Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

BudgetFunnel: Coarse-to-Fine Global Budget Allocation for Trace-Driven Serverless Cold-Start Mitigation

xuming wu

wuxm@lingnan.edu.cn

Lingnan Normal University

Zhipeng Ye

Lingnan Normal University

Gaoyuan Zhao

Lingnan Normal University

Zihao Lu

Lingnan Normal University

Jiayi Hong

Lingnan Normal University

Wen-Yu Peng

Lingnan Normal University

Zhibin Shao

Shenzhen Polytechnic University

Yan Cao

Luoyang Normal University

Research Article

Keywords: Serverless computing, Cold start, Trace-driven simulation, Prewarming and keep-alive (TTL), Warm-budget allocation, Coarse-to-fine optimization

Posted Date: October 1st, 2026

DOI: https://doi.org/10.21203/rs.3.rs-11046519/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: No competing interests reported.

BudgetFunnel: Coarse-to-Fine Global Budget Allocation for Trace-Driven Serverless Cold-Start Mitigation

Zhipeng Ye[3], Zihao Lu[4], Yuan Zhang[4], Jiayi Hong[4], Wen-Yu Peng[2], Zhibin Shao[5], Xuming Wu[1*], Gaoyuan Zhao[2*], Yan Cao[6]

[1]College of Physical Science and Technology, Lingnan Normal University, Zhanjiang, 524048, China.
[2]School of Mechatronic Engineering, Lingnan Normal University, Zhanjiang, 524048, China.
[3]College of Chemistry and Chemical Engineering, Lingnan Normal University, Zhanjiang, 524048, China.
[4]College of Computer Science and Intelligent Education, Lingnan Normal University, Zhanjiang, 524048, China.
[5]School of Undergraduate Education, Shenzhen Polytechnic University, Shenzhen, 518055, China.
[6]College of Physics and Electronic Information & Research Center for Novel Solar-Blind Ultraviolet and Infrared Photoelectronic Detectors of Henan Province, Luoyang Normal University, Luoyang, 471934, China.

*Corresponding author(s). E-mail(s): wuxm@lingnan.edu.cn; z1797808078@163.com;
Contributing authors: z1662903252@outlook.com; arithmo9@gmail.com; z757996@163.com; llevaii323@gmail.com; valentinsuanne@gmail.com; zhibin_shao@szpu.edu.cn; caoyan@lynu.edu.cn;

Abstract

Serverless cold starts remain a major source of performance degradation, whereas aggressive prewarming can incur substantial warm-resource cost. We present BudgetFunnel, a coarse-to-fine trace-driven framework for allocating a shared warm-retention budget across heterogeneous serverless functions. The framework first screens candidate functions with lightweight workload proxies, evaluates candidate TTL choices using trace-driven simulation, and then applies a precedence-aware greedy allocator. We use the term *global* to denote workload-wide coordination under a shared budget; the greedy allocator is not claimed to be globally optimal. On an audited Azure serverless trace and the Alibaba Cluster Trace–Microservices v2021 workload (used as a cross-workload request-pattern robustness trace), evaluating only 10% of allocation entities ($\gamma = \mathbf{0.10}$) yields about $\mathbf{9}\times$ end-to-end speedup while keeping request-weighted CSR within 1.54% and 1.93%, respectively, of full-evaluation greedy at $\mathbf{1} \times \boldsymbol{B_{ref}}$. Across the full budget sweep, the maximum relative CSR gap to full-evaluation greedy is 2.48% on Azure and 2.16% on Alibaba. On Alibaba, BudgetFunnel reduces CSR by up to 39.4% relative to Global TTL and by up to 21.8% relative to IAT-Mean TTL under matched budgets. Same-scope exact MILP experiments show that the greedy allocator is within 1.42–2.60% of the certified optimum for reduced Azure workloads of 50–200 functions. Request-weighted and slot-level cold-counting semantics preserve the method ranking, and additional candidate-ratio, TTL-grid, cold-penalty, reference-budget, resource-utilization, and provider-specific robustness analyses support the conclusions. Because the public traces are aggregated, latency results are treated as model-based sensitivity analyses rather than direct production measurements.

Keywords Serverless computing, Cold start, Trace-driven simulation, Prewarming and keep-alive (TTL), Warm-budget allocation, Coarse-to-fine optimization

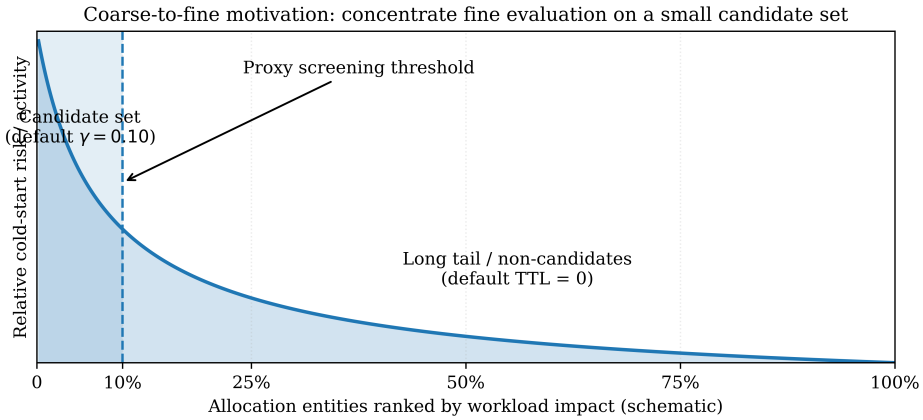


Fig. 1 Coarse-to-fine motivation (schematic). BudgetFunnel uses inexpensive workload proxies to concentrate fine-grained TTL evaluation on a small candidate set (default $\gamma = 0.10$), while non-candidates remain at the default TTL of zero. The curve is schematic and is not presented as an empirical cumulative distribution; the audited Azure workload-skew anchor is reported separately in Fig. 4

1 Introduction

Serverless computing, also known as Function-as-a-Service (FaaS), has become a widely used execution model for cloud-native applications because of its pay-as-you-go pricing and reduced operational overhead [1]. A persistent performance challenge is the *cold start*: when no suitable warm execution environment is available, the platform must provision or restore one before serving the invocation. Production studies show that cold starts can materially affect end-to-end and tail latency [2–4]. Serverless inference workloads further amplify the issue because initialization can include container, runtime, and model-loading overheads [5, 6].

Cold-start mitigation therefore creates a resource trade-off. Keeping execution environments warm can reduce future cold starts, but consumes warm-retention resources during idle intervals. Workload heterogeneity makes this trade-off particularly important: the Azure serverless trace and Alibaba microservice workload used in our evaluation exhibit substantial variation in activity, sparsity, and burstiness across allocation entities [7, 8]. A uniform keep-alive policy can consequently over-warm sparse functions while under-protecting high-impact functions.

Figure 1 illustrates the coarse-to-fine motivation and candidate concentration.

Prior work spans system-level startup acceleration, caching and keep-alive policies, workload prediction, and scheduling [9–14]. Table 1 summarizes representative methods by decision scope, system integration requirements, and primary focus. These approaches are complementary to the question studied here. We do not claim that earlier systems are inherently incapable of global coordination; rather, many were not originally formulated for the specific problem of allocating a *shared warm-retention budget* over many functions using only aggregated traces.

We therefore study the following question: *can most expensive per-function TTL evaluations be avoided without materially degrading the resulting shared-budget allocation?* We propose BudgetFunnel, a coarse-to-fine trace-driven framework with three stages: lightweight proxy screening, fine-grained candidate TTL evaluation, and precedence-aware greedy budget allocation. In this paper, *global* means workload-wide coordination under one shared budget; it does not mean that the greedy procedure is mathematically globally optimal.

The revised evaluation explicitly separates two approximation sources. First, screening quality is measured against *full-evaluation greedy* ($\gamma = 1$), which uses the same allocator but evaluates all functions. Second, allocation quality is measured against a same-scope exact MILP on identical reduced workloads, thereby avoiding the scope mismatch of the previous top-200 comparison. We also quantify sensitivity to request-weighted versus slot-level cold counting, proxy features, TTL grids, cold-penalty models, trace windows, and the definition of B_{ref}.

Our contributions are as follows:

- **Shared-budget formulation.** We formulate TTL-based cold-start mitigation as workload-wide warm-retention allocation under a common resource budget.
- **Coarse-to-fine evaluation.** BudgetFunnel screens candidate functions using lightweight trace statistics and applies fine-grained simulation only to the screened set.
- **Separated approximation analysis.** Full-evaluation greedy isolates screening loss, while same-scope exact MILP experiments quantify the empirical greedy optimality gap.
- **Robust trace-driven evaluation.** Experiments on the Azure serverless trace and Alibaba microservice workload include full budget sweeps, provider-specific robustness windows, candidate-ratio and proxy ablations, alternative cold-counting semantics, TTL-grid and cold-penalty sensitivity, and runtime scaling.

2 Related Work

Serverless (FaaS) cold starts have been studied from diverse perspectives, ranging from system-level optimizations to application-level policies and evaluation methodologies. In addition to reducing initialization time, recent work increasingly treats the warm state as a managed resource: a policy must decide which functions to retain, for how long, and how to react when workload intensity changes. This distinction is important for our setting because the objective is not merely to make one function start faster, but to coordinate retention decisions across many functions under a shared budget.

2.1 System-level Cold Start Mitigation

A significant body of work focuses on optimizing the underlying infrastructure to reduce initialization overhead. Early efforts like SAND [16] proposed application-level sandboxing to mitigate cascading cold starts. To minimize virtualization latency, Firecracker [9] introduced lightweight microVMs, while SOCK [10] optimized container provisioning throughput. Recent advances utilize snapshotting techniques, such

Table 1 Comparison of representative cold-start mitigation strategies by their original decision scope and implementation requirements. The table summarizes published/original formulations and does not imply that a method cannot be extended to another scope.

Strategy	Key Mechanism	Original Decision Scope	System Integration	Primary Focus
Fixed Keep-Alive [7]	Static TTL	Workload-wide / coarse	Low	Simplicity
Histogram / IAT-Mean	History-based TTL selection	Per-function	Low	Inter-arrival history
FaaSCache [12]	Greedy-Dual caching	Cache / function	Medium	Cache hit rate
IceBreaker [13]	Heterogeneity-aware warming	Per-function	Medium–High	Workload heterogeneity
Owl / Defuse [14, 15]	Scheduling / dependency awareness	Workflow / DAG	High	Scheduling / QoS
BudgetFunnel (Ours)	**Coarse-to-fine TTL allocation**	**Shared warm-budget / workload-wide**	**Low–Medium**	**Cost–CSR trade-off**

as FaaSSnap [17], to skip initialization phases, and runtime systems like Nightcore [18] that optimize core scheduling for microsecond-scale latency. Additionally, JITServer [19] decouples JIT compilation to accelerate Java-based functions. While these mechanisms provide a fast execution foundation, they function orthogonally to the keep-alive policies studied in this paper.

