International Journal of Innovative Research in Computer Science and Technology (IJIRCST)
ISSN (Online): 2347-5552, Volume-14, Issue-6, Nov-Dec 2026
DOI: https://doi.org/10.55524/ijircst.2026.14.6.6
Article ID IRP-1800, Pages 48-57
www.ijircst.org

A Critical Survey of Serverless Computing Architectures and Optimization Tradeoffs

Rahul Srivastava[1], and Ranjeet Kumar Dubey[2]

[1, 2] Department of Computer Science and Engineering, KIPM College of Engineering, Gorakhpur, Uttar Pradesh, India

Correspondence should be addressed to Rahul Srivastava; rahulsrivastav1993@gmail.com

Received: 5 Oct 2026; **1st Revision:** 19 Oct 2026; **2nd Revision:** 28 Oct 2026; **3rd Revision:** 5 Nov 2026;
Accepted: 12 Nov 2026; **Published Online:** 23 Nov 2026

Copyright © 2026 Rahul Srivastava et. al. This is an open-access article distributed under the Creative Commons Attribution 4.0 International License (CC BY 4.0), which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

ABSTRACT- In serverless computing the provisioning and managing of servers and runtimes is handled by a cloud platform. The real advantages of serverless computing are dependent on the workload characteristics, platform controls and deployment choices. In this survey, we review 18 research publications published from 2017 to 2026, one cloud-computing standard, and six technical or specification sources. This review is a focused narrative review, based on a bibliography-led starting set and chosen later studies, and is not a comprehensive systematic review. We compare eleven empirical studies on warm-instance retention, snapshot restoration, workflow preparation and fusion, autoscaling, container placement, production runtimes, and large-language-model serving. The platform, workload, baseline, measurement boundary, reported result and limitations are provided for each comparison in each study. The synthesis shows that reported gains are conditional: warm capacity may reduce startup delay at the cost of consuming reserved resources; snapshot methods depend on storage behavior and safe state reuse; workflow fusion may reduce coordination at the cost of parallelism; and learned scaling policies may not transfer to workloads unlike those used in training. Production-runtime and accelerator-serving studies indicate that control-path coordination, contention, and model loading can still be major sources of delay. The survey provides an evidence-based comparison linking optimization goals to their experimental settings and shifted costs, rather than ordering non-comparable speedup numbers. It defines evaluation priorities such as end-to-end latency, resource consumption, total cost, correctness, portability, security, and energy.

KEYWORDS— Autoscaling, Cold starts, Function-as-a-Service (FaaS), Performance optimization, Serverless computing.

I. INTRODUCTION

In serverless computing, a platform takes care of the server provisioning, runtime maintenance, and much of the effort of managing available capacity. This allows developers to spend more time on the application logic, but they are still responsible for the behavior of the application. Two services commonly used in serverless applications. Function-as-a-Service (FaaS) executes application code in response to an event. Backend-as-a-Service (BaaS) provides managed services such as storage, messaging, and authentication. These models are complementary in the larger serverless ecosystem [1], [2]. The platform takes care of some operational tasks, but servers are still there and applications still need to be designed to handle failure. Serverless is a good fit for some workloads, but not for others. Elastic execution and reduced idle capacity may benefit event-driven tasks that run only occasionally. The picture can be different for applications that have continuous incoming requests, use lots of memory, load big dependencies, or access remote data over and over. These requirements impact response time and cost. The deployment of applications has shown the different ways serverless can be used [3]. Hence, architectural suitability, resource efficiency and the experienced performance by users should be evaluated as separate questions.

A. Contributions and Scope

We are interested in how serverless architecture relates to the evidence used to judge optimization techniques. We first describe the execution path and the boundaries around the application state. We then look at representative methods and the conditions under which they were tested, and ask where the reported benefits apply and where important questions remain. Security, economic cost, portability and sustainability are taken into consideration throughout as they can limit what an otherwise promising optimization can achieve. We do not try to compare empirically developer productivity or platform market share.

In this survey, we contribute an evidence-based synthesis across multiple layers of a serverless system. As opposed to reviews that are often structured by application domains, broad challenges, security, or cold-start taxonomies, we compare selected optimization experiments by (i) the execution stage they target, (ii) the platform and workload, (iii) the baseline and measurement boundary, (iv) the reported outcome, and (v) the costs or correctness constraints that might be shifted to another layer. Table 1 positions this contribution against selected previous reviews; Table 2 consolidates the experimental evidence; and Table 3 translates recurring limitations into research questions and evaluation requirements. The unique contribution is not a novel optimization algorithm or

benchmark, but a cross-study interpretation of when reported improvements may be transferable and what must be measured prior to deployment. The survey does not calculate pooled effect sizes, does not claim to be exhaustive, and does not identify a single technique that is best for all.

B. Review Methodology

The review was started with the 18 references that were retained in the previous version of this manuscript. This starting set was bibliography-driven, not exported from a database search. One report on serverless applications reported the same material as a later journal article; we kept the journal article [3], leaving 17 records from the original set. The final bibliography consists of 25 sources: 18 research publications [1]–[6], [11]–[17], [20], [22]–[25], one cloud-computing standard [9], and six technical or specification sources [7], [8], [10], [18], [19], [21]. Table 2 provides a summary of the empirical evidence for optimization from eleven research articles. The other sources support the architectural discussion, the comparison with previous reviews, the context of application and the descriptions of the platform. These numbers represent the material retained for this review. They should not be taken to be the result of an exhaustive database-screening exercise.

Search and source checks were completed 28 September 2026; research coverage was from 2017 to that date. We identified relevant records by searching ACM Digital Library, IEEE Xplore, ScienceDirect, USENIX, arXiv, and author or institutional repositories on the web. The topic searches captured for the review were "serverless cold start optimization", "serverless autoscaling reinforcement learning", and "serverless security survey". We also searched for single titles and for DOI “10.1145/3510611”. No complete logging of every query, search-result counts and native database exports. Therefore, we do not describe the search as comprehensive or fully reproducible. In parallel, the official platform documentation and specifications were reviewed.

