Cold start performance optimization in serverless computing platforms

Chenye Kang

{ckang009@mymail.edu.sim.sg}

School of Information Technology, Singapore Institute of Management (SIM), Singapore

Abstract. Serverless Computing, through its Function as a Service (FPS) model, delegates infrastructure management, resource scaling, and runtime environment maintenance to the cloud platform, reducing application deployment and operational complexity. However, creating execution environments on demand introduces cold start latency and significantly impacts tail latency for interactive services, real-time data processing, and event-driven applications. This paper analyzes the main influencing factors related to the formation mechanism and optimization path of cold starts, focusing on code download, instance startup, runtime initialization, and user code initialization. It further summarizes methods such as container reuse, snapshot recovery, warm-up and elastic scaling, intelligent scheduling, and machine learning-based instance lifecycle management, comparing their differences in startup speed, resource overhead, implementation complexity, and load adaptability. The paper also illustrates the engineering implementation in a real-world platform using Alibaba Cloud Function Compute's instance pre-allocation, minimum instance count, concurrency configuration, and elasticity strategies. Analysis shows that a single method typically only optimizes a localized part of the cold start chain. For low-latency and cost-sensitive scenarios, a more effective solution is to dynamically combine multiple optimization strategies based on function runtime, dependency scale, call cycle, and business service level.

Keywords: Serverless Computations, Cold start time, FaaS Cloud Performance & Scheduling Optimization.

1 Introduction

Serverless Computing is an important form of cloud computing's evolution towards fine-grained resource abstraction and event-driven execution. In the typical Function as a Service (FaaS) model, developers mainly focus on function logic, triggering conditions, and dependency configuration, while instance creation, resource allocation, scaling up and down, and fault recovery are uniformly

ICIAAI 2026, May 22-24, Singapore, Singapore

DOI 10.4108/eai.22-5-2026.2365240

completed by the cloud platform. Compared with long-term resident virtual machines or container services, Serverless can dynamically allocate computing resources based on the arrival of requests and charge according to actual execution time or resource consumption, thereby reducing idle resource occupation and infrastructure maintenance costs [1]. Platforms such as AWS Lambda, Azure Functions, Google Cloud Functions, and Alibaba Cloud Function Compute have formed a relatively complete event-driven service system. The highly elastic resource management mechanism also brings about the cold start problem. When a function has not been called for a long time, the platform reclaims idle instances, or a sudden surge in traffic exceeds the capacity of existing instances, new requests may trigger the creation of a new execution environment. Cold start usually includes steps such as code or image acquisition, instance or container startup, runtime loading, dependency initialization, and user code initialization [2]. These steps increase the response time of the first request and may amplify tail latency such as P95 and P99. For latency-sensitive tasks such as Web APIs, online inference, real-time stream processing, and IoT event response, even with low average latency, occasional cold starts can impact service level objectives [3].

Existing research has developed two main approaches. One directly shortens the initialization chain by reducing environment build time through container reuse, runtime sharing, lightweight images, and snapshot recovery; the other reduces the probability of cold starts by increasing the proportion of available hot instances through warm-up, predictive scaling, instance keep-alive, and intelligent scheduling [4-6]. These two approaches have different trade-offs regarding resource cost, isolation strength, prediction accuracy, and implementation complexity. Based on this, this paper classifies and compares the main optimization techniques starting from the formation mechanism of cold starts, and discusses the applicable conditions of different methods in conjunction with actual cloud platform configurations to form a clearer performance optimization path.

2 Cold Start Formation Mechanism and Key Influencing Factors

2.1 Cold Start Execution Chain

Serverless functions typically go through stages from request to actual execution, including scheduling, resource preparation, runtime environment startup, and function initialization. If a reusable hot instance exists, the request can directly enter the execution stage; if the platform needs to create a new instance, a cold start will be triggered. Taking a typical cloud function platform as an example, the cold start overhead can be broken down into code download or image pull, instance or container creation, language runtime initialization, dependency loading, and user initialization logic execution. Different platforms differ in isolation technology and resource pool design, but the above stages constitute the main source of cold start latency.