2.2 Prewarming and Caching Policies

Complementary to system optimizations, policy-based approaches decide *when* and *how long* to keep instances warm. Transparent auto-scaling caches [20] and layer-wise container sharing [6, 21] have been proposed to amortize startup costs. A key line of research models keep-alive as a caching problem: FaasCache [12] adapts the Greedy-Dual-Size algorithm to serverless, drawing inspiration from web caching policies like TinyLFU [22]. Adaptive TTL strategies, such as IceBreaker [13], leverage workload heterogeneity to optimize warm times for different function types. Other heuristics include dependency-guided scheduling (Defuse [15]) and predictive prewarming [23]. However, complex learning-based schedulers like FaaSRank [24] often incur high training overheads. In contrast, BudgetFunnel focuses on workload-wide allocation under a shared warm budget using a lightweight trace-driven pipeline.

2.3 Workload Prediction and Adaptive Control

Prediction-based approaches use recent invocation history, temporal regularities, or learned representations to anticipate future demand and trigger prewarming before an invocation arrives. Such methods can reduce cold-start exposure when the workload is sufficiently predictable, but their benefit depends on forecast quality and on the cost of maintaining a model for each function or service. They also typically optimize

prediction accuracy or a local latency objective, whereas a shared-budget allocator must compare heterogeneous functions and TTL upgrades using a common resource currency. In our framework, sparsity, burstiness, and periodicity are deliberately used as lightweight screening features rather than as a full demand-forecasting model. This design keeps the screening stage reproducible on aggregated traces and leaves the final decision to the trace-driven cost–benefit evaluation. The distinction also clarifies the role of predictive pre-warming [23]: it is complementary to BudgetFunnel and could be incorporated as an additional proxy or as an online extension, but it is not required by the current offline formulation.

2.4 Resource-Constrained Scheduling

Resource management under strict budgets is critical, especially in edge environments [25, 26]. Performance-aware schedulers like Owl [14] and Aquatope [27] aim to meet QoS targets by managing resource contention. Recent works have explored cost-efficient computing on harvested resources [28], which aligns with our budgeted formulation. Surrogate-based optimization frameworks like Hydro [29] also address hyperparameter tuning in datacenters. Our work specifically targets workload-wide allocation of a shared prewarming budget using a coarse-to-fine strategy; this scope differs from general-purpose scheduling and resource-management objectives [30].

2.5 Trace-Driven Simulation and Benchmarking

Due to the difficulty of reproducing large-scale production environments, trace-driven simulation has become a standard methodology. SimFaaS [31] provides a discrete-event simulator for evaluating cold-start policies, while PanOpticon [32] offers comprehensive performance modeling. To ensure rigorous evaluation, recent community efforts have established standardized benchmarks such as ServerlessBench [33] and SeBS [34]. Furthermore, public Azure serverless traces [7] and Alibaba microservice traces [8] provide complementary workload signals for trace-driven robustness studies. We use the Alibaba trace only as a workload-shape stress test under the common TTL abstraction, not as direct evidence of production FaaS cold-start measurements. Our work builds on these methodologies to conduct large-scale sweep studies over budgets and policy parameters.

Positioning. BudgetFunnel does not claim that prior keep-alive or caching methods are fundamentally unable to coordinate globally. Its contribution is a specific trace-driven mechanism for reducing the cost of evaluating a shared warm-budget allocation problem through coarse-to-fine screening.

3 Methodology

3.1 Overview

We study trace-driven cold-start mitigation in serverless platforms under a *unified global warm budget*. Figure 2 summarizes the three-stage pipeline. Our framework, BudgetFunnel, follows a coarse-to-fine pipeline: **(Stage 0)** compute lightweight

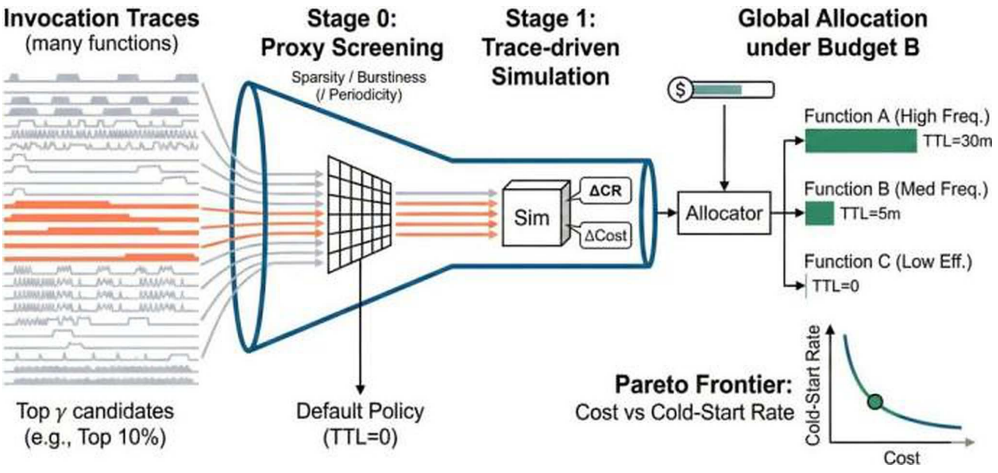


Fig. 2 Overview of BudgetFunnel. Invocation traces are processed by (Stage 0) proxy screening to select top-γ candidate functions, (Stage 1) trace-driven simulation to estimate per-function ΔCR and ΔCost over a TTL grid, and (Stage 2) a precedence-aware global allocator that assigns per-function TTLs under a unified warm budget B, yielding a Pareto frontier of cost versus cold-request rate.

per-function workload proxies and select a small candidate set; **(Stage 1)** run trace-driven simulation only for candidates over a discrete TTL grid to estimate marginal cold-start reduction and warm-cost increases; and **(Stage 2)** perform a global, budget-constrained allocation by selecting TTL upgrades across functions using a precedence-aware greedy procedure. This design reduces fine-grained evaluation cost while empirically preserving allocation quality close to a full-evaluation reference.

3.2 Trace Model and Notation

Let $\mathcal{F}$ be the set of functions. We discretize time into T slots of length Δ seconds (default $\Delta = 60$s). From invocation traces, we obtain the number of requests of each function in each slot: $a_f(t) \in \mathbb{Z}_{\geq 0}$ for $f \in \mathcal{F}$ and $t \in \{1, \ldots, T\}$.

TTL policy. We study keep-alive (TTL) policies. Each function f is assigned a TTL $\theta_f \in \Theta$ (minutes), meaning that after its last active slot, the platform keeps one warm execution environment available for θ_f minutes.

3.3 Policy Space and Practical Defaults

Time granularity. Unless otherwise stated, we use 1-minute slots ($\Delta = 60$s) and aggregate invocation counts accordingly.

TTL grid. We use a discrete TTL grid (minutes):

$$\Theta = \{0, 0.5, 1, 2, 5, 10, 20, 30, 60\}. \tag{1}$$

This grid is a practical discretization spanning short and long retention intervals. Sect. 4.7 and the Supplementary Information report TTL-grid sensitivity.

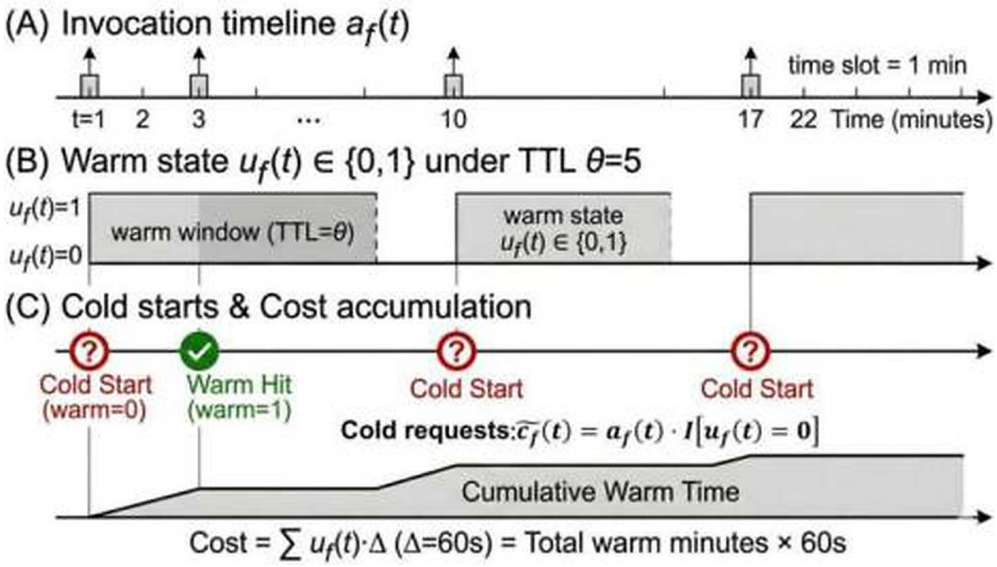


Fig. 3 Warm-state semantics under TTL and cost accumulation. (A) Aggregated invocation timeline $a_f(t)$ in 1-minute slots. (B) Warm indicator $u_f(t) \in \{0,1\}$ under TTL θ: after an active slot, the instance remains warm for a fixed retention window. (C) A cold start (cold requests) occurs when $a_f(t) > 0$ and $u_f(t) = 0$; warm cost accumulates as $\sum_t u_f(t)\Delta$, where $\Delta = 60$s

Default policy for non-candidates. For BudgetFunnel with $\gamma < 1$, functions outside $\mathcal{F}_{cand}$ are fixed at $\theta_f = 0$ and do not enter Stage 2. When $\gamma = 1$ (the full-evaluation reference), all functions are candidates. This definition makes the screening approximation explicit and attributes any quality gap between $\gamma < 1$ and $\gamma = 1$ to the excluded functions.

3.4 Warm-State Semantics and Cold-Start Counting

Public traces are typically aggregated (e.g., per-minute counts) and do not contain per-request start times and service times. Figure 3 illustrates the adopted warm-state abstraction. Therefore, we adopt a standard trace-driven warm-state abstraction with clear counting semantics.

TTL discretization. Given slot length Δ, we discretize a TTL value θ (minutes) into the number of *warm-retention slots*:

$$k(\theta) = \left\lceil \frac{60\theta}{\Delta} \right\rceil . \tag{2}$$

With $\Delta = 60$s, $k(0.5) = 1$, so 0.5-minute TTL retains warmth for 1 subsequent 1-minute slot. This discretization can introduce up to one-slot timing error. Because the available public workload traces are aggregated, we quantify robustness through alternative cold-counting semantics and TTL-grid sensitivity rather than treating the abstraction as a reconstruction of exact per-request timing.