We included studies in English that explicitly addressed serverless architecture or operation, and provided a significant review, workload characterization, or evaluated approach. First, titles and abstracts were screened. Subsequently, accessible methods, experimental conditions and limitations of the papers considered relevant were examined. We filtered out general cloud conversations that were not focused on serverless, promotional accounts, duplicate findings, and any numerical claims that we could not verify. The only resolved overlap in the retained bibliography was the report merged with the journal publication [3]. We cannot provide a reliable number of all search hits rejected prior to final selection due to incomplete earlier screening records. If the full text of the publisher was not available, we consulted the author or repository version.

For each empirical paper we recorded the intervention, execution layer, platform and version if reported, workload, baseline, metrics, reported result and stated or apparent limitations. We considered a result numerical only if the metric and comparator (or absolute measurement boundary) could be determined from the cited study. Otherwise, we summarized the reported direction qualitatively. We did not assume unobserved experimental settings. The screening was done in stages: first, title and abstract relevance was checked, then accessible methods, experimental conditions and limitations were examined. As the original bibliography and search logs were not complete, it is not possible to reconstruct any reliable hit count, duplicate count (beyond the known overlap), or study-quality score. The selection is therefore purposeful and descriptive and not a systematic review according to PRISMA guidelines. The manuscript discloses known sources, search phrases, date, eligibility criteria and extraction fields to improve transparency. A future update should include database exports, complete query strings, screening decisions and a versioned extraction sheet.

C. Position Relative to Existing Surveys

The wider field has already been surveyed at some length. For example, Shafiei et al. [4] discuss the opportunities, application domains, challenges, proposed solutions and open problems in serverless. Their review also discusses performance and resource optimization, so revisiting those topics alone would not be a unique contribution. The present survey takes a narrower evidence-oriented approach, comparing selected studies based on their targeted execution stage and platforms, workloads, baselines, metrics and limitations. It also includes selected later work on heterogeneous container placement [11], production runtime behavior [5] and accelerator-based model serving [6]. The goal is to clarify the circumstances under which the reported results may be relevant, not to provide a full overview of all serverless research.

Table 1: Comparison with Existing Surveys and Related Reviews

Study	Scope	Research literature covered	How optimization is discussed	Type of analysis	Open questions and this survey’s contribution
Shafiei et al. [4], 2022	Opportunities, applications, and broad challenges	Studies arranged across eight application domains and nine challenge areas	Reviews proposed solutions, including performance and resource management	Discusses application feasibility alongside challenges and solution limits	Outlines broad open problems. Our survey adds selected newer systems studies and compares their test conditions explicitly.
Li et al. [12], 2023	Threats to security and possible defenses	Research proposals and security features of industrial platforms	Considers security mechanisms and operational consequences	Examines gaps between research proposals and deployed capabilities	A security review does not measure the latency or cost of optimization. Here, isolation is considered a constraint on evaluation.

Golec et al. [13], 2025	The causes and mitigation of cold-start delay	A dedicated systematic review, including its accepted 2024 version	Groups caching, application-level, and artificial intelligence (AI) or machine learning (ML) methods	Compares mitigation categories, data sets, platforms, and unresolved issues	Our comparison considers how startup methods relate to workflows, placement, and full-request latency.
Eismann et al. [3], 2021	Why applications adopt serverless, when they use it, and how	89 applications drawn from open-source, industry, academia, and science	Uses application properties to inform suitability	Characterizes real applications rather than using one common optimization benchmark	We use these varied workloads to explain why a result may not carry over to another application.
Present survey	Architectural limitations and optimization trade-offs	18 research publications from 2017–2026, including 11 empirical studies in Table 2	Warm-instance retention, restoration, fusion, scaling, placement, production runtimes, and model loading	Records environments, baselines, metrics, evidence types, and limitations	Focuses on repeatability, interactions across layers, controller transfer, state correctness, and measured energy. Selective coverage limits wider claims.

Table 1 shows that the earlier works answer different questions: Shafiei et al. [4] present a broad survey of opportunities and challenges, Li et al. [12] focus on security, Golec et al. [13] systematize cold-start mitigation, and Eismann et al. [3] characterize application adoption and workload patterns. The contributions are not replaced by this survey. Its added value is the side-by-side examination of the selected optimization experiments and the conditions behind their reported results. In particular, the comparison makes explicit that a result measured at the sandbox, worker, workflow or model-serving boundary answers a different performance question. The somewhat selective evidence base also limits the range of claims that can be made.

II. SERVERLESS COMPUTING ARCHITECTURE

Serverless applications frequently use a mix of FaaS and BaaS. FaaS runs code in response to events; BaaS provides managed capabilities through application programming interfaces (APIs). An individual function can be deployed and scale independently, but does not necessarily mean it's a whole microservice. Several functions may be interdependent and depend on persistent back-end services [1, 2]. ISO/IEC 22123-2:2023 describes general concepts in cloud computing [9], but does not define serverless performance or guaranty identical application behavior across providers.

Figure 1 shows the flow of an invocation from the event arrival, through routing and queueing, environment setup, function execution, and access to outside services. Requests can be HTTP, message queues, or timers. If there is a suitable existing runtime, the function can reuse it, otherwise it has to provision, initialize or restore before the code can run. One instance can handle multiple requests but a sudden spike in traffic can still result in cold starts while other instances are still warm. The lifetime of an environment is determined by the platform policy rather than any application-level guaranty of persistence [14], [15], [16], [17].

If an application needs durable data, it must store it explicitly, for example in a database or object store. A later invocation may happen to find information in local memory or in temporary files, but correctness cannot depend on that information still being there. Event sources differ also in how they deliver and retry.

