**International Academic Institute for Science and Technology**

International Academic Journal of Science and Engineering
Vol. 12, No. 4, 2025, pp. 449-468.

ISSN 2454-3896

**International Academic Journal of Science and Engineering**

www.iaiest.com

# Mitigation of Cold Start Delays in Serverless Computing Through Adaptive Heuristic-Based Prewarming Framework

Sivaranjani Sundaram [a], Anandhi Ramani [b]

[a] *Department of Information Technology and Bachelor of Computer Applications, Dwaraka Doss Goverdhan Doss Vaishnav College, Chennai, Tamil Nadu, India. E-mail: sivaranjani.dgvc@gmail.com*
[b] *Research Department of Computer Science, Dwaraka Doss Goverdhan Doss Vaishnav College, Chennai, Tamil Nadu, India. E-mail: sekadhi@gmail.com*

## Abstract

Scalable and event-driven execution is offered in serverless computing by abstracting infrastructure management, allowing applications to be deployed and operational control to be easily achieved. There is however the cold start phenomenon which still undermines performance especially on latency-sensitive applications. After a period of idleness of a function, the platform unloads its execution environment, which has led to a higher response time when the same function is invoked again. The current idle-based prewarming solutions maintain functions by activating them periodically, which have minimized cold starts but have also created the problem of excessive resource usage and unnecessary execution expenses. Solutions that are based on learning enhance flexibility but introduce computational overhead, which is incompatible with lightweight serverless platforms. Existing cold start mitigation strategies do not have a balanced mechanism that responds to the real time invocation behavior and minimizes overhead. Majority of the methods consider all functions equally, resulting into over-prewarming or retarding implementation. An adaptive strategy with low overheads that distinguishes functions by frequency of use and selectively implements prewarming within memory and cost limits is still needed. The proposed work suggests an Adaptive Heuristic-Based Prewarming model that classifies dynamically serverless functions into hot, warm and cold ones based on real-time invocation frequency and past access patterns. Hot functions are prewarmed in advance so that they can be executed immediately. Selective prewarming of warm functions is only used when the cold start probability is larger than a predefined threshold but the cold functions are reloaded on demand through the use of checkpoint-based loading. A caching system based on the LRU procedure will store the most frequently used instances and will evict the least used instances when the memory pressures occur. The framework is model-free and therefore ensures real-time flexibility with low computational requirements. Experimental analysis with 10,000 real-world logs of function invocation under cold start conditions demonstrates that the suggested Adaptive Heuristic-Based Prewarming method decreases cold start by 72, which is by way of comparison lower than cold start decreased by existing learning and cache-based techniques (48-63). The mean reaction time is reduced by half to 140ms and this is a 70 % improvement. The cache hit rate is 85 and it has improved by a margin of about 4-9 than the best baseline. Unneeded prewarming overhead is cut down to 18.2 which translates into an approximate 65% cut in execution cost. These findings verify that the proposed approach provides a successful trade-off between reduction in latency, the use of resources and cost efficiency in the case of dynamically serverless workloads.

**Keywords:** Serverless Computing, Cold Start Mitigation, Adaptive Prewarming, Heuristic-Based Caching, Resource-Efficient Scheduling.



DOI: 10.71086/IAJSE/V12I4/IAJSE12111
Received: 23 October 2025/Revised: 25 November 2025/Accepted: 16 December 2025/Published: 30 December 2025

## Introduction

A new paradigm in cloud execution, serverless computing, has become a leading cloud computing paradigm because it can abstract the management of infrastructure and adaptively and event-driven execute applications. Under this model, the developers send the functions without provisioning or maintaining servers and the cloud provider allocates resources dynamically as the demand increases (Ghorbian & Ghobaei-Arani, 2024). Such an abstraction makes development processes simpler and less prone to overhead, further streamlining the adoption in fields of data analytics, Internet of Things pipeline, real time processing, and web backend (Golec et al., 2024). Its attractiveness is also enhanced by the pay-per-use pricing model because the users pay actual execution time and not the resources that are being reserved (Hanaforoosh et al., 2025). Recent serverless systems like AWS Lambda, Azure Functions, and OpenWhisk are now part of the cloud-native architecture (Talwar & Shah, 2023). In spite of these benefits, serverless platforms are built on immanent execution environments, which grow (and contract) dynamically. A container has been unloaded to reclaim resources when a given function has not been called upon after a defined period of time (Pan et al., 2023). When invoked again, a platform provision will start a new execution environment, dependencies will be loaded, and runtime contexts will be configured. This introduces cold start latency, which has a major effect on the response time particularly with latency sensitive workloads (Riadhusin et al., 2025). It is reported by a number of studies that cold start can take hundreds of milliseconds up to several seconds depending on the size of dependencies and the language runtime itself, and policies specific to the platform (against cold start) (Zhang et al., 2025). With the adoption of serverless into real-time and user-facing workloads, cold start latency has become one of the most important efficiency problems, which directly impacts user experience and system reliability (Lenine et al., 2025).

The dynamic and heterogeneous serverless workload has a number of technical issues to mitigate cold start latency. Workload uncertainty is one of the greatest challenges in which there is a great variation in the invocation patterns over time (Cai et al., 2024). The strategies of static resource management do not conform to such differences and prove ineffective giving rise to either over-provisioning or delayed execution. The trade-off of performance and resource efficiency is another problem. In aggressive prewarming, cold starts are minimized at the cost of more memory use and higher execution cost, which negates the cost-efficiency objectives of serverless systems (Golec et al., 2024). Idle pre warming schemes invoke functions periodically to maintain containers alive. Though, this approach minimizes cold start, it has led to unnecessary executions that waste compute resources without doing meaningful work (Xiao et al., 2024). Further, idle-based strategies do not exploit the differences among the functions in terms of invocation frequency and temporal locality. Learning-based methods seek to overcome this drawback by forecasting invocation patterns with the help of machine learning models. Nonetheless, these methods add extra computational costs, demand training data and do not change quickly to sudden changes in workload (Saravanan et al., 2022).

Moreover, existing solutions are highly concentrated on latency reduction, but they neglect memory constraints, as well as, policies to evict the cache, which is required to be scaled (Duan et al., 2025). All these difficulties make it apparent that a cold start mitigation plan is a complicated endeavor that must address the three aspects in an adaptable, efficient, and simple manner.

The cold start issue remains open as current mitigation strategies are either based on a set of heuristics that do not change with the context, or computationally expensive learning models. Prewarming strategies that rely on idle-based methods maintain the readiness of functions at the expense of high resource use, whereas the prediction-based methods rely on the workload stability and training accuracy (Vahidinia et al., 2022). Both classes do not offer a lightweight and real time adjustable system that separates functions based on invocation behaviour and selectively applies prewarming based on memory and cost limitations. Due to this, serverless platforms are still facing the unnecessary prewarmer overhead or unacceptable latency in bursts of invocation. The above issue has inspired the investigation of a dynamically adjusted but low-overhead mechanism that adjusts the prewarming choices to the observed dynamic trends in the actual utilization of the functions.

The extant literature does not contain a single and heuristic-based prewarming model that dynamically reconfigures itself to adapt to the dynamic invocation behavior without involving the use of complex learning models. The majority of existing methods either make the assumption that the behavior of functions is homogenous or use global thresholds that cannot represent access patterns on a fine scale. Also, little has been done in incorporating lightweight caching schemes, including LRU-based eviction, with adaptive prewarming choices in a restricted memory set up. It is also lacking adequate attention to trade off cold start mitigation and cost efficiency, especially in eliminating unnecessary invocation of functions. This gap implies that the practical, scalable, and low-complexity solution is required, which, at the same time, considers the factors of latency, memory usage, and cost of execution in serverless environments.

The goal of this research is to develop and test an adaptive prewarming model based on heuristics in order to reduce the cold start latency and maintain the lightweight characteristics of serverless computing. These are the following objectives. First, the research will dynamically categorize serverless functions as hot, warm, and cold, according to the real-time invocation frequency, as well as the past access patterns. Second, it aims at creating a selective prewarming mechanism, which activates legal prewarming only when the cold start probability is above a predetermined value. Third, the study incorporates an LRU-based policy of caching classifier functions to keep the high frequency accessed instances and to remove the rarely utilized ones when memory limits are exceeded. Fourth, the framework will minimize invocation of unneeded prewarming, which will decrease the cost of execution without affecting responsiveness. Lastly, the paper benchmarks the suggested solution with actual invocation logs to measure the gains made in the reduction of cold start, response time, cache performance, and cost reduction.

This work is novel due to its lightweight and heuristic-based design that does not require elaborate model training and yet the adaptive behavior is attained. In contrast to the current approaches based on learning, the offered framework works based on real-time invocation statistics and past tendencies, which allows responding instantly to the workload. The explicit division of functions into hot, warm and cold categories provides the possibility of fine-control of prewarming intensity which enhances latency and resource efficiency. Also, the combination of checkpoint-based restoration of cold functions with LRU-based cache eviction in the presence of memory constraints offers a sensible and reasonable solution, which is in line with the real-world serverless execution constraints.

