In-Production Characterization of an Open Source Serverless Platform and New Scaling Strategies

Nima Nasiri
University of British Columbia

Nalin Munshi
University of British Columbia

Simon Moser
IBM

Marius Pirvu
IBM

Vijay Sundaresan
IBM

Daryl Maier
IBM

Thatta Premnath
IBM

Norman Böwing
IBM

Sathish Gopalakrishnan
University of British Columbia

Mohammad Shahrad
University of British Columbia

Abstract

Serverless computing has become more popular and evolved to support more complex tasks than the original Function as a Service (FaaS) model. The design of serverless systems has advanced to accommodate application demands and offer flexibility. Careful characterization of modern serverless systems and understanding of current gaps are warranted. Publicly available datasets on workloads in select production serverless systems do not fully represent all offerings or capture traces at the required time resolution to identify changes in application-level request-response patterns.

We characterize – and make available – production traces from a major public serverless provider with over 1.9 billion invocations spanning over two months. Our dataset is the first to characterize an open-source platform with large-scale trace data, millisecond-scale arrival times, and user configurations for pod concurrency and minimum pod scaling. Using this dataset, we provide new insights for optimizing serverless platforms.

With our insights, we design and implement FeMux, a serverless lifetime management system. FeMux multiplexes lightweight forecasters and uses a Representative Unified Metric (RUM) to decouple metrics from serverless platform implementations. This allows providers like us to update metrics flexibly or support multiple system objectives simultaneously. We prototype FeMux on the Knative platform and evaluate its benefits.

CCS Concepts: • **Computer systems organization** → **Cloud computing**; • **Computing methodologies** → **Learning paradigms**.

EUROSYS '26, Edinburgh, Scotland UK

ACM ISBN 979-8-4007-2212-7/26/04
https://doi.org/10.1145/3767295.3769377

Keywords: Cloud Computing, Serverless, Resource Management, Forecasting, Characterization

1 Introduction

Cloud providers strive to deliver serverless platforms that combine scalability and cost-efficiency through pay-per-use billing, while minimizing resource waste. At the same time, providers must meet rising performance expectations—such as low platform delay, rapid scaling, and fast execution. This inherent tension between efficiency and performance has shaped the evolution of serverless platforms.

Serverless computing began with the Function-as-a-Service (FaaS) model, where users deployed individual functions and providers handled all provisioning. Since then, the serverless model has expanded to support a much broader range of applications—from simple functions to long-running batch jobs and complex, user-defined containers. This transition, referred to by practitioners as "Serverless 2.0", reflects a shift away from simple function execution toward supporting general-purpose applications, as seen in offerings like Google Cloud Run and IBM Cloud Code Engine [12, 16, 22]. Today's platforms let users deploy custom containers capable of concurrent executions and multilingual logic. Configuration options such as minimum scale and concurrency limits allow users to manage cold starts and resource reuse more directly.

Motivated by these shifts in serverless usage and platform capabilities, this paper presents a fresh characterization of modern serverless workloads. We characterize and open-source[1] a large-scale dataset from the serverless offering of a major public cloud provider, IBM. Spanning two months and capturing over 1.9 billion invocations, our dataset is the first to include millisecond-level timing data, up/down scaling events, invocation-to-pod mappings, and detailed user configurations for CPU, memory, and concurrency. Uniquely,

[1] https://github.com/ubc-cirrus-lab/ibm-cloud-code-engine-traces

our trace is derived from an open-source platform – Knative [30] – allowing researchers to reproduce behavior and experiment directly using local deployments. While prior datasets—such as Azure's 2019 and 2021 traces [45, 57], or Huawei's more recent traces [26, 27]—have catalyzed research, their limited resolution, incomplete metadata, and narrow application scope reduce their applicability to today's evolving workloads. Our dataset yields new insights with implications for serverless system design and management. These insights stem both from the fidelity of our data and from the evolving behavior of serverless workloads—some of which challenge prevailing assumptions. For example, while recent literature suggests that cold start times are improving, our logs reveal that *tail cold start times are actually increasing*. This trend aligns with the shift toward fully managed, multi-runtime containers [27], which can introduce heavier startup paths. We also observe that most workloads are configured with non-zero minimum scale—suggesting that users often prioritize latency over efficiency, even at increased cost.

We translate these insights into two system-level principles for lifetime management: (1) decoupling platform optimization from specific metrics, and (2) adapting forecasters to workload characteristics, using our Representative Unified Metric (RUM). We prototype these ideas in FeMux, a lifetime management system built atop Knative [30], which uses RUM to optimize across shifting metrics and traffic patterns via lightweight forecaster multiplexing.

Our main contributions include:

- **The first large-scale serverless dataset from a production open-source platform (Knative)**: over 1.9 billion invocations spanning two months, with millisecond-scale timing, pod mappings, user-specified resource settings, and concurrency configurations—surpassing prior traces in fidelity and breadth.
- **A modular, low-latency forecasting multiplexer-based lifetime manager (FeMux) that outperforms prior work**: Compared to FaasCache, IceBreaker, and Aquatope, FeMux reduces cold starts by up to 64%, lowers memory waste by up to 25%, and improves overall optimization objectives by 30–78%. Our approach uses supervised learning and has acceptable training times (a few hours) as well as short inference times (a few milliseconds) for deployment.
- **A flexible optimization framework (RUM) that enables simultaneous support for multi-tier objectives**: FeMux supports dynamic prioritization (e.g., latency vs. cost), achieving a 45% cold-start-time reduction for latency-sensitive workloads with 35% lower memory waste overall.
- **Real-system validation through a Knative prototype**: We integrate FeMux into Knative Serving and evaluate our implementation. FeMux supports over 1,200 applications per forecasting pod with a p99 forecasting latency of 25 ms—demonstrating feasibility in practical settings.

2 Background

2.1 Evolution of Serverless Compute

FaaS was the initial form of serverless compute, where users upload function and the provider provisions resources for execution upon invocation. We use *invocation* to refer to both user requests and other triggers, e.g., timers or system events. Serverless workloads typically execute in containers or micro-VMs [2], where platforms such as Openwhisk [38] and Knative [30] orchestrate invocations, scheduling/load balancing, and adaptively scale instances based on load. Once a container is active, the language runtime is set up by pulling an image. For languages such as Java, the language runtime is heavy and can be slow to start (e.g., >1 s), while interpreted languages such as Python and Javascript are faster [48].

The FaaS model demonstrated the core benefits of serverless computing, such as an event-driven architecture, high scalability from zero instances, and a pay-per-use model. However, provider-side limitations on programming languages, containerization, and APIs were limiting the possibility of expansion to a broader set of applications. As a result, over the past five years, the notion of what serverless is has slowly grew to well beyond the original FaaS model. The *"idea of serverless functions replacing the traditional web frameworks or APIs has almost disappeared"* [16]. This shift is referred to as "Serverless 2.0" among practitioners, and can be seen in new serverless offerings such Google Cloud Run (vs. Google Cloud Functions) and IBM Cloud Code Engine (vs. IBM Cloud Functions) [12, 22].

It may appear that FaaS provides the same workload flexibility as to Serverless 2.0, but the difference is non-trivial. FaaS platforms (e.g., AWS Lambda) are designed to takes user code and run it on pre-initialized container images for standard runtimes (e.g., Python, Go). Custom runtimes are available, but are not as flexible as custom containers. Modern serverless platforms, or container-as-a-service platforms (e.g., AWS Fargate) offering serverless capabilities (e.g., pay-per-use, event-driven, scale down to zero) can pull custom containers. While increasing initialization overheads, they allow for multilingual applications, custom base images, custom operating systems, and multi-container pods as unit of scaling. Apart from the expanded use cases, these services typically include more configurability options with independent memory/CPU allocation, concurrent execution on a single instance, longer runtimes, and more storage.

IBM's *workload*s comprise of approximately 75% *applications*, 15% *batch jobs*, and 10% *functions*, which are run simultaneously on top of the same platform (Knative). Applications, having the highest flexibility, are uploaded using container images and can make up the entire back-end system for users, while functions are uploaded as code snippets. Users can configure application pods to have a concurrency limit of 1 or more (default 100), while each function instance

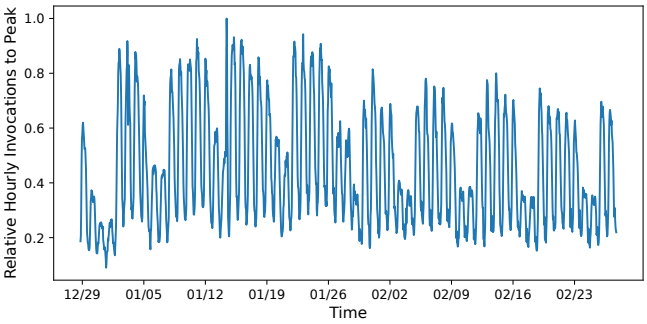

Figure 1. Across 62 days, there is a peak-to-trough span of approximately 60% of weekdays and 40% on weekends relative to peak traffic. Apart from increased traffic during weekdays, there is a seasonal increase in traffic in January.

handles at most one execution concurrently. Further, function containers are initialized using standard images complex applications use custom images which can take longer to load (§3.3). Batch jobs don't receive HTTP inbound requests and are triggered manually or by events (e.g., timers).

2.2 Resource Management Policies

Several avenues of prior work tackle a range of resource management problems, including: managing the lifetime and scaling of resources, reducing cold start durations, scheduling functions to resources, resource sharing, and right-sizing resource configurations to avoid over-allocation. Lifetime management policies use forecasting or heuristics to decide when to scale resources [20, 40, 45, 60]. More recently there are policies exploring scaling resources by layers rather than whole containers [56]. Scaling decisions are split between predictively scaling resources ahead of invocation (pre-warming), and keeping resources alive when expecting future traffic (keep-alive). Schedulers assign functions to resources, with some systems handling independent functions [28], and more recent work tackling complex applications which are organized as Directed Acyclic Graphs, and the consideration required for pre-warming, data movement, resource sharing, and co-location [33, 34]. Through improved lifetime management and container scheduling, cloud providers can improve both service times and resource efficiency. Since variable traffic and user misconfiguration lead to over-allocating resources, there are tools that help users right-size their resource configurations [9, 17, 36], and prior work that adjusts resource allocation based on historical consumption data for each workload [60]. Providers typically bill for resource allocation as opposed to consumption, a practice that can seemingly profit from users over-allocating. Nevertheless, such over-allocation translates into higher opportunity costs for the provider throughout the execution and post-execution idle phases.

	Azure '19 [45]	Azure '21 [57]	Huawei Pub '22 [26]	Huawei '24 [27]	**IBM**
Req. Time Accuracy	min	ms	min	min*	ms
Execution Durations	ms (Daily)	ms (Per Req.)	N/A	μs (Per Min.)	ms (Per Req.)
Platform Delay	N/A	N/A	N/A	μs	ms
CPU/Mem. Allocation	No	No	No	Yes	Yes
CPU/Mem. Usage	Yes/No	No	No	Yes	No
Concurrency & Min. Scale Configs	N/A	N/A	N/A	No	Yes
Scale Up/Down Events	No	No	No	Yes/No	Yes
Duration (days)	14	14	26	31	62
Total # Invocations	12.5 B	2 M	2.5 B	85 B	1.9 B
Open Source Platform	✗	✗	✗	✗	✔

Table 1. Comparison of existing serverless datasets with ours. Our dataset spans the longest, has complete millisecond-scale data, and includes data for both Memory and CPU configuration data at the cost of missing resource usage statistics. *Huawei '24 includes microsecond-scale invocation-level data (<1B requests) for one day, and platform delays for each cold start event at the component-level.

