Advances in Brain-Computer Interfaces and Neural Integration

PRIME OPEN ACCESS

Volume 2, Issue 1

Research Article

Date of Submission: 08 Apr, 2026
Date of Acceptance: 07 May, 2026
Date of Publication: 14 May, 2026

Seminar Cloud Computing Cold Start Problem in Serverless Computing

Dincer Atasoy*

Igdir University, Turkey

***Corresponding Author:** Dincer Atasoy, Igdir University, Turkey.

Citation: Atasoy, D. (2026). Seminar Cloud Computing Cold Start Problem in Serverless Computing. *Adv Brain-Computer Interfaces Neural Integr, 2*(1), 01-10.

Abstract

This study examines the 'cold start problem' in serverless computing. Serverless computing is known for its ability to provide services on-demand and cost-effectively. However, it faces a challenge known as cold starts. This occurs when there is a delay in starting serverless functions after they have not been used for some time. This delay can affect the performance and user experience, especially in applications where time is critical. We look at different solutions offered by commercial platforms and academic research. A key focus is on the FaasCache method, which uses a caching technique to keep functions ready and reduce cold starts. We compare this with other methods like pre-warming functions, using different types of servers, and predicting usage patterns with AI. The paper also discusses new challenges, especially when using AI applications with serverless computing, which have their own issues with resource use and start-up times.

Keywords: Serverless Computing, Cold Start Problem, faaS Optimization Techniques

Introduction

Serverless computing, also known as Function as a Service (FaaS), is a cloud computing execution model where the cloud provider dynamically manages the allocation and provisioning of servers. In serverless computing, developers write and deploy code, and the cloud provider handles the underlying infrastructure, scaling, and management tasks. This model allows developers to focus on writing code without worrying about the server and infrastructure management. It operates on a pay-per-use principle, leading to cost efficiencies, particularly in scenarios where compute needs are only required for discrete intervals [1].

There are four main advantages of serverless computing which make it very convenient to use.

- **Cost-Efficiency:** Serverless computing uses a pay-as-you-go model, meaning you only pay for the compute time you consume. This can result in lower costs compared to traditional cloud service models.

- **Scalability:** Serverless architectures automatically scale with the application's needs, handling increases in traffic or workload without manual intervention [2].

- **Developer Productivity:** Developers can focus on writing code and developing applications, as the main-tenance of servers and infrastructure is handled by the cloud provider [3].

- **Quick Deployments and Updates:** Deployment of applications is quicker in a serverless environment as the infrastructure setup is minimal [3].

Even if it has many advantages, it also has major problems and challenges that current researchers try to solve. The most straightforward problem is the limited control. It gives developers limited control over the underlying infrastructure, which can be a limitation for complex applications with specific requirements. In addition, there are concerns about security and vendor lock-in with applications potentially becoming overly reliant on a single provider's ecosystem and

underlying infrastructure being shared by other users which potentially can be a soft spot for the security of an application [4].

Apart from these challenges, the biggest problem in serverless computing is the cold start problem. It occurs when a function is inactive for a certain period and then enters a 'cold' state, necessitating reinitialization of the container and the function. This process involves several key phases [5]:

• **System Communication:** Initially, the serverless platform must allocate resources for the function. This involves communication between the serverless provider's management system and the underlying infras-tructure to provision and allocate necessary resources.

• **Container Runtime Preparation:** Most serverless platforms use containers to isolate and execute functions. During a cold start, a container must be instantiated. This includes setting up the container's runtime environment, which involves loading and configuring the necessary runtime libraries and dependencies for the function.

• **User Process Initialization:** Once the container is ready, the user's function code is loaded into the con-tainer. This phase includes the initialization of the function's runtime environment (like Node.js, Python, etc.), compiling or interpreting the code, and executing any initialization code defined in the function.

• **Data Loading:** If the function requires external data or needs to connect to databases or other services, this phase involves establishing those connections and loading necessary data. This can include authentication processes, API calls, and reading from or writing to databases.

Each of these phases contributes to the total initialization time of a function, known as the cold start latency. Understanding and optimizing these phases are crucial for improving the performance of serverless applications, especially for time-sensitive operations [5].

Solving the cold start problem is crucial for maintaining the high performance and scalability promised by serverless architectures [3,4]. As serverless computing continues to gain popularity due to its cost-efficiency and scalability advantages, ensuring minimal latency during function initialization becomes necessary for a seamless user experience. This is especially important in scenarios where serverless functions are expected to handle unpredictable traffic, as any delay in response can lead to user dissatisfaction and potential loss of service reliability.