Warm indicator. Let $u_f(t) \in \{0,1\}$ indicate whether function f is warm at the *beginning* of slot t. We simulate warm retention with a per-function counter $r_f(t) \in \mathbb{Z}_{\geq 0}$ representing remaining warm slots. Initially $r_f(1) = 0$. At the beginning of slot t, $u_f(t) = \mathbb{I}[r_f(t) > 0]$. At the end of slot t, the counter updates as:

$$r_f(t+1) = \begin{cases} k(\theta_f), & \text{if } a_f(t) > 0, \\ \max\{r_f(t) - 1, 0\}, & \text{if } a_f(t) = 0. \end{cases} \tag{3}$$

This model represents one logical warm state per function with TTL refresh on activity. It does not model concurrent instances, queueing, or platform autoscaling; these limitations bound the scope of the conclusions.

Cold-start indicators (slot-level and request-weighted). A cold start occurs when requests arrive while the function is cold at the beginning of that slot. We define:

$$c_f(t;\theta_f) = \mathbb{I}\big[a_f(t) > 0\big]\mathbb{I}\big[u_f(t) = 0\big], \tag{4}$$

$$\widetilde{c}_f(t;\theta_f) = a_f(t)\mathbb{I}\big[u_f(t) = 0\big]. \tag{5}$$

Here, c_f denotes a cold slot and $\widetilde{c}_f$ denotes cold requests. Eq. (4) counts whether a slot experiences a cold start event (at most one per slot), while Eq. (5) approximates how many requests are cold within the slot when only aggregated counts are available. **We use request-weighted counting (Eq. (5)) as the primary metric in this paper**, and report slot-level results as a robustness check. On Azure at $1 \times B_{ref}$, the method ranking is unchanged under request-weighted and slot-level counting.

Warm cost (warm instance-seconds). We measure warm cost as warm instance-seconds:

$$\mathrm{Cost}(\boldsymbol{\theta}) = \sum_{f \in \mathcal{F}} \sum_{t=1}^{T} u_f(t;\theta_f) \cdot \Delta. \tag{6}$$

This captures cumulative logical warm-retention time across functions under the single-warm-state abstraction; it is not a direct measurement of physical multi-instance memory consumption.

3.5 Problem Formulation

Primary objective: request-weighted CSR. We define request-weighted cold-start rate (CSR) as:

$$\mathrm{CSR}(\boldsymbol{\theta}) = \frac{\sum_{f \in \mathcal{F}} \sum_{t=1}^{T} \widetilde{c}_f(t;\theta_f)}{\sum_{f \in \mathcal{F}} \sum_{t=1}^{T} a_f(t)}. \tag{7}$$

Budgeted optimization. Given a global warm budget B (warm instance-seconds), we solve:

$$\min_{\boldsymbol{\theta} \in \Theta^{|\mathcal{F}|}} \ \mathrm{CSR}(\boldsymbol{\theta}) \quad \text{s.t.} \quad \mathrm{Cost}(\boldsymbol{\theta}) \leq B. \tag{8}$$

This formulation defines a shared workload-wide budget. It specifies the optimization objective, but does not imply that the proposed greedy solver attains the global optimum.

3.6 Stage 0: Proxy Screening (Coarse)

The key observation is that under a global budget, only a small subset of functions tends to dominate cold-start risk. We identify candidate functions using lightweight statistics computed from $\{a_f(t)\}_{t=1}^{T}$.

Sparsity.

$$s_f = \frac{1}{T}\sum_{t=1}^{T} \mathbb{I}[a_f(t) = 0]. \tag{9}$$

Burstiness. We use normalized variance:

$$b_f = \frac{\mathrm{Var}(a_f(t))}{\mathrm{Mean}(a_f(t)) + \epsilon}, \quad \epsilon = 10^{-6}. \tag{10}$$

If $\sum_t a_f(t) = 0$ (never invoked), we drop the function from screening since it does not affect the objective.

Periodicity. We compute a lightweight daily autocorrelation proxy. Let τ be the lag corresponding to 24 hours in slots ($\tau = \frac{24 \cdot 3600}{\Delta}$; $\tau = 1440$ for $\Delta = 60$s).

$$p_f = \frac{\sum_{t=1}^{T-\tau}(a_f(t) - \bar{a}_f)(a_f(t+\tau) - \bar{a}_f)}{\sum_{t=1}^{T}(a_f(t) - \bar{a}_f)^2 + \epsilon}. \tag{11}$$

If $T \le \tau$, we set $p_f = 0$ (insufficient horizon) to avoid unstable estimates.

Robust normalization and proxy score. We robustly center each feature by its median. To match the audited implementation and avoid numerical instability when many functions have identical feature values, the scale uses MAD when it is sufficiently nonzero; otherwise it falls back to the standard deviation, and finally to 1 for a constant feature:

$$\widehat{x}_f = \frac{x_f - \mathrm{median}(x)}{s_x}, \quad x \in \{b, p, 1-s\}, \tag{12}$$

where $s_x = \mathrm{MAD}(x)$ if $\mathrm{MAD}(x) \ge 10^{-4}$, $s_x = \mathrm{Std}(x)$ if MAD is smaller but $\mathrm{Std}(x) \ge 10^{-4}$, and $s_x = 1$ otherwise. This safeguard only prevents division by a numerically degenerate scale; it does not introduce a learned parameter. We then define the intentionally simple equal-weight proxy:

$$\mathrm{Proxy}(f) = \widehat{b}_f + \widehat{(1-s)}_f + \widehat{p}_f. \tag{13}$$

The equal weighting is not assumed to be theoretically optimal; the feature-ablation experiment evaluates the individual and combined contributions empirically.

Candidate set. We select the top-γ fraction functions by proxy score:

$$\begin{aligned} \mathcal{F}_{cand} &= \text{Top-}\gamma\big(\mathrm{Proxy}(f)\big), \\ \gamma &\in \{0.05, 0.10, 0.20\}. \end{aligned} \tag{14}$$

Algorithm 1 Per-function TTL simulation (trace-driven)

Require: Trace counts $\{a_f(t)\}_{t=1}^{T}$, slot length Δ, TTL θ
Ensure: Warm cost $\mathrm{Cost}_f(\theta)$, cold requests $\mathrm{CR}_f(\theta)$

```
k ← ⌈60θ/Δ⌉; r ← 0
Cost ← 0; CR ← 0
for t = 1 to T do
    u ← 𝕀[r > 0]
    Cost ← Cost + u · Δ
    if a_f(t) > 0 then
        if u = 0 then
            CR ← CR + a_f(t)
        end if
        r ← k {refresh TTL on activity}
    else
        r ← max{r − 1, 0}
    end if
end for
return Cost, CR
```

3.7 Stage 1: Fine-Grained Candidate Evaluation (Simulation)

For each candidate function $f \in \mathcal{F}_{cand}$ and TTL $\theta \in \Theta$, we run trace-driven simulation under the warm semantics in Sect. 3.4 to estimate: (i) warm cost $\mathrm{Cost}_f(\theta)$ and (ii) cold requests $\mathrm{CR}_f(\theta) = \sum_t \widetilde{c}_f(t;\theta)$. We compute marginal improvements over baseline $\theta_0 = 0$:

$$\begin{aligned} \Delta\mathrm{CR}(f,\theta) &= \mathrm{CR}_f(0) - \mathrm{CR}_f(\theta), \\ \Delta\mathrm{Cost}(f,\theta) &= \mathrm{Cost}_f(\theta) - \mathrm{Cost}_f(0). \end{aligned} \tag{15}$$

Caching and reuse. Stage-1 results $\{\mathrm{Cost}_f(\theta), \mathrm{CR}_f(\theta)\}$ are computed once per trace window and reused across budgets B in Stage 2.

3.8 Stage 2: Global Budget Allocation

We allocate budget globally by selecting TTL upgrades across functions based on marginal benefit per marginal cost. Let TTL grid be ordered as $\theta^{(0)} < \theta^{(1)} < \cdots < \theta^{(m)}$ with $\theta^{(0)} = 0$. For each function f, upgrading from $\theta^{(i)}$ to $\theta^{(i+1)}$ yields:

$$\begin{aligned} \Delta\mathrm{CR}_{f,i} &= \Delta\mathrm{CR}\left(f, \theta^{(i+1)}\right) - \Delta\mathrm{CR}\left(f, \theta^{(i)}\right), \\ \Delta\mathrm{Cost}_{f,i} &= \Delta\mathrm{Cost}\left(f, \theta^{(i+1)}\right) - \Delta\mathrm{Cost}\left(f, \theta^{(i)}\right). \end{aligned} \tag{16}$$

We define gain ratio:

$$g_{f,i} = \frac{\Delta\mathrm{CR}_{f,i}}{\Delta\mathrm{Cost}_{f,i} + \epsilon}, \quad \epsilon = 10^{-9}. \tag{17}$$

Algorithm 2 Precedence-aware global greedy allocation under budget B

Require: Candidates $\mathcal{F}_{cand}$, TTL grid $\{\theta^{(i)}\}_{i=0}^{m}$, per-function tables $\Delta\mathrm{CR}(f,\theta), \Delta\mathrm{Cost}(f,\theta)$, budget B
Ensure: TTL assignment $\boldsymbol{\theta}$

```
Initialize θ_f ← 0 for all f; remaining budget R ← B
For each f ∈ F_cand, set current level i_f ← 0
Initialize a max-heap H
for all f ∈ F_cand do
    Push next upgrade (f, i_f) with key g_{f,i_f} into H if i_f < m
end for
while H not empty do
    Pop (f, i) with largest g_{f,i} from H
    Compute ΔCost_{f,i} using Eq. (16)
    if ΔCost_{f,i} ≤ R then
        Accept upgrade: i_f ← i_f + 1; θ_f ← θ^{(i_f)}
        R ← R − ΔCost_{f,i}
        if i_f < m then
            Push the new next upgrade (f, i_f) with key g_{f,i_f} into H
        end if
    end if
end while
return θ
```

The ratio-based procedure is a discrete heuristic and is not guaranteed to be globally optimal. Sect. 4.7 compares it with a same-scope MILP reference on identical reduced workloads.

Precedence constraint (critical for feasibility). Upgrades for a function must respect grid order: $\theta^{(i+1)}$ can only be selected after $\theta^{(i)}$. To guarantee feasibility, we use a *precedence-aware incremental greedy* algorithm: at any time, each function exposes only its *next* upgrade item.

Non-candidates. For $f \notin \mathcal{F}_{cand}$, we fix $\theta_f = 0$; only candidate functions expose TTL upgrades to the Stage 2 heap. The $\gamma = 1$ reference removes this restriction by treating every function as a candidate.