2.2 Runtime, Dependencies, and Image Scale

Programming language and runtime characteristics significantly affect initialization time. Runtimes that require loading virtual machines, libraries, or complex dependency graphs typically have higher startup costs, while lightweight runtimes have relatively shorter initialization chains. The scale of

function packages, container images, and third-party dependencies also directly determines the download, decompression, and loading overhead. The original text uses the startup differences between Java and Go/Node.js as representative factors, a judgment largely consistent with current cloud platform optimization recommendations. In engineering practice, reducing irrelevant dependencies, simplifying image layers, and avoiding loading unnecessary resources during initialization are low-cost and widely applicable optimization methods.

2.3 Request Patterns and Resource Configuration

The probability of a cold start is closely related to the request arrival pattern. Continuous and stable requests help maintain hot instances, while prolonged idle periods, low-frequency triggering, and sudden concurrency are more likely to lead to instance reclamation or scaling, thus triggering new cold starts. Resource specifications also affect the execution speed during the initialization phase. Some platforms increase available CPU resources with memory configuration; therefore, appropriately increasing function specifications can shorten initialization time but increases the cost per execution. Instance concurrency capacity determines the number of requests a hot instance can handle simultaneously. Increasing appropriate concurrency can reduce the frequency of new instance creation, but it is necessary to ensure function thread safety, resource consumption, and downstream connection capacity to support concurrent execution.

3 Cold Start Optimization Techniques

3.1 Container Reuse and Runtime Sharing

Container reuse is the most basic way to mitigate cold starts. After a function finishes execution, the platform retains its execution environment for a certain period of time, allowing subsequent requests to directly use the initialized instance. This method does not require changes to the function logic, is easy to integrate with existing Serverless architectures, and can significantly reduce the number of startups in high-frequency call scenarios. Its limitation is that its effectiveness depends on the instance retention strategy and request intervals. When idle time is too long or sudden traffic exceeds the existing capacity, the platform still needs to create new instances. Further runtime sharing methods reduce repeated initialization by reusing the language runtime environment, basic dependencies, or underlying isolated resources, which can improve resource utilization, but also place higher demands on multi-tenant isolation and resource management.

3.2 Snapshot Recovery

The core of snapshot technology is to save the already initialized execution environment as a recoverable state, and directly restore the snapshot when a new request arrives, instead of re-executing the entire startup process. Works such as SEUSS and Catalyzer show that snapshot and initialization elimination mechanisms can reduce the startup time in some Serverless scenarios to milliseconds or even lower [5-7]. This method is suitable for functions with heavy runtime initialization, high dependency loading costs, and sensitivity to first-packet latency. Its main costs come from snapshot storage, version management, recovery consistency, and isolation mechanisms.

When the function depends on external connections, random states, or non-serializable resources, additional handling of post-recovery state reconstruction is required.

3.3 Warming Up and Predictive Scaling

Warming up strategies shift some of the cold start cost to before requests arrive by creating and maintaining a certain number of instances in advance. Fixed warming up can directly improve tail latency but increases idle resource consumption. Predictive scaling further utilizes historical call volumes, time periods, and operational metrics to estimate future load and dynamically adjusts the number of warmed-up instances accordingly. For businesses with obvious diurnal cycles, weekday patterns, or fixed batch processing windows, predictive scaling can achieve a good balance between resource cost and response performance. Its performance is highly dependent on prediction accuracy. Over-prediction can lead to resource waste when facing sudden events, load shifts, or irregular traffic, while under-prediction may still result in cold starts.

3.4 Intelligent Scheduling and Adaptive Instance Lifecycle Management

Intelligent scheduling focuses on the matching relationship between requests and existing hot instances. The scheduler can comprehensively consider functional dependencies, instance state, concurrency utilization, runtime, and node load to prioritize allocating requests to instances that can be executed directly, thereby improving the reuse rate of hot instances. In recent years, reinforcement learning and other methods have been used to determine instance keep-alive time, scaling up and down timing, and resource allocation, enabling lifecycle strategies to continuously adjust according to load changes [8,9]. These methods have strong dynamic adaptability, but training stability, state space design, policy interpretability, and online trial-and-error costs remain key issues that need to be controlled in actual deployment.

