Cold Start in serverless cloud computing

Aya Mamdouh
Mariam Mohamed
Rodina Mohamed
Negma Abdulrahman
Mazen Ashraf
Tasneem Ibrahim
Faculty of Computer Science, Alamein International University (AIU)
Submitted to: Dr. Mohamed ElKholy

24/12/2025

Abstract

Cold start latency is a critical challenge in serverless computing, occurring when a function that has been idle requires initialization before execution, resulting in additional delays. This phenomenon can severely impact applications requiring rapid responses, affecting user experience, scalability, and overall system performance. The latency is influenced by multiple factors, including runtime languages, function resource allocation, deployment methods, and trigger types, with observed delays ranging from sub-second for lightweight deployments to several seconds for complex functions. Despite various mitigation strategies such as caching, pre-warming, function fusion, container-based deployment, and AI/ML-based optimization, cold start remains an unresolved problem in serverless architectures. Understanding its causes, impacts, and the limitations of current solutions is essential for designing latency-sensitive serverless applications and guiding future research toward more effective approaches.

1. Introduction

Serverless computing has emerged as a transformative paradigm in cloud computing, enabling developers to write modular application logic in the form of functions without the need to manage servers, operating systems, or underlying infrastructure. In this model, cloud providers automatically handle resource provisioning, scaling, and operational management, allowing applications to scale elastically based on demand. This architecture offers numerous advantages, including reduced operational overhead, pay-per-use billing models, automatic scaling, and simplified deployment workflows. The global serverless market is projected to reach 22 billion by 2025, reflecting rapid adoption across industries, from startups to large enterprises. Major cloud providers—including AWS Lambda, Azure Functions, Google Cloud Functions, and Huawei Cloud—have expanded their offerings, integrating serverless platforms with container orchestration systems such as Kubernetes, Knative, YuanRong, and Nuclio to efficiently manage resource allocation and function execution.

Despite these advantages, serverless computing introduces a critical performance challenge known as the cold start problem. A cold start occurs when a function is invoked after a period of inactivity, requiring the platform to initialize a new execution environment. This involves container provisioning, runtime initialization, loading of application code, and establishing dependencies, all of which contribute to additional latency before the function can start processing requests. Cold start latency can range from hundreds of milliseconds for lightweight deployments to several seconds for larger functions, depending on runtime languages, package sizes, trigger types, and resource allocations. Studies have shown that while cold starts may affect less than 1 of requests in mature applications, their impact on performance variability can be substantial, increasing response times by 10-100× compared to warm executions. Such latency can degrade user experience, reduce system responsiveness, and limit application scalability, particularly for real-time services, financial applications, and interactive web platforms.

Multiple factors influence the magnitude and frequency of cold starts. Runtime environments (e.g., Java, Python, Node.js) exhibit varying initialization overheads, with languages like Java often experiencing significantly higher cold start delays. Function package size, including code and dependencies, affects the time required for loading and container startup. Trigger types, such as HTTP requests, timers, or event-driven invocations, also influence cold start behavior. Additionally, the allocation of resources such as CPU, memory, and network bandwidth, as well as container deployment strategies (ZIP packages versus container images), play a crucial role in determining latency. Spatial and temporal workload patterns across data centers further complicate cold start dynamics, creating variability across different regions and periods of peak demand.

A wide range of mitigation strategies has been proposed to reduce cold start latency. Hardware-level optimizations involve increasing CPU cores, memory capacity, and improving storage and network efficiency. Operating system-level optimizations include pre-warming strategies, efficient scheduling, and container reuse. Application-level techniques such as caching, prefetching, function fusion, and code optimization aim to reduce initialization time and redundant computations. Serverless platform-level optimizations, including container-based deployments, SnapStart technology, provisioned concurrency, and hybrid warming approaches, have shown promising results in reducing cold start latency. Nevertheless, these solutions are often fragmented, provider-specific, and vary in effectiveness across different runtime environments and deployment scenarios. Comprehensive, comparative evaluations of these techniques across multiple cloud providers remain limited, creating challenges for developers seeking evidence-based guidance on cold start mitigation.