3.9 Budget Sweeps and Pareto Frontiers

To report cost–performance trade-offs, we sweep budgets on a logarithmic grid:

$$B \in \{0.5, 1, 2, 5, 10, 20, 30, 50\} \times B_{ref}. \tag{18}$$

We compute B_{ref} by running a reference policy once (global TTL = 5 minutes) using the same simulator and trace window. For the audited Azure representative window, this gives $B_{ref} = 2.47 \times 10^7$ warm instance-seconds. The reference-budget sensitivity analysis additionally uses 1-min and 10-min reference TTLs, which yield 1.18×10^7

and 3.86×10^7 warm instance-seconds, respectively (Supplementary Information). For each B, we run Algorithm 2 to obtain $\boldsymbol{\theta}(B)$ and then evaluate CSR via Eq. (7).

3.10 Complexity and Practical Considerations

Runtime complexity. Proxy screening is a single pass over aggregated traces: $O(|\mathcal{F}| \cdot T)$. Fine-grained evaluation is applied only to candidates: $O(|\mathcal{F}_{cand}| \cdot |\Theta| \cdot T)$ (Algorithm 1). Global allocation uses a heap over at most $|\mathcal{F}_{cand}| \cdot (|\Theta| - 1)$ incremental upgrades: $O(M \log M)$ with $M \approx |\mathcal{F}_{cand}| \cdot (|\Theta| - 1)$.

Speed–quality trade-off. By selecting $\gamma \ll 1$, we reduce the number of fine-grained function–TTL evaluations. At $\gamma = 0.10$, the audited implementation achieves about $9\times$ end-to-end speedup, with a 1.54% CSR gap to full evaluation on Azure and 1.93% on Alibaba at $1 \times B_{ref}$.

Optional metric (slot-level CSR). For completeness, we also report slot-level CSR: $\mathrm{CSR}_{slot} = \frac{\sum_{f,t} c_f(t;\theta_f)}{\sum_{f,t} \mathbb{I}[a_f(t)>0]}$, which counts cold-start events per active slot. The Azure $1 \times B_{ref}$ ranking is identical under the two definitions (Full-eval, BudgetFunnel, IAT-Mean, Histogram, Global TTL).

4 Experiments

We evaluate BudgetFunnel using trace-driven simulation on two public workload sources: the Azure serverless invocation trace and Alibaba Cluster Trace–Microservices v2021. Table 2 defines the principal notation and audited simulation settings used below. Tables 4–6 report the main quantitative results in consecutive order. The Alibaba dataset is used as a cross-workload request-pattern robustness trace; each microservice is mapped to the common allocation-entity abstraction, and simulated CSR/CSO values should not be interpreted as measured Alibaba FaaS cold-start statistics. Our goal is to characterize cost–cold-start trade-offs under a shared warm budget, quantify screening and greedy approximation separately, and test whether conclusions are robust to trace and modeling assumptions.

Research Questions. RQ1 (Effectiveness): Under matched warm budgets, how does BudgetFunnel compare with representative TTL baselines on the Azure serverless trace and Alibaba microservice workload? **RQ2 (Screening approximation):** How much quality is lost by evaluating only a fraction of allocation entities, and what runtime reduction is obtained? **RQ3 (Allocation approximation):** How close is the precedence-aware greedy allocator to the exact MILP optimum on identical reduced Azure workloads? **RQ4 (Robustness):** Are the conclusions stable across cold-counting semantics, proxy features, TTL grids, provider-specific trace windows, cold-penalty models, and reference-budget definitions? **RQ5 (Scalability):** How does runtime scale from top-1k to top-5k entities?

4.1 Datasets

Window protocol. The main Azure experiment uses a 3-day window with 60-s slots. Because the public Azure invocation archive contains 14 consecutive daily

Table 2 Summary of key notations and simulation configurations. Key notation and audited simulation settings used in the evaluation

Symbol	Description / Definition	Value / Range
System Model & Workload		
$\mathcal{F}$	The set of all serverless functions in the workload, ranked by invocation frequency.	$\lvert\mathcal{F}\rvert = 1,000$ (Top-1k)
$a_f(t)$	The number of invocation requests for function f arriving at time slot t.	From Traces
Δ	The granularity of time slots for simulation and decision making.	60 seconds
B	The global budget constraint for logical warm-retention time (instance-seconds).	$[0.5 \dots 50] \times B_{ref}$
Algorithm Parameters (BudgetFunnel)		
γ	**Candidate Screening Ratio.** The fraction of top-ranked functions selected for fine-grained simulation (Stage 1).	**0.10 (Default)**
Θ	The discrete set of candidate TTL values (keep-alive duration) available for allocation.	$\{0, 0.5, 1, 2, 5, 10, 20, 30, 60\}$ min
$g_{f,i}$	The marginal gain ratio (ROI) of upgrading function f to the next TTL level.	Computed dynamically
Evaluation Metrics		
L_c/L_w	Latency penalty model: Cold start latency vs. Warm execution latency.	10s / 50ms
O_f	Per-request cold-start overhead of function f (used by CSO).	Log-normal, $\sigma = 1$ (main CSO)
CSR	**Cold-Start Rate.** Request-weighted percentage of invocations that incur a cold start.	Objective (Minimize)
CSO	**Cold-Start Overhead.** Total cold-start overhead weighted by per-function cold penalties.	Report (Lower is better)
WMT	**Wasted Warm Time.** Warm-retention time without serving any request (resource wastage).	Report (Lower is better)
Cost	Total accumulated logical warm-retention time across all functions.	Constraint ($\leq B$)

files, the non-overlapping robustness protocol uses four 3-day windows: W1=Days 1–3, W2=Days 4–6, W3=Days 7–9, and W4=Days 10–12. Days 13–14 are reserved as a 2-day temporal hold-out and are not included in the four-window aggregate. Alibaba Cluster Trace–Microservices v2021 provides a 12-h MSRTQps horizon at 60-s granularity; the main Alibaba experiment uses the full 12 h. Its robustness analysis uses five non-overlapping 2-h subwindows: W1=0–2 h, W2=2–4 h, W3=4–6 h, W4=6–8 h, and W5=8–10 h. The remaining 10–12 h segment is reserved as a hold-out/additional window. Thus, robustness windows are provider-specific and reflect the native trace horizons rather than being forced to a common duration.

Azure trace. We use public Azure invocation traces from the AzurePublicDataset repository [35]. Following common practice in trace-driven serverless studies, we aggregate per-function invocation counts into 1-minute slots. Unless otherwise stated, we evaluate the top-1k functions ranked by total invocations in the representative 3-day window.

Alibaba microservice workload. We use *Alibaba Cluster Trace–Microservices v2021* from the official `alibaba/clusterdata` repository [36], version `cluster-trace-microservices-v2021`, specifically the MSRTQps module. The audited input set comprises `MSRTQps_0.csv` through `MSRTQps_24.csv`. The records span 12 h at 60-s granularity. The MCR fields are call rates (calls/s). During source-field auditing, we found that `HTTP_MCR` in `MSRTQps_0.csv` is numerically identical to `HTTP_RT` for 99.76% of valid records, whereas the remaining shards show only about 0.13% median exact equality. To avoid contaminating the workload reconstruction with this shard-specific source anomaly, the primary Alibaba signal uses provider-side RPC calls only. For microservice f and minute slot t, we sum `providerRPC_MCR` over all replicas $i \in I_f(t)$ and reconstruct the per-minute request-count proxy as

$$a_f(t) = \text{round}\left(60 \sum_{i \in I_f(t)} \text{providerRPC_MCR}_{f,i,t}\right). \tag{19}$$

We do not add consumer-side RPC rates because they represent calls issued by the service to downstream services. We also exclude `HTTP_MCR` from the primary reconstruction because of the audited shard anomaly described above. Invalid/NaN timestamps, non-positive reconstructed request counts, and never-active services are removed. We rank services by $\sum_t a_f(t)$ over the full 12-h horizon, select the top 1000, and break ties by ascending `msname`. Each selected microservice is treated as a function-like allocation entity under the common TTL-retention abstraction. Because this is a microservice workload trace rather than a native FaaS invocation trace, Alibaba results are interpreted as cross-workload robustness evidence only. Because the Alibaba horizon (12 h, and 2 h in the robustness subwindows) is shorter than the 24-h autocorrelation lag, the periodicity feature in Eq. (11) is set to zero by the stated insufficient-horizon rule; Alibaba screening therefore relies on the burstiness and non-sparsity components of the same proxy definition. The audited providerRPC-only reconstruction yields approximately 4.96×10^8 request-count proxy events over the selected top-1000 workload, with 30.9% mean sparsity and a 69.1% active-slot ratio.

Trace statistics. Table 3 reports the audited representative-window statistics supplied by the final preprocessing logs. The Alibaba total is the aggregated request-count proxy described above rather than a count of native FaaS invocations.

4.2 Simulator and Policy Space

Simulator. We implement policies on a trace-driven simulator (SimFaaS) [31], which models warm retention and cold-start events under TTL-based keep-alive. Warm-state semantics, TTL discretization, and cold-request counting follow Sect. 3.4. This

Table 3 Audited workload statistics and evaluation scope. Values marked ≈ are rounded preprocessing summaries.

Trace	Entity count	Main horizon	Total request signal	Avg. sparsity	Active-slot ratio
Azure serverless	1000	3 days	$\approx 8.55 \times 10^8$ invocations	≈ 18.4%	≈ 81.6%
Alibaba Microservices v2021*	1000	12 h	$\approx 4.96 \times 10^8$ request-count proxy	≈ 30.9%	≈ 69.1%

*Alibaba reconstructs a per-minute request-count proxy from `providerRPC_MCR` only, summed across replicas and multiplied by 60. `HTTP_MCR` is excluded after a shard-level field audit identified an anomaly in `MSRTQps_0.csv`; the workload is not a native FaaS cold-start trace.

simulator is a trace-driven TTL-retention abstraction rather than a full model of production concurrency and autoscaling.

Time granularity and TTL grid. We use 1-minute slots and the discrete TTL grid defined in Eq. (1).

Warm-budget. We measure cost in warm instance-seconds (Eq. (6)). To obtain Pareto curves, we sweep budget B using the grid in Eq. (18), where B_{ref} is computed by running the global TTL = 5 min reference policy on the same trace window using the same simulator; for the audited Azure representative window, $B_{ref} = 2.47 \times 10^7$ warm instance-seconds. Sensitivity to 1-min and 10-min reference policies is reported in the Supplementary Information. Unless otherwise stated, the candidate screening ratio is $\gamma = 0.10$; sensitivity to $\gamma \in \{0.05, 0.10, 0.20\}$ is reported in Sect. 4.7 and the Supplementary Information.