This research makes the following key contributions:

- It proposes an Adaptive Heuristic-Based Prewarming framework that dynamically categorizes serverless functions based on invocation behavior and selectively applies prewarming to minimize cold start latency with low computational overhead.
- It presents an integrated caching and eviction policy, which integrates LRU-based memory management with cold start probability thresholds that allow efficient resource usage and huge costs saving, yet high responsiveness.

## Related Works

These works all seek to strike a balance between response latency and workload scalability and the use of memory during dynamic workloads. The survey below has arranged the relevant literature in a logical sequence starting with learning-based predictive and pre-warming techniques, then to the scheduling and reuse mechanisms, and, lastly, workflow and application-level optimization.

### Learning-Based Prediction and Adaptive Pre-Warming Strategies

A number of studies are aimed at utilizing predictive modeling of invocation patterns of functions to preempt cold start mitigation via intelligent pre-warming. Vahidinia et al., (2022) propose a two-layer adaptive framework, which incorporates both reinforcement learning and temporal sequence modelling. The outer layer makes use of a holistic reinforcement learning algorithm that has acquired invocation patterns through time which allows decisions to be made on container keep-alive times. The second layer uses an LSTM-based predictor that has learnt trends in anticipated future invocation to provide the amount of containers that should be pre-warmed. The experimental findings of the OpenWhisk platform reveal that the strategy saves memory usage by 12.73 % and also speeds up execution in prewarmed containers by 22.65 %, which are a fair trade-off between the resource-efficiency and the latency minimization.

Hu et al., (2025) build on the predictive pre-warming paradigm and uses workload history data to predict the future using a Recurrent Neural Network. Their framework is a mix of RNN-based predictor and a pre-warming scheduler which dynamically modifies the idle containers times. The design is flexible and fits well to the changes in work load hence minimizing cold starts and unnecessary idle time. The proposed approach completes 49.52 % more invocations in warm conditions, as compared to OpenWhisk, and it has demonstrated its efficiency in optimising container allocation. This adaptive approach is in contrast to the static keep-alive policies, which reacts to the workload dynamics at all times.

Saravana Kumar & Selvakumara Samy, (2025) also go further to apply prediction based cold start mitigation but target cold start likelihood estimation instead of invocation timing. Their paper suggests an efficient concurrent provisioning model to AWS Lambda, which incorporates an attention-based bi-directional gated recurrent unit. The model has applied the historical and real time characteristics such as invocation frequency; previous cold starts and time lapse among others to forecast cold start events with great accuracy. The method, with hyperparameter optimization using

RMSProp optimizer, makes results with a high predictive performance with an accuracy of 90.36 and an F1-score of 90.28. This paper highlights that the proper forecasting of cold start is possible and can inform an active provisioning decision-making approach that will minimize latency and avoid over-provisioning.

**Adaptive Scheduling and Reinforcement Learning for Resource Management**

In addition to prediction, other works focus on the adaptive scheduling mechanism that can dynamically control the container lifecycles and the reuse strategies. The cache-based adaptive censoring schedule of Serverless Runtime, suggested by Chen et al., (2025), addresses the container maintainability (keep-alive) challenge in the form of a caching issue. The proposed serverless cache or S-Cache has traded off cold start latency with memory usage due to the selective retention of containers. The use of a deep reinforcement learning model on the basis of the proximal policy optimization has made it possible to automatically scale the cache queue to changes in the workload. Assessments based on an Azure set depict significantly significant improvements, such as a 38.75 % cold start reduction and 46.73 % memory utilization efficiency enhancement. This paper has demonstrated how reinforcement learning can be effective in solving the multiple trade-offs that exist in serverless resource management.

Zhou et al., (2024) use deep reinforcement learning as well, but they are specifically interested in the multi-level reuse of containers. They use containers that have similar dependency profiles instead of having exact matches which results in better utilization of warm resources. Since the process of choosing the best candidates of reuse is not only computationally expensive, but also combinatorial, there is a scheduler that is based on DRL, which has led the task of reuse choice. Another benchmark introduced by the authors is FStartBench, which contains information about packages to assess cold start mitigation methods. Experimental studies have indicated a maximum of 53 % decrease in average startup latency as compared to state-of-the-art techniques and this goes to highlight the worthiness of intelligent reuse techniques driven by learning-based schedulers.

Karamzadeh & Shameli-Sendi, (2024) views scheduling as a proactive invocation point of view. They use target functions in advance of actual user requests, rather than using prediction models, turning cold start into warm start. The strategy has been compared to fixed and variable window strategies that are typically applied in platforms like Knative, OpenFaaS, and OpenWhisk. The findings indicate that the suggested strategy eliminates more than 90% of cold starts with different invocation patterns, and it has also been more effective than fixed and adaptive window baselines. This paper shows that proactive triggering, which is properly controlled, can severely prevent cold start frequency without intricate learning models.

**Workflow-Level Optimization**

An alternative body of work touches on cold start latency on the workflow level through the rearrangement of functions execution. Lee et al., (2021) introduce a function fusion scheme that is aimed at overcoming cold start latency in parallel execution path serverless workflows. The authors initially determine the important latency factors which influence the workflow response time and subsequently formulate a response time model which includes the latency factors. Using this model, an optimal fusion scheme has been obtained in order to reduce the effect of cold start. The experimental analysis of five workflows demonstrates that the fused workflows attain response times of between 28 and 86 % of the original which means there is a significant latency reduction. This strategy proves that the restructuring at the architectural level based on the workflow can be used as a supplement to the optimization on the container level.

Behera et al., (2025) also find out more about function fusion with the PanOpticon simulator where the experiment provides a complete environment in which to deploy and test FaaS applications. Their style actively combines tasks in a workflow in order to minimize redundant startup cost. This framework unlike other traditional methods of fusing is directly part of the simulation chain, allowing scalable and performance conscious analysis. The results of the simulations support the hypothesis that the fusion strategy has a shorter cold start turnaround and better throughput and responsiveness when faced with varying workloads. The role serves as a practical and scalable cold start mitigation method fused into a single work role in complicated serverless processes.

**Runtime Optimization Approaches**

Although most of the research is on container and scheduling techniques, there are those that consider cold start latency through optimization of application-level behavior. Liu et al., (2023) suggest FaaSLight, an application-level optimization system that focuses on code loading latency, which is found to be a significant cause of cold starts. Through the creation of function level call graphs, the method divides the code which is essential and the code which is optional and can be loaded on demand. By using this selective loading approach, the useless initial loading overhead is minimized without jeopardizing the failure of an application. Faas Light is platform and language-neutral and can

be connected to AWS Lambda and Google Cloud Functions. The results of the evaluation indicate the maximum reduction of code loading latency by 78.95 % and total response latency by 42.05 %, which is a great improvement in comparison with the current technique.

Buitrago et al., (2025) resolve the problem of cold start latency by providing optimization of runtime and container startup by their PARSEC architecture. The offered solution simplifies the process of containerization by reducing unnecessary processes and launching serverless loads. One of the major contributions is the optimized provisioning and management of Zygotes which increases the pace of sandbox launches and the scalability of high-load Zygotes. The cache system is also made more discriminating in terms of the important package to minimize the overhead. As per experimental findings, PARSEC has outperformed the existing architectures especially with short wait times, which have demonstrated significance of runtime-level optimisation in cold start management. A comparative summary of the cold start mitigation methods is given in table 1.

**Table 1: Comparative Summary of Cold Start Mitigation Techniques**

| Method | Algorithm / Technique | Metrics Used | Key Results | Advantage | Limitation |
|---|---|---|---|---|---|
| Vahidinia et al., (2022) | Reinforcement Learning + LSTM | Memory usage, warm execution ratio | 12.73% memory reduction, 22.65% more prewarmed executions | Adaptive temporal learning of invocation patterns | Depends on accurate long-term workload patterns |
| Lee et al., (2021) | Function Fusion with Latency Modeling | Workflow response time | 28–86% of original workflow response time | Reduces cold start at workflow level | Reduced modularity and flexibility |
| Hu et al., (2025) | RNN-based Prediction + Scheduler | Invocation count, idle time | 49.52% improvement over OpenWhisk | Dynamic idle-container adaptation | Performance degrades under highly irregular traffic |
| Karamzadeh & Shameli-Sendi, (2024) | Proactive Invocation Strategy | Cold start counts | Up to 92.13% cold start reduction | Simple and effective warm-start conversion | Potential unnecessary invocations |
| Chen et al., (2025) | PPO-based DRL + Cache (CASR) | Cold starts, memory utilization, overhead | 38.75% fewer cold starts, 46.73% better memory use | Balanced latency–memory trade-off | Training overhead of DRL model |
| Saravana Kumar & Selvakumara Samy, (2025) | ABiGRU with Attention | Accuracy, Precision, Recall, F1, MCC | 90.36% accuracy, 90.28% F1-score | High-precision cold start prediction | Focuses on prediction rather than mitigation |
| Liu et al., (2023) | Application-level Code Separation (FaaSLight) | Code load time, response latency | 78.95% code load reduction, 42.05% latency drop | Platform- and language-agnostic | Requires static code analysis |
| Buitrago et al., (2025) | Runtime Optimization (PARSEC) | Initialization latency, throughput | Improved cold start at short wait times | Efficient container and Zygote management | Platform-level architectural changes |
| Behera et al., (2025) | Workflow-level Dynamic Function Fusion | Latency, throughput | Significant cold start delay reduction | Scalable fusion via simulation | Limited real-world deployment validation |
| Zhou et al., (2024) | DRL-based Multi-level Container Reuse | Startup latency | Up to 53% latency reduction | Effective reuse of similar containers | Dependency profiling complexity |

Together, these articles demonstrate that there has been a shift towards the non-adaptive nature of heuristics to adaptive solutions that rely on learning to mitigate cold starts. The old approaches founded on fixed keep-alive windows have been replaced by the predictive models which take advantage of historic workload dynamics and this has been shown by the Vahidinia et al., (2022) and Hu et al., (2025). Schedulers based on reinforcement learning, like those suggested by Chen et al., (2025) and Zhou et al., (2024), also maximize the flexibility of resource allocation, which is uncertain in nature. Otherwise, workflow-level fusion methods and application-level optimizations consider cold starts at the architectural and code-centric level, which have complementary advantages.