Efficiently resolving the cold start issue is not only fundamental to maintaining the operational effectiveness of serverless computing but also to its broader adoption across various industry sectors. Especially for AI-based applications, which are becoming more and more common, solving the cold start problem is important. It can help them to create easily scalable and maintainable systems without worrying about the underlying infrastructure.

To tackle the cold start problem, there are many state-of-art researches. The paper called FaasCache addresses the cold-start problem by drawing an insightful analogy between the resource management of functions in serverless computing and object caching [6]. The authors propose a novel approach, utilizing the Greedy-Dual caching framework, to develop keep-alive policies. These policies are designed to effectively reduce cold-start overheads while optimizing server resource utilization and function latency. The significant contributions of the paper include demonstrating the parallel between caching and function keep-alive, and implementing caching-based techniques in the FaasCache system. The intuitive correlation between caching and function keep-alive makes this paper especially important because we can make many contributions to the serverless computing area by leveraging the power of significant research done in the field of caching.

In the following sections, we discuss:
• Current solutions to solve the cold start problem from commercial serverless platforms as well as novel research from academia
• Detailed discussion of FaasCache paper and its comparison with the other solutions [6].
• New challenges for the cold start problem are aroused by the increasing popularity of AI-based applications.

Related Works

Solving the cold start problem also comes with many challenges. The first challenge is finding a way to predict the invocations of the functions. It is not an easy task to do since usually, the functionless service platforms have fluctuating frequencies depending on the time of day as well as the current workload of the platform. The other challenge is understanding its heterogeneous workflow and coming up with a solution to scale the system to the different application demands such as web-hosting to machine learning inferences. In addition, the serverless platform also needs to optimize the container/VM initializations according to heterogeneous workflows.

There are state-of-the-art research and solutions from academia and commercial serverless platforms. They are discussed in this section.

Solutions from Commercial Serverless Platforms

It is crucial for cloud providers to solve the cold start problem. They can both decrease the latency they provide to the users and they can decrease their costs by utilizing their resources efficiently. In this section, we investigate how they solve the problem.

Amazon Lambda

Amazon has recently launched a feature called SnapStart [7]. This feature greatly reduces the time it takes for Java-based Lambda functions to start up, known as ”cold start” times. SnapStart works by creating a secure snapshot of the function’s environment when the function version is released. This snapshot is saved for fast access. When the function is called, Lambda uses this snapshot to start the function environment quickly, instead of setting it up from the beginning. These snapshots stay ready for use for up to 14 days of not being used. After this period, they are removed. If a function version isn’t active when it’s called, the call won’t work, and Lambda will set up a new snapshot. You can set up SnapStart with tools like AWS SDK, AWS CloudFormation, AWS SAM, and CDK. This method is similar to recent research in this area, which will be discussed in the following section 2.2 [8,9].

Google Cloud Functions

Google Cloud Functions, similar to other serverless platforms, also deal with the problem of cold starts. Their solution is quite simple. Once the first request has been handled, the system keeps the instance running so it can be used again for any following requests. These instances are then recycled if they’re not used for 15 minutes. This is a change from Google’s previous method where they would keep idle instances active for a longer time [10,11].

Azure Function

Azure also came up with a solution that is not complex but can improve efficiency a lot. To improve cold start times, Azure maintains a pool of warm workers [12]. These workers have the Functions runtime already running but are unspecialized. When a function is triggered, Azure allocates a preconfigured server from this pool and specializes it for the specific app, significantly improving cold start times. The method they used is also a widely known solution and used in various works [13].

Solutions from Recent Novel Research

The recent advancements made in addressing the cold start problem, are categorized into two main areas:

- Reducing the start-up time of functions [8,9,14]
- Reducing the frequency of function cold starts [15-18]

Reducing the Start-Up Time of Functions

Many studies have looked at how to make the start-up of serverless functions faster. They are explained in this section.

Prebaking Functions to Warm the Serverless Cold Start

One well-known study, ”Prebaking Functions to Warm the Serverless Cold Start” introduces a new idea called ’prebaking.’ In prebaking, they take pictures (snapshots) of the serverless function while it’s running, at just the right time [9]. Then, they use these snapshots later when the function needs to run again. This method skips a lot of the slow steps that usually happen when a function starts, like setting everything up and getting it ready to run. This is especially helpful for functions that usually take a long time to get ready.

