Edge FaaS: Mitigating Serverless Cold Starts with CoW Template Forking and Lightweight Change Detection

Fengcheng Yu*†
fyu54107@usc.edu
University of Southern California
Los Angeles, CA, United States

Tian Xie*
xietian@usc.edu
University of Southern California
Los Angeles, CA, United States

Abstract

Cold starts inflate tail latency in serverless functions, especially at the network edge where deployments may lack MicroVM support or a full container runtime. We present Edge FaaS, a process-pool gateway that combines a Copy-on-Write template process with a lightweight predictive control plane based on EWMA, CUSUM, and Little's Law. The template process pre-imports heavyweight runtime dependencies and forks new workers on demand, reducing worker spin-up from approximately 900 ms to 100 ms. On a four-cycle bursty-ramp workload, Fixed CUSUM reduces cold-start-classified requests by 31% over a reactive baseline and 26% over an ARIMA forecaster, while maintaining sub-microsecond median predictor latency. Ablations show that CoW is most valuable when worker creation lies on the critical path, and an interval sweep identifies when fixed and adaptive CUSUM variants succeed or fail. These results suggest that classical change-detection techniques, paired with Linux fork and Copy-on-Write, can provide a practical cold-start mitigation strategy for resource-constrained edge inference.

Keywords

edge serverless, cold-start mitigation, copy-on-write template fork, change-point detection, predictive autoscaling, Little's Law

1 Introduction

Serverless functions execute on demand and scale down when idle, a model naturally suited to the bursty traffic patterns of *edge inference nodes* [4, 10, 13, 25, 29]—resource-constrained machines deployed near IoT devices, fare gates, or surveillance cameras to provide low-latency machine-learning inference. The cost of elasticity, however, is the *cold start* [1, 2, 8, 20]: when a request arrives and no warm worker is available, the request must wait for a new one to be spun up, a delay that can dominate end-to-end tail latency [7] by an order of magnitude. Cold-start mitigation has been studied extensively in datacenter serverless [1, 2, 7, 8, 29], but the techniques developed there make assumptions that do not transfer to the edge.

The edge setting we target is doubly constrained. Many such deployments lack the memory headroom for keeping MicroVM snapshots warm (Firecracker reserves ~100 MB per VM even when idle [1, 29]) and cannot spend tens of milliseconds on every scaling decision for a heavyweight forecaster: tens of milliseconds of ARIMA [5] inference per scaling decision is a non-trivial fraction of the very cold-start budget the forecaster is trying to mitigate. What edge nodes *do* have is a Linux kernel with a fully featured `fork(2)` [26] and Copy-on-Write (CoW) page sharing [26]— primitives that, properly leveraged, can serve as both the data-plane isolation mechanism *and* the cold-start optimization vehicle.

We exploit this observation in **Edge FaaS**, a process-pool serverless gateway combining two complementary ideas. First, a **Copy-on-Write template process** pre-imports heavyweight runtime dependencies (e.g., Pillow, OpenCV) once at gateway startup; each new worker is forked from the template rather than `execve`'d [26], so the dominant cost of a fresh Python worker—interpreter boot plus large-extension import—is paid once and amortized across all subsequent forks, reducing per-worker spin-up from ~900 ms to ~100 ms (9×). Second, a **lightweight predictive control plane** combines an exponentially-weighted moving average (EWMA [22]) of measured request rate, a CUSUM [21] change-point detector that fires *during* the pre-spike traffic ramp rather than after the spike, and Little's Law [18] to translate the predicted rate into a target worker count. The full predictor has sub-microsecond median per-tick latency, with tail latency more than two orders of magnitude below ARIMA [5] (§5.4, Table 1).

On a four-cycle bursty-ramp workload (§4.1), Edge FaaS reduces cold-start counts by **31% over a reactive baseline** (48±19 vs. 70±1, n=5, 95% CI) and **26% over an ARIMA [5] forecaster**, while occupying the Pareto frontier of cold-start count versus memory footprint when memory is measured using PSS—a metric that avoids double-counting CoW shared pages. A factorial ablation reveals that CoW's contribution *scales with predictor aggressiveness*: it cuts cold-start counts in half for a reactive scaler but provides statistically inconclusive benefit to a conservatively-tuned fixed-CUSUM [21]. An ablation over six inter-burst intervals from 5 s to 120 s identifies regime-specific failure modes for fixed and adaptive CUSUM [21], framing the choice between them as a workload-cadence knob rather than a strict comparison.