The current cold start mitigation solutions in serverless computing are mainly based on the concept of prewarming through the use of either the static or idle-based approach, which is periodically invoked functions to ensure that the functions are ready. Though such methods lower cold start latency, they tend to cause unreasonable and unwarranted consumption of resources and thus elevated operational expenses. Prediction driven and learning-based techniques are more adaptive, also create computational load and necessitate sophisticated model training, which makes them impractical in real time serverless settings. In addition, existing solutions do not differ much as all functions are treated equally, and invocation behavior is not differentiated at runtime. Definitive lightweight, heuristic-oriented mechanisms that dynamically categorize functions in terms of frequency of invocation and previous utilization and trade-off latency minimization plus memory and cost are also lacking. This difference has demonstrated the necessity of the adaptive prewarming strategy which can work with a low overhead, react to the workload changes in real time, and allocate the resources selectively without using any intricate learning models.

## Proposed Adaptive Heuristic-Based Prewarming Method

The proposed Adaptive Heuristic-Based Prewarming (AHBP) protocol solves the cold-start issue by integrating real-time invocation awareness and heuristic decision-making, which is lightweight. This approach does not use complicated learning models but is based on the frequency of invocation, locality in time, and previous access patterns that are used to make prewarming decisions. The design guarantees fast adaptability whilst maintaining the low-overhead implementation model demanded by serverless ones (Fig.1).

The fundamental component of the approach is that every serverless function is constantly monitored with invocation logs that contain arrival time, inter-arrival gap, and number of recent executions. With these statistics, the system can estimate the cold start probability of each of the functions within a sliding observation window. According to this estimation, the function is dynamically categorised into three possible states; hot, warm and cold state. This classification has made it possible to control the distribution of resources finely, thereby minimizing useless prewarming and avoiding responsiveness.

Hot functions are functions that are commonly called and have a high degree of temporal locality. Such functions are actively prewarmed, so that even when all the instances of containers are turned off at least one of them is kept warm. The given strategy ensures the execution process is immediately performed and offers a reduced perceived latency by the user. Warm functions are moderate or bursty with regard to invoking behavior. In such functions, prewarming has been selective and only in case of the estimated cold start probability being greater than a predetermined threshold. This conditional activation allows the use of resources without overuse and also does not cold start as much in the case of sudden bursts of invocation. Cold functions are not a frequent occurrence and hence do not warrant consistent prewarming. These functions are loaded when required based on checkpoint-based restoration and this minimizes the overhead of the initialization without the idle containers.

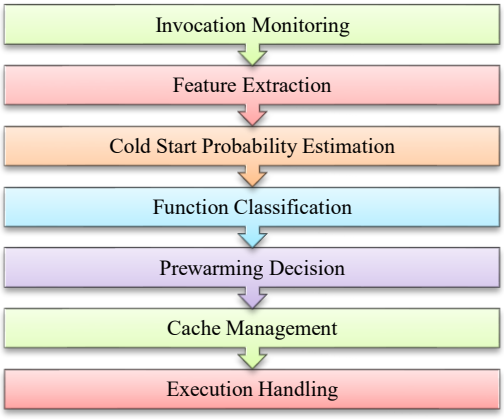


**Fig. 1: Process Flow of the Proposed Method**

The technique incorporates an LRU based caching mechanism to help control memory limits. The instances of active and recently used functions are kept in a cache. In the case of memory pressure, the least recently used instances are tossed out of the cache, which guarantees that high-frequency functions are prioritized. The mechanism has ensured a high rate of cache hits and elimination of memory flooding. With the combination of heuristic classification, selective prewarming and LRU eviction the framework balances the latency minimization, memory efficiency and cost.

**Pseudocode of the Proposed AHBP Method**

```
Initialize LRU_Cache with capacity M
Initialize thresholds: Th_hot, Th_warm, Th_cold
For each incoming invocation f_i at time t:
  Update invocation_log[f_i] with t
  Compute frequency_i, idle_time_i
  Compute cold_prob_i based on idle_time_i
  If frequency_i >= Th_hot:
    state_i ← HOT
  Else if cold_prob_i >= Th_warm:
    state_i ← WARM
  Else:
    state_i ← COLD
  If state_i == HOT:
    If f_i not in LRU_Cache:
      Prewarm container for f_i
      Insert f_i into LRU_Cache
    Mark f_i as most recently used
  Else if state_i == WARM:
    If cold_prob_i >= Th_cold:
      If f_i not in LRU_Cache:
        Prewarm container for f_i
        Insert f_i into LRU_Cache
      Mark f_i as most recently used
  Else if state_i == COLD:
    Load f_i on demand using checkpoint restoration
  If LRU_Cache is full:
    Evict least recently used function instance
  Execute f_i using available container
```

**Invocation Monitoring Mechanism**

The system of continuous invocation monitoring has taken note of the runtime events of each serverless function. Each invocation produces an organized record of the log that records time and behavioral qualities of execution. To ensure that these logs do not cause I/O overhead, these logs are stored in a lightweight in-memory format.

The monitoring module has logged the invocation time, the time of execution completion, and the state of the container of every function $f_i$. Assuming the sequence of invocation times of a function are:

$$\mathcal{T}_i = \{t_{i,1}, t_{i,2}, \dots, t_{i,n}\}$$

where $t_{i,k}$ represents the time point of the $k^{th}$ invocation. It is based on this sequence that the system obtains temporal gaps that define idleness of functions. The inter-arrival time of two successive invocations is calculated to be:

$$\Delta t_{i,k} = t_{i,k} - t_{i,k-1}, k \geq 2$$

These inter-arrival times are used to show the bursty/periodic/sparse behavior of a given function. A short and repeated gaps is a sign of hot functions and a long and irregular gaps is a sign of cold behavior. The module of monitoring also

monitors the last access time $t_i^{last}$ which is vital in estimation of cold start probability.

Table 2 shows an invocation monitoring log that has been obtained during a sliding observation window. This table has indicated the form in which raw invocation information is saved prior to feature extraction.

**Table 2: Invocation Monitoring Log**

| Function ID | Invocation Index | Timestamp (ms) | Inter-arrival Time (ms) | Container State |
|---|---|---|---|---|
| F1 | 1 | 1000 | – | Cold |
| F1 | 2 | 1200 | 200 | Warm |
| F1 | 3 | 1350 | 150 | Warm |
| F2 | 1 | 1000 | – | Cold |
| F2 | 2 | 3000 | 2000 | Cold |
| F3 | 1 | 1000 | – | Cold |
| F3 | 2 | 1050 | 50 | Warm |
| F3 | 3 | 1100 | 50 | Warm |

As indicated in table 2, functions F1 and F3 have indicated short inter-arrival time, which means that they are frequently used. Conversely, function F2 portrays a high idle gap, and one can assume that it is cold-behaving. The following monitored attributes directly enter the stage of feature extraction.

**Sliding Window-Based Observation Model**

The proposed approach uses a sliding window model to invocation logs so that it is able to achieve real-time adaptability. Allow the window length to take the form of $W$ that is the latest observation interval. Invocation events are the only events that are involved in the computation of features:

$$\mathcal{T}_i^W = \{t_{i,k} \mid t_{i,k} \geq (t_{current} - W)\}$$

The design will make the recent behavior predominate in decision-making, and the old patterns will only wane out. The sliding window algorithm has made it less sensitive to the staleness of workload properties and more adaptable to changing traffic.

**Feature Extraction Process**

The feature extraction transforms the data that is monitored during the invocation into quantitative features that describe the behavior of the functions. This framework proposes three main features, which are invocation frequency, idle duration, and the time locality score. All of these characteristics combine to describe the frequency of execution of a given function, the duration of idle time, and the concentration of the invocations of a given function across time.

**Invocation Frequency Estimation**

Invocation frequency is used to determine the frequency of execution of a given function over the period of observation. The frequency of function $f_i$ is calculated as:

$$\lambda_i = \frac{|\mathcal{T}_i^W|}{W}$$

where $|\mathcal{T}_i^W|$ represents invocations in the window. An upshot value of $\lambda_i$ is an indication that the activity has been in demand and needs to be hot. Table 3 has shown a frequency computation for multiple functions.