Taking these snapshots involves three steps. First, they run the function in the normal way [9]. This first run does all the usual things like getting the resources ready, setting up the environment, and loading any needed files. Then, at the best time during this run—after the function is all set up but before it starts doing its specific job—they take a snapshot of what the function is doing. This snapshot is made using a tool called Checkpoint/Restore In Userspace (CRIU), which captures everything about the function at that moment, like what’s in its memory and its network connections. Lastly, they keep these snapshots and use them for later runs of the same function. Instead of starting from the beginning each time, they use these pre-made snapshots. This way, they avoid a lot of the slow steps and make the function start faster.

Catalyzer

”Catalyzer” is another new study that looks at how to make serverless platforms start faster [8]. The paper employs a very similar approach to Prebaking where the state of a running sandbox is saved into a checkpoint image, including both the application state and the sandbox state [9]. However, in Catalyzer VM-based sandboxing is used by extending gVisor implementation [8,19].

There are three ways Catalyzer starts a sandbox. The first is called ’Cold Boot,’ where they create a new sandbox from the saved image [8]. The second, ’Warm Boot,’ uses already running sandboxes to make starting new ones faster. The third way, ’Fork Boot,’ uses a special sandbox that’s already set up to quickly start as many new instances as needed. They do this using a new system tool called ’sfork.’

Another interesting thing Catalyzer found is that serverless functions, when they run, only use a small part of the memory and files they needed when they first started [8]. Using this insight, Catalyzer only brings back the essential

parts of the application and system when they are needed, instead of loading everything all at once. This method, where it loads different parts separately, has been really effective. It made starting up a Python Django system 6.3 times faster and a Java SPECjbb system 7 times faster. This shows how Catalyzer's work can really speed up how fast serverless systems start.

FaaSLight

Another study that's working on the cold start problem is "FaaSLight" [14]. This research aims to make serverless functions start faster by focusing on the time it takes to load the application code. FaaSLight's main idea is to separate and only load 'optional functions' when they're needed. 'Optional functions' are parts of the code that aren't always necessary right from the start. By doing this, FaaSLight makes sure that only the most essential parts of the code are loaded first, which helps reduce the time it takes for a function to start.

FaaSLight uses two key strategies: The first one is called 'Static Analysis-Based Technique [14].' This strategy looks at the code and figures out which parts are essential and need to be ready right away. FaaSLight tries to identify as many of these essential parts as possible. The second strategy is 'Conservative On-Demand Loading.' Instead of getting rid of optional functions, FaaSLight replaces them with a special loader that can bring them in when they're actually needed. This way, even if the system mistakenly thinks a function isn't necessary, it can still be loaded quickly if it turns out to be needed, ensuring the application works correctly.

FaaSLight's method is different from some other approaches, like Catalyzer because it doesn't depend on the type of platform or programming language used [8]. It works directly on the application level, which means it can be used with different programming languages and on popular serverless platforms like AWS Lambda and Google Cloud Functions without needing to change the system underneath. Tests have shown that FaaSLight can greatly reduce the time it takes to load code (by up to 78.95%, averaging 28.78%), which means the overall time for a function to start and respond can be reduced by up to 42.05% (averaging 19.21%). This makes it a really useful tool for improving the speed of serverless functions.

Reducing the Frequency of Function Cold Starts

IceBreaker

Another key method to tackle the cold start issue in serverless computing is to reduce how often these cold starts happen. A lot of studies are looking into this, and one recent paper, "IceBreaker" offers some interesting ideas [15]. The main point in "IceBreaker" is that we can't always predict when a function will be needed. Because of this, the paper suggests using different types of servers for different functions, depending on how likely they are to be needed soon. This means mixing more powerful (high-end) servers with less powerful (low-end) ones, making it more cost-effective to keep functions ready.

"IceBreaker" points out that the usual ways of predicting when functions will be used (like using ARIMA time-series or histogram-based predictions aren't great at handling quick changes or many functions being used at the same time [15,16]. To improve this, the paper introduces a new way to predict function use with something called Fast Fourier transformation. This new method works better than the old ones, especially in dealing with changing patterns of function use. The results from their tests show that "IceBreaker" can do a better job than previous methods, even beating them by up to 27% in some cases. This makes it a pretty significant step forward in managing serverless functions more effectively.