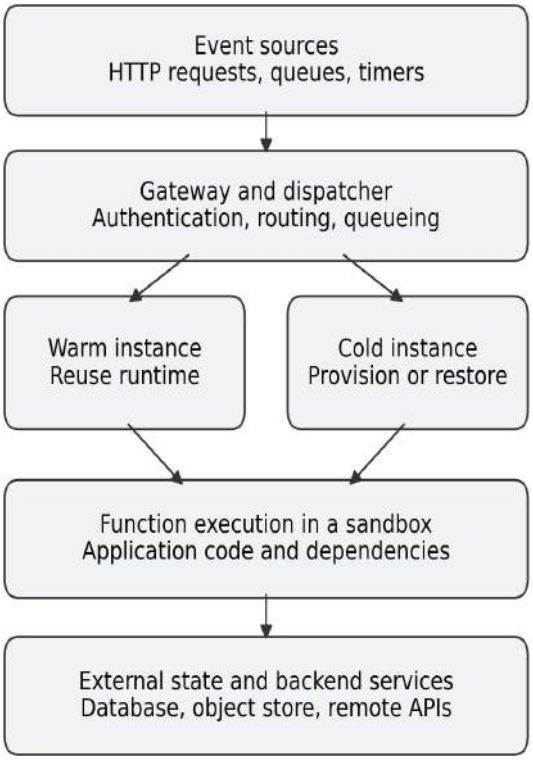

Figure 1: Invocation path and external state boundary. Original schematic synthesized from [1], [15]–[17].

Splitting functions can reduce the impact of some failures, yet the workflow can still be interrupted by downstream service outages, retries, and timeouts. To avoid incorrect side effects from those failures, we need appropriate retry rules and idempotent operations.

Platforms can isolate execution using containers, micro virtual machines (microVMs) or other sandboxes . The design should decouple tenants, host and application dependencies. That’s what matters. Faster restore only helps if it also maintains credentials, connections, state and isolation [12], [17]. These responsibilities differ in a self-managed system: the operator still operates the cluster and runtime, while developers have a serverless interface [8], [18].

III. CONDITIONAL BENEFITS AND DEPLOYMENT COSTS

A. Cost and Workload Suitability

Paying for function execution rather than a running application instance is appealing when requests are sporadic and the tasks fit the service's execution model. This advantage can be diminished when the request volume, execution time, memory requirements or reserved warm capacity increase. If a function is busy almost all the time, there is little chance to benefit from scaling to zero. The costs should be equitable and should be based on the same workload and reliability objectives as tested against a provisioned virtual machine or container service, including the operational effort of each alternative [1], [2]. Full application bill is more than function calls and execution time. This could include reserved capacity, workflow orchestration, database and storage access, monitoring, logs, traces, and data transfer. Retries and cold-start mitigation can increase both the work done and the waiting time. In a multi-stage workflow, the cost for external services can even exceed the function charge. These are cost categories to be included in an analysis, not evidence of a measured price difference. Any comparison of deployment has to be made in relation to the relevant region, current tariffs, quotas and configuration [7], [8], [10], [18], [19].

B. Scalability and Operational Responsibility

Autoscaling removes the need for manual capacity changes, but the speed and usefulness of scaling still depends on control-loop delays, concurrency limits, available resources, and the throughput of backend services. Adding more function instances for example will not help if you have a database bottleneck. The managed fault-tolerance features also work on a particular service configuration and failure model. Overall availability is still dependent on event delivery, external dependencies, regional incidents and recovery procedures. Serverless therefore simplifies some operational decisions, but doesn't make the whole application resilient to failure. Managed event triggers, deployment tools, and integrated monitoring make routine development and operations easier [7], [8], [10], [18], [19]. They also add configuration options and can add additional testing or migration work. None of the studies considered here measures a broad improvement in developer productivity, release cycles or time to market. Hence, we talk about these as potential benefits and not as outcomes established by empirical comparison.

C. Resource Use and Sustainability

Improving resource use by reducing idle capacity may not necessarily mean lower electricity consumption or lower carbon emissions with better utilization. A warm pool is reserving memory for later use. Snapshots consume storage and input/output bandwidth. Optimization controllers perform their own computations. Moving work between locations can increase network traffic [20]. The Azure study assesses a resource-management policy rather than energy and carbon, over the full life cycle. It alone cannot prove that serverless computing has a smaller environmental footprint.

To support an assertion of energy savings, an evaluation would have to measure comparable work completed at similar latency and availability. It should take into account the active and idle machines, the storage, the network usage and the boundary chosen for the system. Operational carbon estimation would also require assumptions on electricity carbon intensity and explicit assumptions on time and location; embodied impacts would require their own accounting scope. The studies compared here do not provide a common basis for such calculations, and therefore this survey makes no quantitative claim about energy or carbon savings.

IV. CHALLENGES AND DESIGN CONSTRAINTS

Cold starts can include dependency loading, runtime initialization, restoration, control-plane operations, and resource contention. Starting a sandbox is just one part of the delay that a user might experience. Production observations in [5] explain why a fast sandbox operation cannot substitute for a measurement of the entire invocation path. For a latency sensitive application, the evaluation should look at warm and cold requests, time spent in queue, and the distribution of response times, particularly the 95th and 99th percentile (p95 and p99). Even if the application's function code is portable, it may be difficult to move it from one provider to another. It can be associated with a particular environment by event bindings, access control policies, storage APIs, deployment definitions and workflow behavior. CloudEvents provides a common way to represent event data [21], but does not prescribe how identity, persistence, or retries are implemented. Portability should not be assumed without testing those differences.

Service plans have different limits on execution time, memory, concurrency and size of deployment [7, 8, 10, 18, 19]. Longer jobs might need checkpoints, workflow orchestration, or some other execution service. Decomposing a computation into functions is not always free, it may introduce extra invocations and state transfers. A meaningful evaluation should thus report limits for the specific provider, service version and hosting plan in use rather than generalize from another platform. To understand what happened during a serverless request, you often need to piece together events across triggers, retries, function instances, and calls to backend services. Such traces may be partial and expensive to retrieve. Developers must also deal with vulnerable dependencies, permissions that are too broad, unsafe inputs, and possible cross-tenant isolation failures [12]. These application-level security responsibilities still apply even when using a managed runtime. When data is shared or updated, duplicate side effects and consistency rules need to be considered. In practice an optimization is useful only if these requirements are still satisfied.