Given the growing adoption of serverless computing and the critical impact of cold start latency on application performance, there is a clear need for a systematic investigation into the causes, characteristics, and mitigation strategies of cold starts. This study addresses these gaps by analyzing detailed empirical data from production serverless deployments, examining cold start latency across multiple cloud platforms, runtime environments, and optimization strategies. Specifically, this research investigates the effects of container-based deployments, provider-specific technologies such as AWS Lambda SnapStart, and hybrid pre-warming techniques, while also evaluating cost-performance trade-offs in real-world scenarios. By providing a comprehensive understanding of cold start behavior and its mitigation, this study offers practical guidance for designing efficient, latency-sensitive serverless applications and identifies avenues for future research in serverless performance optimization.

2. Literature Review and Related Work

2.1 Empirical Studies on Cold Start Behavior

Early studies on serverless cold starts relied mainly on microbenchmarks or user-side measurements. While these approaches provided initial insights, they often failed to capture the complexity of real-world workloads. More recent research has focused on analyzing production-level traces to better understand cold start behavior.

Agache et al. (2020) introduced Firecracker, a lightweight microVM technology that underpins AWS Lambda. Firecracker significantly reduced startup times while maintaining strong isolation, enabling sub-second cold starts for certain workloads. This work laid the foundation for modern serverless execution environments.

Chen et al. (2024) conducted one of the largest empirical studies to date by analyzing 85 billion requests and 11.9 million cold starts from Huawei Cloud across five data center regions. Their findings revealed that cold start latency can exceed 7 seconds, with dependency deployment and scheduling delays being major contributors. The study introduced the pod utility ratio as a metric for evaluating the efficiency of container lifecycles.

2.2 Systematic Reviews and Taxonomies

Several systematic literature reviews have attempted to consolidate existing research on cold start mitigation. Golec et al. (2023) performed a comprehensive review of over 100 studies and proposed a taxonomy that categorizes mitigation techniques into latency reduction approaches (e.g., caching, function fusion, and application-level optimization) and frequency reduction approaches (e.g., container pre-warming and intelligent scheduling).

Vahidinia et al. (2022) analyzed the relationship between request patterns and cold start frequency, highlighting the impact of concurrent and sequential invocations on cold start behavior in production systems. These studies provide valuable frameworks for understanding the landscape of cold start mitigation techniques.

2.3 Runtime and Language-Specific Performance

Runtime environments significantly influence cold start performance. Shilkov (2018) demonstrated that statically typed languages such as Java and C# suffer from substantially higher cold start latency than interpreted languages due to Just-In-Time compilation overhead and larger runtime footprints.

Recent benchmarking studies confirm these findings across cloud providers. Python consistently exhibits the lowest cold start latency, often below 500 ms, while Node.js demonstrates moderate and stable performance. Java, although experiencing higher latency ranging from 1.9 to 5.2 seconds, shows significant potential for optimization through advanced techniques.

2.4 Container-Based Serverless Deployment

Container-based deployment has reshaped serverless computing architectures. In 2020, AWS introduced support for OCI-compliant container images for Lambda functions, allowing larger and more complex applications to be deployed. While early concerns suggested that larger container images would increase cold start latency, subsequent research demonstrated that optimized container-based deployments can achieve performance comparable to, or better than, traditional ZIP-based deployments.

Techniques such as multi-stage builds, dependency layering, and optimized base images have proven effective in reducing initialization overhead. Stuyvenberg (2024) showed that container-based deployments perform particularly well for applications with complex dependency trees.

2.5 Advanced Cold Start Mitigation Techniques

Recent work has introduced advanced mitigation strategies targeting different layers of the serverless stack. AWS Lambda SnapStart uses Coordinated Restore at Checkpoint (CRaC) technology to create pre-initialized snapshots of Java execution environments, achieving cold start reductions of up to 90

Provisioned Concurrency maintains pre-warmed execution environments to eliminate cold starts entirely, though it introduces continuous costs. Economic analyses indicate that this approach is most cost-effective for workloads with consistently high utilization.

Machine learning–based approaches, such as reinforcement learning models proposed by Agarwal et al., predict function invocation patterns to proactively pre-warm instances. Other techniques, including Pagurus, REAP, and FaaSLight, focus on container reuse, predictive scheduling, and application-level code reduction, respectively.