4.3 Metrics

Primary metric: request-weighted cold-start rate (CSR). We report the global request-weighted cold-start rate:

$$\mathrm{CSR}(\boldsymbol{\theta}) = \frac{\sum_{f \in \mathcal{F}} \sum_{t=1}^{T} \widetilde{c}_f(t; \theta_f)}{\sum_{f \in \mathcal{F}} \sum_{t=1}^{T} a_f(t)}, \tag{20}$$

where $\widetilde{c}_f(t; \theta_f) = a_f(t) \cdot \mathbb{I}[u_f(t) = 0]$ counts cold requests under aggregated traces (Sect. 3.4). For completeness, we also report slot-level CSR in the Supplementary Information.

Cost. We report warm instance-seconds: $\mathrm{Cost}(\boldsymbol{\theta})$ in Eq. (6).

Cold-start overhead (CSO) with heterogeneous cold penalties. Prior work shows that cold-start overhead is highly heterogeneous across functions, and lowering the cold-start rate alone may not translate to a proportional reduction in end-to-end

latency. Therefore, we additionally report the *total cold-start overhead*:

$$\mathrm{CSO}(\boldsymbol{\theta}) = \sum_{f \in \mathcal{F}} \sum_{t=1}^{T} O_f \cdot \widetilde{c}_f(t; \theta_f), \tag{21}$$

where O_f denotes the per-request cold-start overhead of function f (seconds/request), and $\widetilde{c}_f(t; \theta_f)$ is the number of cold requests (Eq. (20)). Because the public traces do not provide measured per-function cold penalties, the main heterogeneous-CSO results sample O_f from a log-normal model $O_f \sim \mathrm{LogNormal}(\mu, \sigma^2)$ with $\sigma = 1.0$ and $\mu = \ln(10) - \frac{1}{2}\sigma^2$, so the untruncated distribution has mean 10 s. We clip O_f to $[1, 60]$ seconds for numerical stability. Separately, the fixed-penalty sensitivity experiment sets $O_f = L_c$ for all functions with $L_c \in \{1, 5, 10\}$ s. Clipping changes the realized empirical mean of the heterogeneous samples, especially for larger σ; therefore, the σ sweep is interpreted as a tail-shape-plus-clipping sensitivity analysis rather than as a strictly fixed-post-clipping-mean experiment. For reproducibility, for each trace window we sample $\{O_f\}$ *once* with a fixed seed (42), and reuse the same sampled $\{O_f\}$ across all methods and *all budgets*. We also report the normalized overhead per request, $\mathrm{CSO/Req} = \mathrm{CSO} / \sum_{f \in \mathcal{F}} \sum_{t=1}^{T} a_f(t)$ (seconds/request), to make results comparable across trace windows.

Wasted warm time (WMT) as resource wastage. Keeping containers warm without serving requests consumes retention capacity without serving traffic. We quantify this as wasted warm time (WMT), defined as the total warm-retention time that does not serve any invocation:

$$\begin{aligned} \mathrm{WMT}(\boldsymbol{\theta}) = \sum_{f \in \mathcal{F}} \sum_{t=1}^{T} \Delta\, \mathbb{I}[u_f(t) = 1] \\ \cdot\, \mathbb{I}[a_f(t) = 0]. \end{aligned} \tag{22}$$

measured in instance-seconds. We report WMT/Cost and ActualCost/B for all major methods at $1\times$, $5\times$, and $10 \times B_{ref}$. This matched-budget comparison checks whether CSR improvements are obtained by disproportionately larger idle-warm retention. WMT/Cost is the fraction of consumed warm-retention cost spent in slots with no invocation, while ActualCost/B indicates how fully a method uses the available budget.

Tail latency (P99) via a cold-start penalty model. To quantify user-perceived tail latency without requiring per-request service times, we adopt a simple penalty model. Each request incurs warm latency L_w if it hits a warm instance, and incurs cold-start latency L_c otherwise:

$$\ell = \begin{cases} L_c, & \text{if cold start occurs,} \\ L_w, & \text{otherwise.} \end{cases} \tag{23}$$

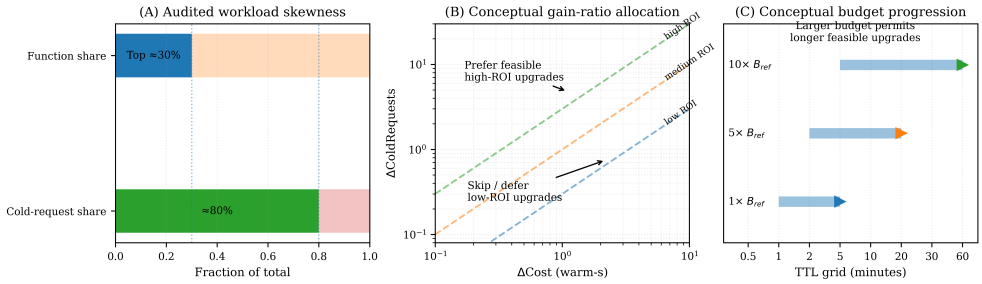


Fig. 4 Workload skewness and allocation mechanism. (A) Audited Azure workload anchor: approximately 80% of cold requests are contributed by the top approximately 30% of functions, confirming a pronounced long-tail structure. (B) Conceptual ROI view of the Stage-2 allocator: upgrades with larger cold-request reduction per unit warm cost are prioritized subject to precedence and budget constraints. (C) Conceptual effect of increasing the warm budget: the feasible allocation can progressively include longer TTL upgrades. Panels (B) and (C) are mechanism illustrations rather than empirical scatter/distribution measurements

We report P99 over per-request latency samples. Because traces are aggregated into counts $a_f(t)$, we compute P99 using a *weighted quantile*: each slot contributes $a_f(t)$ requests with latency L_c when $u_f(t) = 0$, and $a_f(t)$ requests with latency L_w otherwise. Unless otherwise stated, we use $L_w = 50$ms and $L_c = 10$s, and report sensitivity with $L_c \in \{1, 5, 10\}$s. Because CSR is below 1% in several Azure settings, P99 often collapses to the modeled warm latency; we therefore treat P99 only as an auxiliary model-based indicator and emphasize CSR and CSO/Req.

Trade-off reporting. We plot Pareto frontiers of **Cost vs. CSR** by sweeping budgets. When available, we additionally report the P99 tail-latency proxy under the cold-start penalty model.

Efficiency. To quantify scalability, we report runtime of (i) proxy screening, (ii) candidate evaluation, and (iii) global allocation, as well as the speedup over evaluating all functions.

ROI definition (for allocation interpretation). To interpret allocation decisions, we quantify the marginal ROI of upgrading function f from TTL level θ_i to θ_{i+1} as $g_{f,i} = \Delta\text{ColdReq}_{f,i}/\Delta\text{Cost}_{f,i}$, where $\Delta\text{ColdReq}_{f,i}$ is the reduction in cold requests under the simulator and $\Delta\text{Cost}_{f,i}$ is the additional warm instance-seconds incurred by the upgrade.

Screening quality (for γ ablation). Let $\mathcal{H}$ denote the set of top-10% functions with the highest ROI (computed from full evaluation, $\gamma = 1$). For a screened candidate set $\mathcal{F}_{cand}$, we report Recall@Top-10% $= |\mathcal{F}_{cand} \cap \mathcal{H}|/|\mathcal{H}|$ and Precision $= |\mathcal{F}_{cand} \cap \mathcal{H}|/|\mathcal{F}_{cand}|$.

4.4 Workload Skewness and Allocation Mechanism

The audited Azure trace exhibits substantial workload skew: approximately 80% of cold requests are concentrated in the top approximately 30% of functions. This observation motivates non-uniform warm-budget allocation, because a relatively small

subset of functions accounts for most cold-start exposure. Fig. 4A reports this audited long-tail anchor. Panels B–C illustrate the allocation logic only: BudgetFunnel prioritizes feasible TTL upgrades with high marginal cold-request reduction per unit warm cost, while larger budgets permit progressively longer retention choices. We deliberately present these panels as mechanism illustrations rather than as additional empirical measurements.

4.5 Baselines

We compare BudgetFunnel with policies evaluated under the same trace window, warm-state semantics, TTL grid, and budget. Baseline implementations are kept within the TTL/keep-alive policy family so that they are reproducible from aggregated traces.

B1: No keep-alive (TTL=0). No function is retained warm ($\theta_f = 0$ for all f).

B2: Global fixed TTL. All functions share one TTL $\theta^{global} \in \Theta$. We select the best feasible global TTL under each budget without per-function oracle information.

B3: IAT-Mean TTL. For each function, let $t_{f,1} < \cdots < t_{f,n_f}$ denote the active-slot indices. When $n_f > 1$, the baseline computes the arithmetic mean active-slot inter-arrival time, $\bar{\Delta}t_f = \frac{\Delta}{60(n_f-1)} \sum_{j=2}^{n_f}(t_{f,j} - t_{f,j-1})$ minutes; for $n_f \leq 1$, the target is zero. A single workload-wide scale factor λ is selected by 20 iterations of binary search so that the resulting policy is feasible under budget B. Each scaled target $\lambda\bar{\Delta}t_f$ is snapped downward to the largest TTL in Θ not exceeding that target. This baseline is therefore a mean-IAT history heuristic rather than an exponentially weighted moving average.

B4: Histogram TTL. For each function, we compute active-slot inter-arrival times and use the empirical 95th percentile as the unbudgeted target TTL. We apply the same workload-wide scaling and downward grid-snapping procedure used for IAT-Mean TTL to satisfy B. Thus, neither history-based baseline uses BudgetFunnel's simulated $\Delta CR/\Delta Cost$ values for budget projection.

B5: Full-evaluation greedy ($\gamma = 1$). We evaluate all functions over the TTL grid and then run exactly the same precedence-aware greedy allocator as BudgetFunnel. This is a *full-evaluation reference* that isolates screening loss; it is not an optimization upper bound.

Exact MILP validation on reduced workloads. To quantify the allocation approximation separately, we compare greedy and an exact same-scope MILP on the audited Azure representative workload. For each $N \in \{50, 100, 200\}$, we select the top-N functions by total invocations from the same preprocessed top-1000 set; ties are broken by ascending function ID. The reference budget is recomputed inside each reduced scope,

$$B_{ref,N} = Cost_{\mathcal{F}_N}(\text{Global TTL} = 5 \text{ min}), \tag{24}$$

and the comparison uses $B = 1 \times B_{ref,N}$. For a fixed reduced set $\mathcal{F}_N$, binary variable $x_{f,\theta}$ selects exactly one TTL per function:

$$\begin{aligned} \min_{x} \quad & \sum_{f \in \mathcal{F}_N} \sum_{\theta \in \Theta} x_{f,\theta}\, CR_f(\theta) \\ \text{s.t.} \quad & \sum_{f \in \mathcal{F}_N} \sum_{\theta \in \Theta} x_{f,\theta}\, Cost_f(\theta) \le B, \\ & \sum_{\theta \in \Theta} x_{f,\theta} = 1, \quad \forall f \in \mathcal{F}_N, \\ & x_{f,\theta} \in \{0,1\}. \end{aligned} \tag{25}$$

We solve the 0–1 formulation with Google OR-Tools CP-SAT 9.10.4067 using a 60-s time limit, one search worker, relative gap limit 0, and random seed 42. All three audited runs return `OPTIMAL`; the reported MILP objective values are therefore exact optima for these evaluated reduced formulations. Greedy uses the identical reduced function set, TTL grid, simulated cost/benefit tables, and $B_{ref,N}$.