V. CRITICAL LITERATURE REVIEW AND OPTIMIZATION EVIDENCE

A. Looking at the Proof

The sources of evidence in this review are not all of the same kind. Different research questions are addressed by a literature survey, a study of production workloads, a replay

of recorded traces, and a prototype experiment. Li et al. [12] provide a review of security threats and defenses, rather than analysis of one deployable mitigation. A taxonomy of cold-start approaches is presented in [13]. Eismann et al. [3] describe application usage patterns which help to judge the suitability of a workload, but do not measure a particular scaling controller. Therefore, these works are included in the discussion but not as additional empirical optimization results in Table 2.

Even within optimization experiments, latency measures reported do not always have the same meaning. Forking a sandbox, fulfilling a cold invocation at a worker, answering an entire workflow, and producing the first token of a model are all distinct stages. Figure 2 shows where representative techniques intervene along the invocation path, and what resource or correctness costs they may incur. It is a conceptual diagram (not a plot of measured time spent in each stage).

B. Restoration and Retention

Warm-pool approaches [15] keep instances that can respond to incoming requests without re-initialization. This reduces some start-up times assuming an instance is available in the first place and initialization would take a long time otherwise. Requests submitted when the pool is exhausted still incur setup costs. The reported prototype and simulation evidence the authors chosen conditions, but do not prove the approach in case of arbitrary bursts in a shared multi-tenant environment.

Record-and-prefetch (REAP) [16] records the pages touched by a function and prefetches them ahead of future snapshot-based invocations. Its measurements include work at the execution worker with the local solid-state drive (SSD) and the host page cache cleared for cold runs. Catalyzer [17] addresses the problem by restoring the sandbox and reusing existing state. It is not the response time that can be expected by every remote caller. It is the best-case startup , below one millisecond , measured for a particular operation with a running template . That narrow measurement might not account for storage requests and control-path delays . Which is why the distinction is important. .

Shahrad et al. [20] characterize invocation patterns using Azure production traces and develop an adaptive policy to keep instances alive. Real traces capture changes in request arrivals that a simple synthetic model may miss, although the findings remain tied to the provider and period observed. Rather than comparing their largest reported speedups, such retention and restoration studies are best compared by the amount of ready or stored state required to achieve a stated latency goal.

C. Workflow Composition and Control of Resources

Xanadu [22] attempts pre-computing downstream functions, before they are invoked, based on known or predicted workflow paths. Instead, function fusion [23] eliminates some of the boundaries between successive functions. Both want to shorten workflows, but each brings a different risk. Speculative preparation can be wasteful if the expected branch is not taken, and fusion can halt tasks that were previously running in parallel. The relevant outcome is not just the number of invocation steps removed, but the critical path workflow latency and resource usage.

Schuler et al. [24] tune concurrency in Knative using reinforcement learning (RL). Zafeiropoulos et al. [25] also study RL-based scaling but with agents designed for Kubeless. Their results cannot point out the best controller as they are different in their decisions, reward functions and test settings. Also, the assessment must consider training and exploration costs, workload changes, and service-level objective (SLO) breaches. ComboFunc [11] adopts a different approach with different container sizes and adjusts resources in place. Its cluster experiments relate faster scaling to placement decisions, but the level of runtime control required is not available to typical users of many managed FaaS services.

D. Evidence from Production and Accelerators

AFaaS [5] gives evidence from an informed production run time. Its results show that even with faster sandbox setup, control-path coordination, contention and initialization of user code can still introduce significant delay. So faster restoration alone is unlikely to solve all cold-start problems. The study shows what can be done in its own runtime environment, but the same gains would need to be checked independently on other platforms." HydraServe [6] considers another type of cold start, the time to first token (TTFT) generation for a large language model (LLM). The GPU experiments apply four-way pipeline parallelism. In comparison with ServerlessLLM, a distinction between cached and uncached model loading is made. Delays due to fetching model weights, initializing graphics processing units (GPUs), and a limited network capacity are distinct from small central processing unit (CPU) functions. So, the improvements reported by HydraServe should not be directly compared to component level numbers from REAP or Catalyzer.

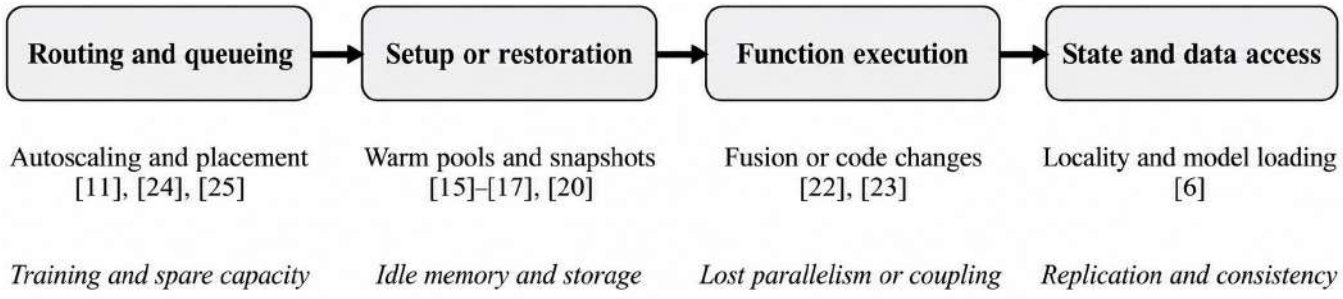

Figure 2: Optimization targets and displaced costs. Original analytical schematic based on the optimization studies and runtime evidence discussed in [5], [6], [11], [15]–[17], [20], [22]–[25]

Figure 2 groups interventions by the part of the invocation path they affect. Autoscaling and placement mainly affect request routing, queueing, and availability of execution capacity. Warm pools and snapshots are about setup or restoration. Fusion changes amount of work and coordination between functions. Locality or model-loading methods affect access to state and data. The lower labels refer to costs that may be shifted elsewhere when optimizing a single stage: training overhead, idle memory, reduced parallelism, and replication or consistency requirements. The figure is shown as a framework for reading the studies, and not as a quantitative comparison of the delays measured.