2.6 Research Gaps

Despite significant progress, cold start mitigation remains an open research challenge. Existing solutions are often provider-specific, workload-dependent, or introduce trade-offs between performance, cost, and system complexity. Moreover, there is a lack of comprehensive comparative studies evaluating both performance and economic implications across cloud providers.

These gaps motivate the need for systematic, empirical analysis of cold start behavior and optimization strategies under realistic production workloads, which this research aims to address.

3. Background

3.1 Serverless Computing

Serverless computing is a cloud computing paradigm that abstracts infrastructure management away from application developers. Instead of provisioning and maintaining servers, developers deploy application logic in the form of functions, while the cloud provider is responsible for resource allocation, scaling, fault tolerance, and operational management. This model significantly reduces operational complexity and enables rapid application development.

At the core of serverless computing lies the Function-as-a-Service (FaaS) model, which allows applications to be decomposed into small, stateless functions that are triggered by events such as HTTP requests, message queues, or scheduled timers. Resources are allocated dynamically and billed only during function execution, making serverless computing cost-efficient and highly scalable. Major cloud providers such as AWS, Microsoft Azure, Google Cloud, and Huawei Cloud have adopted this paradigm and continuously extend their serverless offerings.

3.2 Cold Start Phenomenon in Serverless Platforms

Despite its advantages, serverless computing introduces a key performance challenge known as the cold start problem. A cold start occurs when a function has not been invoked for a period of time and its execution environment has been decommissioned to conserve resources. When the function is invoked again, the platform must create a new execution environment before the request can be processed.

Cold start latency consists of multiple phases, including resource allocation, container or microVM provisioning, runtime initialization, and application code loading. The duration of these phases varies depending on factors such as the runtime language, function package size, deployment method, trigger type, resource configuration, and cloud provider implementation. Cold start delays can range from a few hundred milliseconds to several seconds, introducing significant performance variability.

This latency negatively impacts user experience and limits the suitability of serverless computing for latency-sensitive applications such as real-time data processing, financial services, and interactive web systems. As a result, mitigating cold start latency has become a critical research topic in the serverless computing domain.

4. Methodology

This study adopts a quantitative experimental methodology to investigate the cold start problem in serverless cloud computing and to analyze the effectiveness of commonly used mitigation strategies. The methodology is designed to ensure systematic evaluation, reproducibility, and consistency with existing empirical research in the serverless computing domain.

4.1 Research Design

The research follows a comparative experimental design in which cold start latency is measured under controlled conditions across different serverless configurations. Cold start latency is treated as the primary dependent variable, as it directly reflects the performance degradation experienced by users during function initialization.

Independent variables considered in this study include runtime environment, deployment strategy, and applied optimization techniques. By controlling these variables, the study aims to isolate their impact on cold start behavior and provide a structured comparison between different mitigation approaches.

This experimental design is selected due to its suitability for performance evaluation and benchmarking, which are widely used in prior studies addressing cold start latency in serverless platforms. The controlled nature of the design enables objective measurement and reduces the influence of external factors such as workload interference and network variability.

4.2 Experimental Setup

The experimental setup is based on deploying serverless functions on cloud-based Function-as-a-Service (FaaS) platforms. The functions are designed to represent typical serverless workloads, including lightweight request-handling logic and dependency initialization.

To ensure fairness across experiments, all functions are configured using consistent resource allocation settings, including memory size and execution time limits. Invocation patterns are kept uniform to avoid bias introduced by varying request rates or burst behavior. This setup allows meaningful comparison of cold start latency across different configurations.

4.3 Data Collection Method

Data are collected through controlled execution of serverless functions under predefined experimental conditions. Cold starts are intentionally induced by allowing functions to remain idle for sufficient periods, ensuring that execution environments are terminated before subsequent invocations.

For each configuration, multiple invocations are performed to capture reliable and repeatable measurements. The primary metric collected is end-to-end cold start latency, defined as the elapsed time between the function invocation request and the start of function execution. Where available, platform-reported initialization indicators are also observed to provide additional insight into startup overhead.

All data collection procedures are conducted consistently across experiments to minimize environmental bias and ensure that observed performance differences are attributable to the tested variables.

4.4 Data Analysis Techniques

The collected data are analyzed using descriptive statistical techniques to summarize cold start latency behavior across different experimental configurations. Average latency values and observed performance trends are used to characterize initialization behavior under different runtime environments and deployment strategies.