4.6 Main Results

Budget–CSR trade-offs. Fig. 5 summarizes the audited trade-offs. On Azure, BudgetFunnel obtains CSR values of 0.00396, 0.00120, and 0.00085 at 1×, 5×, and $10 \times B_{ref}$, respectively, compared with 0.00390, 0.00118, and 0.00084 for full-evaluation greedy. Across the full Azure budget sweep, the maximum relative gap to full evaluation is 2.48%. On Alibaba, the corresponding BudgetFunnel values are 0.00900, 0.00310, and 0.00195, with a maximum full-sweep gap of 2.16%. Relative to Global TTL, the largest Alibaba CSR reduction is 39.4%; relative to IAT-Mean TTL, the largest reduction is 21.8%. These statements refer to the audited budget points and should not be interpreted as uniform dominance over every conceivable policy.

Multi-window robustness. Across the four valid non-overlapping 3-day Azure windows (Days 1–12), BudgetFunnel obtains 0.00396 ± 0.00017 CSR versus 0.00390 ± 0.00017 for full-evaluation greedy at $1 \times B_{ref}$. The corresponding Global TTL, IAT-Mean TTL, and Histogram TTL averages are 0.00631 ± 0.00036, 0.00471 ± 0.00024, and 0.00513 ± 0.00026, respectively. Each window independently re-selects the top-1000 functions and recomputes its own B_{ref}; per-window values are reported in the Supplementary Information.

Fixed-budget summary. Table 4 reports audited Azure values. P99 equals the modeled 50-ms warm latency at these points because the request-weighted cold fraction is below the 1% quantile threshold; consequently, P99 is auxiliary and CSR/CSO are more discriminative.

Resource-utilization comparison. BudgetFunnel remains close to full-evaluation greedy in idle-warm efficiency: the WMT/Cost difference is 0.9, 0.8, and 0.8 percentage points at 1×, 5×, and $10 \times B_{ref}$, respectively, while both methods use similar fractions of the available budget. BudgetFunnel also has lower WMT/Cost than Global TTL,

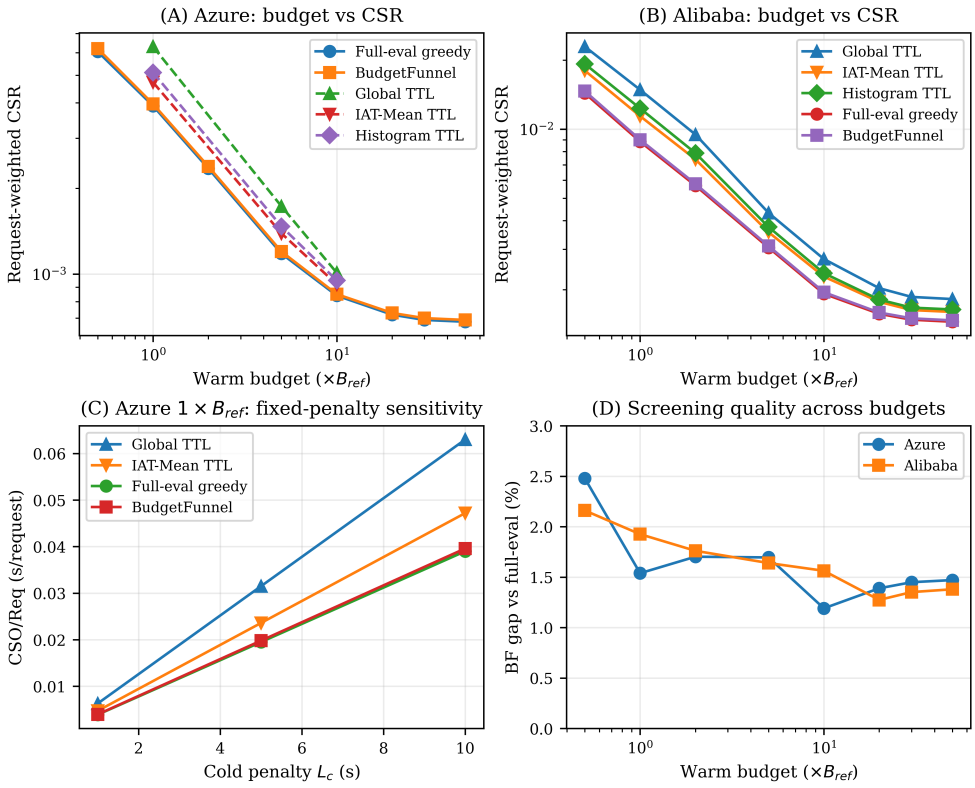


Fig. 5 Audited end-to-end results. (A) Azure CSR versus normalized warm budget; full-evaluation greedy and BudgetFunnel are shown over the complete budget sweep, while representative baselines are shown at the audited fixed-budget points. (B) Alibaba CSR versus budget for all evaluated methods. (C) Model-based CSO/Req sensitivity to fixed cold penalties at Azure $1\times B_{ref}$. (D) Relative CSR gap between BudgetFunnel and full-evaluation greedy over the complete Azure and Alibaba budget sweeps.

IAT-Mean TTL, and Histogram TTL at all three audited budgets. Thus, its CSR quality is not obtained by disproportionately larger idle warming.

Budget saturation. ActualCost/B falls from 99.1% at $1 \times B_{ref}$ to 78.6% at $10 \times B_{ref}$, indicating that the finite TTL grid and available useful upgrades make the budget progressively non-binding. High-budget convergence is therefore interpreted as saturation rather than additional evidence of method equivalence.

4.7 Ablation Studies

A1: Candidate ratio γ. Fig. 6 shows the Azure speed–quality trade-off at $1 \times B_{ref}$. At $\gamma = 0.05$, screening yields a 14.8× speedup but a 9.74% relative CSR gap to full evaluation. Increasing to $\gamma = 0.10$ raises Recall@Top-10% to 0.86, reduces the CSR gap to 1.54%, and still gives a 9.0× speedup. At $\gamma = 0.20$, recall reaches 0.96 and

Table 4 Audited fixed-budget performance results on Azure traces (lower is better). Matched resource-utilization metrics are reported separately in Table 5.

Method	**Metric**	$1 \times B_{ref}$	$5 \times B_{ref}$	$10 \times B_{ref}$
TTL=0	CSR	1.0000	1.0000	1.0000
Global TTL	CSR	0.00630	0.00173	0.00101
IAT-Mean TTL	CSR	0.00472	0.00139	0.00092
Histogram TTL	CSR	0.00511	0.00147	0.00095
Full-evaluation greedy	CSR	**0.00390**	**0.00118**	**0.00084**
BudgetFunnel ($\gamma = 0.10$)	CSR	0.00396	0.00120	0.00085
BudgetFunnel	CSO/Req (s/request)	0.0378	0.0117	0.0084
BudgetFunnel	model-based P99 ($L_c = 10$s)	0.05 s	0.05 s	0.05 s

Table 5 Matched resource-utilization diagnostics on Azure. WMT/Cost is lower when less consumed warm-retention time is idle; ActualCost/B shows budget utilization. Values are percentages.

Method	**1× WMT/ Cost**	**5× WMT/ Cost**	**10× WMT/ Cost**	**1× Actual Cost/B**	**5× Actual Cost/B**	**10× Actual Cost/B**
Global TTL	69.8%	81.5%	87.9%	100.0%	96.8%	82.6%
IAT-Mean TTL	66.4%	78.3%	84.9%	99.0%	95.6%	80.8%
Histogram TTL	68.7%	80.6%	86.5%	98.7%	95.1%	81.4%
Full-evaluation greedy	**61.9%**	**73.8%**	**80.1%**	99.5%	95.7%	79.8%
BudgetFunnel	62.8%	74.6%	80.9%	99.1%	94.8%	78.6%

the CSR gap falls to 0.51%, but speedup drops to 4.7×. Alibaba shows the same qualitative trade-off: at $\gamma = 0.10$, recall is 0.81, the CSR gap is 1.93%, and speedup is 9.0× (Supplementary Information).

A2: Proxy feature ablation. No single proxy feature reproduces the full proxy quality. Burstiness-only, non-sparsity-only, and periodicity-only produce CSR gaps of 8.21%, 6.15%, and 18.21%, respectively. Combining burstiness and non-sparsity reduces the gap to 2.82%, while using all three features reduces it further to 1.54%. Thus, periodicity is weak alone but provides additional information in combination with the other features.

A3: Alternative cold-counting semantics. At Azure $1 \times B_{ref}$, request-weighted CSR ranks Full-eval, BudgetFunnel, IAT-Mean, Histogram, and Global TTL from best to worst (0.00390, 0.00396, 0.00472, 0.00511, 0.00630). Slot-level CSR changes

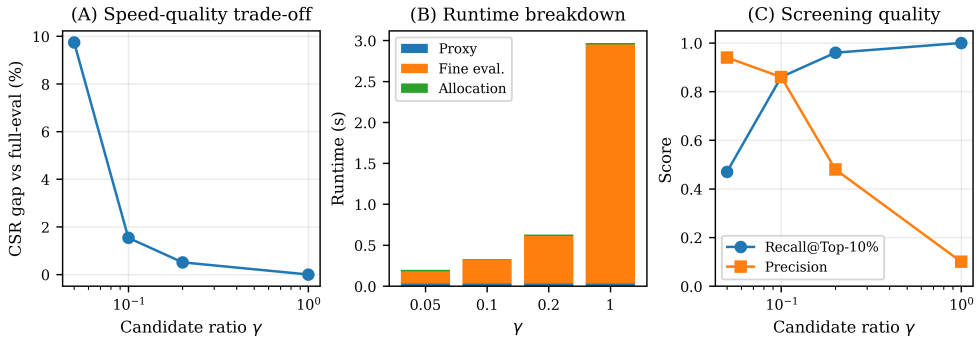


Fig. 6 Candidate-ratio ablation on Azure at $1 \times B_{ref}$. (A) Relative CSR gap to full evaluation versus candidate ratio. (B) Proxy, fine-evaluation, and allocation runtime. (C) Recall@Top-10% and precision of the screened set.

the absolute values (0.0091, 0.0093, 0.0108, 0.0119, 0.0138) but preserves the identical ranking. This supports the comparative conclusion while acknowledging that aggregated traces cannot reconstruct the exact number of per-request cold starts.