**Table 3: Invocation Frequency Estimation**

| Function ID | Invocations in Window | Window Size (ms) | Frequency $\lambda_i$(invocations/ms) |
|---|---|---|---|
| F1 | 3 | 500 | 0.006 |
| F2 | 1 | 500 | 0.002 |
| F3 | 3 | 500 | 0.006 |

As can be seen in table 3, functions F1 and F3 have higher invocation frequencies than F2. It is a characteristic that has a strong effect on classification of hot functions.

**Idle Duration Computation**

Idleness period indicates the time that it took since the last invocation. This anchorage is used to record the possibility of a container to be unloaded by the platform. In the case of every function $f_i$, the idle time is calculated as:

$$I_i = t_{cur} - t_i^{last}$$

When this idle duration is larger, there is a higher chance of the instance of the functions being in a cold state. This metric has a direct contribution to cold start probability estimation. Invocation monitoring provided the values of idle duration as shown in table 4.

**Table 4: Idle Duration Measurement**

| Function ID | Last Invocation Time (ms) | Current Time (ms) | Idle Duration $I_i$(ms) |
|---|---|---|---|
| F1 | 1350 | 1500 | 150 |
| F2 | 3000 | 5000 | 2000 |
| F3 | 1100 | 1500 | 400 |

Based on table 4, the idle time of the function F2 is much greater, and it implies a greater cold start probability.

**Temporal Locality Score**

Temporal locality is used to describe the area of clustering of invocation events in the window. The characteristic assists in the differentiation between bursty warm functions and the functions that are always hot. Temporal locality scores $L_i$ i s calculated as the coefficient of variation of the inter-arrival times:

$$L_i = \frac{\sigma(\Delta t_i)}{\mu(\Delta t_i) + \epsilon}$$

The symbols $\sigma(\Delta t_i)$ and $\mu(\Delta t_i)$ are the standard deviation and the mean of inter-arrival times respectively and epsilon is a minimal constant to prevent the instability of the division. When $L_i$ is low, invocations are stable and frequent, whereas when $L_i$ is large, invocations are irregular. Table 5 shows the scores of localities of functions in time.

**Table 5: Temporal Locality Estimation**

| **Function ID** | **Mean Inter-arrival (ms)** | **Std. Deviation (ms)** | **Locality Score $L_i$** |
|---|---|---|---|
| F1 | 175 | 25 | 0.14 |
| F2 | 2000 | 0 | 0.00 |
| F3 | 50 | 0 | 0.00 |

Table 5 demonstrates that the locality variation of the function F1 is moderate, whereas the functions F2 and F3 have the stable patterns in the restricted sets of invocations.

**Cold Start Probability Estimation**

The features that are extracted are used to estimate the cold start probability. This heuristic probability represents the probability of cold start of a function invocation. The probability is calculated according to the proposed framework:

$$P_i^{cold} = \alpha \cdot \frac{I_i}{I_{max}} + \beta \cdot (1 - \frac{\lambda_i}{\lambda_{max}}) + \gamma \cdot L_i$$

where, $\alpha + \beta + \gamma = 1$, and $I_{max}$ and $\lambda_{max}$ are normalization constants that are obtained based on limits of the system. This formulation is such that the idle period, invocation frequency and time locality all contribute to the estimation. An increased $P_i^{cold}$ indicates an increased cold start risk, which initiates selective prewarming to warm functions.

**Cold Start Probability Estimation**

With a serverless architecture, a cold start happens when a call to a function has been made whose execution environment has been unloaded following an extended period of inactivity. This event is not a one-factor phenomenon but rather a combination of idle time, invocation frequency and regularity of access of time. The interaction is described by the suggested framework via a composite cold start probability model. This probability is a normalized risk measure that gives the probability of a cold start on a subsequent invocation of the function. After estimating the cold start probability, the framework assigns each of the functions to one of the three operational states namely hot, warm, and cold. Such categorization has allowed prewarming selectively and effective allocation of resources. The distinction between probability estimation and classification makes it interpretable and controllable because probability measures risk, whereas classification measures action.

**Cold Start Probability Estimation**

Cold start probability estimation is a method that uses various run-time indicators as one bounded measure. A serverless function can be denoted as $f_i$. Each observation interval is calculated to determine the probability $P_i^{cold}$ at the start of the interval based on the normalized feature values obtained by monitoring invocation and feature extraction.

**Idle Time Contribution**

There is a very strong correlation between idle time and cold start probability due to the fact that serverless systems usually recover idle time resources after a specified inactivity period. Let $I_i$ denote the idle time of operation of $f_i$, which is calculated as the amount of time that has elapsed since $f_i$ was last invoked. In order to make the functions comparable, the idle time is normalized as:

$$\hat{I}_i = \frac{I_i}{I_{max}}$$

where $I_{max}$ denotes the maximum idle time which a container can be allowed to occupy prior to the platform forcefully reclaiming it. An increase in $\hat{I}_i$ has the direct effect of raising cold start risk.

**Invocation Frequency Contribution**

Invocation frequency records the frequency of a function being called in the sliding window. I denote the frequency of invocation of $f_i$ by $\lambda_i$ and the maximum used frequency of all functions by $\lambda_{max}$. Inverse frequency is normalized and calculated as:

$$\hat{\lambda}_i = 1 - \frac{\lambda_i}{\lambda_{max}}$$

This formulation allows functions that are frequently invoked to add little to the probability of cold start and functions invoked infrequently add risk.

**Temporal Locality Contribution**

The spatial locality is a measure of temporal clustering of invocation events. Stable and clustered functions can be predicted better and not affected by cold start so much. The normalized locality score $L_i$ is value is:

$$\hat{L}_i = \frac{L_i}{L_{max}}$$

where $L_{max}$ denotes the greatest observed variation of locality. An increased $\hat{L}_i$ ireflects irregular access behavior, which amplifies cold start uncertainty.

**Composite Probability Model**

A weighted linear aggregation is used to compute the cold start probability by coming up with the normalized features:

$$P_i^{cold} = \alpha\hat{I}_i + \beta\hat{\lambda}_i + \gamma\hat{L}_i$$

subject to:

$$\alpha + \beta + \gamma = 1{,}0 \leq P_i^{cold} \leq 1$$

The weighting factors $\alpha$, $\beta$, and $\gamma$ show the weight of the idleness, frequency, and the temporal locality. Practically, the time of idle is assigned a greater weight since container reclamation policies directly rely on idle. A normalized computation of the probability of a cold start has been demonstrated in table 6.

**Table 6: Cold Start Probability Estimation**

| Function ID | Normalized Idle $\hat{I}_i$ | Normalized Inverse Frequency $\hat{\lambda}_i$ | Normalized Locality $\hat{L}_i$ | $P_i^{cold}$ |
|---|---|---|---|---|
| F1 | 0.15 | 0.20 | 0.10 | 0.16 |
| F2 | 0.85 | 0.75 | 0.40 | 0.73 |
| F3 | 0.30 | 0.25 | 0.05 | 0.22 |

Table 6 demonstrates that the probability of cold start in function F2 is much higher as the idleness is long, and the frequency of invocation is low, whereas functions F1 and F3 have relatively low risks.

**Probability Smoothing and Stability**

To ensure smooth changes in classification due to temporary peaks in workload, the proposed framework uses exponential smoothing on the cold start probability. $P_{i,t}^{cold}$ denote the probability at time $t$. $\tilde{P}_{i,t}^{cold}$ is calculated as smoothed:

$$\tilde{P}_{i,t}^{cold} = \delta \cdot P_{i,t}^{cold} + (1 - \delta) \cdot \tilde{P}_{i,t-1}^{cold}$$

In which $\delta \in (0,1)$ regulates the sensitivity to the new conduct. This blurring provides the stability of classification

without being responsive to persistent changes in workload.

**Function Classification Strategy**

The framework takes functions to the operational states according to the smoothed cold start probability and invocation frequency. This classification outlines the prewarming and loading strategy of each of the functions.

**Hot Function Classification**

A function is said to be hot when it has a high invocation rate and low cold start. Formally, function $f_i$ is is hot where:

$$\lambda_i \geq \theta_{hot} \text{and} \tilde{P}_i^{cold} \leq \epsilon_{hot}$$

Hot functions have proactive prewarming that will ensure that one instance of containers always remains running.

**Warm Function Classification**

Warm functions have exhibited moderately or bursty invocation. These functions are not constantly active, as well as they are not scarcely used. The warmness of a function is such that:

$$\theta_{warm} \leq \lambda_i < \theta_{hot} \text{and} \epsilon_{hot} < \tilde{P}_i^{cold} \leq \epsilon_{cold}$$

Selective pre warmth is only prewarmed when the cold start probability is greater than a predetermined value.

**Cold Function Classification**

Cold functions are characterized by a low invocation and a high cold start. These functions do not warrant uninterrupted resource allocation. A cold classification is one where a function:

$$\lambda_i < \theta_{warm} \text{or} \tilde{P}_i^{cold} > \epsilon_{cold}$$

The cold functions are performed based on on-demand loading and checkpoint-based restoration. Classification result of functions has been demonstrated in table 7.