Table 2 presents a comparative overview of the experimental scope, findings and limitations of the selected serverless optimization studies. Studies are classified based on the main optimization focus. The first group deals with warm-instance retention and snapshot restoration [15]–[17], and [20] studies instance retention based on production traces. The second group is concerned with workflow execution and resource allocation via speculative scheduling, function fusion, reinforcement learning, and container placement [11], [22]–[25]. The latter studies focus on production-informed runtime optimization and accelerator-based model serving [5], [6].

Table 2: Comparative Analysis of Serverless Optimization Techniques: Experimental Results, Platforms, and Limitations

Study and approach	Test platform and configuration	Workload and relevant assumptions	Comparator and measurements	Findings reported	Conditions and limitations
Lin and Glikson [15] Warm pool	Knative prototype running on a Google Cloud instance with 4 vCPUs and 15 GB RAM; simulation conducted separately	A Go HTTP service and a Python image classifier; simulated services move between zero and one instance	Unmodified Knative without pooled warm instances; first-response latency and simulated tail latency	Numeric: Across 10 prototype trials, the classifier's mean first response decreased from 39.25 s to 7.458 s.	Instances kept ready occupy resources; contention and migration may still occur. The simulation does not show general behavior during scale-out.
Ustiugov et al. [16] REAP	vHive with Containerd and Firecracker; one worker, local SSD, and 256-MB/one-vCPU guests	FunctionBench tasks and a minimal handler; repeated working sets and cold runs with the host page cache cleared	Conventional snapshot restoration; cold-invocation latency at the worker and page faults	Numeric: Mean cold-invocation latency improved by 3.7× over the baseline snapshot method.	The recorded working set helps later calls. Changes in memory access, storage throughput, and distributed control paths may affect the benefit.
Schuler et al. [24] Adaptive concurrency	Knative v0.12 on IBM Cloud Kubernetes, using nine service nodes with 16 vCPUs and 64 GB each	Synthetic CPU-heavy, memory-heavy, and waiting workloads; scaling to zero allowed between rounds	Knative's default concurrency setting; throughput, latency, and resource measures	Qualitative summary: The learned policies increased throughput over the default for workloads tested.	The evaluation uses one configurable application and a simple reward. Exploration and workload drift make broader transfer uncertain.
Wen et al. [11] ComboFunc	Knative v1.7 with Kubernetes v1.27.1; 16 nodes, each offering 8 CPU cores and 16 GB	File, video, web, and ML tasks; scaled Azure traces; in-place vertical resource adjustments enabled	Default Knative with uniform container sizes and no in-place resizing; time to scale and modeled resource expense	Numeric: File-processing resources scaled 2.85× faster than with default Knative.	Requires operator control of placement and resizing. Modeled resource expense is not a full application bill; traces are scaled to the test cluster.
Daw et al. [22] Xanadu	Prototype workflow engine using several isolation modes; compared with Knative and OpenWhisk	Directed acyclic workflows, synthetic chains, and emulated shopping and image-processing pipelines	Reactive preparation on the comparison platforms; overhead latency, processor use, and memory costs	Qualitative summary: Speculative and just-in-time preparation reduced cascading setup delays in the workflows studied.	Needs knowledge or predictions about execution paths. Wrong predictions waste capacity; workflow structure and isolation mode influence results.
Lee et al. [23] Function fusion	Experiments on AWS Lambda with automatic function fusion and a workflow-level response model	Five workflows with branches and parallel stages; only functions eligible for fusion are combined	Original workflows without fusion; complete response time under specified warm and cold conditions	Numeric: Across five workflows, response times were 28–86% of those for the original workflows.	Fusion may remove parallel execution, separate scaling, or deployment independence. Its benefit depends on workflow structure.

Du et al. [17] Catalyzer	Runtime built on gVisor; microbenchmarks on an i7/32-GB machine and server tests on a 96-core/256-GB host	C, Java, Python, Ruby, and Node.js functions; restoration, templates, and sandbox-fork configurations	gVisor and restoration variants, Docker, Hyper containers, and Firecracker; startup and function response times	Numeric: Sandbox fork reached 0.97 ms startup for a C hello-world handler with a template already running.	This is a best-case startup measurement, not end-to-end latency. Template memory, safe state reuse, and platform integration still matter.
RL agents [25] Autoscaling	Kubeless on three Intel NUC Kubernetes nodes; real-world and simulated setups	Changing traffic, with Q-learning, DynaQ+, and deep Q-learning agents; rewards account for service quality and resources	Configured scaling approaches and the RL agents; latency, throughput, violations, and resource use	Qualitative summary: Adaptive resource allocation was demonstrated in the real and simulated environments.	Small cluster and training/reward rules tied to the environment. Safety during adaptation and transfer to other settings remain untested.
Shahrad et al. [20] Hybrid keep-alive	Characterization of Azure production traffic with trace-based tests of a retention policy	Different invocation rates and idle intervals; histogram-guided retention with fallback behavior	Fixed keep-alive rules; cold-start rate and resource/memory consumption	Qualitative summary: The policy reduced cold starts while using fewer resources than compared fixed-retention methods.	Findings depend on one provider's traces and time window. Reductions in reserved memory do not establish energy or carbon benefits.
Chai et al. [5] AFaaS	Production-informed AFaaS runtime at Ant Group; concurrent and sustained CPU tasks	Changes to the runtime control path, pooled resources, and initialized code templates; 18 months of deployment experience	Catalyzer under the assessed production conditions; end-to-end delay and startup stability	Numeric: End-to-end latency improved by 1.80×–8.14× against Catalyzer in the reported scenarios.	The gains come from coordinated changes to the runtime. Other workloads and systems require portability checks and independent reproduction.
Lou et al. [6] HydraServe	Testbed (i): four A10 machines (one GPU each) and four V100 machines (four GPUs each); 16 Gbps per server, remote model storage, pipeline parallelism = 4	OPT, Llama 2/3, and Falcon models; cold TTFT with overlapping model retrieval and runtime initialization	Serverless vLLM and ServerlessLLM, tested both with and without RAM model caching (no high-speed SSD); cold-start TTFT and separate SLO tests	Numeric: TTFT improved by 2.1×–4.7× over serverless vLLM and by 1.7×–3.1× over uncached ServerlessLLM [6, Figure 7].	Cached ServerlessLLM remains a distinct comparator. Model choice, GPU hardware, connectivity, and cache conditions affect the result; small CPU functions differ.