4. Method Comparison and Strategy Selection

Table 1. Comparison of Serverless Cold Start Optimization Methods

Method	Core Mechanism	Startup Latency Improvement	Resource Overhead	Implementation Complexity	Applicable Scenarios
Container Reuse	Retain and reuse initialized instances	Moderate	Low to Moderate	Low	Stable or high-frequency workloads
Snapshot Restoration	Restore a pre-initialized runtime environment	High	Moderate	Moderate to High	Heavy runtimes and latency-sensitive services
Fixed Prewarming	Keep instances ready in advance	High	High	Low	Predictable capacity and

					strict SLA scenarios
Predictive Scaling	Scale out in advance based on historical data and runtime metrics	Moderate to High	Moderate	Moderate	Periodic or trend-driven workloads
Intelligent Scheduling / Reinforcement Learning	Dynamically reuse warm instances and optimize instance lifecycle management.	Moderate to High	Dynamically Adjustable	High	Complex workloads and large-scale resource pools

Table 1 shows the comparison of serverless cold start optimization methods. There is no single optimal solution for different technologies. Container reuse is simple to implement and suitable as a platform's default mechanism; snapshot restoration directly compresses the initialization path, making it more suitable for functions with high startup costs and latency sensitivity; warm-up and predictive scaling reduce the probability of cold starts by improving the availability of hot instances, but incur additional resource costs; intelligent scheduling and reinforcement learning can dynamically optimize based on real-time status, making them more suitable for large-scale platforms with significant load variations. Real-world systems typically employ a combined strategy, such as using container reuse as a foundation to ensure stable capacity for critical services with a minimum number of instances, then combining timed or metric-driven scaling to handle predictable peaks, and finally using a scheduler to improve the efficiency of hot instance utilization.

Strategy selection should consider at least four dimensions. First, function initialization cost, including image size, runtime, and dependency loading time; second, load predictability, including call frequency, periodicity, and burstiness; third, service level objectives, especially tail latency constraints such as P95 and P99; and fourth, resource budget. For low-frequency but extremely latency-sensitive functions, fixed warm-up or snapshot recovery is more valuable; for request-intensive and high-concurrency businesses, increasing instance concurrency and hot instance reuse rate usually has better cost-effectiveness; for traffic with obvious periodicity, it is suitable to combine timed scaling with predictive models.

5 Alibaba Cloud Function Compute Cold Start Optimization Case

The latest documentation for Alibaba Cloud Function Compute defines cold start as the process of preparing the execution environment and code, including code download, function instance container startup, runtime initialization, and user code initialization. The platform recommends reducing the impact of cold start from multiple levels, including code packages, runtime, resource specifications, and instance strategies [10]. For example, deleting irrelevant dependencies and

redundant files can reduce code download and decompression time; for runtimes with heavy initialization, long-tail latency can be reduced by choosing a lighter language or optimizing startup logic; under fixed concurrency conditions, increasing memory specifications can obtain more CPU resources, thereby speeding up initialization.

For cold starts that are difficult to eliminate on the business side, Function Compute provides minimum instance count and elasticity strategies. By setting the minimum number of instances to a value greater than 0, the platform pre-allocates instances, allowing some requests to directly enter the prepared execution environment. Simultaneously, the minimum number of instances can be dynamically adjusted based on time schedules or metric thresholds to cover known peak periods and changes in operational status. This method significantly reduces the probability of cold starts for latency-sensitive services; however, idle pre-allocated instances still incur resource costs, thus requiring a combination of business peaks/valleys and service level targets to determine the scale.

Instance concurrency configuration reduces the number of instances created from another perspective. When a single instance is allowed to handle multiple requests concurrently, the platform only needs to create new instances after existing instances have reached their concurrency limits, thus reducing the number of cold starts and improving resource utilization for I/O-intensive functions. Therefore, cold start optimization in actual platforms forms a multi-layered combination of "lightweight code and dependencies—instance pre-allocation—concurrency reuse—elastic strategies." This model corresponds to the initialization optimization and cold start frequency control in the aforementioned research, also indicating that engineering systems place greater emphasis on the synergy of performance, cost, and resource utilization.