Serverless in the Wild

"Serverless in the Wild" is another important study that looks at reducing the number of cold starts in functionless servers [16]. The main idea of this paper is a new way to manage resources called the Hybrid Histogram Policy. This policy's goal is to lessen the chances of cold starts and use resources better. It does this by waiting for a certain amount of time, called a "pre-warming window," after the last time the application runs. The idea is to get ready for another use of the application during this time.

There are a few big challenges with this policy [16]. First, it's hard to guess when the functions will be used because their use can change a lot. Also, different applications need different things, and some are used only rarely. Another challenge is to keep track of each application's use in a way that doesn't cost too much. The Hybrid Histogram Policy tries to solve these problems by adapting to how often and in what pattern each application is used. After an application is used, the policy removes it to save resources. Then, before the next time it's likely to be used, the policy gets the application ready again. This keeps the application running for a while, so it's ready to use quickly. This way, the policy helps to reduce cold starts and makes sure resources are used well.

In their research the authors compare their hybrid policy with the fixed keep-alive policy, which is often used in serverless platforms [16]. They demonstrate that the hybrid approach greatly reduces cold starts and also uses memory resources more efficiently. For instance, a fixed policy of keeping instances active for 2 hours results in almost 30% more memory being wasted compared to a 10-minute baseline. In contrast, the hybrid policy, particularly when using a histogram-based approach, significantly cuts down on cold starts and uses much less memory. For example, a fixed

10-minute keep-alive policy leads to about 2.5 times more cold starts compared to the hybrid policy with a 4-hour range, even though they use the same amount of memory. Furthermore, the fixed 2-hour policy achieves a similar rate of cold starts as the 4-hour range of the hybrid policy, but it uses about 50% more resources.

Help Rather than Recycle

There's another study called "Help Rather Than Recycle" that looks at a different way to reduce cold starts in serverless computing [17]. This paper noticed that while some functions have to wait for their containers to start up (cold starts), other containers that are already running and ready (warm containers) are not being used much. So, the paper suggests an idea: why not share these warm containers between different functions? This way, the warm containers get used more, and overall, everything runs more efficiently.

The solution they propose is a special way of managing containers, called 'Pagurus.' Pagurus uses three kinds of containers, each with its own job to make sure everything runs smoothly and quickly. Private Containers: These are containers used by only one specific function. They are dedicated to that function and don't get shared with others.

Zygote Containers: These are basic containers that are set up but don't have any specific function's code or data in them yet. They have common things that many functions need and keep each function's information private. Helper Containers: These are made from zygote containers. When a specific function needs a container and there are no warm private containers available, a zygote container gets turned into a helper container for that function. This helps avoid the delay of starting a container from scratch.

Pagurus also uses a technique called the Similarity Filtered Weighted Random Sampling (SF-WRS) algorithm. This algorithm helps decide which functions should get help based on how similar their needs are and how often they face cold starts. They tested Pagurus with AWS and Azure functions and found it really effective. The tests showed that Pagurus could reduce the number of cold starts by 84.6% on average in Azure, bringing down the time it takes to start from hundreds of milliseconds to just 16 milliseconds.

Reinforcement Learning-Based Approach

Another recent work introduces an innovative approach to mitigate the cold start problem in serverless computing by leveraging an AI-based solution, specifically a Reinforcement Learning (Q-Learning) agent [18]. This agent is designed to analyze critical factors such as CPU utilization to predict function invocation patterns. The primary objective is to preemptively prepare function instances based on these identified patterns, thereby aiming to decrease the frequency of function cold starts.

The implementation of this Reinforcement Learning agent is integrated with the Kubeless serverless platform. It operates by discretizing environment states, actions, and rewards based on metrics like CPU utilization per instance, the availability of function instances, and the success or failure rate of responses. This method enables the agent to gradually learn and understand the invocation patterns, facilitating informed decision-making to optimize the required number of function instances over time within controlled environment settings.

The agent determines which functions are likely to be called next by using the invocation patterns it has learned. After that, the agent pre-warms these features [18]. Pre-warming is the process of getting the function instances ready so they don't have the usual cold start wait and can start immediately.

