Reproduction and Extension of CASR: A Cache-Based Adaptive Scheduler for Serverless Computing with Novel K=4 Queue Granularity Analysis

Anmol Krishna
School of Computer Engineering
KIIT University
Bhubaneswar, India
anmolkrishna80@gmail.com

***Abstract*—Serverless computing has emerged as a dominant cloud paradigm where functions are executed on-demand. Cold starts remain a critical performance bottleneck causing delays of 0.1 to 80 seconds per invocation. This paper presents an independent reproduction and experimental extension of CASR (Cache-Based Adaptive Scheduler for Serverless Runtime), originally proposed by Chen et al. [3]. CASR combines W-TinyLFU caching with Proximal Policy Optimization reinforcement learning to minimize cold start latency and wasted memory time simultaneously. We evaluate our implementation on the Microsoft Azure Functions 2019 dataset containing 1,332,032 daily invocations against five baseline algorithms across three workload types. CASR eliminates wasted memory time across all evaluated workloads whereas baseline policies incur measurable memory waste. CASR reduces cold start rate by up to 14.929 percentage points compared to FaaSCache. We extend the original work by investigating K=4 queue granularity, finding a reduction of up to 5.9 percentage points in cold start rate over the original K=3 design. All implementation code is available at: https://github.com/Krishn4nmol/CASR_Project**

***Index Terms*—serverless computing, cold start optimization, reinforcement learning, PPO, caching, W-TinyLFU, container management, FaaS, reproducibility study**

I. Introduction

Serverless computing, also known as Function-as-a-Service (FaaS), has transformed cloud application development by allowing developers to deploy individual functions without managing server infrastructure. Major providers including AWS Lambda, Google Cloud Functions, and Microsoft Azure Functions have adopted this model [1].

Despite its advantages, serverless computing suffers from the cold start problem. When a function is invoked after inactivity, the provider must initialize a new container, download function code, and install dependencies before execution begins. This introduces latency from 0.1 seconds for simple HTTP functions to over 80 seconds for complex machine learning workloads [2] [8].

Existing approaches either minimize cold starts at the expense of high memory waste, or minimize memory waste while suffering from frequent cold starts. CASR, proposed by Chen, Liu, Lin, Guo and Peng [3], addresses this trade-off through a stratified cache (S-Cache) and a PPO reinforcement learning agent.

This work is primarily a reproduction and implementation study of the CASR framework proposed by Chen et al. [3]. Our main contribution is an independent Python implementation verified on real Azure Functions data, and an experimental investigation of K=4 queue granularity which was not explored in the original paper.

The contributions of this paper are:

- Independent Python implementation of CASR including W-TinyLFU caching and PPO reinforcement learning agent
- Evaluation on the Microsoft Azure Functions 2019 dataset against five baseline algorithms across three workload types
- Novel K=4 queue granularity experiment extending the original K=3 design showing up to 5.9 percentage point cold start reduction
- Analysis of the cold start versus memory efficiency trade-off under constrained simulation conditions

The remainder of this paper is organized as follows. Section II provides background. Section III describes our implementation. Section IV presents experimental setup. Section V presents K=3 results. Section VI presents the K=4 experiment. Section VII discusses findings. Section VIII concludes.

II. Background and Related Work

A. Serverless Computing

Serverless platforms execute stateless functions in isolated containers. Each invocation either reuses a warm container or initializes a new one (cold start). Overhead depends on function complexity and required libraries [1].

B. Cold Start Problem

Cold starts significantly impact user experience. Golec et al. [8] provide a comprehensive taxonomy of cold start solutions, classifying approaches into caching-based, AI/ML-based, and

application-level optimization categories. Shahrad et al. [2] characterized Azure workloads showing high invocation variability that makes cold start prediction challenging.

C. Container Keep-Alive Strategies

Fixed TTL: Containers are kept warm for a fixed duration. Simple but wastes memory for infrequently invoked functions.

Histogram-based: Historical invocation patterns predict future calls. Achieves low cold start rates but assumes stationary workloads.

FaaSCache: A greedy-dual caching approach for container management [4].

D. W-TinyLFU Caching

W-TinyLFU combines a window cache for new items with a main cache for frequent items using a frequency sketch for popularity estimation [5]. CASR adapts this algorithm for container lifecycle management.