3 Characterization

3.1 The Need for New Serverless Characterization

Table 1 summarizes four publicly released serverless datasets, and refer to the Huawei datasets from the year they were collected. We provide a new dataset to provide the research community with a more holistic view of the serverless landscape, and to overcome existing limitations:

1. **New perspective:** A limitation of prior serverless datasets is their reliance on only two providers: Microsoft and Huawei. We address this gap by providing data collected from a third major public cloud serverless provider.
2. **Open-source platform**: Prior industry datasets are gathered from proprietary serverless platforms. We provide the first dataset gathered from an open source serverless system (Knative [30]). This allows the community to replay traces meaningfully on a publicly known platform.
3. **Serverless 2.0:** Existing datasets, except for Huawei '24 [27], strictly describe FaaS workloads, which do not capture the wider range of possibly longer running and more complex workloads supported by Serverless 2.0.
4. **High volume of fine-grained data:** Azure '19 [45] and Huawei '22 [26] datasets have minute-scale invocation data, providing per-minute invocation counts (1440 daily data points). Azure '21 and Huawei '24 datasets include trace data at millisecond and microsecond granularities, but only capture under 2 M requests, and 1 day of data, respectively, in those granularities. We provide high volume (1.9 B) of millisecond-granularity data, for over two months (Fig. 1).

Our characterization highlights immediate optimization opportunities for production-grade serverless platforms. Our dataset, to be open sourced with this paper, comprises 1,283 traces from a subset of the total projects running on IBM's second most popular sub-region, with 75% of traffic volume relative to the most popular sub-region. A limitation of our

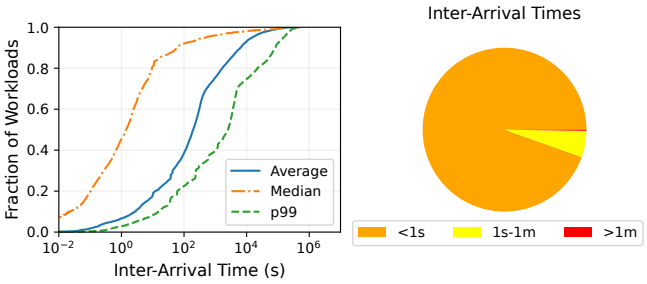

Figure 2. (Left) The distribution of median IAT values shows a significant gap compared to the 99th percentile IAT values. (Right) More than 94% of all IAT values are sub-second.

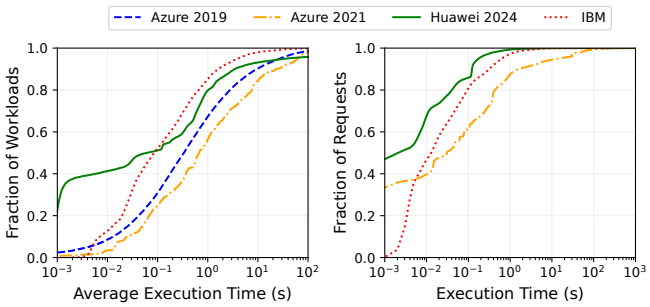

Figure 3. Newer datasets (ours and Huawei '24) show shorter execution times. 82% of our applications and 96% of invocations have sub-second average execution times.

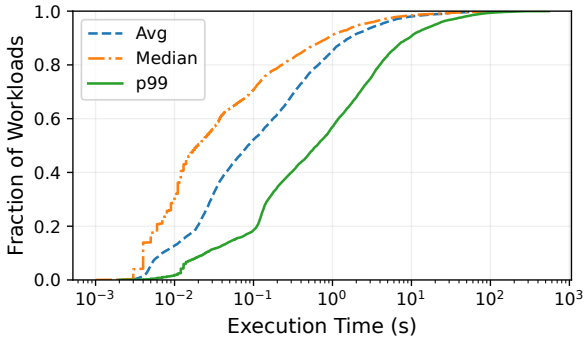

Figure 4. High exec time variability in our workloads.

characterization is that only batch jobs can be distinguished– we refer to functions and applications as workloads since they cannot be separated based on the available data. Our reported numbers on the fraction of each workload type are based on all serverless cloud workloads, as opposed to the subset included in this dataset. In the rest of this section, we identify several lines of research for lifetime management policies that optimize cold starts and resource efficiency, and right-sizing tools and dynamic resource allocation policies which help users efficiently configure resources and providers to improve utilization. Readers interested in more insights are referred to the supplementary material.

One of the limitations of our dataset is the lack of system utilization data, but this aspect was not part of the system's logging facility. Our traces provide request data, including invocation time, application, pod, and execution time, alongside daily per-app configuration metadata, including pod concurrency. These are sufficient to reconstruct the scaling behavior of our workload.

3.2 Second-Scale Resource Management

Invocation Inter-Arrival Times (IATs). We observe that over 94.5% of invocations have sub-second IATs, and 5.2% have sub-minute IATs (Fig. 2-Right). Traffic patterns can vary significantly across workloads and throughout the lifetime of each application (Fig. 2-Left): 46% and 86% of workloads have sub-second and sub-minute median IATs, respectively. As 99.8% of the total traffic has sub-minute IATs, and over 86% of workloads have sub-minute median IATs, there is potential for fine-grained resource management.

The disparity between median and 99th percentile inter-arrival times for over 95% of workloads further indicates the intermittent nature of traffic: in addition to fine-grained resource management, there is still a necessary consideration for long periods with no traffic. By considering the coefficient of variation (CV) for inter-arrival times, we approximate the variability of workloads, where a CV larger than one represents high variability ($\sigma > \mu$). Over 96% of workloads from our dataset have a CV above 1, while 78% of workloads from the Azure '21 dataset have high variability[2].

Lastly, Azure '19 characterization reported that 19% of applications had an average IAT of less than one minute, and 45% had an average IAT less than an hour [45]. In our workloads, these figures have increased to 26% and 69%, respectively, indicating increased traffic volumes for mid-popularity serverless applications. This can be attributed to the increased adoption of serverless among developers.

Execution Times. We examine the distribution of execution times from our trace and previously available serverless execution time data (Fig. 3). The Azure '19 dataset only offers daily per-app average statistics, which is why it is missing on the right subfigure. We observe shorter execution times for our workload and the Huawei '24 workload relative to prior years. 82% of workloads from our dataset have average execution durations under one second. This fraction was lower in the Azure '19 dataset, at 70% [45]. This trend has been reported before; a 2021 Datadog report found that the median execution time of AWS Lambda functions dropped from 130 ms to 60 ms between 2019 and 2020 [13]. Several factors likely contribute to the ongoing reduction in serverless execution times: 1) developers' growing proficiency with the serverless paradigm, including breaking down complex

[2]Only the Azure '21 dataset and ours include precise invocation times (Table 1), enabling us to calculate the CV values of IATs.

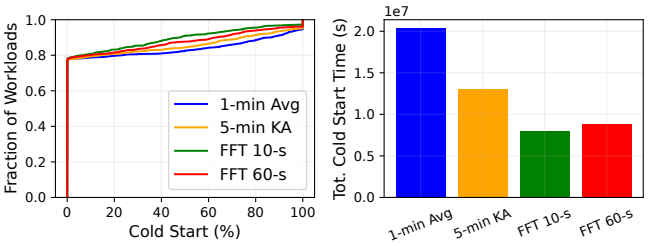

Figure 5. (Left) FFT with a 10-second timestep achieves the lowest fraction of cold starts across all workloads, and (Right) reduces total cold start durations by 60% and 38% relative to a 1-min moving average and 5-minute keep-alive respectively.

functions into smaller stages for independent scaling; 2) the increased adoption of serverless workflows [34]; and 3) the rising use of serverless for latency-sensitive, user-facing applications.

We also observe significant execution duration variability within each workload, as the median value for average workload execution time is 10 ms while the median value for p99 workload execution time is 800 ms (Fig. 4). Additionally, some workloads experience longer request completion times during higher traffic periods (Appendix B.2).

Serverless platforms immediately scale their first container. However, providers such as AWS and Huawei, and prior work [20, 40, 45, 60], scale down containers one or more minutes after the container starts idling. We evaluated the scaling behavior of AWS Lambda by sending requests to a variable execution function with inter-arrival times ranging from 1 to 8 minutes. We observed that all requests with an idle time (completion to next arrival) exceeding 6 minutes were cold. Similarly, Joosen et al. [27] report that Huawei's serverless platform uses a 1-minute keep-alive policy. Knative's default autoscaling policy adjusts the number of pods every two seconds, but uses a 1-minute sliding window for determining the number of pods to scale.

Predictive scaling policies with more granular timescale can reduce container lifetimes, while also preventing cold starts. We explore scaling pods in 10-second time steps using an event-based simulator. Per-app traffic is captured by an application's average concurrency, as used in Knative [30]. We use a Fast Fourier Transform (FFT) forecaster from prior work [26, 40] to evaluate the benefits of forecasting, with both 60-second (minute-scale) and 10-second scaling frequencies to evaluate finer-grained lifetime management. Platforms such as Knative passively adjust the number of pods based on traffic rate every two seconds, so a 10-second predictive scaling period would not be excessive.

Sub-Minute Scaling. We observe that predictive scaling reduces the fraction of cold starts across all workloads (Fig. 5-Left). FFT with a 10-second timestep reduces total cold start by 60% the 1-minute moving average with 2-second scaling intervals, used by Knative's autoscaling policy (Fig. 5-Right).

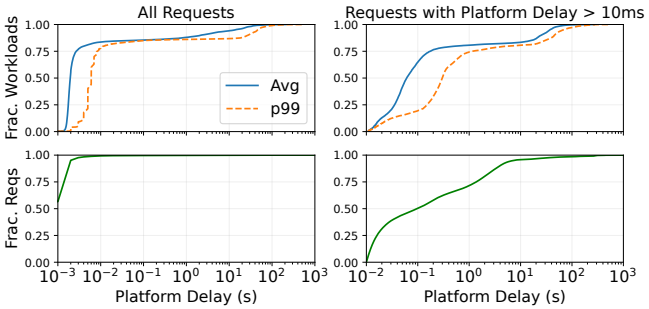

Figure 6. Most executions experience short platform delays, but the tails are very long, exceeding 300 s.

It also reduces total cold start duration by 38% compared to a simple 5-minute keep-alive policy used in AWS Lambda. Further, FFT with a 10-second timestep improves cold starts across all workloads relative to a 60-second timestep, and reduces the aggregate cold start duration by 11%. Additional resource allocation for all of these policies is less than 1%, due to the prevelance of user-configured minimum scale pods (§3.4).

> ***Implication 1****: Most inter-arrival and execution times are sub-second, while existing lifetime management systems scale down in minute-level. High frequency predictive management can improve resource efficiency and performance.*

3.3 Platform Delay

We characterize platform delay—the difference between total service time and execution time—which captures the combined impact of cold starts, queuing, inter-component latency, and network delays. Unlike prior studies from proprietary platforms like Huawei's [26, 27], our analysis is based on a Knative-enabled platform.

We explore platform delay distributions across workloads (Fig. 6-top) and invocations (Fig. 6-bottom). The left plots include all data; the right focus on delays over 10 ms for clarity. Most executions have delays under 1 ms, and 73% of workloads have 99th-percentile delays below 10 ms.

Two main factors, explored further (§3.4), contribute to these low delays: (1) Serverless 2.0 platforms use many-to-one execution models, where each pod handles many concurrent requests—reducing the need for frequent scaling; (2) Minimum pod counts ensure baseline resource availability.