6 Development Trends and Key Challenges

Cold start optimization is evolving from single-point acceleration to cross-layer collaboration. First, snapshots, lightweight isolation, and runtime sharing will continue to reduce environment startup costs and form closer synergy with language runtimes, container images, and hardware virtualization. Second, predictive scaling will increasingly combine online monitoring and adaptive control, continuously updating instance capacity based on changes in call patterns, rather than relying on fixed historical windows. Third, as AI inference, GPU functions, and large model services enter Serverless scenarios, model file loading, GPU memory initialization, and dependency recovery will significantly amplify cold start overhead, making resource reservation and model caching new optimization priorities.

Simultaneously, optimization strategies need to pay more attention to cost constraints and interpretability. While excessive warming can reduce latency, it weakens the advantage of Serverless's on-demand resource usage; overly complex prediction and reinforcement learning strategies may increase the computational burden on the platform's control plane. A more feasible future direction is to establish a multi-objective optimization framework oriented towards service level goals, dynamically selecting container reuse, snapshots, warming, and scheduling strategies based on function startup profiles, load patterns, and resource prices, and continuously adjusting

them during runtime. Edge computing and IoT scenarios will also introduce factors such as heterogeneous devices, network instability, and limited node capacity, requiring cold start management to extend from the central cloud to distributed edge resource pools.

7 Conclusion

Cold start is a critical performance issue in Serverless Computing under highly elastic and on-demand resource allocation mechanisms, with its impact concentrated in the initial request and high-quantile tail latency. Cold start latency is comprised of multiple stages, including code and image acquisition, instance startup, runtime loading, dependency initialization, and user code initialization, and is influenced by factors such as programming language, dependency scale, request patterns, instance concurrency, and resource specifications. Existing optimization methods can be summarized into two categories: shortening the initialization path and reducing the probability of cold starts. Container reuse, snapshot recovery, preheating and predictive scaling, intelligent scheduling, and adaptive lifecycle management each correspond to different performance and resource trade-offs.

Comprehensive research and cloud platform practice show that a combined approach is more suitable for cold start optimization geared towards actual business needs. By reducing basic startup costs through lightweight function packages, ensuring high-latency sensitive functions through snapshots or preheating, matching periodic loads with elastic scaling, and improving hot instance reuse rates through concurrent configuration and intelligent scheduling, a more stable balance can be achieved between response time, resource utilization, and operating costs. As Serverless extends to AI inference, edge computing, and heterogeneous resources, research on cross-layer collaboration, adaptive strategy selection, and multi-objective resource management will become an important direction for performance optimization.

References

[1] Li, Z., Guo, L., Cheng, J., et al.: The serverless computing survey: A technical primer for design architecture. ACM Computing Surveys (CSUR), (2022), 54(10s): 1-34

[2] Wang, L., Li, M., Zhang, Y., et al.: Peeking behind the curtains of serverless platforms. 2018 USENIX Annual Technical Conference (USENIX ATC 18), (2018): 133-146

[3] McGrath, G., Brenner, P. R.: Serverless computing: Design, implementation, and performance. 2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW), IEEE, (2017): 405-410

[4] Mohan, A., Sane, H., Doshi, K., et al.: Agile cold starts for scalable serverless. 11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19), (2019)

[5] Du, D., Yu, T., Xia, Y., et al.: Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, (2020): 467-481

[6] Agarwal, S., Rodriguez, M. A., Buyya, R.: A reinforcement learning approach to reduce serverless function cold start frequency. 2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid), IEEE, (2021): 797-803

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

[8] Golec, M., Walia, G. K., Kumar, M., et al.: Cold start latency in serverless computing: A systematic review, taxonomy, and future directions. ACM Computing Surveys, (2024), 57(3): 1-36

[9] Mampage, A., Karunasekera, S., Buyya, R.: A deep reinforcement learning based algorithm for time and cost optimized scaling of serverless applications. Future Generation Computer Systems, (2025), 173: 107873

[10] Alibaba Cloud.: Best Practices for Optimizing Cold Starts in Function Compute [EB/OL]. Alibaba Cloud, (2026-06-26)