Upon evaluation, it is observed that the performance of the proposed solution is similar with the baseline model, which is the default auto-scale feature of Kubeless [18]. While this may initially appear as no significant improvement, the authors claim that the findings underscore the potential of Reinforcement Learning-based approaches in effectively addressing the cold start issue in serverless computing in the future. This perspective opens up avenues for further exploration and refinement of AI-driven solutions in this domain.

These contributions show significant improvements when dealing with the serverless computing cold start prob-lem. Each paper offers unique solutions and insights, collectively advancing our understanding and capability in optimizing serverless computing environments.

Main Paper Discussion: FaasCache

Background

Understanding the Function Keep-Alive Concept

In serverless computing, after a function is executed within a container, there's an option to keep the container active instead of shutting it down immediately. This approach, known as the function keep-alive strategy, allows for the reuse of the container for future executions of the same function. This method significantly cuts down on the delay typically associated with starting a container, known as the cold-start problem, which might take around 100 milliseconds.

Creating effective keep-alive policies is complex due to the vast differences in functions in terms of how often they are used, their resource demands, and the varying delays in starting them up. For example, a study of FaaS workloads

indicated that the time between function calls and the memory requirements could vary greatly [16]. This variability intensifies the balance that needs to be struck between performance and server utilization in keep-alive policies. Furthermore, the dynamic nature of FaaS workloads demands innovative methods for resource allocation and scalable solutions, which are also part of the policy development process.

This concept is also used by novel works discussed in Section 2 [15-18].

Caching

To tackle the challenges posed by maintaining function keep-alive and efficient resource provisioning, especially given the diverse and dynamic nature of workloads, they turn to the principles of caching [6]. Caching is a mature field that has successfully addressed similar challenges through sophisticated eviction algorithms that exploit temporal locality, such as the Least Recently Used (LRU) method and its variants [20,21].

Methodology

Reducing the Frequency of Function Cold Starts by Utilizing Caching Analogies

The concept of maintaining a function in a 'warm' state in serverless computing can be likened to the caching of an object in traditional caching systems. A 'warm' function execution parallels a cache hit in this analogy. By aligning the keep-alive strategy with well-established caching methodologies, they can effectively apply these principles to enhance serverless computing frameworks [6].

Key Factors in Policy Design

Developing an effective caching-based keep-alive policy requires considering several important factors:

- **Initialization Time:** The start-up time for functions varies, depending on their code and data dependencies.

- **Total Running Time:** This includes both the initialization and execution phases.

- **Resource Footprint:** This encompasses the CPU, memory, and I/O usage, which differ widely based on the application's needs.

Policy Explanation

The policy, inspired by the Greedy-Dual-Size-Frequency (GDSF) object caching concept is tailored for serverless computing. Unlike traditional caching policies like LRU or LFU, which do not account for object sizes, this policy considers the resource footprint as a key factor [22]. Essentially, the policy functions as a decision-making tool for terminating function containers. They assign a priority to each container, calculated based on factors like cold-start overhead and resource footprint.

Priority Calculation

The priority in the GDSF keep-alive policy is derived from the Greedy-Dual caching principle, accounting for varying eviction costs across containers [22]. The priority for each container is a function of the frequency of function invocation, its duration, and its size, calculated as follows:

$$\text{Priority} = \text{Clock} + (\text{Frequency} \times \text{Cost}) / \text{Size} \quad (1)$$

- **Clock:** This component captures the recency of execution, with a logical clock maintained per server, updated at each eviction. Each container usage updates this clock, thereby adjusting its priority.

- **Frequency:** This measures how often a function is invoked.

- **Cost:** This represents the termination cost, equivalent to the total initialization time.

- **Size:** This denotes the resource footprint of the container.

Optimizing Resource Allocation Using Server Provisioning Policies

In serverless computing, a key challenge is managing the trade-off between performance efficiency and the allocation of server resources. A well-designed resource provisioning algorithm is crucial for optimizing this balance. Resource provisioning involves deciding the size and capacity of servers to effectively handle FaaS workloads, taking into account the frequency of function calls, their resource requirements, and the time interval between function invocations.

Static Provisioning

A crucial aspect of their approach is determining the right server size to accommodate the workload, which directly impacts the effectiveness of the keep-alive policy. The server's capacity must be aligned with the workload requirements. They use cache hit-or-miss ratio curves as a model to understand and manage the server resources. These curves are a classic tool in caching, helping to predict how a cache performs at different sizes. By analyzing these curves, they

decided the most efficient server size for a particular workload. Essentially, they look for the point on the curve where increasing the cache size brings diminishing returns, indicating the most efficient use of resources.