In this table, "Numeric" indicates that the metric and comparator, or absolute measurement boundary, reported can be traced to the cited study. "Qualitative summary" refers to the direction or nature of the reported findings without converting them to a common effect size . It does not mean that the original study does not contain numerical results. Reported findings should be interpreted according to the experimental conditions, platforms, workloads, hardware configurations, baselines, and measurement boundaries of their respective studies. For example, sub-millisecond sandbox startup, a workflow response-time improvement, and model time-to-first-token (TTFT) are different performance indicators and should not be compared or combined directly. Therefore, Table 2 aims to briefly outline the experimental assumptions, reported results, and limitations of the reviewed techniques, rather than to provide a numerical ranking of their performance. Source: Authors' comparative analysis; referenced publications.

E. Evidence Synthesis and Limitations

The comparison yields three lessons. You should not only measure an improvement at the component you are optimizing but also at the boundary of the complete application. Lower latency can be obtained by keeping resources available, changing scheduling decisions or reorganizing the workflow, so cost and correctness need to be checked together. Finally, production studies provide valuable evidence about operating systems at scale, but a result seen in one runtime cannot be automatically transferred to another provider or workload. These considerations motivate the evaluation questions in Section VII.

The evidence reviewed has evident limitations. We have selected older versions of the platforms, synthetic tests, trace-based evaluations, prototypes, and observations in production by individual providers. Published data may be biased toward positive results. Furthermore, the baselines, caching setups, hardware, levels of concurrency and study periods all differ, making it hard to compare numbers directly. We have not reproduced the experiments and

calculated an overall effect. If the necessary evidence is missing, we consider the wider applicability of a technique as unknown, rather than conclude that it cannot work.

F. Cross-technique Synthesis and Deployment Implications

The studies show the optimization choice should be based on the dominant source of delay, not the largest headline speedup. Warm pools or snapshot restoration can be beneficial if initialization is a significant factor and request arrivals are reasonably predictable. However, reserved memory, storage traffic, cache behavior, and state-safety checks should be considered in the decision. If the critical path of a workflow is dominated by sequential function setup, speculative preparation or fusion can reduce coordination, but speculation can consume capacity on unused branches and fusion can remove parallelism or independent scaling. Autoscaling and placement policies are relevant where queueing and capacity shortages dominate, but their training overhead, controller stability and operator access must be taken into account. Model-weight transfer, GPU initialization and cache state in model-serving workloads need to be treated differently than traditional CPU-function cold starts. These observations lead to the following qualitative decision guide. It is one interpretation of the selected studies, not a universal prescription: (1) prefer warm retention when burst patterns and latency objectives justify the idle-memory cost; (2) prefer snapshots when initialization state is reusable and storage restoration is not the new bottleneck; (3) prefer workflow preparation or fusion when sequential setup dominates and lost parallelism is acceptable; (4) prefer learned autoscaling only with representative training traces, safe fallback behavior, and explicit SLO monitoring; (5) prefer placement or heterogeneous sizing when the operator controls the cluster and can measure end-to-end effects; and (6) prefer LLM-serving optimizations with TTFT, model-cache state, GPU/network utilization, and model-specific workloads. In all cases, the final decision should be based on total application cost and performance of the completed request, not on an isolated component metric. Future comparative experiments should report at least median, p95 and p99 latency; throughput and successful completions; cold-start frequency; CPU, memory, storage and network consumption; failure and retry rates; and the full cost boundary. Where energy or carbon claims are made, measurements should cover compute, idle capacity, storage, and networking for equivalent completed work and state the electricity mix and accounting boundary. Changes in code, dependencies and credentials should be followed by testing of reused snapshots, templates and warm state including invalidation and tenant-isolation scenarios.

VI. PLATFORMS, TOOLS, AND INTEROPERABILITY

The platform a function runs on determines the system a developer can change in an application. Managed FaaS services expose configuration and integrations, but generally keep host scheduling and sandbox internals out of reach. Self-managed services can give access to those components but then their operators have to maintain the cluster. The descriptions below compare available architectural controls, not price or performance rankings. Documentation was verified in Sept. 2026.

AWS Lambda [10] executes functions in response to events, and integrates those functions with other managed services. Developers define handlers, triggers, permissions and supported resource options; AWS manages the execution environment. So, techniques that require modification of the sandbox or host scheduler can't be assumed to work with standard Lambda configuration alone.

Invocation options are HTTP requests and event integrations. Google Cloud Run functions [7] support managed builds from source code and deployment as containers. Both Cloud Functions and Cloud Run APIs apply to this deployment model. You should look for the service generation and configuration used when comparing published results, rather than assuming every historical Google Cloud Functions experiment is the same.

Microsoft Azure Functions [19] provides event triggers and bindings, local development support and integrated monitoring. Durable Functions extend the model to coordinated work flows. Experiments should declare the hosting arrangement and the backend services that a function depends on . Workflow support can help with some coordination, but it does not eliminate the fees or correctness issues inherent in storing persistent state.

Knative Serving [18] runs request-driven containers on Kubernetes and provides autoscaling, revisions, traffic routing, and configurable concurrency. Apache OpenWhisk [8] connects triggers to executable actions and supports container-based deployment. Such systems can give the researchers a better control over the execution environment. That control carries responsibility for cluster capacity, networking, storage and monitoring, all of which matter when calculating resource cost.