Contributions. This paper makes four contributions:

- An edge-oriented CoW-template worker pool that amortizes heavyweight runtime initialization across forked workers, reducing measured worker spin-up latency by 9× without requiring containers, MicroVMs, or snapshot/restore (§3.2).
- A lightweight predictive control plane that composes EWMA baseline tracking, CUSUM ramp detection, and Little's Law sizing into an O(1) pre-warming loop (§5.1).
- A factorial ablation that decomposes the contribution of CoW from the predictor (§5.2), and a six-point inter-burst interval sweep that characterizes the failure modes of fixed-drift versus adaptive-threshold CUSUM (§5.3).
- A PSS-based Pareto analysis that fairly compares memory footprints across baselines differing in worker count (§5.4), correcting a common over-count of CoW shared pages.

*Both authors contributed equally to this research.
†The GitHub repository for this project is available at: https://github.com/ffengc/edge-faas-cpp.

2 Background and Related Work

2.1 Cold-start mitigation in serverless.

Datacenter serverless systems reduce cold starts through several complementary mechanisms [6, 20, 27]. Lightweight virtualization, such as Firecracker [1], reduces MicroVM boot time while preserving strong isolation; snapshot-based systems such as Catalyzer avoid repeated initialization by restoring pre-initialized execution state [8]; and warm-pool schedulers exploit fleet-level resource slack to keep functions close to ready [9, 16, 19, 23, 28]. Recent production work [7] also revisits fork-based starts under the memory and security constraints of large serverless deployments. These techniques are effective in cloud platforms, but they assume infrastructure that the edge setting we target may not provide: a hypervisor or container runtime, snapshot support, image distribution, or a fleet-level scheduler. Edge FaaS instead studies the narrower setting of a single Linux gateway where process creation and Copy-on-Write page sharing are the available primitives.

2.2 Resource-constrained edge inference.

Edge inference nodes—IoT gateways, surveillance cameras, and fare gates—run ML inference close to the data source [14, 17, 29], where round trips to the cloud can be prohibitive in latency or bandwidth. The deployments we target still provide a Linux process abstraction, but cannot assume the full cloud stack: KVM-backed MicroVMs [15], container image distribution, or a fleet-level scheduler. In this setting, process-level workers created with `fork()` [26] and backed by Copy-on-Write page sharing form a practical execution substrate. The resulting cold-start problem is different from container or MicroVM startup: the dominant cost is not image launch or VM boot, but repeatedly importing the language runtime and heavyweight ML dependencies.

2.3 Time-series forecasting for autoscaling.

Time-series forecasting for autoscaling has used Box–Jenkins ARIMA [5] and, more recently, deep recurrent networks such as LSTM [11]. These methods [3, 5, 11, 12, 24] are useful when enough history and compute budget are available, but they introduce two costs that matter in our target edge control loop: they require history before producing stable forecasts, and their per-decision overhead can be nontrivial relative to the cold-start latency being mitigated. We instead ask whether a simpler control plane is sufficient for short-horizon pre-warming: an exponentially weighted moving average [22] for baseline tracking, the cumulative-sum (CUSUM [21]) change-point detector [21] for spike anticipation, and Little's Law [18] for translating arrival rate into a worker count. None of these primitives is individually novel; our contribution is their composition into an $O(1)$ edge control loop and the empirical characterization of where it succeeds and fails.

3 Design

3.1 System Architecture

Edge FaaS is a single-machine gateway implemented as a C++ TCP server that fronts a pool of Python worker processes. Figure 1 gives an overview. Three components run inside the gateway: a single-threaded *Reactor* that handles all `epoll`-based accept/recv/send

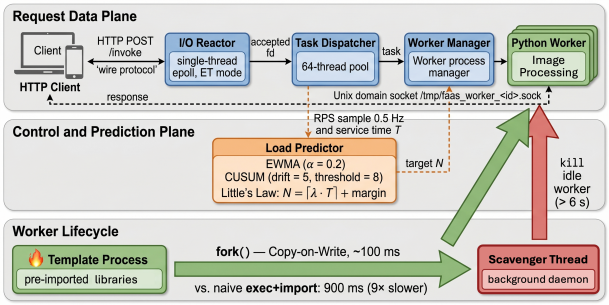

Figure 1: Edge FaaS architecture. The single-threaded Reactor accepts requests over `epoll`/ET; the 64-thread DispatchPool performs blocking UDS I/O to Python workers; the Predictor scales the WorkerPool, whose workers are forked Copy-on-Write from a template process that pre-imports heavyweight runtime dependencies.

in edge-triggered mode; a 64-thread *DispatchPool* that performs the blocking Unix-domain-socket exchange between the gateway and each Python worker; and a *Predictor* that ticks every two seconds, reads the observed request rate, and adjusts the size of the *WorkerPool*, which manages worker lifecycle on top of the template process described in §3.2. Strict separation between the Reactor and DispatchPool prevents any blocking I/O from stalling the event loop even when tens of half-second worker requests are simultaneously in flight. The Predictor itself runs on the Reactor thread because its $O(1)$ per-tick cost is negligible relative to the two-second control interval and is measured explicitly in §5.4.