Long-tail delays, however, remain. About 20% of workloads have p99 delays exceeding one second, with extremes above 400 s. Such delays are largely caused by cold starts from custom containers, which involve the loading and initialization of complex dependencies. Joosen et al. observed similar cold starts over 10 s with custom images on Huawei's platform [27].

***Implication 2**: While serverless systems provide low delays for most workloads, the complexity of modern applications leads to long-tail latency. Mitigating these tails requires new techniques and necessitates lifetime managers to consider cold start durations—not just cold start counts or percentages.*

3.4 Resource Configuration Patterns

Azure2019 included daily memory consumption statistics [45], and Huawei2024 includes consumption and allocation for both function and complex application workloads, but **this study is the first to include resource allocation alongside concurrency and minimum scale configurations**. This information sheds light on user behavior, as well as on opportunities for autoconfiguration and dynamic rightsizing. The data suggests that users change configurations for up to 10% of workloads; we report median configuration across 62 days (Fig. 7). We discuss each one next.

CPU and Memory Allocation. Default configuration for CPU is 1 vCPU. 44.8% of workloads were configured with less than 1 vCPU, and only 4.4% with more than 1 vCPU (up to 8 vCPUs). The default memory configuration is 4 GB. We see more memory reduction from the default config compare to CPU: 53.6% of workloads requested less than 4 GB, and 4.5% went beyond this limit (up to 48 GB). A respective 50.8% and 41.9% of CPU and memory allocations at their default value, which may indicate over-allocation of resources for those workloads. During the 62-day period, 2.3% and 3.3% of workloads updated their CPU and Memory configurations, respectively, with an approximately even number of users increasing/decreasing up to 2GB of memory and 0.5 vCPU.

Minimum Pod Scale. Not long ago, FaaS developers who needed to keep their containers warm had to send artificial traffic [32]. Serverless 2.0 support provisioned concurrency: allowing users to configure a minimum number of warm containers to mitigate cold starts [3, 25]. The default value for this option is zero in our system, which permits the number of pods to scale down to zero. However, we see that 58.8% of workloads requested to have one (53.8%) or more (4.9%) minimum pod scale. This high number showcases the importance of cold start mitigation to users and a clear need to reduce cold start times or mitigate them entirely.

***Implication 3**: Almost 60% of workloads had a minimum pod scale of at least one (default is 0), signifying that service times are still important to users. Cold start mitigation and reduction should be improved to reduce the need for idle pods.*

Container Concurrency Limit. The serverless model has evolved to have containers that can be configured to run hundreds of executions concurrently. We observe that 93.3% of workloads use Knative's default container concurrency of 100. With far more workloads having their default CPU/memory settings changed than their concurrency configurations, it appears adjusting resource allocations is a more intuitive for users. 3.2% of workloads configured concurrency higher than 100 (up to 1000). Containers set with higher concurrency require less scaling and avoid long cold starts for custom images, but their static resource allocation can lead to over-provisioning during low traffic.

***Implication 4**: Serverless systems have evolved to support containers running hundreds of executions concurrently. This amplifies resource wastage when containers run at lower concurrency than configured.*

4 Enhanced Lifetime Management

Using our characterization insights, we build on recent research in adaptive lifetime management of serverless applications. We revisit two seemingly obvious assumptions that prior studies have overlooked:

(1) **There is no consensus on performance metrics or efficiency metrics to use** (Table 2). Existing systems cannot dynamically adapt to different metrics, as they use decoupled metrics for optimizing system components (e.g., Mean Absolute Error) and evaluating the performance and efficiency of their overall system–making it difficult to support new metrics or different tiers of service [31].

(2) **Existing solutions use one forecasting approach** [20, 40, 45, 60]. We find that individual forecasting models and heuristics do not adapt to all applications, given the variable nature of serverless workloads [26, 28, 44].

We translate these two insights into two implementation-level principles for building extensible lifetime management systems that adapt to evolving serverless metrics and workloads [26, 34], incorporate advances in predictive models [35], and maintain low overheads. First, we introduce and use a Representative Unified Metric (RUM) to decouple specific metrics from the operation and implementation of serverless platforms (§4.1). This allows providers to update or extend metrics of interest or simultaneously support multiple sets of metrics for different service tiers without impacting platform design and implementation. RUMs are meant to be used as the objective function for optimization *and* evaluation: aligning the optimization of the overall system and its components. RUMs enable systems to adapt to changing metrics. Second, we propose automatically selecting forecasters based on different types of applications and traffic patterns, and optimizing forecasters using RUM rather than common error metrics (§4.2).

4.1 Towards Flexible Metrics

Our characterization (§3) reveals diversity in workload behavior that motivates a more flexible approach to lifetime management. For instance, while most workloads experience low platform delays, about 20% exhibit long-tail latencies

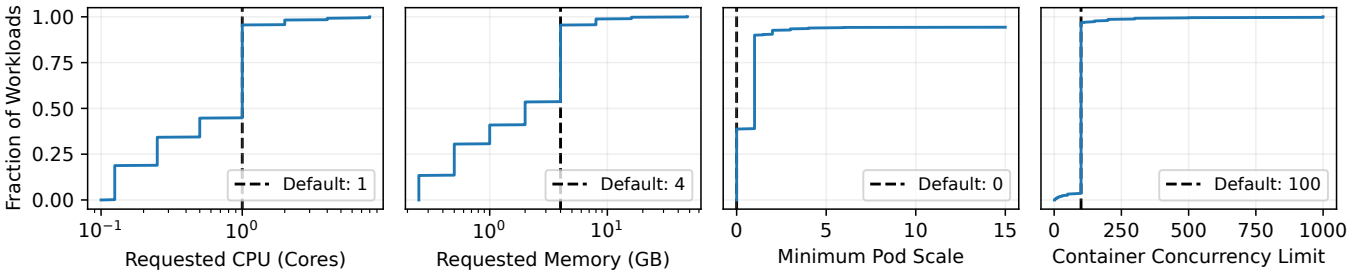

Figure 7. The distribution of workload configurations illustrates how users alter the default configurations.

Criteria	Metric	SITW [45]	FaasCache [20]	IceBreaker [40]	AQUATOPE [60]
Performance	Cold Start % Per App	●			
	Overall Cold Start %		●		●
	Service Time		●	●	
	Number of Cold Starts		●		
Efficiency	Wasted Memory Time	●			
	Allocated Memory Time				●
	Total Keep-alive Cost ($)			●	

Table 2. There is no consensus on the preferred performance and efficiency metrics for serverless lifetime management. Metrics are aggregated over all apps unless stated otherwise.

exceeding one second (**Implication 2**). This highlights that a single, fixed optimization metric is insufficient. A system that optimizes only for cold start *percentage* might ignore a workload with few but extremely long cold starts, leading to poor user experience. Furthermore, our findings show that users often default to costly configurations like non-zero minimum pods (**Implication 3**) and high concurrency limits (**Implication 4**) to manually prioritize latency over cost.

The evidence from our characterization drives our first design principle: **decoupling platform optimization from specific, hard-coded metrics**. To achieve this, we employ **RUM**s to encode efficiency and performance trade-offs in a tunable objective. This decouples system logic from metric definitions and supports multi-tier services and evolving goals. Unlike prior systems that fix metrics in code or use them solely for evaluation (Table 2), we use RUM to drive both component optimization and platform-wide decisions.

Providers must be able to evolve optimization goals without redesign. With Serverless 2.0, user-defined containers amplify variability, making design with static metrics untenable. 9% of apps see over 10s delay, while most remain below 1s (§3.3)—emphasizing the need for per-app flexibility.

> ***Implication 5**: A serverless lifetime management system should allow for (1) updating the set of optimization metrics and (2) simultaneously optimizing for various metrics on the same platform.*

RainbowCake [56] takes a first step by combining memory and latency in one policy; RUM generalizes this idea. We introduce two example RUM formulations. The first reflects trade-offs seen in prior work and uses public cloud data:

$$w_1 \times (\text{cold start seconds}) + w_2 \times (\text{wasted GB·seconds}) \quad (1)$$

This metric captures the trade-off between latency and memory. The weight ratio w_2/w_1 reflects how much memory providers are willing to waste to avoid one cold start second.

We estimate w_1 and w_2 using publicly available data from the three largest cloud providers. We first take a weighted average of keep-alive times across AWS, Azure, and Google [37, 48] based on market share [1] to infer a provider-agnostic keep-alive time of 537 seconds. Given a median memory consumption of 150 MB for Azure workloads [45], we conclude that up to 80.5 GB-seconds are wasted per cold start. Next, using a similar analysis, we found that the weighted average of cold start times across providers and languages is 0.808 seconds. We consider cold start times for Python, JavaScript, Go, and Java across the same three providers: first weighing cold start times across languages based on popularity [13, 48] for each provider, then weighing cold start times across providers based on market share [1].

With 80.5 GB-seconds of wasted memory per cold start, and an average cold start duration of 0.808 seconds, we deduce that providers waste $\frac{80.5}{0.808} \approx 99.7$ GB-s per cold start second. Consequently, we set $w_1 = 1$ and $w_2 = \frac{1}{99.7}$. Unless stated otherwise, we use the default RUM with these weights in later sections, setting all cold start durations to 0.808 seconds. Fixed cold start durations ensure fairer comparisons to prior work as some did not consider variable cold start durations, and minimizes the scope of our assumptions as cold start durations are not available in public cloud datasets.

$$w_1 \times \sqrt{\frac{\text{cold start seconds}}{\text{execution time}}} + w_2 \times (\text{wasted GB·seconds}) \quad (2)$$

This alternative metric emphasizes cold start mitigation for short-lived executions. We do not claim universality for either. Instead, we show that RUM enables optimization across diverse workloads. FeMux can support multiple such formulations simultaneously (§5.1.2).

4.2 Towards Flexible Traffic Forecasting

Our analysis (§3.2) showed the intermittent nature of serverless traffic, with over 96% of workloads exhibiting high variability in their inter-arrival times ($CV > 1$). This confirms that a 'one-size-fits-all' forecasting approach, common in prior work, is unlikely to be optimal. This insight drives our **second design principle: adapting forecasters to workload characteristics**.

Active lifetime management policies rely on workload forecasting to predict incoming traffic: informing keep-alive and pre-warming decisions. This section is focused on the importance of forecasting flexibility. For the following studies, we used simulate forecasts for 13k randomly chosen, representative[3] applications from the Azure 2019 trace [18, 45].

4.2.1 Not All Accuracy Metrics Are the Same. Accuracy of forecasters can be measured by standard statistical metrics. Prior work [40, 60] uses metrics including Mean Absolute Error (MAE) [40] and Symmetric Mean Absolute Percentage Error (SMAPE) [60] to evaluate and optimize workload forecasters. However, generic accuracy metrics do not align with system's performance metrics, like fewer cold starts or lower resource wastage. We show the merit of directly using metrics of interest (RUM) for assessing and tuning forecasters by comparing Autoregressive (AR) and FFT forecasters model used in prior work [26, 40]. Each forecaster predicts the average concurrency (used in Knative's Autoscalar) for each application over time. We assess the predictive power per application, based on MAE and RUM. Even with identical forecasters, different metrics yield different conclusions. AR is superior for 65.2% of applications when assessed using MAE. However, FFT outperforms AR for 68.9% of apps for the metric that we ultimately care about, the RUM. Common error metrics assign equal weight to all errors, regardless of their impact on system performance. An insight guiding FeMux's design is to adaptively use forecasters based on the metric of interest, rather than relying on raw statistical error metrics.