CloudEvents [21] defines the representation of event information; it is not a FaaS runtime nor a framework to deploy applications. Sharing information between services is facilitated by a common event format, but to move a workflow you need compatible payload definitions, identity rules, state access, delivery behavior and execution semantics. Open interfaces can help with integration, but cannot guaranty two providers will deliver the same performance.

VII. FUTURE RESEARCH DIRECTIONS

Table 3 translates the gaps identified in the reviewed work into research questions with corresponding evaluation requirements. These are priorities implied by the present comparison, not assertions that no researcher has ever considered them. Future studies should at least provide their workload definitions, platform and runtime versions, memory and concurrency settings, cache and warm-state policies, baseline configurations, repeated-run variation, and measurement boundaries.

The practical implication for platform operators is to expose enough telemetry to distinguish queueing, provisioning, initialization, execution, and backend access time. Distributed traces should be used by application developers to first identify the dominant stage, then select an optimization that targets that stage and does not violate state, security or portability requirements. Where possible,

researchers should release reproducible workload definitions, platform versions, configuration files, baseline settings and raw or aggregated measurement data. These steps would make future comparisons more transferrable and reduce the risk of optimizing one component and making the complete request path worse. While testing a technique in isolation is useful, combined experiments are also needed. Warm pools, snapshots, function fusion and learned scaling policies can work synergistically or against each other. The comparisons should be made with a common workload and service-level objective (SLO), and report latency distributions, throughput, completed requests, failures, resource consumption, and total cost of running the application. Optimizations that affect shared state or isolation also need to be tested for correctness and security . If these properties degrade execution time is faster . This is not enough .

Table 3: Research questions derived from the evidence

Direction	Evidence gap or limitation	Question for future work	What to Measure
Cold starts and combined optimizations	Retention and restoration [15], [16], [17] shorten different startup stages; production observations [5] also point to control-path and contention delay.	Under a fixed resource budget, when do warm pools, snapshots, and runtime changes work better together than separately?	Isolated and combined tests; sparse traffic and bursts; p95/p99 end-to-end delay, ready memory, storage traffic, and checks of isolation.
Multi-cloud interoperability	Common event representation [21] addresses only part of portability; provider bindings and interfaces differ [7], [8], [10], [18], [19].	Which state, identity, and retry behaviors remain consistent when the same workflow is moved between cloud providers?	Matching contracts and traffic on at least two providers; migration work, semantic differences, transfer charges, failures, and tail latency.
Edge and cloud placement	Experiments in controlled clusters [25], [11] do not demonstrate performance across small, heterogeneous edge installations.	Can placement and retained warm state still achieve a latency goal when links are unreliable or local capacity changes?	Physical edge hardware, changing latency and bandwidth, intermittent connections, warm/cold requests, privacy limits, and resource or energy use.
Observability	Table 2 studies use different timing boundaries and instrumented runtimes; a complete asynchronous request path may be unobserved.	Can distributed tracing separate queueing, initialization, state access, and retries without creating excessive overhead?	Known event paths, controlled failures, coverage and accuracy of traces, timing overhead, and logging expense.
AI-based autoscaling	RL scaling methods [25], [24] use particular actions, reward definitions, and training workloads.	How can learned scaling adapt to changing demand while limiting unsafe exploration and SLO breaches?	Unseen and shifting traffic; tuned fixed and non-learning baselines; training/inference resources, adaptation time, SLO breaches, and fallback handling.
Stateful and long-running workflows	Speculation and fusion [22], [23] change function boundaries; basic FaaS applications still place persistent state outside execution [1].	Can state locality and checkpoints reduce delay while preserving consistency and preventing duplicate effects?	Defined consistency rules; retries, crashes, network partitions, recovery times, duplicate operations, data traffic, and total application cost.
Energy and carbon accounting	Reported resource savings [11], [20] are not whole-system energy results; accelerator model loading brings additional costs [6].	After standby and data movement are counted, does lower resource use translate into less energy for each successful task?	Equivalent completed work and SLOs; host, network, and storage energy; explicit accounting scope and carbon intensity with uncertainty.
Security preserving optimization	Reusable state and faster startup [16], [17], [5] must still respect the security and permission concerns discussed in [12].	How can shared infrastructure and reusable templates preserve tenant isolation after code or credential changes?	Stated threat model; tests for refreshed secrets, invalidated snapshots, tenant isolation, dependency updates, overhead, and secure recovery.

VIII. CONCLUSION

The survey reviewed the serverless optimization results in terms of execution architectures and experimental conditions that generated them. Earlier reviews have addressed the broad opportunities, challenges, security concerns, and cold-start mitigation. The narrower contribution of the present survey is to relate selected optimization interventions to their platforms, workloads, baselines, measurement boundaries, reported outcomes, and displaced costs. The review does not propose a pooled estimate or a universal ranking and the evidence base is purposive, not exhaustive.

None of the techniques is the best choice for all workloads across the 11 empirical studies. Warm instances can help to mitigate some startup delays, but they do consume capacity. Snapshots and templates are derived from working set patterns, storage access and safe state reuse. Workflow preparation and fusion can reduce selected critical paths. Speculation can waste resources and fusion can reduce parallelism. Controller design, training workloads, and system-level privileges determine autoscaling and placement. The production runtimes and LLM serving show that after one startup component has been accelerated, control operations, contention, model transfer and accelerator initialization can still be important.

So, the practical choice should be guided by the workload's arrival pattern, latency objective, resource budget, state and consistency requirements of the workload, security model, and the available platform controls. Future evaluations should use matched workloads and baselines, disclose platform and runtime versions, report latency distributions and completed work, and account for compute, memory, storage, networking, retries, and idle capacity. If you reuse state or change execution boundaries, you also need to consider correctness, tenant isolation, and performance. The reviewed literature does not show that serverless is necessarily cheaper or more energy-efficient than provisioned alternatives. Such claims necessitate comparable measurements at the system level. The primary conclusion is conditional: an optimization is useful if it shows its benefit on the application boundary, and its displaced costs and operational risks are measured under the intended workload.