3.2 Copy-on-Write Template Process

A naive worker pays the full cost of interpreter boot (~100 ms), heavyweight package import (~500 ms for Pillow), and socket setup (~100 ms) on every spin-up, totaling ~900 ms per fresh worker—roughly twice the 500 ms of inference work each worker exists to perform. This cost dominates whenever the WorkerPool must scale up under bursty traffic.

We pay it *once* at gateway startup by spawning a *template process*: a Python interpreter that imports all heavyweight dependencies and then blocks on a control Unix socket waiting for fork requests. When the WorkerPool needs a new worker, it sends the worker's UDS path to the template, which calls `os.fork()` [26] and returns the child's PID. The fork inherits the parent's address space via Linux Copy-on-Write: imported library code, native extension data, and much of the Python interpreter state remain shared unless modified, so the child avoids re-importing heavyweight dependencies. Per-worker warm-up time drops to ~100 ms (fork-call overhead plus per-worker socket bind), a 9× reduction in the measured worker spin-up path.

3.3 Predictive Pre-Warming Control Plane

Every two seconds, the gateway records the requests served in that window as the observed rate λ_t (requests per second) and feeds it to the Predictor, which produces a target worker count N_t. The Predictor chains three composable stages, summarized in Figure 2.

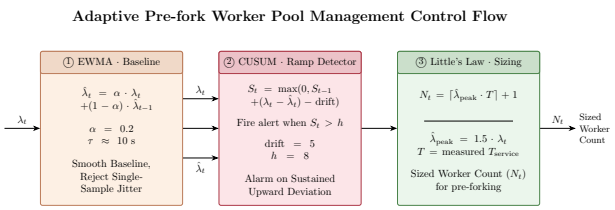

Figure 2: Predictive pre-warming control flow. The Predictor first smooths the observed request rate with an EWMA baseline, then uses a CUSUM statistic to detect sustained upward deviations during the ramp, and finally applies Little's Law to translate the predicted peak rate into a target worker count for pre-forking.

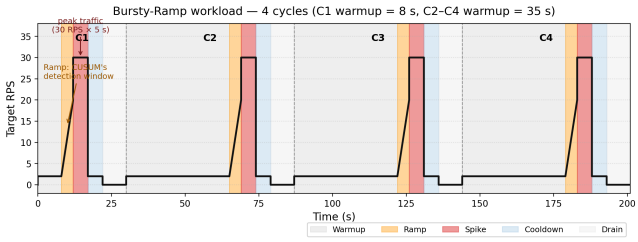

Figure 3: Bursty-Ramp four-cycle workload. C1 uses an 8 s warmup (cold regime); C2–C4 use 35 s warmups so the EWMA baseline and the worker pool fully return to idle between cycles. The orange Ramp band is the pre-spike signal a predictive scheduler can detect; the red Spike is the peak it must provision for.

EWMA baseline tracking [22]. An exponentially-weighted moving average maintains a slowly-adapting estimate of the periodic baseline rate:

$$\hat{\lambda}_t = \alpha \lambda_t + (1-\alpha)\hat{\lambda}_{t-1}, \tag{1}$$

with $\alpha = 0.2$ in our deployment, yielding an effective time constant of $\tau \approx 10$ s. The baseline is intentionally lagging: a single high λ_t does not push it up, so subsequent CUSUM accumulation interprets the deviation as a real signal rather than as a re-baselined normal.

CUSUM change-point detection [21]. A cumulative-sum statistic accumulates *sustained* positive deviations of the observed rate over the baseline:

$$S_t = \max\big(0,\ S_{t-1} + (\lambda_t - \hat{\lambda}_t) - k\big), \tag{2}$$

where a drift constant k filters single-tick noise; S_t resets toward zero whenever λ_t tracks $\hat{\lambda}_t$. An alarm fires when S_t exceeds a threshold h. We use $k = 5$, $h = 8$, tuned against the ramp slope of our workload (§4.1). On alarm, the predicted rate for the upcoming peak is overshot to $\hat{\lambda}_{\text{peak}} = 1.5\lambda_t$ to provision for the spike that the ramp telegraphs, and S_t is reset to zero.

Little's Law [18] worker sizing. The target worker count translates the predicted arrival rate into an integer pool size via Little's Law:

$$N_t = \lceil \hat{\lambda}_{\text{peak}} \bar{T} \rceil + M, \tag{3}$$

where $\bar{T}$ is the EWMA-smoothed measured per-request service time (reported back by each completed request) and $M = 1$ is a safety-margin add. The WorkerPool then forks $\max(0, N_t - \text{alive})$ new workers from the template; a background scavenger thread reaps workers idle longer than 6 seconds, keeping the pool responsive in both directions without an explicit scale-down rule.