Building Hit-Ratio Curves

To create these hit-ratio curves, they examine the reuse distances of functions. A function's reuse distance is the sum of the memory sizes of distinct functions called between two consecutive calls of the same function. Understanding these distances helps us gauge the required cache size to avoid misses. A cache size larger than these distances ensures efficient function management.

Elastic Dynamic Scaling

They also adapt their server resources dynamically with the workload changes. The policy uses hit-ratio curves to guide a dynamic scaling policy, adjusting server resources to match current demand. This dynamic scaling is based on a proportional control system [23]. It adjusts the memory size of the virtual machine in line with the rate of cold starts. In simpler terms, when fewer functions are being invoked, it reduces the server resources, and vice versa. This approach ensures that server resources are always in tune with actual usage, preventing both resource wastage and performance bottlenecks.

Limitations and Future Works

A notable limitation in applying caching strategies to FaaS is the difference in handling multiple containers for the same function, unlike unique objects in caching. This difference is especially seen at smaller cache sizes, leading to inaccuracies in hit-ratio estimations. Addressing these differences forms a key area for future exploration.

Their current provisioning approach, reliant on periodically updated hit-rate curves, isn't fully real-time. Ad-vancing towards online, adaptive hit-rate curves that can promptly respond to changing function characteristics is a primary focus for future enhancements.

Comparison with Other Works

FaasCache paper has many novel ideas to solve the cold start problem [6]. In our categorization, it falls under "Reducing the Frequency of Function Cold Starts" 2.2.2 category. Therefore, we first compare it with these works.

Comparison with Papers in Reducing the Frequency of Function Cold Starts Category

In addressing the cold start problem in serverless computing, several innovative approaches have been proposed. FaasCache paper with its intuitive use of caching analogies, stands out for its straightforward methodology, offering an easily implementable solution [6]. This contrasts with IceBreaker which introduces the concept of using high-end and low-end servers based on function demand [15]. While IceBreaker's idea is novel, integrating its idea of using high-end and low-end servers together strategy with the caching-based approach of FaasCache could potentially create a more balanced and cost-effective system since then FaasCache can help to IceBreaker to better predict about their priorities and then we can further increase our utilization.

Meanwhile, Serverless in the Wild suggests the Hybrid Histogram Policy, focusing on pre-warming windows to reduce cold starts [16]. Though this presents a proactive approach, it doesn't provide the same level of simplicity and intuitiveness found in FaasCache's caching methodology. The policy, while valuable, might be less efficient in practice as it is discussed in FaasCache paper [6].

On the other hand, Help Rather Than Recycle explores an inventive concept of reusing containers across functions [17]. This approach promises efficiency improvements but is hampered by its complexity, particularly in maintaining secure and isolated environments within shared containers. While innovative, these practical challenges make the straightforward approach of FaasCache more appealing for immediate application.

Lastly, the reinforcement learning-based approach marks a promising direction, leveraging AI to anticipate func-tion invocation patterns. Despite its potential, its current performance does not offer a substantial improvement over baseline models, positioning FaasCache as a more effective choice for the present. The AI-driven solution, however, holds significant promise for the future, as machine learning models continue to evolve.

Method/Platform	Strategy	Advantages	Limitations	Improvements
Amazon Lambda	SnapStart for Java functions	Rapid startup, secure snapshot saving	Limited to Java, snapshot expires after 14 days	Significant reduction in Java function startup
Google Cloud Functions	Keep instances running	Simplicity, reuses instances for following requests	Instances recycled after 15 mins of inactivity	Reduction in instance startup time
Azure Functions	Pool of warm workers	Quick allocation from preconfigured servers	May have resource efficiency concerns	Reduced cold start times for new invocations
Prebaking Functions	Snapshots of running state	Skips slow startup steps	Complexity in snapshot management	Faster function startup by skipping setup steps
Catalyzer	Snapshots of VM-based sandboxing	Fast startup, efficient memory/file usage	Complexity in sandboxing, VM-dependency	Up to 7 times faster startup for Java systems
FaaSLight	Load optional functions on-demand	Reduces load time, application-level optimization	May need fine-tuning for specific functions	Up to 42.05% reduction in function start time
IceBreaker	Heterogeneous server types	Cost-effective, adaptive resource allocation	Requires predictive modeling for function use	Up to 27% better performance in managing functions
Serverless in the Wild	Hybrid Histogram Policy	Resource efficient, pre-warming windows	Complexity in resource management, prediction	2.5 times fewer cold starts than fixed 10-minute keep-alive policy
Help Rather Than Recycle	Container sharing	Efficient resource use, reduces cold starts	Complexity in managing shared containers	84.6% reduction in cold starts on Azure
Reinforcement Learning	AI-driven prediction	Innovative, anticipates invocation patterns	Performance similar to baseline models	Comparable performance with potential for future improvements
FaasCache	Greedy-Dual Caching Framework	Reduces cold-start overheads, optimizes resource utilization	Requires management of function keep-alive policies	2x to 3x increase in warm-starts, 2× more requests served