> ***Implication 6**: Forecasters used in an adaptive lifetime management system need to be objective-aware.*

4.2.2 Not All Applications Are the Same. Prior work uses a single forecaster for all applications [6, 40, 45, 60]. Considering the diverse serverless workload patterns, using a single forecaster for all applications is debatable. To illustrate this with a simple example, we compare the performance of those two forecasters (discussed in §4.2.1) using RUMs. In doing so, we use the same set of Azure traces and classify the applications based on the invocation count in 12 days of data and report RUM per category. FFT performs better for applications with fewer than 1 million invocations, while AR performs better for applications with more (Fig. 8-Left).

[3]The invocation frequency distribution follows that of the full dataset.

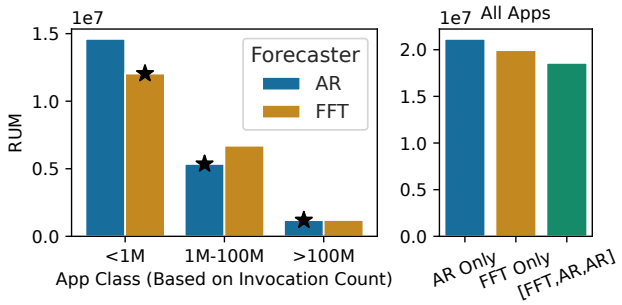

Figure 8. A simple app classification by invocation volume shows forecasting quality varies across app classes with different forecasters (Left). Using the right forecaster per class reduces RUM (Right). FeMux automates classification and feature/forecaster selection for custom RUMs.

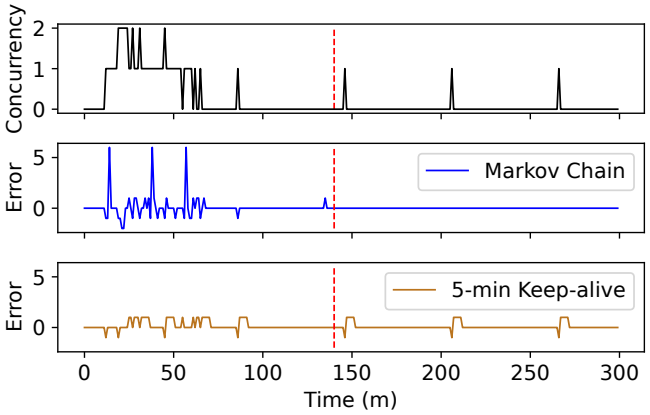

Figure 9. Forecasters' suitability can change over time.

Fig. 8-Right compares the aggregate RUM for all applications when only one forecaster (AR or FFT) is used, versus using the optimal forecaster for each class. Here, the categorization criteria (invocation count), thresholds (1M and 100M), and forecaster set (AR and FFT) were selected for simplicity, aiming to convey key insights. We will show how a broader set of forecasters and more advanced classification can achieve superior performance (§4.3). FeMux is the first lifetime management system exploring this angle.

4.2.3 Not All Times Are the Same. Applications can experience different invocation phases [45]. Suitability of different forecasters can change over time. Fig. 9 illustrates this by comparing a fixed 5-minute keep-alive policy with a Markov Chain (MC) forecaster using a real trace (hash ending `..a427be`). The 5-minute keep-alive policy performs better initially with higher traffic variability, while MC learns to predict periodic traffic perfectly in the second hour.

> ***Implication 7**: The framework should be able to choose the most suitable forecaster for each epoch because application behavior evolves over time. This should be done with low overhead.*

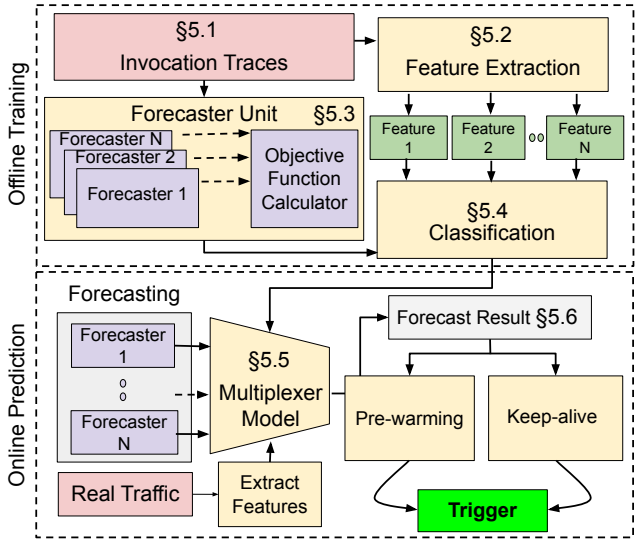

Figure 10. FeMux system overview.

4.3 FeMux

To address the design implications listed in prior sections, we present FeMux (Fig. 10), a serverless lifetime management framework. FeMux multiplexes between forecasters based on the current pattern of each application. This is done by periodically extracting latent features from application traces, which are used as input to a classification model that selects which lightweight forecaster is used to predict each application's traffic pattern. The classification model is trained *offline* on fleet-level traces. Providers already track traffic, execution time, and memory data [26, 45] needed for FeMux. By switching between forecasters as opposed to using an ensemble, FeMux only executes one lightweight forecaster at a time with an average inference time less than 7 ms (§5.2), while maintaining high accuracy.

4.3.1 Data Representation. Prior work commonly represents an application's past and forecasted activity using idle times [45, 47, 55] or the number of invocations per minute [40, 60]. Knative [30] has an *average concurrency* representation, closer to the latter. Since our service is Knative-based and we prototype FeMux on Knative, we use the same representation. It can be changed for other settings.

4.3.2 Feature Extraction. Invocation traces contain latent features useful for selecting the best forecaster. Statistical tests are typically used in time series forecasting to extract such features, including linearity, stationarity, and periodicity. We evaluate the use of standard statistical features that can be derived from publicly available traces. Nevertheless, FeMux's design allows for seamless integration of supplementary features.

Stationarity and ***linearity*** are key properties in time series forecasting, as many models assume one or both. Linearity implies a linear relationship between values, while stationarity means constant mean and variance. Identifying these traits aids in model selection. For instance, an AR [51] model assumes that a time series is stationary and linear; for piece-wise linear and non-stationary time series, Self-Excitation Threshold AutoRegressive (SETAR) [11] performs well [50]. We use the Broock-Dechert-Scheinkman test [7] and Augmented Dickey-Fuller test [15] to determine linearity and stationarity, respectively.

Periodicity is assessed by analyzing harmonics. A finite number of prominent harmonics indicates a periodic or quasiperiodic nature, depending on time series length. FFT extracts harmonics, which help determine periodicity and make predictions [26, 40]. Joosen et al. [26] have found that FFT outperforms all but one of the eight ML and statistical models they use for forecasting.

Density is the number of invocations within a time window. It serves not only to categorize applications by popularity (§4.2.2), but also as a proxy for forecasting complexity. When workloads do not contain discernible patterns, general trends are followed by Exponential Smoothing [21] forecaster, which forecasts based on a moving average, and Double Exponential Smoothing (Holt [10]), which uses a second moving average to account for trends in the data.

For feature calculation and analysis, we divide invocation data into *blocks*. We use static windows as block boundaries, avoiding the high cost of continuous analysis per application. While static block sizes may cause patterns to span boundaries, this trade-off simplifies implementation. Due to the Broock-Dechert-Scheinkman linearity test requiring a minimum block size of 400 data points, we set a block size of 504 minutes to balance performance and pattern detection (sensitivity study in Supplementary Material). Feature extraction takes <5 ms and is done once per block.

4.3.3 Forecaster Unit. Instead of using a single forecaster, FeMux's Forecaster Unit comprises a set of low-latency forecasters, with one selected online based on recent feature values and the trained model. Each forecaster must have low prediction overhead to minimize costs for the provider.

Forecasters. Our forecasters use two hours of past traffic and predict the incoming minute worth of traffic, both of which can be adjusted by providers. For stationary and linear patterns, we include AR, and include SETAR for non-stationary patterns that are piece-wise linear. For periodic traces we include FFT, and for dense traces include Exponential Smoothing and Holt models. We also implemented a Markov Chain forecaster [24, 29, 41] for applications with repetitive invocation patterns. The intention behind selecting these diverse forecasters is to showcase the framework's capability in adaptively choosing forecasters based on the features of each block. Providers can use their preferred set of forecasters and metrics of interest (using RUM).

During offline training, we simulate forecasts for 13k applications (§5.1) to tune forecast parameters based on RUM.

Using empirical results from a range of parameter values between 1 and 20, we pick 10 lags for AR and SETAR, with up to two thresholds for SETAR, and have FeMux use the top 10 harmonics from the FFT. Further, Exponential Smoothing and Holt have dynamic parameter selection, and our Markov Chain uses four states. Dynamically adapting forecaster hyperparameters to different RUMs is future work.

4.3.4 Classification. Any classification method can map block features to forecasters during offline training. Providers to re-train classification models with new data periodically (e.g., per month or quarter). Since we use Azure 19 traces for many of our experiments, we train once for our experiments given the two-week span of the trace. K-means clustering, run on a 16-core Intel Xeon E5 with 64 GB of memory, processes blocks from 13k applications in under 10 minutes, with negligible training overhead due to high amortization. To put it in perspective, each AWS Lambda function receives a monthly free tier of 400,000 GB-s of compute time [5], where functions receive CPU and memory with a fixed ratio of $\frac{1vCPU}{1.769GB}$ [4]. So, the above monthly training comes to less than 0.000738%[4] of the free-tier memory and less than 0.000327%[5] of the free-tier CPU allocation of each application. Even if a user deploys 1,000 applications, this overhead is less than 1% of their per-app free tier share.

Feature values are first standardized using a transformer (`StandardScaler`), removing the mean and scaling to unit variance. Then, FeMux uses K-means to group blocks with similar features. It then assigns each cluster to the forecaster that has the lowest RUM sum across all blocks within the cluster. The forecaster with the lowest RUM value across all blocks is set as the default forecaster, which FeMux uses when there is not enough invocation data to form a block. K-means clustering reduces RUM by over 15% compared to supervised models such as decision trees and random forests. Supervised models optimize for labelling the best forecaster for each block, but mislabelled blocks might assign forecasters that perform poorly. FeMux mitigates this by clustering similar blocks and assigning the best forecaster on average, improving tolerance to misclassification.

4.3.5 Resource Allocation. When an application has sufficient invocation history to form a new block, FeMux asynchronously extracts the features and uses the pre-trained classification model (§4.3.4) to select the forecaster for the next block. Traffic forecasts are used to control lifetime management parameters (§2.2). At the start of each interval, FeMux scales enough compute units to keep the predicted number alive. It also has two overriding rules, similar to prior work [20, 40, 45]: compute units are not preempted mid-execution, and any compute units that are provisioned due to a cold start are kept alive until the end of the interval.

[4] $[(10m \times 60\frac{s}{m} \times 64GB) \div 13k] \div 400,000GB.s = 0.000738\%$

[5] $[(10m \times 60\frac{s}{m} \times 16c) \div 13k] \div (400,000GB.s \times \frac{1c}{1.769GB}) = 0.000327\%$

4.3.6 Training Overhead. We observe that FeMux's training and inference overhead is suitable for large-scale production environments. For offline training, feature extraction from the Azure 2019 dataset (12.5B invocations) takes under 4 hours, while classifier training completes in under 10 minutes. For our dataset (1.9B invocations), feature extraction and classification take less than 30 and 5 minutes, respectively. Providers can retrain monthly or daily depending on QoS requirements, and retraining can be done incrementally by adding or replacing blocks.

5 Evaluation

Similar to prior work [20, 40, 45], we use simulations (§5.1) as our primary evaluation methodology. This allows to assess the effectiveness of FeMux at production scales for extended times. Such scale and duration are also critical to test the combination of various workloads and to press the design in infrequent scenarios.