Adaptive (standardized) CUSUM variant. The fixed k and h above are tuned for one ramp magnitude. To reduce this dependence, we additionally implement a standardized CUSUM that normalizes each deviation by a running exponentially weighted mean absolute deviation:

$$\sigma_t = \beta|\lambda_t - \hat{\lambda}_t| + (1-\beta)\sigma_{t-1}, \tag{4}$$

with $\beta = 0.3$ in our deployment, corresponding to an effective time constant of roughly 6.6 s under the 2 s predictor tick. The standardized score is then

$$z_t = \frac{\lambda_t - \hat{\lambda}_t}{\sigma_t + \epsilon}, \tag{5}$$

where ϵ prevents division by zero. Adaptive CUSUM applies the same accumulation-and-threshold logic as fixed CUSUM, but on z_t rather than the raw RPS deviation. This makes the alarm threshold less dependent on the absolute RPS scale, but also introduces a cadence dependence through the decay rate of σ_t; we evaluate this trade-off in §5.3.

4 Methodology

4.1 Workload: Bursty-Ramp 4-Cycle

We evaluate Edge FaaS on a synthetic four-cycle *Bursty-Ramp* workload (Figure 3) chosen to expose each predictor's response to recurring traffic spikes. Each cycle consists of five phases. *Warmup* holds the offered load at a low baseline of 2 RPS; *Ramp* climbs from 2 to 20 RPS over 4 seconds, providing the pre-spike signal that an anticipatory predictor can detect; *Spike* sustains 30 RPS for 5 seconds, the peak the predictor must provision for; *Cooldown* returns to 2 RPS for 5 seconds; and a *Drain* phase of 8 zero-RPS seconds lets the WorkerPool's 6-second scavenger reap idle workers before the next cycle. Cycle **C1** uses an 8-second warmup—the truly cold regime, with no prior history—while **C2–C4** use 35-second warmups, long enough for the EWMA baseline and the worker pool to fully decay while still leaving sufficient history for ARIMA [5]. A request is classified as *cold* if its end-to-end RTT exceeds 700 ms, midway between the warm-path latency (~502 ms) and the cold-start fallback (~800 ms). Requests are issued by a wall-clock-paced Python load generator that does not stall on slow responses, so the offered RPS matches the workload definition regardless of server-side queueing.

4.2 Baselines

We compare Edge FaaS (Fixed CUSUM [21]) against four baselines spanning the design space. **Static-15** pre-forks 15 workers at startup and keeps them alive indefinitely (no scavenger), representing a static warm-pool baseline sized for the spike plateau. Since the spike sustains 30 RPS and each worker spends approximately 0.5 s per request, Little's Law [18] gives a steady-state concurrency demand of $30 \times 0.5 = 15$ workers. **Reactive** disables prediction and sizes the pool directly from the current observed rate via Little's Law, producing the worst-case "always-one-step-behind" policy. **ARIMA** runs a Box–Jenkins ARIMA(2,1,2) model [5] in a separate

Python process, queried over a Unix domain socket every two seconds; it falls back to Reactive when fewer than ten history samples are available. **Adaptive CUSUM** is the standardized z-score variant introduced in §3.3. All five modes share the same data plane (Reactor, DispatchPool, WorkerPool, CoW template) and differ only in the Predictor's update rule.

4.3 Setup and Measurement

Experiments run on a single Linux 6.8 (Ubuntu 22.04) virtual machine with 6 GB RAM and 2 vCPUs; the gateway and load generator communicate over localhost loopback. Each trial starts from a fresh gateway process and an empty worker pool, with leftover Unix-domain sockets and worker processes cleaned before the next run. Each main-result configuration ⟨mode, CoW state⟩ is executed for n=5 independent trials, where CoW state indicates whether new workers are forked from the pre-loaded template or launched through the fresh-import path; each sweep point ⟨W, mode⟩ is likewise executed for n=5.

Confidence intervals are reported as ± half-width at 95% level, computed with Student's t at $n-1$ degrees of freedom. Memory is reported as *Proportional Set Size* (PSS), sampled periodically from `/proc/<pid>/smaps_rollup` for the gateway, template, and worker processes and averaged over each run. PSS divides each shared page by the number of processes sharing it, so summing across processes does not double-count CoW-shared memory and gives a fair comparison across baselines that differ in worker count.

Predictor inference latency is measured in-process via a `std::chrono::high_resolution_clock` bracket around each predictor call; ARIMA latency includes the domain-socket round-trip to the forecaster process. Each Python worker performs a representative inference task (Pillow image colour inversion) and includes a calibrated `time.sleep(0.5)` so that $\bar{T} \approx 500$ ms across all workers regardless of background host load.

5 Evaluation

5.1 Cold-Start Counts Across Predictors

Figure 4 reports total cold-start counts per run across the five predictor modes on the Bursty-Ramp workload, with 95% confidence intervals over n=5 independent trials. **Fixed CUSUM (ours) reduces cold starts by 31% over the Reactive baseline** (48 ± 19 vs. 70 ± 1), and by 26% over ARIMA (48 ± 19 vs. 64 ± 12). The disjoint confidence intervals between Fixed CUSUM and Reactive indicate that the improvement is larger than the observed trial-to-trial variability.