E. Proximal Policy Optimization

PPO is a stable policy gradient algorithm with a clipped objective function [6]. Its stability makes it suitable for dynamic resource management in cloud environments.

III. Implementation

A. System Overview

Fig. 1 illustrates the overall system architecture of our CASR implementation.

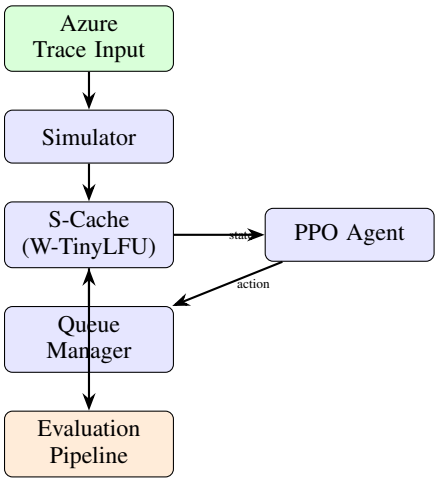

Fig. 1. CASR System Architecture. Azure function traces feed into the simulator which drives the S-Cache. The PPO agent observes S-Cache state and issues scaling actions to the Queue Manager every $\Delta = 10,000$ invocations.

The implementation comprises eight Python modules: config.py for hyperparameters, simulator.py for dataset loading, scache.py for W-TinyLFU S-Cache, environment.py for the RL environment, ppo_agent.py for the PPO agent, baselines.py for comparison algorithms, train.py for the training procedure, and evaluate.py for evaluation.

B. Implementation Environment

Experiments were conducted with the following specifications:

- OS: Windows 11
- Processor: AMD Ryzen 7 8840HS with Radeon 780M Graphics
- RAM: 32GB
- Python: 3.11.9
- PyTorch: 2.11.0
- NumPy: 2.4.4
- Gymnasium: 1.3.0

C. S-Cache Design

The S-Cache organizes containers into K queues based on cold start overhead. For K=3:

TABLE I
S-Cache Queue Structure (K=3)

Queue	Range	Type	Azure %
0	0–1s	Lightweight	9.4%
1	1–60s	Medium	85.3%
2	60+s	Heavy ML	5.0%

Each queue uses W-TinyLFU with a window cache (20% capacity) and main cache (80% capacity) [5]. A key innovation from [3] accounts for busy containers:

$$C_{available} = C_{total} - C_{running} \tag{1}$$

where $C_{available}$ is containers available for new requests, C_{total} is total containers in memory, and $C_{running}$ is currently executing containers.

D. PPO Agent Design

The agent scales queue capacities every $\Delta = 10,000$ invocations [6].

State: 7 metrics per queue (capacity, length, invocations, cold starts, evictions, running containers, WMT). K=3 gives 21 dimensions; K=4 gives 28 dimensions.

Actions: Expand (+25%), shrink (-25%), or maintain each queue. K=3 gives $3^3 = 27$ actions; K=4 gives $3^4 = 81$ actions.

Reward: Following [3]:

$$R = -(\theta \cdot R_1^{norm} + (1-\theta) \cdot R_2^{norm}) \tag{2}$$

where $\theta = 0.8$, R_1 represents cold starts per step, and R_2 represents change in wasted memory time.

E. Baseline Algorithms

Five baseline algorithms are implemented:

- **S-Cache:** W-TinyLFU without RL scaling
- **LCS:** Least Cold Start eviction policy
- **FaaSCache:** Greedy-dual caching [4]
- **Hist:** Histogram-based keep-alive policy
- **Fixed:** Fixed 10-minute keep-alive policy

IV. Experimental Setup

A. Dataset

Experiments use the Microsoft Azure Functions 2019 dataset [7] containing real anonymized production traces. Day 1 contains 1,332,032 invocations:

TABLE II
Azure Dataset Distribution

Queue	Range	Calls	Pct.
0	0–1s	124,663	9.4%
1	1–60s	1,135,757	85.3%
2	60+s	66,988	5.0%

The top 2,000 most frequent functions are selected to simulate a single server workload, yielding 100,000 evaluation calls per workload.