Table

Possible Integrations with Papers in Reducing the Start-Up Time of Functions Category

Because the methodology of FaasCache significantly differs from the papers in this category 2.2.1, they cannot directly be compared. However, it is a good idea to integrate them together by using the methodology of FaasCache to increase the number of times cold start happens and use the methodologies of the other papers to make the start-up faster.

We can use the idea of checkpointing to decrease the time of starting the container discussed in Catalyzer and Prebaking papers [8,9]. Also, we can use the idea of analyzing the code to detect the unused parts and lazily load these parts both at the system level as discussed in Catalyzer and at the application level as discussed in FaasLight [8,14].

In this way, we can further create a better-utilized and more efficient solution for the cold start problem.

Comparison Table

There is a comparison of all discussed solutions for the cold start problem in Table 3.4.

New challenges

There are many state-of-art solutions for the current cold start problem and all of them have their own way of solving the problems, which has their own trade-offs. However, the cold start problem will see more challenges in the future, especially with the sudden increase of LLM-based applications. Currently, there are two problems in the area of functionless AI:

• Splitting GPU resources among multiple containers is challenging because GPUs are not inherently de-signed for fine-grained resource sharing. Unlike RAM, which can be easily partitioned and allocated in small chunks, GPUs are optimized for intensive, parallel processing tasks. Dividing their processing power into smaller, isolated segments for different containers disrupts their efficiency and can lead to resource contention, where multiple tasks compete for the same GPU resources, causing performance degradation. There are works that try to tackle this problem [24,25].

• GPU-based AI applications generally require more initialization time compared to other applications. This increased initialization time is due to the complex setup processes involved in preparing a GPU for execution, such as loading large AI models into memory and initializing various libraries and drivers. This process is more time-consuming than initializing CPU-based applications, which typically involve simpler and faster setups. This inherent delay amplifies the cold start problem in serverless environments, where rapid scaling and quick response times are critical. While improvements suggested in Section 2.2.1 can be applied here, it is still a challenge to overcome the problem for AI-based applications because they have bigger memory footprints and more computing need. This problem is also addressed in recent novel research and they try to come up with a solution to solve the problem [26,27].

To address the challenges in functionless computing with GPUs, the current focus is on developing serverless GPU platforms that offer dynamic scaling for cost efficiency and support for multiple model frameworks [28]. These platforms aim to minimize cold start latency and inference time, while providing scalable infrastructure and comprehensive analytics, thereby enhancing performance and user experience in AI applications.

Conclusion

This paper's investigation into the cold start problem in serverless computing highlights a complex issue that requires balancing technical aspects with efficiency and cost. The solutions we reviewed, such as FaasCache, show creative ways to address this challenge. FaasCache is notable for its practical use of caching ideas in serverless computing. However, as serverless computing evolves, especially with more AI applications being used, the cold start problem becomes more complex. Future research should focus on improving these solutions, keeping in mind the changing and more complex tasks that serverless computing will handle. The aim is to make serverless computing more efficient, with quick and easy deployment of functions in the cloud.