A4: TTL-grid sensitivity. The default grid yields CSR 0.00396 with 0.290 s fine-evaluation time. Removing the 0.5-min point leaves CSR unchanged (0.00396) and reduces fine-evaluation time to 0.259 s. A coarse grid increases CSR to 0.00410, while a denser grid improves it marginally to 0.00391 at 0.354 s. The principal quality/runtime conclusion is therefore not tied to one precise grid.

A5: Greedy versus the exact MILP optimum. On top-N reduced Azure workloads, with $B_{ref,N}$ recomputed separately for each scope, exact MILP versus greedy CSR is 0.00281 versus 0.00285 for $N = 50$, 0.00312 versus 0.00318 for $N = 100$, and 0.00346 versus 0.00355 for $N = 200$. CP-SAT returns `OPTIMAL` for every audited instance. The corresponding greedy optimality gaps are 1.42%, 1.92%, and 2.60%. Exact MILP runtimes grow from 0.09 to 1.87 s, whereas greedy takes 0.001–0.007 s. Thus, the allocator is empirically close to the exact optimum on these reduced scopes while remaining substantially faster.

4.8 Policy Attribution and Micro-dynamics

Fig. 7 explains why BudgetFunnel performs well under a unified budget: it concentrates warm budget on bursty high-impact functions while reclaiming budget from sparse-tail functions where warming yields little benefit.

4.9 Scalability and Runtime Analysis

Table 6 reports mean±std over 10 repeated runs. At top-1k, $\gamma = 0.10$ reduces total runtime from 2.967 ± 0.063 s to 0.329 ± 0.010 s (9.0×). At top-5k, runtime falls from 14.567 ± 0.185 s to 1.611 ± 0.030 s (9.0×). Fine-grained candidate evaluation dominates the runtime, consistent with the intended coarse-to-fine design.

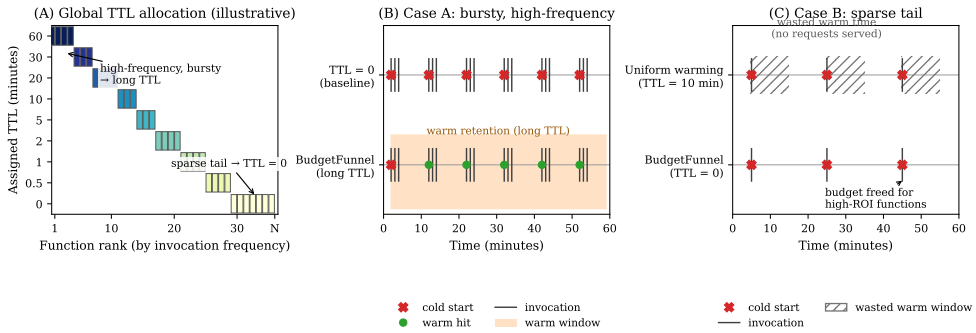


Fig. 7 Policy attribution and micro-dynamics (illustrative under the trace-driven abstraction). (A) Global allocation map: longer TTLs are assigned to high-frequency and bursty functions, while sparse-tail functions receive shorter TTLs or TTL= 0. (B) Case A (bursty, high-frequency): BudgetFunnel uses a longer TTL to convert cold to warm hits under controlled cost. (C) Case B (sparse tail): BudgetFunnel avoids wasting budget (TTL= 0), freeing budget for high-ROI functions

Table 6 Runtime breakdown and scalability in seconds, reported as mean±std over 10 runs.

Scale	γ	Proxy	Fine	Alloc.	Total	Speedup
top-1k	0.10	0.031 ± 0.002	0.294 ± 0.009	0.004 ± 0.001	$\mathbf{0.329 \pm 0.010}$	**9.0**×
top-1k	1.00	0.031 ± 0.002	2.921 ± 0.061	0.015 ± 0.002	2.967 ± 0.063	1.0×
top-5k	0.10	0.143 ± 0.005	1.455 ± 0.027	0.013 ± 0.002	$\mathbf{1.611 \pm 0.030}$	**9.0**×
top-5k	1.00	0.143 ± 0.005	14.349 ± 0.181	0.075 ± 0.004	14.567 ± 0.185	1.0×

All runtime experiments were conducted on an Intel Xeon E5-2620 v4 @ 2.10 GHz system with 8 physical cores (16 logical threads) and 32 GB RAM, running Ubuntu 20.04.6 LTS x86_64. The software environment used Python 3.9.18, NumPy 1.26.4, Pandas 2.1.4, and Google OR-Tools 9.10.4067. Each runtime entry reports mean±std over 10 runs. Main experiments used one execution thread; CP-SAT used `num_search_workers`=1.

5 Discussion and Limitations

The evaluation uses aggregated workload traces and therefore should be interpreted as a trace-driven TTL-retention abstraction, not a reconstruction of complete production serverless dynamics. Azure provides serverless invocation counts, whereas Alibaba Microservices v2021 supplies a microservice request-count proxy used only for cross-workload robustness. First, Eq. (5) counts all requests in a cold slot as cold-exposed; the slot-level sensitivity analysis shows that this changes absolute rates but not the evaluated method ranking. Second, the warm-state model tracks one logical warm state per function and omits concurrent instances, queueing, and autoscaling interactions. Third, latency and CSO are derived from explicit penalty models because the public traces do not provide per-request service-time measurements; P99 is consequently

reported only as an auxiliary model-based indicator. Fourth, the main formulation uses one warm-retention budget rather than simultaneous CPU, memory, network, or energy constraints. Finally, BudgetFunnel is an offline trace-driven allocator; adapting it to non-stationary workloads would require periodic retraining or online updates.

The robustness experiments reduce, but do not eliminate, these external-validity limitations. Validation on an open-source FaaS platform such as OpenWhisk or Knative would be a useful next step.

6 Conclusion

We presented BudgetFunnel, a coarse-to-fine trace-driven framework for allocating a shared warm-retention budget across serverless functions. The framework reduces the number of expensive per-function TTL evaluations through proxy screening and then uses a precedence-aware greedy allocator over the simulated marginal benefits and costs. The revised evaluation separates screening loss from allocation loss: at $\gamma = 0.10$, BudgetFunnel provides about $9\times$ end-to-end speedup while remaining within 1.54% of full-evaluation greedy on Azure and 1.93% on Alibaba at $1 \times B_{ref}$; over the complete budget sweep, the maximum gap remains below 2.5%. Same-scope exact MILP experiments place the greedy allocator within 1.42–2.60% of the certified optimum for reduced Azure scopes of 50–200 functions. Provider-specific multi-window, cold-counting, proxy-feature, TTL-grid, cold-penalty, reference-budget, and matched resource-utilization analyses show that the principal comparative conclusions are stable under the tested settings.

These results support BudgetFunnel as a scalable approximation for trace-driven shared-budget allocation, rather than as a globally optimal production autoscaling solution. Future work should validate the framework on an executable serverless platform and extend the cost model to multi-instance and multi-resource constraints.

Supplementary information

The accompanying Supplementary Information is supplied as Online Resource 1 (`ESM_1.pdf`; caption: Supplementary Information for BudgetFunnel) and reports multi-window robustness, candidate-ratio and proxy ablations, alternative cold-counting semantics, TTL-grid and cold-penalty sensitivity, the same-scope MILP comparison, reference-budget sensitivity, complete budget sweeps, and runtime details. Online Resource 2 (`ESM_2.zip`), **Supplementary Code S1**, contains the essential reference implementation of BudgetFunnel, including the trace-driven TTL simulator, proxy screening, precedence-aware budget allocation, same-scope MILP validation, preprocessing/data-audit utilities, and documented runtime/MILP configurations. The online resources are provided as separate files and are not part of the numbered main-text reference list.

Acknowledgements

The authors thank the maintainers of the public Azure and Alibaba trace repositories for making workload data available to the research community.

Declarations

Ethics approval and consent to participate

Not applicable. This study uses publicly available, aggregated workload traces and does not involve human participants, animals, or identifiable personal data.

Consent for publication

Not applicable.

Materials availability

Not applicable. The study uses no physical materials; the public datasets are described in the Data availability statement.

Funding

This research was supported by the Natural Science Foundation of Guangdong Province (2025A1515012277), the Research Foundation for Advanced Talents of Lingnan Normal University (ZL2022001), the Higher Education Teaching Research and Reform Project of Lingnan Normal University (2022057), the Lingnan Normal University Special Support Plan for High-Level Talents (ZL2519), Modernized Marine Ranching Industry Talents in 2025 (000-0009925009), the Henan Provincial Natural Science Foundation (252300423518), the Key Scientific Research Projects of Higher Education Institutions in Henan Province (25A140012), Shenzhen Polytechnic Research Fund (6023310019K), and the Post-doctoral Later-stage Foundation Project of Shenzhen Polytechnic University (6023271020K).

Competing interests

The authors declare no competing interests. The funding sponsors had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript; or in the decision to publish the results.

Data availability

The Azure Functions 2019 trace and Alibaba Cluster Trace–Microservices v2021 workload used in this study are publicly available from their original repositories [35, 36]. Raw provider traces are not redistributed and should be obtained from those repositories. The supplementary files are Online Resource 1 (`ESM_1.pdf`) and Online Resource 2 (`ESM_2.zip`); the latter contains code and audit utilities only.

Code availability

Online Resource 2 (`ESM_2.zip`), identified as **Supplementary Code S1**, provides the essential reference implementation and preprocessing/data-audit utilities used to inspect and validate the proposed method. It does not redistribute the raw Azure or Alibaba traces.

Author contributions

Zhipeng Ye: Methodology, Software, Validation, Formal analysis, Writing – original draft. **Zihao Lu:** Investigation, Visualization, Validation, Formal analysis. **Yuan Zhang:** Formal analysis, Writing – review & editing. **Jiayi Hong:** Software, Validation, Writing – review & editing. **Wen-Yu Peng:** Editing. **Zhibin Shao:** Investigation, Validation, Data curation. **Xuming Wu:** Conceptualization, Supervision, Resources, Funding acquisition, Writing – review & editing. **Gaoyuan Zhao:** Conceptualization, Supervision, Project administration, Writing – review & editing. **Yan Cao:** Conceptualization, Project administration.