FeMux itself is implemented in Python and Go, and fully automated once deployed. The classifier model is loaded in the forecasting pod. FeMux scales both horizontally (more pods), and vertically (larger pods) as required. We also prototype FeMux in Knative and evaluate the deployment (§5.2).

5.1 Simulation Results

To ensure fair comparison to prior work [20, 40, 45, 60], which all used the Azure '19 dataset [18], we use the same. It also allows us to demonstrate that FeMux's benefits are not dependent on our own dataset from IBM. The Azure '19 dataset includes daily average execution time and per-minute invocation counts for 14 days, and daily app-level memory consumption for 12 days—we use 12 days for evaluation. Applications consist of one or more functions, with app-level resource provisioning; each application instance executes on a separate compute unit, and at most one instance of each function runs on each compute unit [45].

We discard traces with NaNs for memory or execution duration, zero duration, or no invocations in the first 12 days, using the remaining 19k with a 70-30 train-test split. The training set is divided in half to form a train and validation set. Subtraces are generated by randomly sampling data across applications with three levels of traffic: over 100M invocations, between 1M and 100M invocations, and under 1M. We use the full 12-day trace, and uniformly distribute invocations within each minute similar to prior work [20, 45, 58]. To maintain realistic scaling limitations in our simulation, we use the limits set by AWS Lambda, which scales up no more than 500 instances per minute once there are over 3,000 [42].

Comparing designs built for different metrics is a common pitfall in prior work. Direct comparison of adaptive lifetime management systems can be skewed, as each system has its own assumptions on the metrics desired by users and

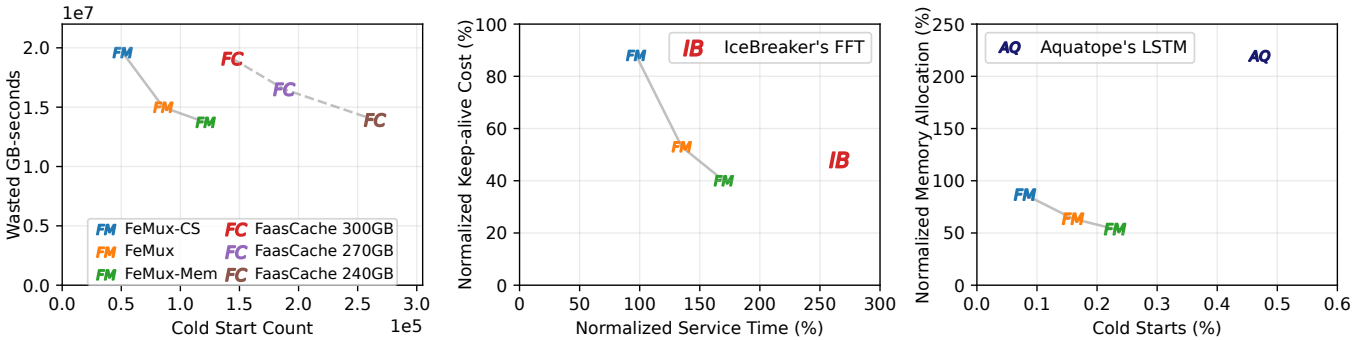

Figure 11. FeMux outperforms prior work based on their original metrics. For IceBreaker [40], service times include cold start durations and execution times, and keep-alive cost is based on total allocated memory (Middle). Aquatope uses aggregate cold start percentage and memory allocation normalized to a 10-minute keep-alive (Right).

providers (Table 2). For instance, striving to reduce overall cold start percentages [60] leads to optimizing for frequently invoked applications, whereas striving to improve the tail behavior of cold start percentages for different applications [45] requires efficient handling of rarely invoked applications. Neither of these strategies is obviously superior. The provider may choose either goal: reducing overall costs inherently favors popular applications, attracting new users favors considering all applications equally, or a combination.

To ensure fair representation of prior work, we 1) tune all baselines based on the same training dataset (§**??**); 2) simulate their performance using the same data representation as their experiments (e.g., invocations per minute for IceBreaker), and 3) adapt FeMux to use similar metrics as each prior work to ensure fair, apples-to-apples comparisons.

5.1.1 Comparison to Prior Work.

Comparing adaptive lifetime management systems is notoriously difficult, as each is optimized for a different set of performance and efficiency goals (Table 2). A direct comparison using a single, fixed metric would unfairly favor the system originally designed for it. To address this, for each state-of-the-art system, we use the RUM to optimize for the exact same metrics used in the original work, and in doing so we are able to get closer to an 'apples-to-apples' comparison. Our experiments demonstrate that FeMux shows greater adaptability.

We evaluate the benefits of having a lifetime management policy that adapts to traffic patterns by comparing FeMux and FaasCache, as FaasCache uses a fixed cache size. Further, we observe how forecaster switching leads to better predictions than using FFT by comparing with IceBreaker, and LSTM-based prediction networks by comparing with Aquatope. RUM's flexibility allows us to compare FeMux with each system using its respective metrics. FaasCache, Icebreaker, and Aquatope cannot be fairly compared with each other due to their different fixed objectives.

RUM enables optimizing the defined Pareto optimality of underlying metrics, and allows system optimization for a mutable metric. This means FeMux can be optimized for, and evaluated with any set of RUM definitions. For the sake of evaluation, we use four specific RUMs. Three are variants of the first RUM example (Eq. (1)) with: default RUM weights (Eq. (1)), 4x higher cold start weight (FeMux-CS), and 4x higher wasted memory weight (FeMux-Mem). The fourth one is FeMux with an execution-time aware RUM (FeMux-Exec) (Eq. (2)). The variants of FeMux (CS, Mem, Exec) have the same underlying system, but use different RUM definitions and weights for optimization.

FaasCache. One primary difference between FeMux and FaasCache is adaptability: FaasCache's fixed cache size is either (i) wasting resources from being too large, or (ii) too small and incurs many cold starts from not keeping enough compute units alive. Given the burstiness [28] and variability [26, 45] of serverless workloads, lifetime management systems must dynamically adapt to traffic patterns.

Fuerst and Sharma use both the number of cold starts (n) and the fraction of cold starts ($\frac{n}{N}$) to evaluate the cold start mitigation of FaasCache [20]. In any tested scenario, the number of invocations (N) is fixed, and essentially one metric is the scaled version of the other. We use the former for comparison. For the FaasCache results, we use their open source artifact [19]. While using the Azure traces, FaasCache performs function-level allocation, unlike Azure Functions' application-level allocation model [45]. This constraint in FaasCache's simulator limited us to testing to just single-function applications: 2,523 apps from our test dataset with >18M invocations.

All variants of FeMux are more Pareto optimal than FaasCache with different cache sizes (Fig. 11-Left). FeMux-CS reduces cold starts by over 64% compared to FaasCache with a 300 GB cache size, while wasting 3% more memory. Conversely, FeMux-Mem reduces the number of cold starts by over 54% compared to FaasCache with a cache size of 240 GB, while wasting 1% less memory. Further, FeMux reduces RUM by 30% relative to FaasCache with a cache size of 270 GB. Since cold start durations are fixed, minimizing cold start durations in the default RUM (Eq. (1)) translates to minimizing

cold start count, and wasted GB-seconds is already used by both FaasCache and in the default RUM.

IceBreaker's FFT. Roy et al. evaluate IceBreaker's adaptive lifetime policy using service times and keep-alive costs normalized to a 10-minute keep-alive policy [40]. Service times are the sum of wait times, cold start durations, and execution times; keep-alive costs are measured in dollars based on the wasted GB-second. We assume homogenous resources to evaluate only the adaptive lifetime policy of IceBreaker. FeMux-Mem and IceBreaker incur 40% and 48% of the keep-alive cost when compared to a 10-minute-KA policy, but IceBreaker increases service times by >266% while FeMux-Mem creates a 170% increase (Fig. 11-Middle). FeMux reduces the RUM by 42%, when Icebreaker's metrics are translated into RUM.

IceBreaker's reliance on a single FFT forecaster struggles with both low-traffic applications (where it often forecasts zero) and high-traffic applications with highly variable patterns. This highlights the fundamental limitation of a single-forecaster approach, which FeMux overcomes by dynamically multiplexing between models (e.g., Markov Chains for periodic traffic, Exponential Smoothing for trends) that are better suited for the workload's current behavior.

Aquatope's LSTM. We use Aquatope's open-source artifact with default parameters [59], which trains the LSTM and prediction models independently for each application, uses a 48-minute input window, and the first 7 days of each test trace as training data with the remaining 5 days for testing. Similar to prior work [20, 40, 45], we report training and inference statistics based on CPU computation. Training for Aquatope takes 4x longer than FeMux on the same hardware, and is application-specific; inference times range from 109-308 ms – almost 28x slower than FeMux's.

Aquatope uses aggregate cold start percentage and memory allocation normalized to a 10-minute keep-alive for evaluation. Fig. 11-Right shows that Aquatope allocates 114% more memory than a 10-minute keep-alive and incurs 3.1M cold starts (0.47%), while all variants of FeMux have fewer cold starts and less memory allocation. Further, for RUM, which considers total cold start duration and wasted memory, default FeMux provides a 78% reduction compared to Aquatope. Note that aggregate metrics such as cold start percentage are skewed to be small as high-traffic applications have many live containers, which reduces their fraction of cold starts but incurs additional memory wastage. Despite training a separate LSTM model for each application, Aquatope's complex models adapt too slowly to the bursty, high-frequency traffic patterns common in serverless workloads. FeMux's use of an agile classification model to switch between lightweight forecasters provides better accuracy with significantly lower training (4x faster) and inference (28x faster) overhead.

5.1.2 Supporting Different RUMs Simultaneously.

With FeMux providers can offer differentiated tiers of service through simultaneous use of multiple RUMs in the same platform. To demonstrate this, we consider a case where 10% of applications are *premium (P)*, representing users willing to pay more for better performance, and the remaining 90% are *regular (R)*. Fig. 12 shows that selective use of FeMux-CS (Blue) for premium applications (P_{Blue}) reduces cold start seconds of premium applications by 45%. Compared to a single-objective deployment, which treats all apps equally to reduce cold starts for just 10% of them, the tiered approach cuts memory wastage by 35.4%: (WastedMem($P_{Blue} + R_{Orange}$) = 64.6% ∗ WastedMem($P_{Blue} + R_{Blue}$)).

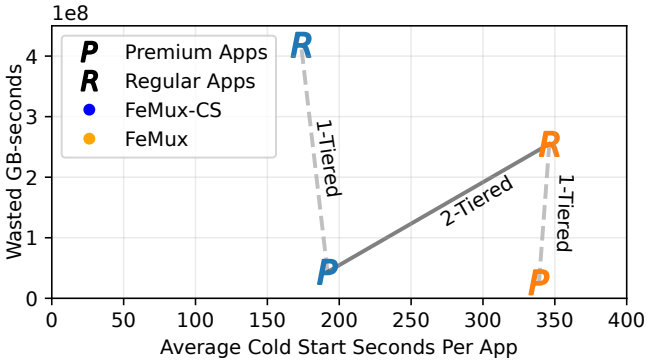

Figure 12. FeMux simplifies building multi-tiered services. This flexibility comes from enforcing flexible application-level RUMs.

5.1.3 Different RUM Definitions.

Apart from evaluating with different weights for the default RUM formulation, we also optimize FeMux with the execution time RUM (Eq. (2)). One key difference between the RUMs is that cold start mitigation for the default RUM is application agnostic, but for the execution time RUM the impact of cold starts is measured relative to the execution time of each application. We train an execution time aware FeMux (FeMux-exec) by using the execution time RUM as the optimization metric and adding a feature capturing the execution duration of the application that each block is extracted from.