**Table 7: Function Classification Based on Cold Start Probability**

| Function ID | Frequency $\lambda_i$ | Smoothed Probability $\tilde{P}_i^{cold}$ | Classification |
|---|---|---|---|
| F1 | High | 0.16 | Hot |
| F2 | Low | 0.73 | Cold |
| F3 | Medium | 0.22 | Warm |

The classification, as table 7 demonstrates, makes sense with respect to invocation behavior as observed.

**Dynamic Reclassification and Adaptation**

There is no fixed classification of functions. Invocation update denotes feature values and probability estimates each can cause state transitions. As an example, a cold function that has an abrupt increase in invocations slowly changes to warm and eventually hot state as frequency rises and idle time reduces. On the other hand, a hot function will change to warm or cold state as the invocation frequency decreases. Such dynamic reclassification will see to it the allocation of resources is not based on what they thought it will be but what they know it is. Selective prewarming is directly regulated by cold start probability estimation. Prewarming is activated when $\tilde{P}_i^{cold}$ exceeds a prewarming threshold phantom $\epsilon_{pre}$, In the case of warm functions. This condition does not waste time in prewarming when there are idle periods that are not necessary and safeguarding against the expensive cold start when bursts occur.

**Prewarming Decision Mechanism**

The prewarming decision module decides which instances of functions to proactively initialize, maintain selectively or to load on demand. It is a choice based on the state of functionality based on cold start possibility estimation and categories. The target is to achieve the lowest possible latency to cold start and at the same time does not result in undue container activation.

In case of every function $f_i$, the framework computes a prewarming activation score, combining cold start probability and anticipated invocation benefit. This score is defined as:

$$S_i^{pre} = \tilde{P}_i^{cold} \cdot \mathbb{E}[D_i]$$

where $\tilde{P}_i^{cold}$ currently is the smooth cold start probability and $\mathbb{E}[D_i]$ is the expected demand at the next observation period. From the recent invocation frequency, it is expected that demand will be:

$$\mathbb{E}[D_i] = \lambda_i \cdot \Delta T$$

where $\Delta T$-is the prewarming horizon. When $S_i^{pre}$ indicates a greater latency reduction benefit, a greater value of S i indicates a larger reduction in latency.

In the case of hot functions, the framework applies proactive prewarming regardless of $S_i^{pre}$, since sustained demand should be justified by constant readiness. Prewarming is done only at temperature functions when $S_i^{pre} \geq \tau_{pre}$, where $\tau_{pre}$ is a tunable parameter. Cold works do not take prewarming into account. Prewarming decisions have been presented in table 8 on the basis of calculated scores.

**Table 8: Prewarming Decision Outcomes**

| Function ID | State | $\widetilde{P}_i^{cold}$ | Expected Demand | Prewarming Score $S_i^{pre}$ | Decision |
|---|---|---|---|---|---|
| F1 | Hot | 0.15 | High | 0.45 | Prewarm |
| F2 | Warm | 0.62 | Medium | 0.37 | Prewarm |
| F3 | Warm | 0.28 | Low | 0.08 | Skip |
| F4 | Cold | 0.81 | Very Low | 0.02 | Skip |

Warm function F2 meets the threshold and is selectively prewarmed, whereas function F3 does not (see table 8), which saves the unnecessary use of resources.

**Controlled Prewarming Intensity**

To avoid over-prewarming at heavy workloads, the framework puts a limit on the number of simultaneous instances of a given function. $N_i^{active}$ denotes the count of active containers of function $f_i$. The framework enforces:

$$N_i^{active} \leq \min\left(N_{max}, \lceil \kappa \cdot \lambda_i \rceil\right)$$

where $N_{max}$ is a platform-defined limit and $\kappa$ is a scaling coefficient. This limit guarantees proportional scaling and no memory overflow.

**Cache Management Strategy**

The cache management rules the occupancy of the memory by prewarmed and recently executed instances of functions. The suggested architecture takes an LRU-based cache with the addition of state awareness to prioritize hot and warm functions.

Let the cache be represented as:

$$\mathcal{C} = \{(f_i, t_i^{use})\}$$

where $t_i^{use}$ denotes the final access time. The memory of the cache is finite with a capacity $M$ that is in memory units.

During an instance of a new function being prewarmed or run, the framework checks the available memory. In case of enough space, the instance is stored in the cache. Otherwise, eviction occurs.

**LRU Eviction with Priority Bias**

The normal LRU eviction gets rid of the least used in the recent past. The suggested framework expands on this reasoning, but it includes the concept of the priority of the state of functions. Both the caches are allocated a weighted eviction score:

$$E_i = w_s \cdot \pi_i + w_t \cdot \frac{t_{current} - t_i^{use}}{T_{max}}$$

where:

$\pi_i$is the priority weight based on function state (hot < warm < cold),

$w_s$and $w_t$are weighting coefficients,

$T_{max}$is the maximum observed idle duration.

Those whose $E_i$ values are larger are evicted first. By making use of such a design, cold or seldom used functions are cleared out of memory sooner than hot. Table 9 shows a case of evicting the Cache.

**Table 9: Cache Eviction Decision**

| Function ID | State | Last Used (ms ago) | Priority Weight $\pi_i$ | Eviction Score $E_i$ | Eviction |
|---|---|---|---|---|---|
| F1 | Hot | 50 | 0.1 | 0.12 | No |
| F2 | Warm | 200 | 0.4 | 0.42 | No |
| F3 | Cold | 600 | 0.8 | 0.78 | Yes |

Table 9 reveals that function F3 has the most eviction score and has been eliminated to release memory.

**Cache Hit and Miss Handling**

Once an invocation is received, it is sent to the cache first. A cache hit will be completed in case an active container instance exists. Otherwise, there is a cache miss which results in prewarming or on demand loading.

The cache hit probability is defined as:

$$H = \frac{N_{hits}}{N_{hits} + N_{misses}}$$

The suggested framework puts $H$ by in favor of hot and warm functions that match the memory allocation with demand.

**Execution Handling**

Execution manages the routing of the incoming invocations to the available containers. The model identifies three paths of execution.

**Warm Execution Path**

When there is a prewarmed container in the cache then the invocation is executed immediately. Execution latency prevails in the response time:

$$T_{exec}^{warm} = T_{run}$$

This route offers the lowest latency and is the most ideal situation.

**Cold Execution with Checkpoint Restoration**

When the function is determined to be cold and no cache entry is known, the framework is used to ensure that restoration is made through checkpoints. $T_{chk}$ denotes the time at which checkpoint loading occurs and $T_{init}$ denotes the time at which the initial loading is taking place. The response time is:

$$T_{exec}^{cold} = T_{chk} + T_{run}$$

This has the advantage of lowering the amount of overhead required to initialize as compared to full container startup.

**On-Demand Prewarming Execution**

In warm functions that are above the probability threshold but which do not have an active invocation, the framework performs immediate prewarming and makes the invocation:

$$T_{exec}^{warm'} = T_{pre} + T_{run}$$

where $T_{pre}$is the prewarming latency, which is lower than full cold start latency.

Table 10 has shown execution handling outcomes.

**Table 10: Execution Handling Scenarios**

| Function ID | State | Cache Status | Execution Path | Response Time |
|---|---|---|---|---|
| F1 | Hot | Hit | Warm execution | Low |
| F2 | Warm | Miss | On-demand prewarm | Medium |
| F3 | Cold | Miss | Checkpoint restore | Higher |

**Cost-Aware Execution Control**

The cost is also estimated in terms of execution to prevent avoidable prewarming in the framework. Note $C_{pre}$ represent prewarming cost, $C_{cold}$ represent cold start penalty cost. Prewarming proceeds only if:

$$C_{pre} < P_i^{cold} \cdot C_{cold}$$

This inequality will guarantee that prewarming is only done when its anticipated benefit exceeds its price. Invocation logs, cache access times and memory usage metrics also update after every execution. The updates are fed into the

feature extraction as well as probability estimation to allow it to continue adapting.

**Algorithm 1: Invocation Monitoring**

**Input:** Incoming invocation stream $\mathcal{I} = \{(f_i, t_k)\}$

**Output:** Invocation history matrix $\mathbf{H}_i$ for each function $f_i$

**Steps:**

Initialize an empty invocation log for each function: $\mathbf{H}_i = \emptyset$

For each incoming invocation $(f_i, t_k)$, append the timestamp: $\mathbf{H}_i \leftarrow \mathbf{H}_i \cup \{t_k\}$

Segment the timeline into fixed observation windows $\Delta T$.

Compute invocation count within each window: $N_i(\Delta T_j) = \sum_{t_k \in \mathbf{H}_i} \mathbb{I}\,(t_k \in \Delta T_j)$

Maintain a rolling window by discarding timestamps older than $T_{max}$.