References

1. Veuvolu, R., Suryadevar, A., Vignesh, T., & Avthu, N. R. (2023, January). Cloud computing based (serverless computing) using serverless architecture for dynamic web hosting and cost optimization. In 2023 International Conference on Computer Communication and Informatics (ICCCI) (pp. 1-6). IEEE.
2. Hellerstein, J. M., Faleiro, J., Gonzalez, J. E., Schleier-Smith, J., Sreekanti, V., Tumanov, A., & Wu, C. (2018). Serverless computing: One step forward, two steps back. arXiv preprint arXiv:1812.03651.
3. Castro, P., Ishakian, V., Muthusamy, V., & Slominski, A. (2019). The rise of serverless computing. Communications of the ACM, 62(12), 44-54.
4. McGrath, G., & Brenner, P. R. (2017, June). Serverless computing: Design, implementation, and performance. In 2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW) (pp. 405-410). IEEE.
5. Wang, L., Li, M., Zhang, Y., Ristenpart, T., & Swift, M. (2018). Peeking behind the curtains of serverless platforms. In 2018 USENIX annual technical conference (USENIX ATC 18) (pp. 133-146).
6. Fuerst, A., & Sharma, P. (2021, April). Faascache: keeping serverless computing alive with greedy-dual caching. In Proceedings of the 26th ACM international conference on architectural support for programming languages and operating systems (pp. 386-400).
7. Reducing java cold starts on aws lambda functions with snapstart.
8. Du, D., Yu, T., Xia, Y., Zang, B., Yan, G., Qin, C., ... & Chen, H. (2020, March). Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (pp. 467-481).
9. Silva, P., Fireman, D., & Pereira, T. E. (2020, December). Prebaking functions to warm the serverless cold start. In Proceedings of the 21st international middleware conference (pp. 1-13).
10. Tips & tricks.
11. Cold starts in google cloud functions.
12. Understanding serverless cold start.
13. Lin, P. M., & Glikson, A. (2019). Mitigating cold starts in serverless platforms: A pool-based approach. arXiv preprint arXiv:1903.12221.
14. Liu, X., Wen, J., Chen, Z., Li, D., Chen, J., Liu, Y., ... & Jin, X. (2023). Faaslight: General application-level cold-start latency optimization for function-as-a-service in serverless computing. ACM Transactions on Software Engineering and Methodology, 32(5), 1-29.
15. Roy, R. B., Patel, T., & Tiwari, D. (2022, February). Icebreaker: Warming serverless functions better with heterogeneity. In Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (pp. 753-767).
16. Shahrad, M., Fonseca, R., Goiri, I., Chaudhry, G., Batum, P., Cooke, J., ... & Bianchini, R. (2020). Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In 2020 USENIX annual technical conference (USENIX ATC 20) (pp. 205-218).
17. Li, Z., Guo, L., Chen, Q., Cheng, J., Xu, C., Zeng, D., ... & Guo, M. (2022). Help rather than recycle: Alleviating cold startup in serverless computing through {Inter-Function} container sharing. In 2022 USENIX annual technical conference (USENIX ATC 22) (pp. 69-84).
18. Agarwal, S., Rodriguez, M. A., & Buyya, R. (2021, May). A reinforcement learning approach to reduce serverless function cold start frequency. In 2021 IEEE/ACM 21st international symposium on cluster, cloud and internet computing (CCGrid) (pp. 797-803). IEEE.
19. Google gvisor: Container runtime sandbox.
20. Cheng, K., & Kambayashi, Y. (2000, October). LRU-SP: a size-adjusted and popularity-aware LRU replacement algorithm for web caching. In Proceedings 24th Annual International Computer Software and Applications Conference. COMPSAC2000 (pp. 48-53). IEEE.
21. O'neil, E. J., O'neil, P. E., & Weikum, G. (1993). The LRU-K page replacement algorithm for database disk buffering. Acm Sigmod Record, 22(2), 297-306.
22. Cherkasova, L. (1998). Improving WWW proxies performance with greedy-dual-size-frequency caching policy. Palo Alto, CA, USA: Hewlett-Packard Laboratories.

23. Pid controllers.
24. Lee, M., Ahn, H., Hong, C. H., & Nikolopoulos, D. S. (2022). gShare: a centralized GPU memory management framework to enable GPU memory sharing for containers. Future Generation Computer Systems, 130, 181-192.
25. Gu, J., Song, S., Li, Y., & Luo, H. (2018, December). GaiaGPU: Sharing GPUs in container clouds. In 2018 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom) (pp. 469-476). IEEE.
26. Wang, Z., Jiang, Z., Wang, Z., Tang, X., Liu, C., Yin, S., & Hu, Y. (2020). Enabling latency-aware data initialization for integrated CPU/GPU heterogeneous platform. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 39(11), 3433-3444.
27. Ang, L. M., & Seng, K. P. (2021). GPU-based embedded intelligence architectures and applications. Electronics, 10(8), 952.
28. The state of serverless gpus.