By weighing all cold starts equally, FeMux (trained on default RUM) incurs 33% less cold start seconds than FeMux-Exec and reduces the default RUM by 7%. Conversely, by incurring cold starts for applications that have low cold start impact due to longer execution times, FeMux-Exec reduces wasted memory by 25% relative to FeMux and achieves a 19% reduction in execution time RUM.

5.1.4 Sensitivity Studies.

We also 1) compare the performance of individual forecasters with their multiplexed combination, 2) compare different combintations of features, and 3) explore the effect of block size (Appendix C). *These results are included in the Supplementary Material file submitted.* While they may be of interest to some readers, they are peripheral to our main technical contributions.

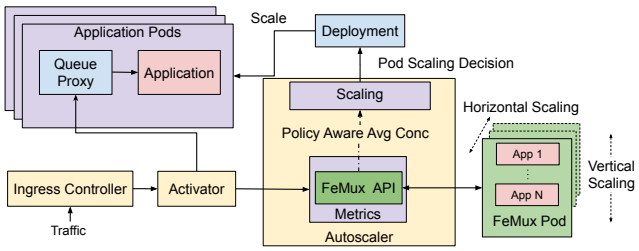

Figure 13. Integration of FeMux in *Knative Serving*.

5.2 Knative Prototype

To validate FeMux in a real-world system, we integrate it into the Knative Serving component as a scalable microservice that intercepts the concurrency metrics flowing to Knative's default Autoscaler (Fig. 13). FeMux's REST API provides a predictive scaling target back to the Autoscaler, effectively overriding Knative's default reactive logic with FeMux's proactive, forecast-based decisions—requiring minimal changes to the core Knative components.

In the original configuration, Knative's Serving component (Fig. 13) routes client traffic via an ingress. The Activator buffers invocations for under-scaled applications, and Autoscaler manages pod scaling using concurrency data. Application pods pair a user container (code execution) with a queue-proxy that provides metrics (every 2s) to the Autoscaler and handles response egress to the ingress. Its default lifetime policy is a 1-minute KA.

All FeMux forecasters are implemented in Python and reside within a pod, while passive policies (e.g., 10-min KA) are directly implemented within the Autoscaler. Each application has a dedicated FeMux instance (a thread in the FeMux pod), as lifetime management happens at the application level. FeMux pods can scale vertically by adding more concurrent threads, or horizontally by scaling up additional FeMux pods. We deploy a horizontal pod scaler to manage scaling FeMux pods, and use `etcd` to persist threads' states.

Each second, the metrics collector forwards the average invocation concurrency to the FeMux API. To match the Azure data and simulations, the FeMux API batches per-second average concurrency to obtain per-minute estimates, which means (i) the forecasts from the FeMux pods are based on per-minute data, and (ii) the FeMux API's scaling decisions are maintained for one minute.

The FeMux API uses application-specific REST endpoints to route minute-scale average concurrency to the respective forecasting thread(s) which reside in FeMux pods. The FeMux pod then outputs the forecasted concurrency which the FeMux API uses to make a scaling decision and communicates to the Autoscaler. When sufficient minutes are captured to complete a new block, application threads asynchronously cluster the newly completed block and start using the assigned forecaster for future forecasts.

Setup. Our evaluation setup consists of a Knative-serving deployment on top of a Kubernetes cluster with 1 controller node and 9 worker nodes. The controller node has 8 cores and 30 GB of memory and worker nodes have 4 cores and 15 GB of memory each, totaling 44 cores and 165 GB of memory. For comparison, clusters from prior work had 38-48 cores and 76-256 GB of memory [20, 40, 45].

Workload. Our workload for the 10-node Knative deployment is composed of 100 randomly sampled applications sourced from the testing portion of our simulation workload. With over 232,000 invocations in a 24-hour period and a peak CPU utilization below 70%, the number of applications, experiment duration, and traffic characteristics follow practices from prior work [45, 60] and production clusters [46, 57]. To maintain representativity [52], we ensure that the distribution of invocations across applications from our Knative workload follows full Azure 2019 dataset's (Fig. 14-Left). Ideally, our work and prior work could access production scale clusters to support large workloads, but this limitation is why simulated studies are the standard for large scale experiments. We replay traces using FaaSProfiler [43, 44], where each invocation executes a Go function that allocates memory and busy waits as defined by the trace.

FeMux Gains. FeMux reduces RUM by 36% compared to Knative's default policy (Fig. 14-MiddleRight). Further, FeMux reduces application cold start percentages by >50% for over 25% of applications (Fig. 14-MiddleLeft), and maintains (within 2%) or improves the cold start percentage while maintaining modest resource wastage, as indicated by the aggregate RUM reduction. For this evaluation, we use the same FeMux trained on the default RUM weights as in simulation, and observe that the RUM values from our simulation are within 13% of deployment.

Knative uses container concurrency, unlike invocation count used by prior work [20, 40, 45, 60] on OpenWhisk. Knative's serving architecture makes scaling decisions every two seconds, and requests are queued in pods and central Autoscaler. Conversely, OpenWhisk sends requests to invokers, which allocate containers to serve requests (no concurrency by default). As prior systems are built upon very different architecture and data representations, we only compare to them in simulation.

Scalability Study. A single FeMux pod with 1 vCPU maintains a mean forecasting latency of 7 ms and a p99 latency of 25 ms, when serving 20 forecasting requests per second. Since application forecasts are performed once per minute, this maps to supporting 1,200 applications per forecasting pod. Our experiment shows that FeMux pods scale out gracefully to accomodate forecasting demands for more applications (Fig. 14-Right). Further, clustering occurs once per block (e.g., 504 minutes) and takes under 10ms.

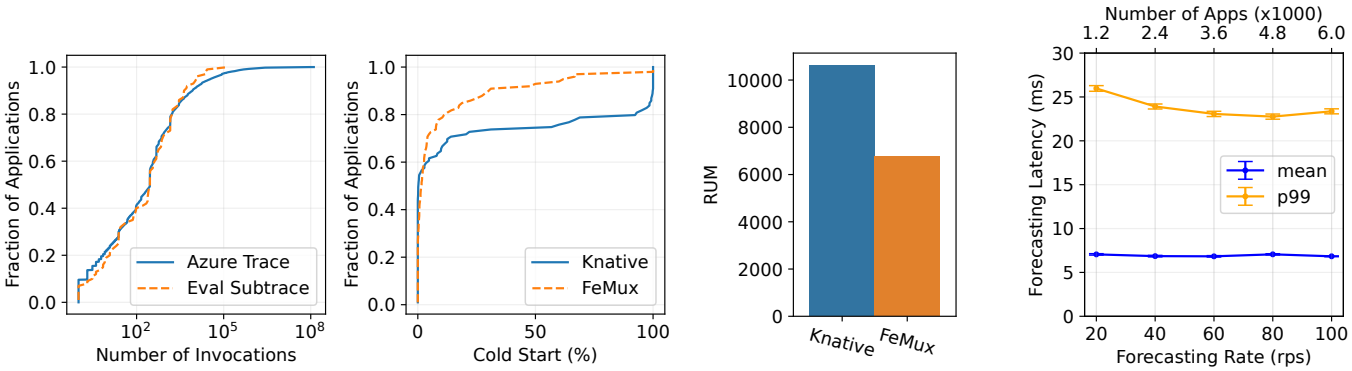

Figure 14. (Left) Distribution of our evaluation subtrace follows the full Azure workload. (Mid-left) FeMux reduces the fraction of cold starts incurred across >40% of applications compared to the Knative baseline. (Mid-right) This enhancement considers memory efficiency, as shown in the reduction of aggregate RUM. (Right) The FeMux pods scale horizontally to accommodate more serverless applications with low forecasting latency.

6 Related Work

Objectives. Combining and parameterizing objectives has been proposed in prior work for scheduling or resource scaling policies [23, 39, 53, 56], but we propose RUM as an abstract metric for designing multi-component systems (e.g., forecasters, forecaster switching model) that adapt to changeable metrics and aggregation techniques.

Adaptive lifetime management. Shahrad et al. [45] consider the idle time between invocations, and deploy a histogram policy for most traces alongside an Autoregressive Integrated Moving Average (ARIMA) [49] for functions that cannot be represented by a histogram. INFLess [55] extends the hybrid histogram policy by introducing separate histograms for hourly and daily timespans, and uses a greedy scheduling algorithm for matching functions to resources. FaasCache [20] instead models the serverless resource management as a caching problem and uses a greedy heuristic to decide which functions are kept alive. Rather than using low-cost heuristics, Barista [6] and Incendio [8] respectively use pre-trained neural networks and clustered reinforcement learning for scheduling functions to compute units. In the context of serverless databases, Diao et al. [14] have recently evaluated resource scaling using an ensemble of forecasting and machine learning models, but inference costs are above our threshold (e.g., >1 s), and scaling is targeted at capacity planning as opposed to individual containers.

Recent work has integrated heterogeneity, dynamic resource allocation, and layered-scaling. IceBreaker [40] introduces heterogeneity within compute units to reduce the overhead of the keep-alive policy, and reduces the number of cold starts using an FFT forecaster. Instead of splitting the resource pool due to heterogeneity, Flame [54] splits a homogenous resource pool into several smaller clusters, and shows the benefit of a centralized cache controller. AQUATOPE [60] uses Bayesian optimization for dynamically allocating resources when scheduling functions, and forecasts future invocations using LSTM encoding and a pre-trained neural network model. Further, RainbowCake [56] assumes containers can be scaled between bare-metal, runtime environment, and user-application layers. By using a keep-alive heuristic based on a parameterized combination of startup cost and memory waste, scaling layers of containers shows efficiency and service time improvements. We compare with FaasCache, IceBreaker, and Aquatope as they are the best performing lifetime management policies that have ms-scale latencies without assuming split resource pools or layered-scaling.

7 Conclusion

We have provided the first large-scale dataset and characterization of a production serverless platform outside Azure and Huawei, and the first public trace from an open-source platform that includes concurrency and minimum scale settings. Using our insights, we introduce a new serverless lifetime management system, FeMux, aimed at increasing the flexibility and quality of resource management for providers.

Acknowledgments

We thank the anonymous reviewers, and specially our shepherd, Yue Cheng, for helping us improve the paper. We also thank Matei Ripeanu and Hassan Halawa for their valuable feedback on this work. This work was supported by IBM CAS Canada Fellowship (project #1139), a British Columbia Graduate Scholarship (BCGS) award, and the Natural Sciences and Engineering Research Council of Canada (NSERC). Computational resources from the Digital Research Alliance of Canada (RAS and RAC allocations), IBM Cloud, AWS Cloud Credit for Research Program, and Oracle for Research Cloud Starter Award facilitated this project.

References

[1] AAG. 2025. The Latest Cloud Computing Statistics. https://aag-it.com/the-latest-cloud-computing-statistics/.

[2] 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)*. USENIX Association, Santa Clara, CA, 419–434. https://www.usenix.org/conference/nsdi20/presentation/agache

[3] Amazon. 2025. Configuring reserved concurrency. https://docs.aws.amazon.com/lambda/latest/dg/configuration-concurrency.html.

[4] AWS. Accessed on 2025-09-16.. Lambda quotas. https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html

[5] AWS. Accessed on 2025-5-14.. AWS Lambda Pricing. https://aws.amazon.com/lambda/pricing/.

[6] Anirban Bhattacharjee, Ajay Dev Chhokra, Zhuangwei Kang, Hongyang Sun, Aniruddha Gokhale, and Gabor Karsai. 2019. Barista: Efficient and scalable serverless serving system for deep learning prediction services. In *2019 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 23–33.