Adaptive CUSUM achieves the lowest cold-start count on this workload (0 ± 0 across all five trials). We do not treat this as a strict dominance result: the interval sweep in §5.3 shows that the standardized variant is sensitive to burst cadence and can fail sharply at short inter-burst intervals. We therefore use the main result to show that lightweight CUSUM-based pre-warming is effective, and use the sweep to distinguish the regimes in which Fixed and Adaptive CUSUM are appropriate.

ARIMA places at 64 ± 12 cold starts, slightly better than Reactive but at substantially higher per-tick inference cost (§5.4). It operates in reactive-fallback mode during the first cycle (history shorter than its 10-sample minimum) and only converges across the four-cycle horizon, illustrating the history-warmup cost a heavyweight forecaster incurs on cold deployments.

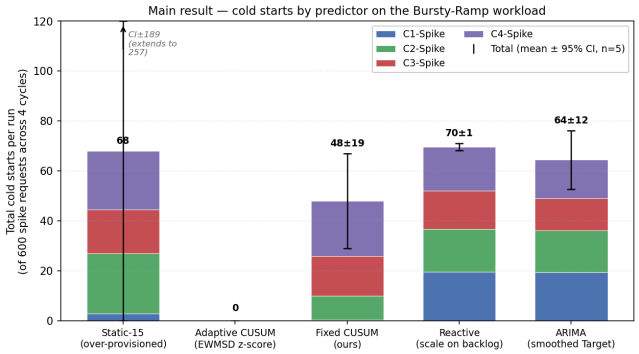

Figure 4: Total cold-start counts per run for five predictor modes on the Bursty-Ramp four-cycle workload. Bars are stacked by spike-cycle (C1–C4); error bars give 95% CI over n=5 trials. Fixed CUSUM (ours) reduces cold starts by 31% over Reactive (48 ± 19 vs. 70 ± 1, disjoint CIs) and 26% over ARIMA. Adaptive CUSUM achieves 0 ± 0 on this workload but is cadence-sensitive (§5.3); Static-15's wide CI reflects a bimodal distribution we attribute to host-side resource contention (§6).

Static-15, despite holding 15 workers permanently warm, registers 68 ± 189 cold starts. The large interval reflects a bimodal outcome across the five trials: some runs complete with no cold starts, whereas others experience substantial tail-latency inflation during burst periods. We analyze this behavior as a resource-contention effect in §6.

5.2 Ablation: CoW × Predictor

Figure 5 decomposes the contribution of the CoW template by rerunning each predictor with CoW disabled (`--no-cow`, where every new worker pays the full ~900 ms fresh-import cost). The pattern is not "CoW helps everyone equally"; instead, CoW's benefit depends on how often and how urgently the predictor creates new workers.

Reactive benefits the most. Reactive's cold-start count nearly doubles without CoW (139 ± 122 vs. 70 ± 1, Δ=+69): with no anticipatory signal, every traffic increase triggers a fresh fork on the request critical path, and CoW's 9× spin-up reduction can determine whether a newly created worker becomes available during the spike or only after much of the spike has passed. The CI on the no-CoW arm is wide because the slow-spawn regime amplifies queueing variance.

ARIMA and Adaptive CUSUM benefit moderately. ARIMA shows Δ=+28 (92 ± 22 without CoW), and Adaptive CUSUM Δ=+11 (11 ± 7 without CoW; from 0 ± 0 with). Both predictors fire multiple scaling actions per cycle, so the per-fork latency is paid often enough to matter, but each individual fork-cost difference is amortized over the between-fire windows.

Fixed CUSUM's CoW effect is statistically inconclusive. Fixed CUSUM shows 48 ± 19 with CoW vs. 36 ± 33 without (Δ=−12, CIs overlap). This reflects the conservatively-tuned CUSUM rule: a

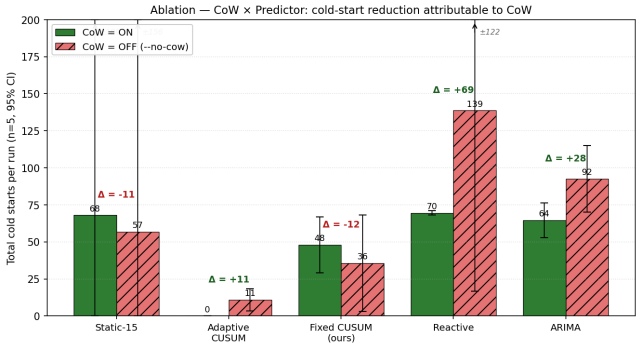

Figure 5: Ablation of the CoW template against each predictor, n=5 trials per cell with 95% CI. Δ above each pair is nocow − cow (positive: CoW saves cold starts). The contribution of CoW scales with predictor aggressiveness: Reactive (no anticipation) loses 69 cold starts without CoW; ARIMA loses 28; Adaptive CUSUM loses 11; Fixed CUSUM and Static show statistically inconclusive deltas.

