kernel Perspective, 2020-10.

[56] Gackstatter, P., Frangoudis, P. A., & Dustdar, S. (2022, May). Pushing serverless to the edge with webassembly runtimes. In

2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid) (pp. 140-149). IEEE.

[57] Ramachandran, U., Gupta, H., Hall, A., Saurez, E., & Xu, Z. (2019, July). Elevating the edge to be a peer of the cloud. In 2019 IEEE 12th International Conference on Cloud Computing (CLOUD) (pp. 17-24). IEEE.

[58] Panda, A., & Sarangi, S. R. (2024). Faasctrl: A comprehensive-latency controller for serverless platforms. IEEE Transactions on Cloud Computing, 12(4), 1328-1343.

[59] Shi, J., Gu, J., Xia, Y., & Chen, H. (2025). Serverless functions made confidential and efficient with split containers. In 34th USENIX Security Symposium (USENIX Security 25) (pp. 1091-1110).

[60] Verma, P., Goel, P., & Rani, N. (2024, April). A review: Cold start latency in serverless computing. In 2024 Sixth International Conference on Computational Intelligence and Communication Technologies (CCICT) (pp. 141-148). IEEE.

[61] Liu, X., Wen, J., Chen, Z., Li, D., Chen, J., Liu, Y., ... & Jin, X. (2023). Faaslight: General application-level cold-start latency optimization for function-as-a-service in serverless computing. ACM Transactions on Software Engineering and Methodology, 32(5), 1-29.

[62] Zhang, Y., & Jacobsen, H. A. (2025, December). Mocha: Scalable and Compliant Function Scheduling for Federated Serverless Computing. In Proceedings of the 26th International Middleware Conference (pp. 398-412).

[63] Kaffes, K., Yadwadkar, N. J., & Kozyrakis, C. (2019, November). Centralized core-granular scheduling for serverless functions. In Proceedings of the ACM symposium on cloud computing (pp. 158-164).

[64] Brown, B. (2025). Serverless Architectures for Scalable Cloud-Based Microservices: Performance and Cost Trade-Offs.

[65] Baresi, L., Hu, D. Y. X., Quattrocchi, G., & Terracciano, L. (2022, May). NEPTUNE: Network-and GPU-aware management of serverless functions at the edge. In Proceedings of the 17th Symposium on Software Engineering for Adaptive and Self-Managing Systems (pp. 144-155).