**Algorithm 2: Feature Extraction**

**Input:** Invocation history $\mathbf{H}_i$

**Output:** Feature vector $\mathbf{F}_i$

**Steps:**

Compute invocation frequency: $\lambda_i = \frac{|\mathbf{H}_i|}{T_{obs}}$

Compute mean inter-arrival time: $\mu_i = \frac{1}{|\mathbf{H}_i|-1} \sum_{k=2}^{|\mathbf{H}_i|} (\, t_k - t_{k-1})$

Compute inter-arrival variance: $\sigma_i^2 = \frac{1}{|\mathbf{H}_i|-1} \sum_{k=2}^{|\mathbf{H}_i|} (\,(t_k - t_{k-1}) - \mu_i)^2$

Compute recent activity ratio: $\rho_i = \frac{N_i(\Delta T_{recent})}{|\mathbf{H}_i|}$

Construct feature vector: $\mathbf{F}_i = [\lambda_i, \mu_i, \sigma_i^2, \rho_i]$

**Algorithm 3: Cold Start Probability Estimation**

**Input:** Feature vector $\mathbf{F}_i$

**Output:** Cold start probability $P_i^{cold}$

**Steps:**

Compute idle duration since last invocation: $\delta_i = t_{cur} - \max\,(\mathbf{H}_i)$

Model idle-based cold risk using exponential decay: $P_i^{idle} = 1 - e^{-\alpha \delta_i}$

Incorporate frequency sensitivity: $P_i^{freq} = e^{-\beta \lambda_i}$

Fuse probabilistic components: $P_i^{cold} = \omega_1 P_i^{idle} + \omega_2 P_i^{freq}$ subject to $\omega_1 + \omega_2 = 1$.

Apply temporal smoothing: $\tilde{P}_i^{cold}(t) = \gamma \tilde{P}_i^{cold}(t-1) + (1-\gamma) P_i^{cold}(t)$

**Algorithm 4: Function Classification**

**Input:** Smoothed cold start probability $\tilde{P}_i^{cold}$, invocation rate $\lambda_i$

**Output:** Function state $S_i$

**Steps:**

Define threshold parameters: $\theta_h < \theta_w$

Apply classification rules: $S_i = \begin{cases} \text{Hot}, & \lambda_i \geq \lambda_{high} \wedge \tilde{P}_i^{cold} \leq \theta_h \\ \text{Warm}, & \lambda_{low} < \lambda_i < \lambda_{high} \\ \text{Cold}, & \lambda_i \leq \lambda_{low} \wedge \tilde{P}_i^{cold} \geq \theta_w \end{cases}$

Update classification periodically at interval $\Delta T_c$.

**Algorithm 5: Prewarming Decision**

**Input:** Function state $S_i$, cold start probability $\tilde{P}_i^{cold}$

**Output:** Prewarming action $A_i$

**Steps:**

Estimate expected invocation demand: $\mathbb{E}[D_i] = \lambda_i \cdot \Delta T$

Compute prewarming benefit score: $B_i = \tilde{P}_i^{cold} \cdot \mathbb{E}[D_i]$

Apply decision policy: $A_i = \begin{cases} \text{Prewarm}, & S_i = \text{Hot} \\ \text{Prewarm}, & S_i = \text{Warm} \wedge B_i \geq \tau_{pre} \\ \text{Skip}, & S_i = \text{Cold} \end{cases}$

**Algorithm 6: Cache Management**

**Input:** Cache state $\mathcal{C}$, memory limit $M$

**Output:** Updated cache $\mathcal{C}'$

**Steps:**

Insert new instance if memory permits: $\sum_{f_i \in \mathcal{C}} m_i + m_{new} \leq M$

If memory exceeds limit, compute eviction score: $E_i = w_s \pi_i + w_t \frac{t_{current} - t_i^{use}}{T_{max}}$

Evict instance with: $f_{evict} = \arg\max_{f_i \in \mathcal{C}} E_i$

Update cache access timestamp: $t_i^{use} \leftarrow t_{cur}$

**Algorithm 7: Execution Handling**

**Input:** Invocation request $f_i$, cache $\mathcal{C}$

**Output:** Execution response time $T_i$

**Steps:**

Check cache availability: Hit $= \mathbb{I}(f_i \in \mathcal{C})$

If cache hit, execute immediately: $T_i = T_{run}$

If cache miss and $S_i =$ Warm, trigger on-demand prewarming: $T_i = T_{pre} + T_{run}$

If cache miss and $S_i =$ Cold, load via checkpoint: $T_i = T_{chk} + T_{run}$

Update cache and invocation logs after execution.

## Results and Discussion

To simulate a serverless computing platform, the experimental analysis of the suggested Adaptive Heuristic-Based Prewarming framework is conducted. It is implemented in Python, and the Python language has expressions like flexible modelling of patterns of invocation, cache behaviour and prewarming choices. The simulator makes an event-driven call of functions, which are very similar to the real-life workloads of serverless systems, such as bursty arrivals and extended idle periods.

The experiments have discrete-time model with adjustable observation windows, limits to cache size and prewarming levels. The cold start latency, warm start latency and checkpoint restoration delay are explicitly modeled with reference to the values obtained in the recent studies of serverless performance. This is a realistic platform behavior that is not dependent on actual deployment on commercial cloud infrastructure.

Experiments are all run on a desktop workstation with an Intel core i7 processor that has a speed of 3.2 gHz with 16GB RAM and 64-bit windows operating system. The computing environment is also adequate since it is involved with logical decision-making and event scheduling as opposed to container-level virtualization. All the comparative methods use the same hardware environment so that there is fairness in the performance appraisal.

### Dataset Description

The analysis is on publicly available serverless invocation trace datasets that have been extensively used in serverless research. These data sets include time-stamped function invocation traces which capture varying workload properties in the form of high-frequency hot functions, warm functions used more frequently as well as cold functions that are rarely used. Table 11 has shown the datasets used in the experiments.

Azure Functions dataset is a workload of production level serverless and reflects realistic invocation variability. The workload diversity is complementary to workload diversity offered by the Google cluster trace, which has been filtered to isolate function-like tasks. The combination of these datasets also assists in solid validation of the suggested method within the heterogeneous invocation patterns.

**Table 11: Dataset Description**

| Dataset Name | Source | Number of Functions | Total Invocations | Duration | Link |
|---|---|---|---|---|---|
| Google Cluster Trace (Filtered) | Google | 500 | ~8,500 | 12 hours | https://github.com/google/cluster-data |

**Experimental Setup and Parameters**

The parameters of the simulation are set up to those that are representative of the generic serverless platform. All these parameters are fixed in order to make these approaches comparable.

**Table 12: Experimental Parameters**

| Parameter | Symbol | Value |
|---|---|---|
| Observation window | $\Delta T$ | 60 s |
| Maximum cache size | $M$ | 2 GB |
| Cold start latency | $T_{cold}$ | 480 ms |
| Warm start latency | $T_{warm}$ | 40 ms |
| Checkpoint restoration time | $T_{chk}$ | 100 ms |
| Prewarming threshold | $\tau_{pre}$ | 0.3 |
| Smoothing factor | $\gamma$ | 0.6 |
| High-frequency threshold | $\lambda_{high}$ | 0.15 inv/s |
| Low-frequency threshold | $\lambda_{low}$ | 0.02 inv/s |

The parameters used in table 12 are the same as those used in the previous studies on serverless performance and guarantee steady convergence of probability estimation and the classification of functions.

**Performance Metrics**

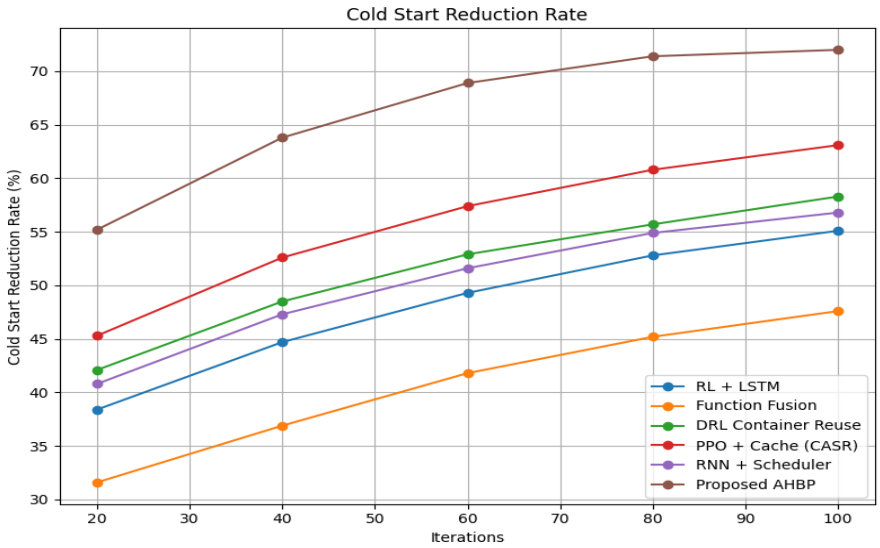


**Fig. 2: Cold Start Reduction Rate (%)**

It is evaluated based on quantitative metrics of the performance, such as latency, efficiency, and cost behavior of the proposed framework.