B. Workload Types

- **Common:** Top 2,000 most frequent functions from Day 1
- **Significant:** Top 2,000 high overhead functions from Day 2
- **Random:** 2,000 randomly selected functions from Day 3

C. Training Configuration

The PPO agent trains for 200 episodes with 100,000 calls per episode, completing in approximately 5 minutes. Hyperparameters follow [3]: learning rate 0.001, discount factor 0.63, GAE lambda 0.95, PPO clip 0.2, $\theta = 0.8$, $\Delta = 10,000$.

V. K=3 Evaluation Results

A. Training Convergence

Training achieved a best reward of -0.0447 over 200 episodes. Wasted memory time remained at 0.000 seconds throughout all training episodes, indicating consistent zero memory waste maintenance.

B. Results

TABLE III
K=3 Results: Common Workload

Algorithm	Cold%	WMT(s)	OH(s)
CASR	88.972	0.000	8.974
S-Cache	89.962	0.000	9.078
LCS	87.152	7.563	9.017
FaaSCache	99.999	0.000	9.646
Hist	61.214	24.970	5.962
Fixed	50.673	1.122	4.900

TABLE IV
K=3 Results: Significant Workload

Algorithm	Cold%	WMT(s)	OH(s)
CASR	96.152	0.000	10.679
S-Cache	94.164	0.000	10.486
LCS	88.142	7.011	10.434
FaaSCache	100.000	0.000	11.132
Hist	61.642	25.705	6.929
Fixed	50.700	1.092	5.656

TABLE V
K=3 Results: Random Workload

Algorithm	Cold%	WMT(s)	OH(s)
CASR	85.070	0.000	9.145
S-Cache	85.842	0.000	9.211
LCS	72.705	5.583	8.329
FaaSCache	99.999	0.000	9.773
Hist	61.324	12.282	6.013
Fixed	51.151	8.719	5.018

C. Analysis

Simulation Context: The observed cold start rates are artifacts of the constrained single-server simulation with limited cache capacity and aggressive eviction policies. The results should therefore be interpreted comparatively across algorithms rather than as representative production deployment metrics.

Memory Efficiency: CASR eliminates wasted memory time in all evaluated workloads, whereas Hist and Fixed policies incur measurable memory waste. Hist incurs up to 25.705 seconds WMT per invocation on the Significant workload while Fixed incurs up to 8.719 seconds on the Random workload.

Cold Start vs FaaSCache: CASR reduces cold start rate by 11.027 percentage points on Common (88.972% vs 99.999%), 3.848 percentage points on Significant (96.152% vs 100.000%), and 14.929 percentage points on Random (85.070% vs 99.999%).

Hist and Fixed Behavior: Hist and Fixed achieve lower cold start rates by retaining all containers indefinitely. This strategy is effective in the constrained 2,000-function simulation but would be unsustainable at production scale with millions of functions competing for memory.

VI. Novel K=4 Experiment

A. Motivation

The original paper [3] evaluates only K=3 queues. Dataset analysis shows 85.3% of Azure calls fall in the 1–60 second range. Splitting this dominant range into finer granularity may allow the agent to apply differentiated management strategies for medium overhead functions.

B. K=4 Queue Design

Queue 1 (1–60s) is divided into two sub-queues:

TABLE VI
S-CACHE QUEUE STRUCTURE (K=4)

Queue	Range	Type
0	0–1s	Lightweight
1	1–30s	Medium Light
2	30–60s	Medium Heavy
3	60+s	Heavy ML

State dimensions increase from 21 to 28 and action space from $3^3 = 27$ to $3^4 = 81$. Training achieved a best reward of -0.1067 over 200 episodes.

C. K=4 Results

Both K=3 and K=4 eliminate wasted memory time across all workloads. Table VII presents cold start comparisons.

TABLE VII
K=3 VS K=4 CASR COMPARISON

Workload	K=3 (%)	K=4 (%)	Reduction (pp)
Common	88.972	88.811	0.161
Significant	96.152	90.252	5.900
Random	85.070	81.655	3.415

D. Analysis

K=4 reduces cold start rate across all workload types. The Significant workload shows the largest reduction at 5.900 percentage points. Training results indicate the agent allocated capacity to Queue 1 (1–30s) and Queue 2 (30–60s) differently in K=4 compared to the merged Queue 1 in K=3, enabling more targeted resource allocation for the dominant medium overhead range.