References

[1] Jonas, E., Schleier-Smith, J., Sridharan, V., Tsai, C.-C., Khandelwal, A., Pu, Q., Shankar, V., Carreira, J., Krauth, K., Yadwadkar, N.J., Gonzalez, J.E., Popa, R.A., Stoica, I., Patterson, D.A.: Cloud programming simplified: a Berkeley view on serverless computing. arXiv:1902.03383 [cs.DC] (2019)

[2] Chai, X., Zhou, T., Hu, K., Tan, J., Bie, T., Shen, A., Shen, D., Xing, Q., Song, S., Yang, T., Gao, L., Yu, F., He, Z., Du, D., Xia, Y., Chen, K., Chen, Y.: Fork in the road: reflections and optimizations for cold start latency in production serverless systems. In: 19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25), pp. 199–218. USENIX Association, (2025)

[3] Nasiri, N., Munshi, N., Moser, S.D., Pirvu, M., Sundaresan, V., Maier, D., Premnath, T., Böwing, N., Gopalakrishnan, S., Shahrad, M.: In-production characterization of an open source serverless platform and new scaling strategies. In: Proceedings of the 21st European Conference on Computer Systems (EuroSys 2026), pp. 2054–2072. ACM, (2026). https://doi.org/10.1145/3767295.3769377

[4] Ustiugov, D., Amariucai, T., Grot, B.: Analyzing tail latency in serverless clouds with STeLLAR. In: 2021 IEEE International Symposium on Workload Characterization (IISWC), pp. 51–62. IEEE, (2021). https://doi.org/10.1109/IISWC53511.2021.00016

[5] Sui, Y., Yu, H., Hu, Y., Li, J., Wang, H.: Pre-warming is not enough: accelerating serverless inference with opportunistic pre-loading. In: Proceedings of the 15th ACM Symposium on Cloud Computing (SoCC 2024), pp. 178–195. ACM, (2024). https://doi.org/10.1145/3698038.3698509

[6] Yu, H., Roy, R.B., Fontenot, C., Tiwari, D., Li, J., Zhang, H., Wang, H., Park, S.-J.: Rainbowcake: mitigating cold-starts in serverless with layer-wise container caching and sharing. In: Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS 2024), pp. 335–350. ACM, (2024). https://doi.org/10.1145/3617232.3624871

[7] Shahrad, M., Fonseca, R., Goiri, I., Chaudhry, G.I., Batum, P., Cooke, J., Laureano, E., Tresness, C., Russinovich, M., Bianchini, R.: Serverless in the wild: characterizing and optimizing the serverless workload at a large cloud provider. In: 2020 USENIX Annual Technical Conference (ATC 20), pp. 205–218. USENIX Association, (2020). https://www.usenix.org/conference/atc20/presentation/shahrad

[8] Luo, S., Xu, H., Lu, C., Ye, K., Xu, G., Zhang, L., Ding, Y., He, J., Xu, C.: Characterizing microservice dependency and performance: Alibaba trace analysis. In: Proceedings of the ACM Symposium on Cloud Computing 2021 (SoCC 2021), pp. 412–426. ACM, (2021). https://doi.org/10.1145/3472883.3487003

[9] Agache, A., Brooker, M., Iordache, A., Liguori, A., Neugebauer, R., Piwonka, P., Popa, D.-M.: Firecracker: lightweight virtualization for serverless applications. In: 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20), pp. 419–434. USENIX Association, (2020). https://www.usenix.org/conference/nsdi20/presentation/agache

[10] Oakes, E., Yang, L., Zhou, D., Houck, K., Harter, T., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: Sock: rapid task provisioning with serverless-optimized containers. In: 2018 USENIX Annual Technical Conference (ATC 18), pp. 57–70. USENIX Association, (2018). https://www.usenix.org/conference/atc18/presentation/oakes

[11] Du, D., Yu, T., Xia, Y., Zang, B., Yan, G., Qin, C., Wu, Q., Chen, H.: Catalyzer: sub-millisecond startup for serverless computing with initialization-less booting. In: Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2020), pp. 467–481. ACM, (2020). https://doi.org/10.1145/3373376.3378512

[12] Fuerst, A., Sharma, P.: 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 (ASPLOS 2021), pp. 386–400. ACM, (2021). https://doi.org/10.1145/3445814.3446757

[13] Roy, R.B., Patel, T., Tiwari, D.: Icebreaker: warming serverless functions better with heterogeneity. In: Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2022), pp. 753–767. ACM, (2022). https://doi.org/10.1145/3503222.3507750

[14] Tian, H., Li, S., Wang, A., Wang, W., Wu, T., Yang, H.: Owl: performance-aware scheduling for resource-efficient function-as-a-service cloud. In: Proceedings of the 13th Symposium on Cloud Computing (SoCC 2022), pp. 78–93. ACM, (2022). https://doi.org/10.1145/3542929.3563470

[15] Shen, J., Yang, T., Su, Y., Zhou, Y., Lyu, M.R.: Defuse: a dependency-guided function scheduler to mitigate cold starts on FaaS platforms. In: 2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS), pp. 194–204. IEEE, (2021). https://doi.org/10.1109/ICDCS51616.2021.00027

[16] Akkus, I.E., Chen, R., Rimac, I., Stein, M., Satzke, K., Beck, A., Aditya, P., Hilt, V.: Sand: towards high-performance serverless computing. In: 2018 USENIX Annual Technical Conference (ATC 18), pp. 923–935. USENIX Association, (2018). https://www.usenix.org/conference/atc18/presentation/akkus

[17] Ao, L., Porter, G., Voelker, G.M.: Faasnap: FaaS made fast using snapshot-based VMs. In: Proceedings of the 17th European Conference on Computer Systems (EuroSys 2022), pp. 730–746. ACM, (2022). https://doi.org/10.1145/3492321.3524270

[18] Jia, Z., Witchel, E.: Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices. In: Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2021), pp. 152–166. ACM, (2021). https://doi.org/10.1145/3445814.3446701

[19] Khrabrov, A., Pirvu, M., Sundaresan, V., Lara, E.: Jitserver: disaggregated caching JIT compiler for the JVM in the cloud. In: 2022 USENIX Annual Technical Conference (ATC 22), pp. 869–884. USENIX Association, (2022). https://www.usenix.org/conference/atc22/presentation/khrabrov

[20] Romero, F., Chaudhry, G.I., Goiri, I., Gopa, P., Batum, P., Yadwadkar, N.J., Fonseca, R., Kozyrakis, C., Bianchini, R.: Faa$t: a transparent auto-scaling cache for serverless applications. In: Proceedings of the ACM Symposium on Cloud Computing 2021 (SoCC 2021), pp. 122–137. ACM, (2021). https://doi.org/10.1145/3472883.3486974

[21] Li, Z., Guo, L., Chen, Q., Cheng, J., Xu, C., Zeng, D., Song, Z., Ma, T., Yang, Y., Li, C., Guo, M.: Help rather than recycle: alleviating cold startup in serverless computing through inter-function container sharing. In: 2022 USENIX Annual Technical Conference (ATC 22), pp. 69–84. USENIX Association, (2022)

[22] Einziger, G., Friedman, R., Manes, B.: Tinylfu: a highly efficient cache admission policy. ACM Transactions on Storage **13**(4), 1–31 (2017) https://doi.org/10.1145/3149371

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

[24] Yu, H., Irissappane, A.A., Wang, H., Lloyd, W.J.: Faasrank: learning to schedule functions in serverless platforms. In: 2021 IEEE International Conference on

Autonomic Computing and Self-Organizing Systems (ACSOS), pp. 31–40. IEEE, (2021). https://doi.org/10.1109/ACSOS52086.2021.00023

[25] Aslanpour, M.S., Toosi, A.N., Cicconetti, C., Javadi, B., Sbarski, P., Taibi, D., Assuncao, M., Gill, S.S., Gaire, R., Dustdar, S.: Serverless edge computing: vision and challenges. In: 2021 Australasian Computer Science Week Multiconference (ACSW 2021), pp. 1–10. ACM, (2021). https://doi.org/10.1145/3437378.3444367

[26] Pan, L., Wang, L., Chen, S., Liu, F.: Retention-aware container caching for serverless edge computing. In: IEEE INFOCOM 2022 – IEEE Conference on Computer Communications, pp. 1069–1078. IEEE, (2022). https://doi.org/10.1109/INFOCOM48880.2022.9796705

[27] Zhou, Z., Zhang, Y., Delimitrou, C.: Aquatope: Qos-and-uncertainty-aware resource management for multi-stage serverless workflows. In: Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS 2022), pp. 1–14. ACM, (2022). https://doi.org/10.1145/3567955.3567960

[28] Zhang, Y., Goiri, I., Chaudhry, G.I., Fonseca, R., Elnikety, S., Delimitrou, C., Bianchini, R.: Faster and cheaper serverless computing on harvested resources. In: Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP 2021), pp. 724–739. ACM, (2021). https://doi.org/10.1145/3477132.3483580

[29] Hu, Q., Ye, Z., Zhang, M., Chen, Q., Sun, P., Wen, Y., Zhang, T.: Hydro: surrogate-based hyperparameter tuning service in datacenters. In: 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23), pp. 757–777. USENIX Association, (2023). https://www.usenix.org/conference/osdi23/presentation/hu

[30] Toosi, A.N., Javadi, B., Iosup, A., Smirni, E., Dustdar, S.: Serverless computing for next-generation application development. Future Generation Computer Systems **164**, 107573 (2025) https://doi.org/10.1016/j.future.2024.107573

[31] Mahmoudi, N., Khazaei, H.: Simfaas: a performance simulator for serverless computing platforms. In: Proceedings of the 11th International Conference on Cloud Computing and Services Science (CLOSER 2021), pp. 23–33. SciTePress, (2021). https://doi.org/10.5220/0010376500230033

[32] Somu, N., Daw, N., Bellur, U., Kulkarni, P.: Panopticon: a comprehensive benchmarking tool for serverless applications. In: 2020 International Conference on Communication Systems & Networks (COMSNETS), pp. 144–151. IEEE, (2020). https://doi.org/10.1109/COMSNETS48256.2020.9027346

[33] Yu, T., Liu, Q., Du, D., Xia, Y., Zang, B., Lu, Z., Yang, P., Qin, C., Chen,

H.: Characterizing serverless platforms with ServerlessBench. In: Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC 2020), pp. 30–44. ACM, (2020). https://doi.org/10.1145/3419111.3421280

[34] Copik, M., Kwasniewski, G., Besta, M., Podstawski, M., Hoefler, T.: Sebs: a serverless benchmark suite for function-as-a-service computing. In: Proceedings of the 22nd International Middleware Conference (Middleware 2021), pp. 64–78. ACM, (2021). https://doi.org/10.1145/3464298.3476133

[35] Microsoft: Azure Functions Trace 2019 (AzurePublicDataset). https://github.com/Azure/AzurePublicDataset/blob/master/AzureFunctionsDataset2019.md. Accessed 16 January 2026 (2019)

[36] Alibaba: Alibaba Cluster Trace–Microservices v2021 (clusterdata). https://github.com/alibaba/clusterdata/tree/master/cluster-trace-microservices-v2021. Accessed 16 January 2026 (2021)

Supplementary Files

This is a list of supplementary files associated with this preprint. Click to download.

- supplementary.pdf