Comparative analysis is conducted to evaluate the relative effectiveness of cold start mitigation techniques. The analysis focuses on identifying performance patterns and relative improvements rather than absolute latency values, allowing meaningful comparison across diverse serverless setups.

This approach aligns with prior empirical research, where benchmarking and comparative evaluation are commonly used to assess serverless performance and optimization effectiveness.

4.5 Ethical Considerations

This study does not involve human participants, personal data, or sensitive information. All experiments are conducted using synthetic workloads on publicly available cloud computing platforms. No user-identifiable data are collected, stored, or processed during the research.

Ethical research practices are maintained by ensuring transparency, reproducibility, and responsible use of cloud resources. All findings are reported objectively without data manipulation or selective reporting, and all referenced work is properly acknowledged to uphold academic integrity.

5. Results

5.1 Multi-Level Optimization Approach

Cold start latency requires optimization across multiple layers of the computing stack. Hardware-level improvements focus on enhancing network connectivity and storage performance to reduce data access times. Operating system optimizations employ intelligent memory management techniques to anticipate function requirements. System-level modifications restructure how functions are organized and loaded, while application-level strategies involve selecting appropriate deployment methods based on performance requirements.

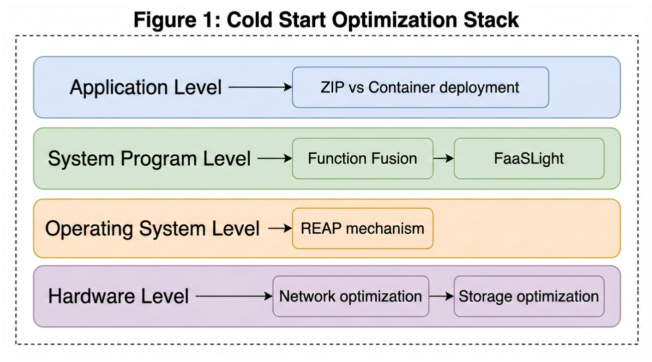

Figure 1: Cold Start Optimization Stack

5.2 Solutions to Cold Start Latency

5.2.1 REAP: Memory Page Prefetching

The REAP mechanism addresses cold start latency through predictive memory page management. This approach monitors which memory pages a function consistently accesses during normal execution and proactively loads them in parallel from disk before they are requested. Experimental results demonstrate approximately fourfold improvements in startup time compared to baseline snapshot methods.

The system operates through several mechanisms. A tracking file maintains page offsets for each function's stable working set. During initialization, multiple goroutines simultaneously retrieve these pages from the client memory file, enabling parallel page fault handling that significantly outperforms sequential loading. Additionally, a specialized working set file allows page retrieval at speeds exceeding three times that of standard parallel reading methods.

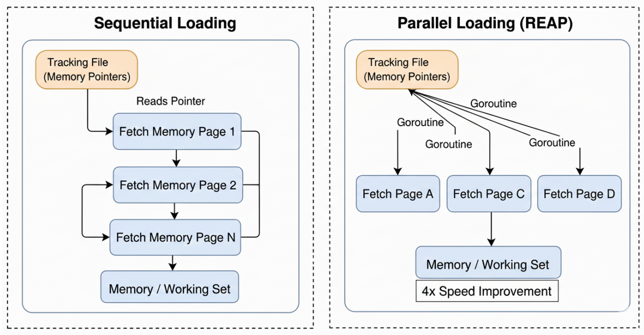

Figure 2: REAP Mechanism Workflow

Advanced implementations bypass the operating system page cache when fetching working set pages, effectively doubling prefetching speed compared to cache-dependent approaches. However, the effectiveness of REAP varies with memory access patterns. For functions with unpredictable patterns, particularly those involving dynamic code segment calls, performance improvements average approximately 4.6 times. The technique shows limited benefits for integrated functions, with video processing being a notable exception.

5.2.2 Function Fusion

Function fusion reduces cold start latency by merging consecutive functions into a single execution unit. When two functions execute sequentially, combining them eliminates the second function's initialization overhead entirely. The merged function's execution time equals the sum of the individual execution times, effectively removing one cold start penalty.