VII. DISCUSSION

A. Trade-off Analysis

Results confirm CASR effectively balances cold start rate and memory efficiency. Algorithms prioritizing cold start reduction sacrifice memory efficiency significantly. CASR eliminates wasted memory time while achieving competitive cold start rates, a property critical for sustainable production deployments.

B. Effect of Queue Granularity

The K=4 experiment demonstrates that finer queue granularity consistently reduces cold start rates. The improvement is most pronounced on the Significant workload (5.900 percentage points), containing high overhead functions that benefit most from differentiated management. Splitting the dominant 1–60 second range enables more precise capacity allocation across function types.

C. Comparison with Original Paper

The original paper [3] reports 38.75% cold start reduction under common workloads. Our reproduction shows higher absolute cold start rates due to simulation constraints. The key findings are consistent: CASR eliminates wasted memory time and reduces cold start rates compared to FaaSCache across all workload types. Differences in magnitude reflect the simplified single-server evaluation environment.

D. Limitations

- Single-server simulation versus multi-server cloud deployment
- 2,000 functions versus millions in production environments
- 200 training episodes may not represent full convergence
- Container warm-up phases not modeled in simulation
- Results limited to Azure 2019 dataset characteristics

VIII. CONCLUSION

This paper presents a reproduction and experimental extension of CASR for serverless cold start optimization. Key findings:

1) CASR eliminates wasted memory time in all evaluated workloads, whereas Hist and Fixed policies incur measurable memory waste of up to 25.705 and 8.719 seconds respectively
2) CASR reduces cold start rate by 3.848 to 14.929 percentage points compared to FaaSCache across all three workload types
3) K=4 queue granularity reduces cold start rate by up to 5.900 percentage points over K=3 while maintaining zero wasted memory time
4) The observed high cold start rates are artifacts of the constrained simulation and should be interpreted comparatively rather than as absolute production metrics

Future directions include extended training beyond 200 episodes, K=5 queue evaluation, multi-server deployment testing, and statistical analysis across multiple independent runs.

ACKNOWLEDGMENT

The author thanks Microsoft Azure for making the Azure Functions 2019 dataset publicly available and Chen et al. [3] for the original CASR framework that motivated this study.

REFERENCES

[1] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking Behind the Curtains of Serverless Platforms," in *Proc. 2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018, pp. 133–146.

[2] M. Shahrad, R. Fonseca, I. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider," in *Proc. 2020 USENIX Annual Technical Conference (USENIX ATC 20)*, Jul. 2020, pp. 205–218.

[3] Y. Chen, B. Liu, W. Lin, Y. Guo, and Z. Peng, "CASR: Optimizing cold start and resources utilization in serverless computing," *Future Generation Computer Systems*, vol. 170, p. 107851, 2025, doi: 10.1016/j.future.2025.107851.

[4] A. Fuerst and P. Sharma, "FaasCache: Keeping Serverless Computing Alive with Greedy-Dual Caching," in *Proc. 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 21)*, New York, NY, USA, 2021, pp. 386–400, doi: 10.1145/3445814.3446757.

[5] G. Einziger, R. Friedman, and B. Manes, "TinyLFU: A Highly Efficient Cache Admission Policy," *ACM Trans. Storage*, vol. 13, no. 4, article 35, pp. 1–31, Nov. 2017, doi: 10.1145/3149371.

[6] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal Policy Optimization Algorithms," *arXiv preprint arXiv:1707.06347*, 2017. [Online]. Available: https://arxiv.org/abs/1707.06347

[7] M. Shahrad et al., "Azure Functions Traces," GitHub, 2020. [Online]. Available: https://github.com/Azure/AzurePublicDataset

[8] M. Golec, G. K. Walia, M. Kumar, F. Cuadrado, S. S. Gill, and S. Uhlig, "Cold Start Latency in Serverless Computing: A Systematic Review, Taxonomy, and Future Directions," *ACM Comput. Surv.*, vol. 57, no. 3, article 65, pp. 1–36, Nov. 2024, doi: 10.1145/3700875.