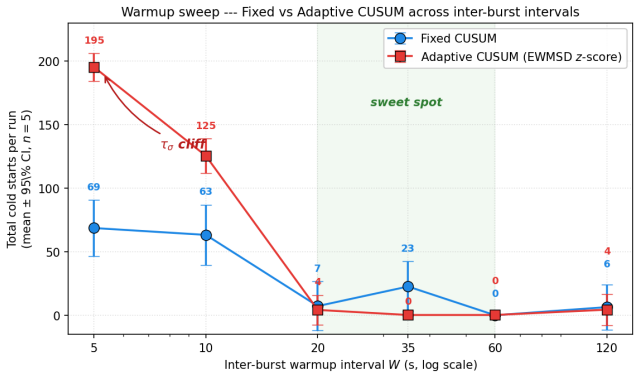

Figure 6: Total cold-start counts across the inter-burst warmup interval W (n=5, 95% CI). Adaptive CUSUM exhibits a τ_σ cliff at short W: with back-to-back bursts the running-scale estimator remains inflated, suppressing the standardized z_t score and delaying the alarm. Fixed CUSUM degrades more gracefully across short W but shows wider CIs in the 35–120 s range. Both perform well in the 20–60 s interval range (shaded).

single alarm per ramp provisions the entire spike's worker pool in one burst, after which the pool stays warm; few subsequent forks occur on the critical path, so CoW's per-fork speedup has limited surface area to exploit. The Static-15 row is similarly inconclusive (Δ=−11, CIs overlap), as that mode forks 15 workers once at startup and never again.

The takeaway is that CoW is a scaffolding that makes the control plane's output *actionable in time*—it is most valuable when paired with a controller that produces frequent or urgent scaling signals (Reactive, ARIMA), and has a smaller, statistically inconclusive effect when the predictor already avoids frequent reactive spawning (Fixed CUSUM, Static).

5.3 Sensitivity: Inter-Burst Interval Sweep

Figure 6 sweeps the inter-burst warmup interval W from 5 s to 120 s, with n=5 trials at each $\langle W, \text{predictor}\rangle$ point. The two CUSUM variants exhibit *distinct failure regimes rather than a strict ordering*; the right choice between them depends on the expected cadence of the deployment.

Adaptive CUSUM exhibits a τ_σ cliff at short intervals. At $W = 5$ s, the standardized variant degrades to 195 ± 11 cold starts per run; the tight CI indicates a structural failure mode rather than trial-to-trial noise. The mechanism follows directly from the running-scale estimator in §3.3: with the roughly 6.7 s EWMSD time constant induced by $\beta = 0.3$ and a 2 s predictor tick, back-to-back bursts prevent σ_t from decaying between cycles. As a result, the same raw RPS deviation is divided by an inflated σ_t, suppressing the standardized z_t score and delaying the CUSUM alarm. The effect attenuates at $W = 10$ s (125±13) and disappears once the inter-burst interval comfortably exceeds the EWMSD decay time.

Fixed CUSUM degrades more gracefully across short W but exhibits occasional ramp-window aliasing. At the same W=5 s point, Fixed reports 69 ± 22—an order of magnitude lower than Adaptive—because its absolute drift threshold remains sensitive even when the baseline is elevated. In the 35 s and 120 s regimes, however, Fixed shows wider CIs (23±20 at W=35 s, 6±18 at W=120 s, both bracketing zero), reflecting bimodal trial-to-trial behaviour where the 2 s measurement window occasionally aliases the ramp slope and delays the alarm by one window.

Both predictors converge in the 20–60 s sweet spot. Across $W \in \{20, 35, 60\}$ s, total cold starts are in the single digits for both modes; the shaded band in Figure 6 marks the interval range where both variants perform well in our workload. The choice between Fixed and Adaptive is therefore best framed as a workload-cadence knob: *Fixed* when burst-to-burst gaps may shorten to the few-second range, *Adaptive* when gaps remain long enough for the EWMSD scale estimate to decay between bursts and tighter cold-start counts at the central operating point matter. We return to the practical implication—an automatic regime-aware selector—in §6.

5.4 Overhead

Predictor inference latency. Table 1 reports the cost of each scaling decision. Reactive and the EWMA/CUSUM variants compute their target in $O(1)$ arithmetic; their median per-tick latency is below the 1 μs resolution of the in-process timer (we report "<1"), with p99 in the low hundreds of microseconds, likely reflecting occasional system noise rather than arithmetic cost. ARIMA, in contrast, incurs a 41 ms median and 72 ms tail, dominated by the Box–Jenkins fit over a 30-sample window plus the Unix-domain-socket round-trip to its forecaster process. The lightweight predictors run *more than two orders of magnitude* below ARIMA's tail.