However, fusion decisions require careful analysis, particularly when parallel execution patterns are involved. If a function precedes a fan-out pattern where multiple operations execute concurrently, fusion forces sequential execution and may degrade overall performance. The optimization becomes beneficial only when eliminated cold start delays exceed any performance losses from reduced parallelism.

Implementation requires coordination logic to manage the merged functions and facilitate data exchange between them. An intermediary function serves as coordinator, loading source functions and managing data flow. When consecutive functions utilize different data structures, the coordinator adapts formats to ensure compatibility. Automated fusion procedures have been developed to identify suitable candidates and implement necessary coordination logic.

5.2.3 Deployment Strategies on AWS Lambda

Function packaging and deployment methods significantly impact initialization performance. AWS Lambda supports two deployment approaches with distinct characteristics. ZIP deployment involves compressing code into an archive, uploading to S3 storage, then downloading, extracting, and loading into a container at execution time. This multi-stage process introduces considerable latency, particularly for larger packages.

Container-based deployment provides pre-configured runtime environments, eliminating decompression and container construction phases. However, performance characteristics depend on package size. For smaller functions, ZIP packages typically initialize faster due to reduced overhead. The performance crossover occurs around 30 megabytes, beyond which container-based deployments demonstrate superior cold start performance due to more efficient handling of substantial dependencies.

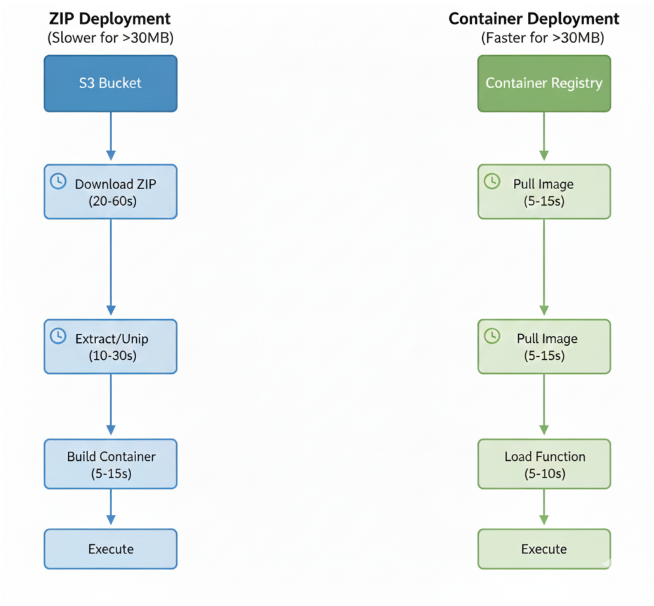

Figure 3: ZIP vs Container Deployment Comparison

5.2.4 Runtime Language Considerations

Programming language selection substantially influences cold start behavior. Java functions exhibit the longest initialization times, often requiring several seconds due to JVM startup overhead and class loading requirements. Python and Node.js functions typically initialize within hundreds of milliseconds, making them more suitable for latency-sensitive applications.

AWS Lambda SnapStart significantly improves Java performance by capturing and restoring runtime state, effectively bypassing traditional initialization overhead. Without SnapStart, Java functions experience multi-second delays. With SnapStart enabled, startup times typically drop below one second, bringing Java performance comparable to traditionally faster runtimes.

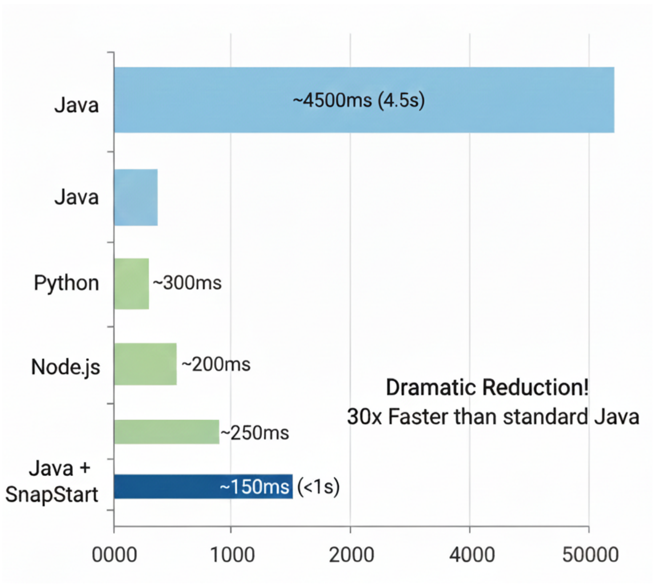

Figure 4: Runtime Cold Start Comparison

5.2.5 Cross-Platform Performance Variations

Cold start behavior varies across cloud service providers even when controlling for runtime and resource allocation. Deploying identical functions with equivalent configurations to AWS, Azure, and Google Cloud Platform reveals measurable startup time differences. These variations reflect fundamental differences in platform architecture, resource provisioning systems, and initialization processes, underscoring the importance of platform-specific optimization and testing.