[7] William A Broock, José Alexandre Scheinkman, W Davis Dechert, and Blake LeBaron. 1996. A test for independence based on the correlation dimension. *Econometric reviews* 15, 3 (1996), 197–235.

[8] Xinquan Cai, Qianlong Sang, Chuang Hu, Yili Gong, Kun Suo, Xiaobo Zhou, and Dazhao Cheng. 2024. Incendio: Priority-based Scheduling for Alleviating Cold Start in Serverless Computing. *IEEE Trans. Comput.* (2024), 1–14.

[9] Alex Casalboni. 2019. AWS Lambda Power Tuning. Retrieved 2024-05-08 from https://github.com/alexcasalboni/aws-lambda-power-tuning

[10] Chris Chatfield and Mohammad Yar. 1988. Holt-Winters forecasting: some practical issues. *Journal of the Royal Statistical Society Series D: The Statistician* 37, 2 (1988), 129–140.

[11] Michael P Clements and Jeremy Smith. 1997. The performance of alternative forecasting methods for SETAR models. *International Journal of Forecasting* 13, 4 (1997), 463–475.

[12] IBM Cloud. Accessed on 2025-5-14.. IBM Code Engine. https://www.ibm.com/products/code-engine.

[13] Datadog. Accessed on 2024-10-3.. The State of Serverless 2021. https://www.datadoghq.com/state-of-serverless-2021/.

[14] Yanlei Diao, Dominik Horn, Andreas Kipf, Oleksandr Shchur, Ines Benito, Wenjian Dong, Davide Pagano, Pascal Pfeil, Vikram Nathan, Balakrishnan Narayanaswamy, and Tim Kraska. 2024. Forecasting Algorithms for Intelligent Resource Scaling: An Experimental Analysis. In *Proceedings of the 2024 ACM Symposium on Cloud Computing (SoCC '24)*. ACM, 126–143.

[15] David A Dickey and Wayne A Fuller. 1979. Distribution of the estimators for autoregressive time series with a unit root. *Journal of the American statistical association* 74, 366a (1979), 427–431.

[16] Mathew Duggan. 2023. Serverless Functions Post-Mortem. (2023). https://matduggan.com/serverless-functions-post-mortem/.

[17] Simon Eismann, Long Bui, Johannes Grohmann, Cristina Abad, Nikolas Herbst, and Samuel Kounev. 2021. Sizeless: Predicting the Optimal Size of Serverless Functions. In *Proceedings of the 22nd International Middleware Conference*. ACM, 248–259.

[18] Rodrigo Fonseca. Accessed on 2025-5-14.. Azure Functions Invocation Trace 2019. https://github.com/Azure/AzurePublicDataset/blob/master/AzureFunctionsDataset2019.md.

[19] Alex Fuerst. Accessed on 2025-5-14.. Faascache. https://github.com/aFuerst/faascache-sim.

[20] 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.

[21] Everette S Gardner Jr. 1985. Exponential smoothing: The state of the art. *Journal of Forecasting* 4, 1 (1985), 1–28.

[22] Google. Accessed on 2025-5-14.. Google Cloud Run. https://cloud.google.com/run.

[23] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. 2014. Multi-resource packing for cluster schedulers. *SIGCOMM Comput. Commun. Rev.* 44, 4 (Aug. 2014), 455–466.

[24] James D Hamilton. 1996. Specification testing in Markov-switching time-series models. *Journal of econometrics* 70, 1 (1996), 127–157.

[25] IBM. 2025. Configuring application scaling. https://cloud.ibm.com/docs/codeengine?topic=codeengine-app-scale.

[26] Artjom Joosen, Ahmed Hassan, Martin Asenov, Rajkarn Singh, Luke Darlow, Jianfeng Wang, and Adam Barker. 2023. How Does It Function? Characterizing Long-Term Trends in Production Serverless Workloads. In *Proceedings of the 2023 ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '23)*. ACM, 443–458.

[27] Artjom Joosen, Ahmed Hassan, Martin Asenov, Rajkarn Singh, Luke Darlow, Jianfeng Wang, Qiwen Deng, and Adam Barker. 2025. Serverless Cold Starts and Where to Find Them. In *Proceedings of the Twentieth European Conference on Computer Systems* (Rotterdam, Netherlands) *(EuroSys '25)*. ACM, 938–953.

[28] Kostis Kaffes, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2022. Hermod: Principled and Practical Scheduling for Serverless Functions. In *Proceedings of the 13th Symposium on Cloud Computing* (San Francisco, California) *(SoCC '22)*. ACM, 289–305.

[29] In Kee Kim, Wei Wang, Yanjun Qi, and Marty Humphrey. 2022. Forecasting Cloud Application Workloads With CloudInsight for Predictive Resource Management. *IEEE Transactions on Cloud Computing* 10, 3 (2022), 1848–1863.

[30] Knative. Accessed on 2025-5-14.. Knative. https://knative.dev/docs/.

[31] Qingyuan Liu, Dong Du, Yubin Xia, Ping Zhang, and Haibo Chen. 2023. The Gap Between Serverless Research and Real-World Systems. In *Proceedings of the 2023 ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '23)*. ACM, 475–485.

[32] Gilad Maayan. Accessed on 2025-05-11.. 5 Ways to Manage Lambda Cold Starts. https://khalilstemmler.com/blogs/serverless/5-ways-to-manage-lambda-cold-starts/.

[33] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. 2022. ORION and the Three Rights: Sizing, Bundling, and Prewarming for Serverless DAGs. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 303–320.

[34] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Eshaan Minocha, Sameh Elnikety, Saurabh Bagchi, and Somali Chaterji. 2022. WISEFUSE: Workload Characterization and DAG Transformation for Serverless Workflows. *Proc. ACM Meas. Anal. Comput. Syst.* 6, 2, Article 26 (jun 2022), 28 pages.

[35] Ricardo P Masini, Marcelo C Medeiros, and Eduardo F Mendes. 2023. Machine learning advances for time series forecasting. *Journal of economic surveys* 37, 1 (2023), 76–111.

[36] Arshia Moghimi, Joe Hattori, Alexander Li, Mehdi Ben Chikha, and Mohammad Shahrad. 2023. Parrotfish: Parametric Regression for Optimizing Serverless Functions. In *Proceedings of the 2023 ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '23)*. ACM, 177–192.

[37] Kim Long Ngo, Joydeep Mukherjee, Zhen Ming Jiang, and Marin Litoiu. 2022. Has Your FaaS Application Been Decommissioned Yet? – A Case Study on the Idle Timeout in Function as a Service Infrastructure. arXiv:2203.10227 [cs.DC]

[38] Openwhisk. Accessed on 2025-5-14.. Open Source Serverless Cloud Platform. https://openwhisk.apache.org/.

[39] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R. Ganger, and Eric P. Xing. 2021. Pollux: Co-adaptive Cluster Scheduling for Goodput-Optimized Deep Learning. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 1–18. https://www.usenix.org/conference/osdi21/presentation/qiao

[40] 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.

[41] Ghazal Sadeghian, Mohamed Elsakhawy, Mohanna Shahrad, Joe Hattori, and Mohammad Shahrad. 2023. UnFaaSener: Latency and Cost Aware Offloading of Functions from Serverless Platforms. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, Boston, MA, 879–896.

[42] Amazon Web Services. 2025. Lambda Function Scaling. https://docs.aws.amazon.com/lambda/latest/dg/lambda-concurrency.html.

[43] Mohammad Shahrad. Accessed on 2025-5-14.. FaaSProfiler. https://github.com/PrincetonUniversity/faas-profiler.

[44] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. 2019. Architectural implications of function-as-a-service computing. In *Proceedings of the 52nd annual IEEE/ACM international symposium on microarchitecture*. 1063–1075.

[45] Mohammad Shahrad, Rodrigo Fonseca, Íñigo 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.

[46] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. 2018. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 69–87.

[47] Jiacheng Shen, Tianyi Yang, Yuxin Su, Yangfan Zhou, and Michael R. Lyu. 2021. Defuse: A Dependency-Guided Function Scheduler to Mitigate Cold Starts on FaaS Platforms. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*. 194–204.

[48] Mikhail Shilkov. 2025. Comparison of Cold Starts in Serverless Functions across AWS, Azure, and GCP. https://mikhail.io/serverless/coldstarts/big3/.

[49] Robert H Shumway, David S Stoffer, Robert H Shumway, and David S Stoffer. 2017. ARIMA models. *Time series analysis and its applications: with R examples* (2017), 75–163.

[50] George C Tiao and Ruey S Tsay. 1994. Some advances in non-linear and adaptive modelling in time-series. *Journal of forecasting* 13, 2 (1994), 109–131.

[51] George Udny Yule. 1927. On a method of investigating periodicities in disturbed series, with special reference to Wolfer's sunspot numbers. *Philosophical Transactions of the Royal Society of London Series A* 226 (1927), 267–298.

[52] Dmitrii Ustiugov, Dohyun Park, Lazar Cvetković, Mihajlo Djokic, Hongyu Hè, Boris Grot, and Ana Klimovic. 2023. Enabling In-Vitro Serverless Systems Research. In *Proceedings of the 4th Workshop on Resource Disaggregation and Serverless* (Koblenz, Germany) *(WORDS '23)*. ACM, 1–7.

[53] Abhishek Verma, Madhukar Korupolu, and John Wilkes. 2014. Evaluating job packing in warehouse-scale computing. In *2014 IEEE International Conference on Cluster Computing (CLUSTER)*. 48–56. doi:10.1109/CLUSTER.2014.6968735

[54] Yanan Yang, Laiping Zhao, Yiming Li, Shihao Wu, Yuechan Hao, Yuchi Ma, and Keqiu Li. 2024. Flame: A Centralized Cache Controller for Serverless Computing *(ASPLOS '23)*. ACM, 153–168.

[55] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. 2022. INFless: A Native Serverless System for Low-Latency, High-Throughput Inference *(ASPLOS '22)*. ACM, 768–781.

[56] Hanfei Yu, Rohan Basu Roy, Christian Fontenot, Devesh Tiwari, Jian Li, Hong Zhang, Hao Wang, and Seung-Jong Park. 2024. RainbowCake: Mitigating Cold-starts in Serverless with Layer-wise Container Caching and Sharing *(ASPLOS '24)*. ACM, New York, NY, USA, 335–350. doi:10.1145/3617232.3624871

[57] Yanqi Zhang, Íñigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. 2021. Faster and Cheaper Serverless Computing on Harvested Resources. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. ACM, 724–739.

[58] Ming Zhao, Kritshekhar Jha, and Sungho Hong. 2023. GPU-enabled Function-as-a-Service for Machine Learning Inference. In *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 918–928.

[59] ZhuangZhaung Zhou. 2025. aquatope. https://github.com/zzhou612/aquatope.

[60] Zhuangzhuang Zhou, Yanqi Zhang, and Christina Delimitrou. 2022. 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*. 1–14.

A Artifact Appendix

A.1 Abstract

Our artifact supplies the scripts and source code to reproduce simulated FeMux results based on the Azure 2019 dataset in our paper, including metric comparison results, forecaster switching analysis (Fig. 8), FeMux performance (Fig. 11), and FeMux with different RUMs (Fig. 12). While all FeMux simulation results can be reproduced by configuring our artifact, we configure our artifact and highlight the steps to reproducing the key steps for brevity.

A.2 Description & Requirements

A.2.1 How to access. All relevant source code, scripts, and datasets alongside instructions are available on the public Git repository (https://github.com/ubc-cirrus-lab/femux) and Zenodo (https://doi.org/10.5281/zenodo.17180431). By following the repository steps, users will pull and clean the Azure 2019 dataset, transform to average concurrency, simulate forecasts, train the FeMux clustering, and simulate the cold starts and wasted memory. With all of the results generated, users can run our provided plotting scripts to generate the plots and results outlined below.