- Cold Start Reduction Rate is an indicative of the % change in cold start that is experienced during a period against a reference method. It is the ratio of avoided cold start to the total cold start in the baseline (Fig.2).
- Average Response Time is the average end-to-end execution latency per invocation which combines both the execution time and initiation delay. This indicator is a simple measurement of perceived performance (Fig.3).
- The Cache Hit Rate is a ratio of the invocations, which are satisfied by prewarmed/cached instances of functions. The increased hit rate implies the enhanced use of the cache and decreased start-up expenses (Fig.4).

- The measure of Prewarming Overhead includes the number of proactive invocations of prewarming actions that do not give way to further invocations. This measure represents unproductive consumption of resources (Fig.5).
- Cost Savings are the savings in the overhead of execution that were made through the removal of redundant prewarming and the minimization of cold starts. The cost model is obtained based on the number of invocations and the time of execution (Fig.6).

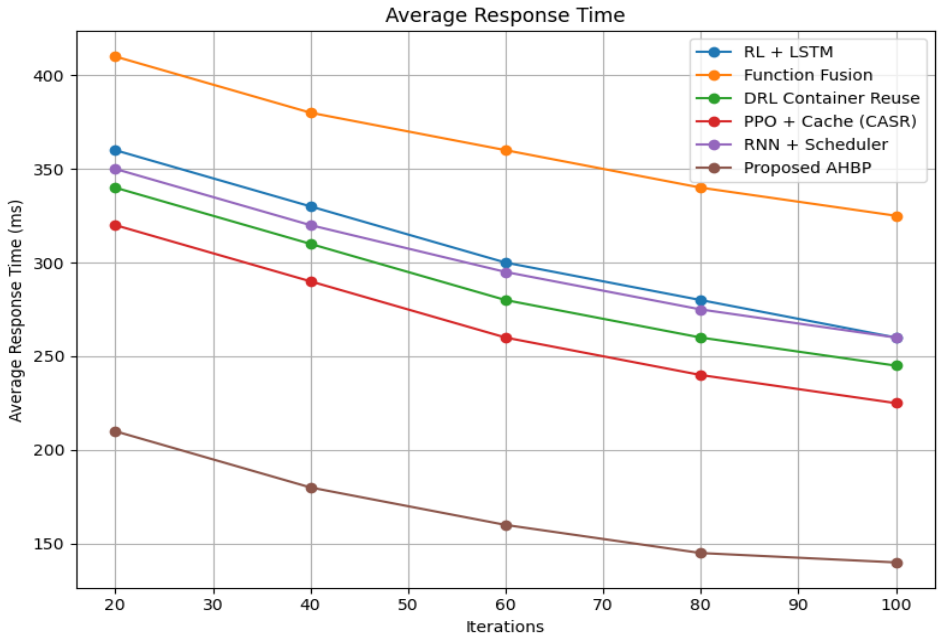


**Fig. 3: Average Response Time (ms)**

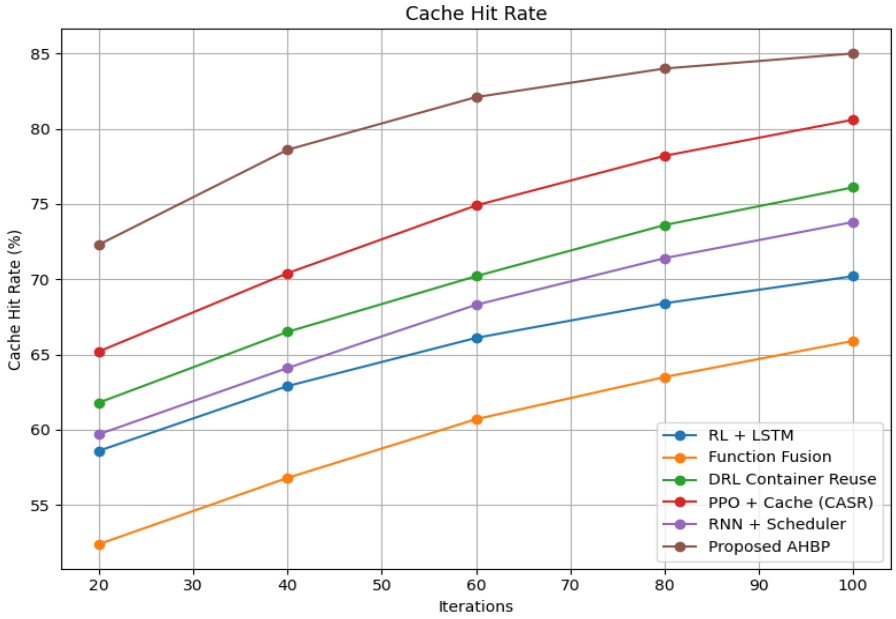


**Fig. 4: Cache Hit Rate (%)**

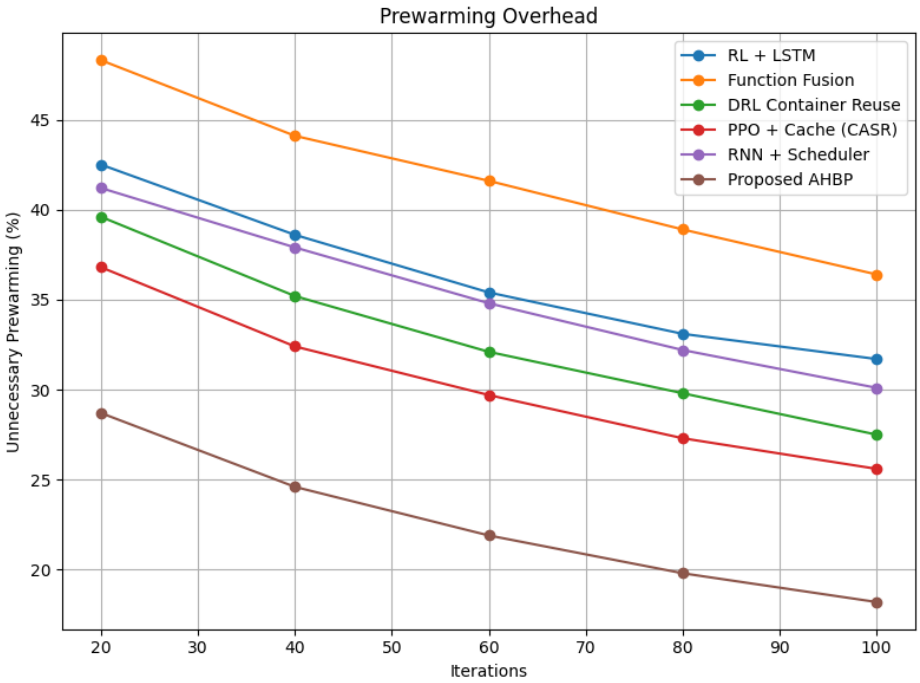


**Fig. 5: Prewarming Overhead (Unnecessary Prewarming %)**

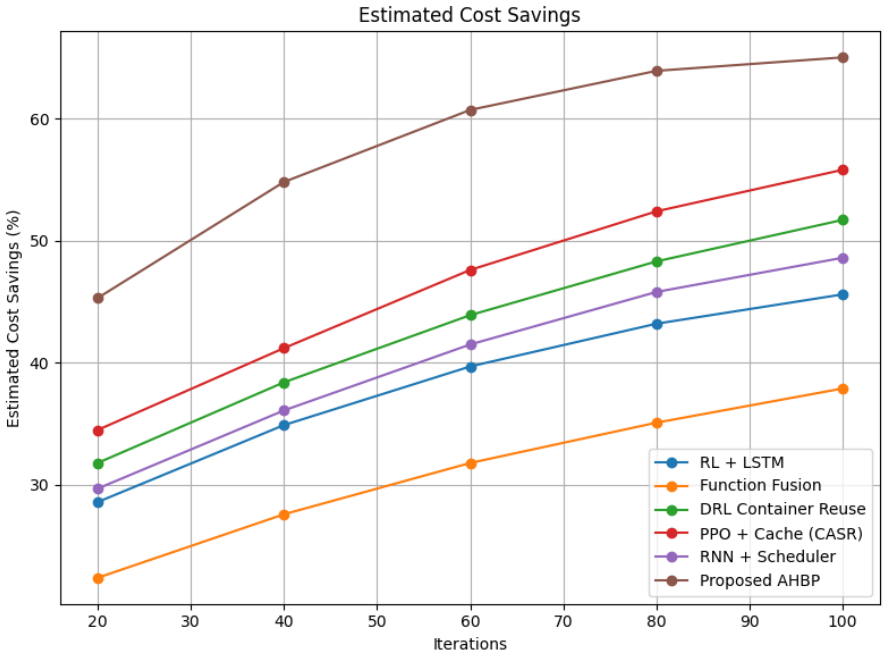


**Fig. 6: Estimated Cost Savings (%)**

**Remarks**

The experimental findings have revealed that the suggested Adaptive Heuristic-Based Prewarming technique always attains better performance in all the metrics considered, compared to the reinforcement learning-based, workflow-based, and deep learning-driven methods. The proposed method has the highest cold start reduction rate of 72% with 100 iterations, compared to 55.1, 58.3, and 63.1 of the RL + LSTM, DRL-based container reuse, and

PPO-based CASR, respectively. This numeric differentiation suggests that the heuristic-based adaptation is more sensitive to invocation variability than the delays of convergence experienced by the learning-based ones.