Memory footprint and the PSS-based Pareto frontier. Figure 7 plots the average PSS (over the run) against total cold-start count for each mode (CoW=ON only, n=5, 95% CI). Fixed CUSUM (ours) sits at the left edge of the frontier with the lowest PSS among non-trivial predictors (2357 MB) and 48 ± 19 cold starts. Adaptive CUSUM occupies the lower-left region at 0 ± 0 cold starts and 2610 MB. Reactive (2780 MB) and ARIMA (2746 MB) lie above; Static-15 sits at 2967 MB—the highest PSS, as expected from its 15 permanently resident workers.

Mode	p50 (μs)	p99 (μs)
Reactive	<1	<1
Fixed CUSUM (ours)	<1	191
Adaptive CUSUM	<1	244
ARIMA	41,087	72,427

Table 1: Predictor inference latency per scaling decision (per-tick), averaged over n=5 trials. ARIMA latency includes the Unix domain-socket round-trip to its forecaster process. The lightweight predictors are sub-microsecond at the median, with p99 below 250 μs.

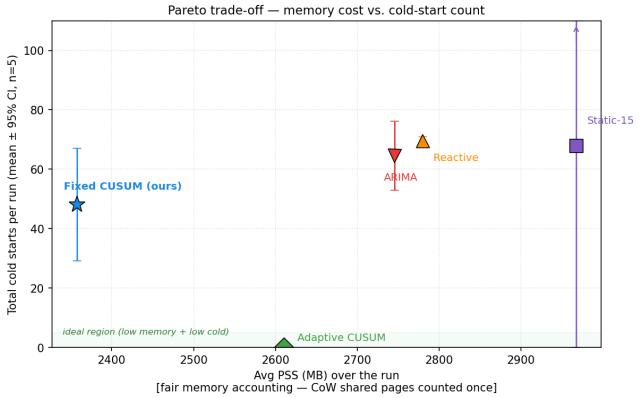

Figure 7: Pareto trade-off between average PSS (physical-memory accounting) and total cold-start count, n=5 with 95% CI. Fixed CUSUM (ours) lies on the left edge of the frontier; Adaptive CUSUM occupies the lower-left region. Static-15 has the highest PSS due to its permanently-resident workers; Reactive and ARIMA neither minimize cold starts nor memory.

We emphasize that PSS is a non-trivial reporting choice: the corresponding RSS totals (which double-count CoW-shared pages) range from 6018 MB to 6957 MB—roughly 2.3× inflated relative to PSS. RSS-based comparisons would mask both the within-mode CoW sharing and the across-mode worker-count differences that PSS exposes, and would mis-rank the modes against true physical-memory cost.

6 Discussion and Conclusion

Limitations. Our predictor is most effective when the workload exposes a detectable pre-spike ramp. In a supplementary step-workload ablation (n=3), where traffic jumps directly from 2 RPS to 30 RPS with no Ramp phase, all CUSUM-based predictors lose most of their advantage: Adaptive degrades from 0 ± 0 to 221 ± 153 cold starts, Fixed from 48 ± 19 to 173 ± 16, and Reactive from 70 ± 1 to 192 ± 3. This confirms that the control plane mitigates predictable bursts rather than instantaneous step changes. Static-15's wide CI in Figure 4 also shows that static warm pools are not a zero-cold upper bound under tight CPU budgets: keeping workers alive removes worker creation from the critical path, but it does not remove host-side scheduling pressure on a 2-vCPU machine.

External validity and future work. All experiments run on a single 6 GB / 2-vCPU Linux VM over localhost loopback. This controlled setup isolates worker-lifecycle and control-plane effects, but does not capture network jitter, NIC interrupts, or bare-metal scheduling effects. We also evaluate one spike magnitude (30 RPS), so Adaptive CUSUM's scale-invariance is algorithmic rather than empirically validated across RPS levels. Future work should evaluate higher request rates using `wrk` or a Rust-based open-loop load generator, replay real serverless traces, and use a regime-aware selector to switch between Fixed and Adaptive CUSUM based on observed burst cadence.

7 Conclusion

We presented Edge FaaS, a process-pool serverless gateway that combines a Copy-on-Write template process with a lightweight EWMA+CUSUM+Little's Law control plane. On a four-cycle Bursty-Ramp workload, Fixed CUSUM reduces cold-start counts by 31% over Reactive and 26% over ARIMA, while maintaining sub-microsecond median per-tick latency. The ablation and interval sweep show that the system's benefit is interpretable: CoW helps most when worker creation is urgent, and the choice between Fixed and Adaptive CUSUM depends on burst cadence. These results suggest that classical statistical-process-control primitives, paired with Linux's native `fork()` and Copy-on-Write page sharing, can provide a practical cold-start mitigation strategy for resource-constrained edge inference.