A.2.2 Hardware dependencies. Further, we recommend configuring our scripts to run on 48 cores and 140GB of Memory to minimize runtime, with 150GB of disk space available. We provide time estimates based on 16 threads, where thread count can be updated globally by changing $num\_workers = 16$ or by changing the parameter at the bottom of each parallelizable script.

A.2.3 Software dependencies. Ubuntu 20.04 or newer and Python +3.10 with the dependencies defined in *requirements.txt*. We also note that users with more than 128 cores need to set the *OPENBLAS_NUM_THREADS* environment variable to a number less than 128.

A.2.4 Benchmarks. None.

A.3 Set-up

After cloning the repository, we make some of the basic directories and download the azure 2019 dataset by running *setup.sh*. We make two adjustments to the source code to save significant time on reproduction. First, we comment out Holt, Exponential Smoothing, and SETAR to significantly cut down forecasting simulation runtime which is the bottleneck for reproduction. Results do not change significantly as these forecasters are selected for under 5 percent of blocks. We also configure FFT to run on application-level traces which affects some of the final results. For our FeMux prototype, we use function-level traces for FFT and the IceBreaker comparison. This can be seen in the initialization function of *forecasting_sim.py*.

A.4 Evaluation workflow

A.4.1 Major Claims.

- *(C1): AR is superior for 65.2% of applications when assessed using MAE, but FFT outperforms AR for 68.9% of apps for RUM Section 4.2.1.*
- *(C2): Using the right forecaster per class reduces RUM (Fig. 8).*
- *(C3): FeMux outperforms IceBreaker (Fig. 11).*
- *(C4): FeMux-CS reduces cold start seconds of premium applications by 45% relative to FeMux, and the tiered approach cuts memory wastage by 35.4% (Fig. 12).*

A.4.2 Experiments. In the preprocessing steps, we start with the Azure 2019 dataset and clean, format, and transform the data for our simulations. Then in the offline simulation steps we generate the forecasts and extract features required for clustering and evaluation. Finally, we have FeMux results which can be used to generate results for our claims in the experiments that follow. All estimated times are based on a 16-core setup. Our time estimates exclude person-time as each step is fully automated apart from a parameter defining the number of cores.

Preprocessing (24 Compute Hours).

1. Make some of the basic directories and download the azure 2019 dataset: */code/setup.sh*
2. Preprocess the data and generate separate dataframes for execution times, invocation counts, and application memory: each of the files in */preprocess*, starting with *preprocess_data.py*.
3. Convert invocations per minute into average concurrency (24h): *transform_azure.py*.
4. Generate training and testing split for applications: *gen_train_test_split.py*

Offline Simulation (130 Compute Hours).

1. Forecasting Simulations (~120 h): *forecasting/forecast.py*
2. Extracting Features (~4 h): *extract/feature_extraction.py*

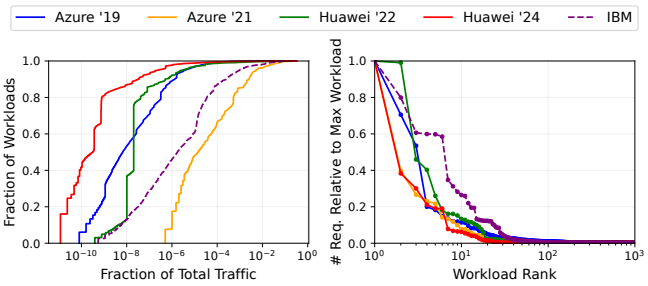

Figure 15. (Left) Our dataset has a smaller proportion of low-traffic workloads compared to prior datasets relative to the average daily invocations, while 63% of Huawei workloads are timer-based. Consequently, (Right) our dataset has the most workloads with 10% of more traffic volume relative to the maximum workload—we only show the top 1000 workloads.

3. Simulate the cold start and memory wastage based on the simulated forecasts (~4 h): *results/gen_results.py*
4. Generate MAE values based on simulated forecasts for comparing RUM and MAE ((~20 m)): *results/maes.py*
5. FeMux simulation: *clustering_pipeline.py* combines the clustering, and simulated performance of the FeMux prototype configured with the named features and forecasters that have already been extracted/simulated above (~1 h).

Plotting Preparation (5 Compute Minutes).

1. */plots/setup.sh* copies generated results from */data/azure* to */plots/data*, and additionally unzips the pre-generated data we provide.
2. Plotters are in */plots/plots/plotters* and should be run within the directory. Each plotter has its corresponding plot output to a pdf file in */output_plots*.

Experiments. With all data generated and moved into */plots/data*, plotting each result takes less than 5 minutes between human intervention and compute time. Steps for generating all plots are the same, so we will describe a single repeatable experiment that generates results for all claims.

Experiment: [Generate Figures]: Results for each claim (*C1-C4*) are generated using the following steps.

[Execution] Run each file in */plots/plotters/*. Plotter names summarize the result and section in the paper. We include *sec_4_switching.py* for completeness, and do not reproduce the FeMux result for the FaasCache comparison in the prior work comparison.

[Results] View the figure associated with each plotter in */plots/output_plots*.

B Additional Insights From Our Dataset

B.1 Cross-Workload Request Variations

Here, we show how the traffic patterns of various workloads in our dataset differ from those in previous datasets. We

avoid comparing absolute traffic numbers, as each dataset represents an unknown fraction of the total traffic served by a different provider. The traffic volumes are higher in our trace for mid-popularity workloads compared to past data based on IAT distributions. This skew is better captured in Fig. 15-Right. IBM has over 30 workloads accounting for 10% or more of the traffic generated by the most popular trace, while Huawei 2022, Azure 2019, Azure 2021, and Huawei 2024 have 18, 12, 10, and 7 respectively. Fig. 15-Left shows the CDF of fraction of total traffic attributed to different workloads. Azure 2021 has the highest normalized traffic across workloads, as the overall volume was 2M requests across 14 days. Among large-scale datasets, with >25M daily invocations, IBM workloads make up the largest fraction of total traffic; our median trace has two to four orders of magnitude higher relative traffic volume compared with the median trace from Azure 2019 and Huawei datasets. We also observe a large difference in traffic volume across regions within the same provider: regions from the Huawei 2024 dataset have total traffic volume ranging from 125M to 39.3B.

Fig. 15-Left also reveals that workloads from most datasets are variable in traffic volume. Huawei's datasets are an exception, showing significant traffic similarities across a large portion of workloads, as evidenced by prominent vertical jumps in the CDF curves. This cross-workload variability affects the choice of models for traffic prediction (considering overhead and delay budget), the scheduling assumptions (regarding scheduling delay), and the aggressiveness of resource reclamation.

B.2 Benefits of Long Traces

Examining individual workload patterns can provide insights into the load variation phases that workloads experienced over extended periods. For instance, workload *A* showcases daily and weekly periodicity, with an increasing trend in traffic throughout in January, before maintaining a weekly average in February. Further, the New Year's Day and spanning the first two weeks of January, workload *B* experiences several hourly peaks between 75k-100k requests per hour, before the traffic returns to the standard hourly peaks of 25k-50k requests per hour (Fig. 16-Bottom). For either of these workloads, a month-long or two-week-long trace would not provide a holistic view of the trends and seasonality of these workloads. Understanding these trends across workloads is valuable for pursuing antagonistic colocation strategies and avoiding simplistic assumptions. Additionally, using long-term and rich traces allows for realistic stress testing of utilization-enhancing techniques.

C Sensitivity Studies

FeMux vs. Individual Forecasters. Fig. 17 compares the performance of individual forecasters with their multiplexed

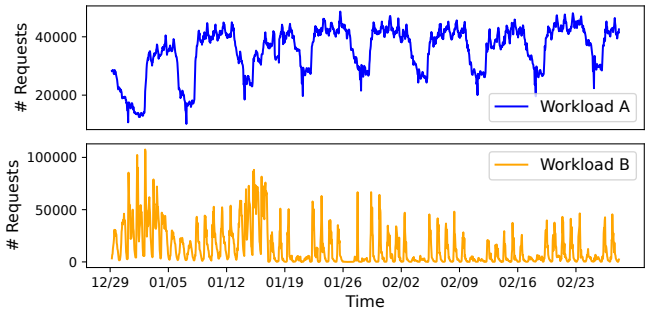

Figure 16. We select two example workloads that showcase the benefits of long traces: given a trace spanning many weeks, we can observe seasonal increases in traffic during early January (Top), and a slowly increasing average load trend in requests over two months (Bottom).

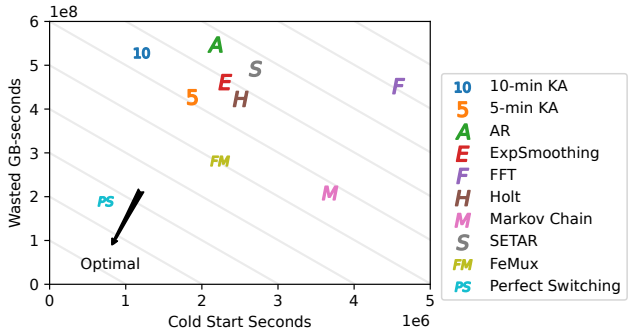

Figure 17. FeMux outperforms individual forecasters included in its set of forecasters.

combination (FeMux). The contours represent lines of constant RUM based on the default weights (Eq. (1)). Some forecasters (e.g., fixed keep-alive or AR) are conservative, minimizing cold starts but increasing wastage. Some (e.g., Exponential Smoothing or Markov Chain) reduce wastage at the cost of more cold starts. FeMux automatically multiplexing across the most suitable forecaster per application. Here, it switched between forecasters for over 65% of applications, with 20% switching between 4 or more different forecasters. The system is flexible, allowing providers to integrate any forecasters they find suitable.

Features. We compare different combinations of features in Fig. 18. First, we see that adding more features improves results by providing differentiation power to the classifier. However, this yields diminishing returns. Secondly, the performance of a set number of features can vary based on the selected features—all combinations that include harmonics perform better. Third, complementary features seem to work better than top-performing features; Density+Harmonics performs better than Stationarity+Harmonics, despite the fact that Stationarity and Harmonics are two best single features. The features chosen in this work are to showcase

the benefits of classification-based forecaster multiplexing. FeMux works with any other feature(s).

Block Size. After evaluating block sizes between 7-24 hours, we observe that increasing the block size reduces FeMux's RUM slightly (<3%). This is because a larger block size helps capture larger patterns which benefits certain forecasters, like Holt which removes trends. However, long block sizes result in slower adaptation to changing application patterns, as a new forecaster can only be selected at block boundaries. To hit a balance, we chose a block size of 504 minutes. The specific figure ensures an integer number of blocks over the 14-day Azure trace (40 blocks).

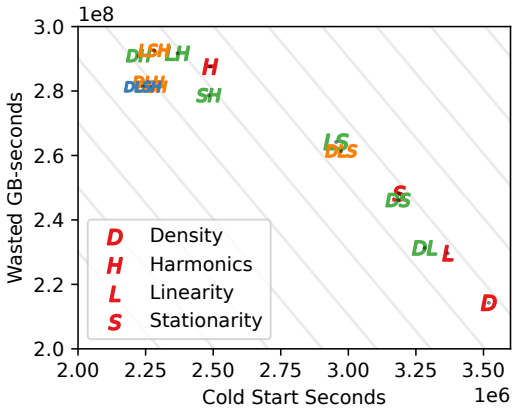

Figure 18. Combining statistical features boosts classifier accuracy by enhancing differentiation. Combinations with same-size share similar colors.