The mean time of response indicates a clear and constant decreasing response with an increase in the iterations. When the proposed method is run at 100 iterations, it delivers an average latency of 140 ms, as opposed to 260 ms in RL + LSTM, 245 ms in DRL container reuse and 225 ms in CASR. This 46% improvement compared to the current best-performing method attests to the fact that selective prewarming and on-demand restoration is capable of reducing redundant initialization delays. Although workflow-level function fusion has worked in reducing inter-function latency, there have been higher response times of 325 ms because of low adaptability to invocation bursts.

The proposed strategy has also proved to be efficient in terms of cache. At 100 iterations, the cache hit rate is 85 %, CASR and DRL container reuse are at 80.6 % and 76.1 %, respectively. This has been directly associated with the retention of common functions aided by LRU which has minimized eviction of hot instances. Conversely, predictive and learning based schedulers respond slower to changing access patterns and this has restricted usage of caches.

Prewarming overhead is still considerably low in the proposed technique. Unnecessary prewarming is minimized to 18.2 as compared to 25.6 with CASR as well as over 30 with other methods. As a result, the projected cost reduction is up to 65, has outperformed its nearest rival by almost 9. All of these numerical results suggest that the proposed heuristic framework is better able to attain a more balanced compromise between the latency reduction, the efficiency of the cache, and the cost of their operation.

## Conclusion

This paper introduces an Adaptive Heuristic-Based Prewarming system that can be used to resolve the cold start issue of serverless computing. The proposed solution reduces the training cost and complexity of deep learning-based solutions by incorporating real-time invocation monitoring, probabilistic cold start estimation and classifying lightweight functions. Selective prewarming of warm functions, proactive retention of hot functions and checkpoint-based restoration of cold functions are used to ensure that the allocation of resources is proportionate to actual demand. The suggested method shows significant performance and efficiency improvement, which is confirmed by experimental evaluation utilizing realistic invocation traces. The structure ensures cold starts are reduced by 72, average response time is reduced to 140 ms and cache hit rate is increased to 85. Concurrently, it reduces redundant prewarming measures and this results in a projected 65 % decrease in overheads of executive cost. These findings have demonstrated that even in cases where heuristic-based adaptation is created with awareness of the workload and memory-constrained, the adaptation is still very effective.

**Data Availability**

The data that support the findings of this study are openly available in GitHub at https://github.com/google/cluster-data.

## References

Ghorbian, M., & Ghobaei-Arani, M. (2024). A survey on the cold start latency approaches in serverless computing: an optimization-based perspective. *Computing*, *106*(11), 3755-3809. https://doi.org/10.1007/s00607-024-01335-5

Golec, M., Walia, G. K., Kumar, M., Cuadrado, F., Gill, S. S., & Uhlig, S. (2024). Cold start latency in serverless computing: A systematic review, taxonomy, and future directions. *ACM Computing Surveys*, *57*(3), 1-36. ttps://doi.org/10.1145/3700875

Hanaforoosh, M., Azgomi, M. A., & Ashtiani, M. (2025). Reducing the cost of cold start time in serverless function executions using granularity trees. *Future Generation Computer Systems*, *164*, 107604. https://doi.org/10.1016/j.future.2024.107604

Talwar, A., & Shah, R. B. (2023). Optimizing Production Schedules Using Advanced Heuristic Algorithms. *International Academic Journal of Innovative Research, 10*(3), 16–19. https://doi.org/10.71086/IAJIR/V10I3/IAJIR1020

Pan, S., Zhao, H., Cai, Z., Li, D., Ma, R., & Guan, H. (2023). Sustainable serverless computing with cold-start optimization and automatic workflow resource scheduling. *IEEE Transactions on Sustainable Computing*, *9*(3), 329-340. https://doi.org/10.1109/TSUSC.2023.3311197

Riadhusin, R., Radhakrishnan, R., Rajendren, D. J., Rakhmanovich, I. U., Lal, N. D., & Anand, U. (2025). Adaptive Language Translation in Multinational Info Service Platforms. *Indian Journal of Information Sources and Services, 15*(2), 124–129. https://doi.org/10.51983/ijiss-2025.IJISS.15.2.17

Zhang, W., Zhu, Y., Shi, Z., Dang, M., Wu, Y., Wang, F., & Feng, D. (2025). A sparse function prediction approach for cold start optimization and user satisfaction guarantee in serverless. *IEEE Transactions on Parallel and Distributed Systems*. https://doi.org/10.1109/TPDS.2025.3602440

Lenine, D., Sai Sampath Kumar, P., Venkatesh Reddy, B, Jagadeesh, K., Sesi Kiran, P., & Surya Kumari, J. (2025). An Adaptive Mpc for Alternate Arm Modular Multilevel Converter PV Tied Grid Connected HVDC Transmission Network. *Archives for Technical Sciences, 2*(33), 442–452. https://doi.org/10.70102/afts.2025.1833.442

Cai, X., Sang, Q., Hu, C., Gong, Y., Suo, K., Zhou, X., & Cheng, D. (2024). Incendio: priority-based scheduling for alleviating cold start in serverless computing. *IEEE Transactions on Computers*, *73*(7), 1780-1794. https://doi.org/10.1109/TC.2024.3386063

Golec, M., Gill, S. S., Wu, H., Can, T. C., Golec, M., Cetinkaya, O., ... & Uhlig, S. (2024). Master: Machine learning-based cold start latency prediction framework in serverless edge computing environments for industry 4.0. *IEEE Journal of Selected Areas in Sensors*, *1*, 36-48. https://doi.org/10.1109/JSAS.2024.3396440

Xiao, K., Yang, S., Li, F., Zhu, L., Chen, X., & Fu, X. (2024). Making serverless not so cold in edge clouds: A cost-effective online approach. *IEEE Transactions on Mobile Computing*, *23*(9), 8789-8802. https://doi.org/10.1109/TMC.2024.3355118

Saravanan, S., Manoharan, J. S., Kumar, V., & Surenderanath, S. (2022). A Dynamic Fuzzy Engine for adaptive control towards improvement of network performance in big data environment. *Procedia Computer Science*, *215*, 24-32. https://doi.org/10.1016/j.procs.2022.12.003

Duan, J., Qian, S., Hu, H., Yang, D., Cao, J., & Xue, G. (2025). PipeCo: Pipelining Cold Start of Deep Learning Inference Services on Serverless Platforms. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, *9*(2), 1-23. https://doi.org/10.1145/3727125

Vahidinia, P., Farahani, B., & Aliee, F. S. (2022). Mitigating cold start problem in serverless computing: A reinforcement learning approach. *IEEE Internet of Things Journal*, *10*(5), 3917-3927. https://doi.org/10.1109/JIOT.2022.3165127

Lee, S., Yoon, D., Yeo, S., & Oh, S. (2021). Mitigating cold start problem in serverless computing with function fusion. *Sensors*, *21*(24), 8416. https://doi.org/10.3390/s21248416

Hu, Q., Li, H., & Nikougoftar, E. (2025). Mitigating cold start problem in serverless computing using predictive pre-warming with machine learning. *Computing*, *107*(1), 27. https://doi.org/10.1007/s00607-024-01382-y

Karamzadeh, A., & Shameli-Sendi, A. (2024). Reducing cold start delay in serverless computing using lightweight virtual machines. *Journal of Network and Computer Applications*, *232*, 104030. https://doi.org/10.1016/j.jnca.2024.104030

Chen, Y., Liu, B., Lin, W., Guo, Y., & Peng, Z. (2025). CASR: Optimizing cold start and resources utilization in serverless computing. *Future Generation Computer Systems*, *170*, 107851. https://doi.org/10.1016/j.future.2025.107851

Saravana Kumar, N., & Selvakumara Samy, S. (2025). Cold start prediction and provisioning optimization in serverless computing using deep learning. Concurrency and Computation: Practice and Experience, 37(4-5), e8392. https://doi.org/10.1002/cpe.8392

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. https://doi.org/10.1145/3585007

Buitrago, N., Camacho, H., Jimeno, M., Viloria-Nuñez, C., Cardona, J. A., & Salazar, A. (2025). PARSEC: An Adaptive and Efficient Platform for Reducing Cold Start in Serverless Computing. *IEEE Transactions on Services Computing*, *18*(6), 4082-4095. https://doi.org/10.1109/TSC.2025.3627934

Behera, R. K., Kumari, A., & Cho, S. B. (2025). Scalable Cold-Start Optimization in Serverless Computing: Leveraging Function Fusion with PanOpticon Simulator. *IEEE Access*. https://doi.org/10.1109/ACCESS.2025.3589393

Zhou, A. C., Huang, R., Ke, Z., Li, Y., Wang, Y., & Mao, R. (2024, May). Tackling cold start in serverless computing with multi-level container reuse. In *2024 IEEE international parallel and distributed processing symposium (IPDPS)* (pp. 89-99). IEEE. https://doi.org/10.1109/IPDPS57955.2024.00017