CONFLICTS OF INTEREST

The authors declare that they have no conflicts of interest.

REFERENCES

[1] Jonas et al., "Cloud programming simplified: A Berkeley view on serverless computing," University of California, Berkeley, Tech. Rep. UCB/EECS-2019-3, 2019, arXiv:1902.03383. Available from: https://arxiv.org/abs/1902.03383

[2] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, "The rise of serverless computing," *Commun. ACM*, vol. 62, no. 12, pp. 44–54, Dec. 2019. Available from: https://doi.org/10.1145/3368454

[3] S. Eismann, J. Scheuner, E. van Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. L. Abad, and A. Iosup, "Serverless applications: Why, when, and how?" *IEEE Softw.*, vol. 38, no. 1, pp. 32–39, Jan./Feb. 2021. Available from: https://doi.org/10.1109/MS.2020.3023302

[4] Shafiei, A. Khonsari, and P. Mousavi, "Serverless computing: A survey of opportunities, challenges, and applications," *ACM Comput. Surv.*, vol. 54, no. 11s, Art. 239, pp. 1–32, 2022. Available from: https://doi.org/10.1145/3510611

[5] Chai et al., "Fork in the road: Reflections and optimizations for cold start latency in production serverless systems," in *Proc. USENIX OSDI*, 2025. Available from: https://www.usenix.org/conference/osdi25/presentation/chai-xiaohu

[6] Lou, S. Qi, C. Jin, D. Nie, H. Yang, Y. Ding, X. Liu, and X. Jin, "HydraServe: Minimizing cold start latency for serverless LLM serving in public clouds," in *Proc. USENIX NSDI*, 2026, pp. 415–430. Available from: https://www.usenix.org/conference/nsdi26/presentation/lou

[7] Google Cloud, "Cloud Run functions overview." Available from: https://docs.cloud.google.com/run/docs/functions/overview

[8] Apache Software Foundation, "Apache OpenWhisk." Available from: https://openwhisk.apache.org/

[9] ISO/IEC 22123-2:2023, *Information technology—Cloud computing—Part 2: Concepts*. Geneva, Switzerland: International Organization for Standardization, 2023.

[10] Amazon Web Services, "AWS Lambda developer guide." Available from: https://docs.aws.amazon.com/lambda/

[11] Z. Wen, Q. Chen, Q. Deng, Y. Niu, Z. Song, and F. Liu, "ComboFunc: Joint resource combination and container placement for serverless function scaling with heterogeneous container," *IEEE Trans. Parallel Distrib. Syst.*, vol. 35, no. 11, pp. 1989–2005, Nov. 2024. Available from: https://doi.org/10.1109/TPDS.2024.3454071

[12] Li, X. Leng, and Y. Chen, "Securing serverless computing: Challenges, solutions, and opportunities," *IEEE Network*, vol. 37, no. 2, pp. 166–173, Mar./Apr. 2023. Available from: https://doi.org/10.1109/MNET.005.2100335

[13] M. Golec, G. K. Walia, M. Kumar, F. Cuadrado, S. S. Gill, and S. Uhlig, "Cold start latency in serverless computing: A systematic review, taxonomy, and future directions," *ACM Comput. Surv.*, vol. 57, no. 3, Art. 65, pp. 1–36, 2025. Available from: https://doi.org/10.1145/3700875

[14] McGrath and P. R. Brenner, "Serverless computing: Design, implementation, and performance," in *Proc. IEEE ICDCSW*, 2017, pp. 405–410. Available from: https://doi.org/10.1109/ICDCSW.2017.36

[15] P.-M. Lin and A. Glikson, "Mitigating cold starts in serverless platforms: A pool-based approach," arXiv:1903.12221, 2019. Available from: https://doi.org/10.48550/arXiv.1903.12221

[16] Ustiugov, P. Petrov, M. Kogias, E. Bugnion, and B. Grot, "Benchmarking, analysis, and optimization of serverless function snapshots," in *Proc. ACM ASPLOS*, 2021, pp. 559–572. Available from: https://doi.org/10.1145/3445814.3446714

[17] 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. ACM ASPLOS*, 2020, pp. 467–481. Available from: https://doi.org/10.1145/3373376.3378512

[18] Knative, "Knative Serving overview." Available from: https://knative.dev/docs/serving/

[19] Microsoft, "Azure Functions overview." Available from: https://learn.microsoft.com/en-us/azure/azure-functions/functions-overview

[20] 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. Available from: https://www.usenix.org/conference/atc20/presentation/shahrad

[21] Cloud Native Computing Foundation, "CloudEvents." Available from: https://cloudevents.io/

[22] N. Daw, U. Bellur, and P. Kulkarni, "Xanadu: Mitigating cascading cold starts in serverless function chain deployments," in *Proc. 21st Int. Middleware Conf.*, 2020, pp. 356–370. Available from: https://doi.org/10.1145/3423211.3425690

[23] S. Lee, D. Yoon, S. Yeo, and S. Oh, "Mitigating cold start problem in serverless computing with function fusion," *Sensors*, vol. 21, no. 24, Art. 8416, 2021. Available from: https://doi.org/10.3390/s21248416

[24] L. Schuler, S. Jamil, and N. Kühl, "AI-based resource allocation: Reinforcement learning for adaptive auto-scaling in serverless environments," in *Proc. IEEE/ACM CCGrid*, 2021, pp. 804–811. Author version: arXiv:2005.14410. Available from: https://ieeexplore.ieee.org/abstract/document/9499535

[25] Zafeiropoulos, E. Fotopoulou, N. Filinis, and S. Papavassiliou, "Reinforcement learning-assisted autoscaling mechanisms for serverless computing platforms," *Simul. Model. Pract. Theory*, vol. 116, Art. 102461, 2022. Available from: https://doi.org/10.1016/j.simpat.2021.102461