5.2.6 Memory Allocation Effects

Memory allocation serves as a critical performance parameter in serverless environments. Increasing allocated memory generally reduces initialization latency, with Java functions showing particularly pronounced benefits. This occurs because cloud providers typically allocate CPU resources proportionally to memory configuration, providing additional computational power that accelerates runtime initialization, dependency loading, and class instantiation.

The relationship between memory and performance exhibits threshold effects. Initial memory reductions produce proportional increases in latency and execution time. However, beyond certain thresholds, performance degradation becomes exponential rather than linear, suggesting potential over-optimization for capacity at the expense of latency and execution performance.

5.2.7 Processor Architecture Considerations

Serverless functions present challenges for modern CPU optimization features. Branch prediction mechanisms, which improve performance by anticipating execution paths, struggle with short-lived functions that complete before predictors establish accurate behavioral patterns. The brief lifespan of serverless functions disrupts micro-architectural features relying on temporal locality. Current branch predictors exhibit elevated error rates for short functions, motivating research into fast-training algorithms or systems that preserve prediction state across invocations.

Last Level Cache performance shows nonlinear characteristics. Research indicates that restricting LLC size produces measurable performance effects, with substantial degradation occurring below approximately 2 megabytes. Above this threshold, LLC does not constitute a bottleneck for typical serverless workloads, suggesting most functions operate effectively within standard cache allocations.

5.2.8 Multi-Tenancy Interference Effects

Serverless platforms experience significant interference in multi-tenant environments. When a function operates above 70% capacity, repeated invocation of another function—approximately 20 invocations within 4 seconds—produces sustained performance degradation. This demonstrates how resource contention in shared infrastructure impacts function performance beyond direct cold start considerations, highlighting complex interactions in production environments.

5.2.9 Cost-Performance Trade-offs

Cold start mitigation strategies involve varying cost implications. Provisioned concurrency nearly eliminates cold starts by maintaining continuously warm function instances, but substantially increases operational expenses since resources remain allocated regardless of utilization. This approach proves prohibitive for cost-sensitive deployments or functions with unpredictable invocation patterns.

Alternative approaches provide more favorable cost-performance balances. Technologies such as SnapStart and container optimization deliver latency reductions of 3-4x without dramatically increasing costs. These methods improve performance through efficient initialization rather than continuous resource reservation, making them more practical for production deployments where both performance and budget constraints apply.

6. Discussion

6.1 Analysis and Interpretation

The findings indicate that multi-level optimization strategies substantially reduce cold start latency in serverless platforms. The *REAP* mechanism delivered approximately four-fold improvements in startup time versus baseline snapshot methods, underscoring the value of predictive memory page prefetching and parallel fault handling. *Function Fusion* eliminated the initialization overhead for consecutive functions in linear workflows; however, in fan-out patterns, fusion can diminish parallelism and may degrade end-to-end latency if the lost parallelism outweighs the removed cold-start penalty.

Regarding deployment, container-based packaging outperformed ZIP archives for functions with larger dependency sets (e.g., packages exceeding 30 MB), primarily because containers avoid decompression and leverage optimized base images. Language-specific results confirmed higher initialization overheads for Java due to JVM startup and class loading; nevertheless, enabling SnapStart reduced Java cold starts to sub-second ranges, bringing performance closer to Python and Node.js for latency-sensitive workloads.

6.2 Implications of the Findings

Practically, systems that require near-instant responses (e.g., interactive web services or financial workloads) can leverage SnapStart or provisioned concurrency to suppress cold starts, accepting the cost premium associated with keeping environments pre-initialized. Workloads with bursty or unpredictable arrival patterns can instead capitalize on memory-aware techniques (e.g., REAP) and container-based deployments to reach better cost–latency trade-offs without permanently reserving capacity.