References

[1] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX symposium on networked systems design and implementation (NSDI 20)*. 419–434.

[2] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. {SAND}: towards {High-Performance} serverless computing. In *2018 USENIX annual technical conference (USENIX ATC 18)*. 923–935.

[3] Saleha Alharthi, Afra Alshamsi, Anoud Alseiari, and Abdulmalik Alwarafy. 2024. Auto-scaling techniques in cloud computing: Issues and research directions. *Sensors* 24, 17 (2024), 5551.

[4] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, et al. 2017. Serverless computing: Current trends and open problems. In *Research advances in cloud computing*. Springer, 1–20.

[5] George EP Box, Gwilym M Jenkins, Gregory C Reinsel, and Greta M Ljung. 2015. *Time series analysis: forecasting and control*. John Wiley & Sons.

[6] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. 2020. SEUSS: skip redundant paths to make serverless fast. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–15.

[7] Xiaohu Chai, Tianyu Zhou, Keyang Hu, Jianfeng Tan, Tiwei Bie, Anqi Shen, Dawei Shen, Qi Xing, Shun Song, Tongkai Yang, et al. 2025. 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)*. 199–218.

[8] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 467–481.

[9] Alexander Fuerst and Prateek Sharma. 2021. 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*. 386–400.

[10] Hassan B Hassan, Saman A Barakat, and Qusay I Sarhan. 2021. Survey on serverless computing. *Journal of Cloud Computing* 10, 1 (2021), 39.

[11] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[12] Pooyan Jamshidi, Aakash Ahmad, and Claus Pahl. 2014. Autonomic resource provisioning for cloud-based software. In *Proceedings of the 9th international symposium on software engineering for adaptive and self-managing systems*. 95–104.

[13] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. 2019. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383* (2019).

[14] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. 2017. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. *ACM SIGARCH Computer Architecture News* 45, 1 (2017), 615–629.

[15] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. 2007. kvm: the Linux virtual machine monitor. In *Proceedings of the Linux symposium*, Vol. 1. Dttawa, Dntorio, Canada, 225–230.

[16] Cheryl Lee, Zhouruixin Zhu, Tianyi Yang, Yintong Huo, Yuxin Su, Pinjia He, and Michael R Lyu. 2024. SPES: Towards optimizing performance-resource trade-off for serverless functions. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 165–178.

[17] Mingran Li, Xuejun Zhang, Jiasheng Guo, and Feng Li. 2023. Cloud–edge collaborative inference with network pruning. *Electronics* 12, 17 (2023), 3598.

[18] John DC Little. 1961. A proof for the queuing formula: L= λ W. *Operations research* 9, 3 (1961), 383–387.

[19] Qingyuan Liu, Yanning Yang, Dong Du, Yubin Xia, Ping Zhang, Jia Feng, James R Larus, and Haibo Chen. 2024. Harmonizing efficiency and practicability: optimizing resource utilization in serverless computing with JIAGU. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. 1–17.

[20] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. {SOCK}: Rapid task provisioning with {Serverless-Optimized} containers. In *2018 USENIX annual technical conference (USENIX ATC 18)*. 57–70.

[21] Ewan S Page. 1954. Continuous inspection schemes. *Biometrika* 41, 1/2 (1954), 100–115.

[22] S. W. Roberts. 1959. Control Chart Tests Based on Geometric Moving Averages. *Technometrics* 1, 3 (1959), 239–250. doi:10.1080/00401706.1959.10489860

[23] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. 2022. Icebreaker: Warming serverless functions better with heterogeneity. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 753–767.

[24] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, et al. 2020. Autopilot: workload autoscaling at google. In *proceedings of the fifteenth european conference on computer systems*. 1–16.

[25] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX annual technical conference (USENIX ATC 20)*. 205–218.

[26] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. 2018. *Operating System Concepts*. Wiley.

[27] Paulo Silva, Daniel Fireman, and Thiago Emmanuel Pereira. 2020. Prebaking functions to warm the serverless cold start. In *Proceedings of the 21st international middleware conference*. 1–13.

[28] Khondokar Solaiman and Muhammad Abdullah Adnan. 2020. WLEC: A not so cold architecture to mitigate cold start problem in serverless computing. In *2020 IEEE international conference on cloud engineering (IC2E)*. IEEE, 144–153.

[29] Kongyange Zhao, Zhi Zhou, Lei Jiao, Shen Cai, Fei Xu, and Xu Chen. 2023. Taming serverless cold start of cloud model inference with edge computing. *IEEE Transactions on Mobile Computing* 23, 8 (2023), 8111–8128.