The observed memory–latency relationship suggests careful configuration of memory (and hence proportional vCPU share) to avoid the non-linear degradation that appears at low allocations. Provider-to-provider variability further implies that engineering teams should test and tune on the target platform (AWS, Azure, GCP) rather than generalize results across clouds, as differences in provisioning pipelines, isolation mechanisms, and runtime bootstrapping can materially affect cold-start behavior.

6.3 Limitations of the Study

Several limitations remain. First, some mitigations (e.g., SnapStart, provisioned concurrency) are provider-specific, which constrains portability and may lock designs to particular cloud ecosystems. Second, fusion introduces coordination complexity and can harm workloads that depend on parallel fan-out; its benefits are contingent on workflow shape and data dependencies. Third, the evaluation emphasizes major providers and common runtimes; edge scenarios, emerging platforms, and specialized languages were not comprehensively assessed. Finally, while cost is discussed qualitatively, a rigorous economic analysis across configurations, diurnal traffic patterns, and regional pricing remains future work.

6. Conclusion

The systematic analysis of serverless computing reveals that while the paradigm offers significant advantages in scalability and operational overhead, cold start latency remains a critical bottleneck affecting Quality of Service (QoS) metrics such as latency, throughput, and user experience. This review highlights that cold start latency is influenced by multiple factors, including the choice of runtime environment, memory allocation, and deployment methods. Specifically, compiled languages like Java exhibit significantly higher latency compared to interpreted languages like Python, although optimization techniques like AWS SnapStart can reduce Java cold starts by approximately 91 percent.

Current mitigation strategies were categorized into latency reduction techniques, such as optimized caching and application-level optimizations, and frequency reduction techniques, such as periodic pinging and AI-based pre-warming. Notable findings challenge conventional deployment wisdom; empirical evidence suggests that container-based deployments outperform traditional ZIP packages for functions exceeding 30MB, achieving 20-40 percent better performance in those scenarios. Furthermore, hybrid optimization frameworks that combine runtime selection, container optimization, and intelligent pre- warming have shown to achieve a 75 percent aggregate reduction in cold starts. Despite these advancements, trade-offs remain regarding resource wastage, cost, and implementation complexity, necessitating a balanced approach based on specific application requirements.

7. Future Work

Based on the open challenges identified in the literature, future research should focus on the following key directions:

- Energy-Aware Optimization: Current pre-warming and keep-alive strategies often result in unnecessary resource consumption. Future research must develop energy-sensitive simulators and ”green” algorithms that minimize the carbon footprint of keeping containers idle without compromising latency.

- Advanced AI/ML Prediction Models: While AI and Deep Learning (DL) models are used for workload prediction, their accuracy varies with sporadic traffic and noisy datasets. Future work should explore Transfer Learning techniques to improve model performance on small datasets and develop open-source cold start datasets to foster reproducibility.

- New Isolation Architectures and WebAssembly: Lightweight virtualization technologies like microVMs (e.g., Firecracker) have improved isolation, but emerging technologies like WebAssembly (Wasm) offer sub-millisecond startup times. Further investigation is needed into Wasm-based serverless platforms and their integration with edge computing environments to minimize initialization overhead.

- Security in Optimization: Techniques such as container reusing and function fusion raise security concerns regarding data sanitization and inter-function isolation. Future frameworks must address these vulnerabilities to ensure that performance optimizations do not compromise the security integrity of the serverless platform.

- Function Fusion for Parallel Execution: While combining sequential functions has proven effective in reducing cold starts, developing fusion methodologies for parallel functions remains a complex challenge that requires further exploration to handle errors and execution dependencies effectively.

References

- 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 Computing Surveys, vol. 57, no. 3, Article 65, Nov. 2024.

- S. Koirala, ”Cold Start Performance in Serverless Computing: A Comprehensive Cross-Provider Analysis of Language-Specific Optimizations and Container-Based Mitigation Strategies,” Department of Computer Science and Engineering, Quantitative Computer Architecture, 2025.

- H. Gao, ”Optimization Research and Solutions for the Cold Start Problem in Serverless Computing,” Highlights in Science, Engineering and Technology, vol. 85, pp. 113-120, 2024.