ALAP: Intent-Based Serverless Computing via Delayed Decision-Making

Prasoon Sinha[1] Kostis Kaffes[2] Neeraja J. Yadwadkar[1]
[1]The University of Texas at Austin [2]Columbia University

Abstract

The lack of an interface that allows users to express their intent, to trade off latency versus cost, continues to be a hurdle in the adoption of serverless computing for latency-critical and cost-constrained applications. Existing systems shy away from providing a user-intent knob mainly because they make resource allocation decisions as soon as a function is registered, when inputs to the function are mostly unavailable. We find that function performance and resource utilization can vary greatly (up to 6× in latency and 4.78× in utilization) across inputs. Thus, to enable intent-based serverless computing, our key insight is to delay making resource allocation decisions until after function inputs are available so that resources can be allocated independently for each input and resource type. We introduce ALAP, a resource management framework for serverless systems that makes decisions *As Late As Possible* to right-size each invocation and meet user-intent efficiently. ALAP uses an online learning agent to predict the required amount of resources to meet an invocation's constraints and schedules these right-sized containers in a cold-start-aware manner. For a range of functions and inputs, ALAP adapts to user intents: it reduces latency and cost by up to 17.5× (3× on avg.) and 13× (3.7× on avg.), respectively, and latency and cost SLO violations by 1.3-2.3× and 1.6-2.2×, respectively, while nearly eliminating CPU and memory waste compared to four state-of-the-art systems.

CCS Concepts

• **Computer systems organization → Cloud computing**; • **Computing methodologies → Machine learning**.

Keywords

Serverless computing, Resource management, Online learning

ACM Reference Format:
Prasoon Sinha[1] Kostis Kaffes[2] Neeraja J. Yadwadkar[1]. 2025. ALAP: Intent-Based Serverless Computing via Delayed Decision-Making. In *ACM Symposium on Cloud Computing (SoCC '25), November 19–21, 2025, Online, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3772052.3772262

1 Introduction

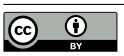
Serverless computing or Function as a Service (FaaS) manages nearly all system administration tasks, including resource management, simplifying cloud usage for programmers [7, 13, 19, 34, 35, 39]. Users simply upload their code while the providers manage resources and autoscaling on behalf of the users. However, commercial platforms, including AWS Lambda [7] and Google Cloud Functions [19], require users to specify a memory limit for each function, forcing them to reverse-engineer the implications of their specification on latency and cost. Most prior work in serverless computing attempts to optimize either function latency [2–4, 17, 25, 32, 41, 55, 58, 69, 70, 78, 82, 85, 88, 92, 97, 111] or costs [12, 26, 67, 83, 102], while some allow users to specify desired latency constraints [14, 16, 52, 112]. However, *none provide an interface for users to express their intent to trade off latency versus cost, as their rigid resource allocation policies cannot adapt to different user intents.* We find that the user intent dictates the optimal resource allocation for a function (Section 2, Figure 1). Thus, providing users with an intent-based interface is imperative for enabling a unified serverless system that works for a range of applications, from latency-critical to cost-constrained functions.

Existing systems shy away from providing a user-intent knob mainly because they make resource allocation decisions as soon as a function is registered, when inputs to the function are mostly unavailable. However, we observe in our characterization study (Section 2, Figure 3) that performance and resource utilization vary greatly across inputs for a function (e.g., 4.78× difference in the number of cores used between two unique video inputs). Thus, it is not feasible to determine how a user-specified intent can be met until function invocation time when the inputs are available.

Our work. To enable intent-based serverless computing, we argue that serverless systems should make resource allocation decisions *as late as possible* to account for the needs of each function invocation.

We characterize serverless functions (Section 2) from several benchmark suites [21, 45, 96] commonly used in research [3, 4, 30, 52, 54, 67, 69, 109]. As Azure [84] and Meta [80] report executing both short-lived (100ms-10s, 70% of functions) and long-running functions (>10s, 20-40% of functions), we study and optimize for both types. These functions include scientific applications, image/video processing, ML training and inference, and web services. We find that function semantics and input properties, beyond just size, greatly affect function execution time and resource utilization. Our observations lead to the following design principles for building an *intent-based serverless system* that meets user's intents while improving resource efficiency for the providers: (a) delay allocation decisions until inputs are available, (b) account for function semantics, and (c) make independent allocations per resource type (cores and memory). Section 2 details our measurement study.

Based on these design principles, we introduce ALAP, a serverless computing platform that makes resource allocation decisions **A**s **L**ate **A**s **P**ossible to account for the needs of each function invocation. ALAP introduces an **interface** that enables users to communicate their intent (e.g., minimize cost or latency, or meet a specific cost or latency target). To account for inputs, **ALAP's Allocator** uses *readily-available* metadata of the function inputs as features



SoCC '25, Online, USA

ACM ISBN 979-8-4007-2276-9/2025/11
https://doi.org/10.1145/3772052.3772262

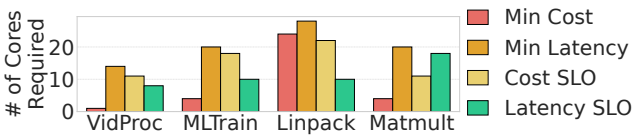


Figure 1: The user intent dictates the optimal number of cores to allocate. See § 2.

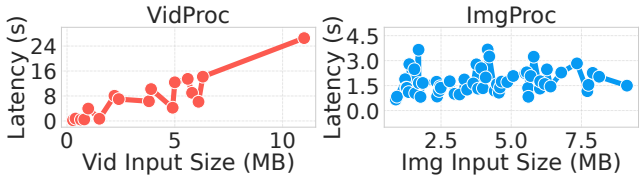


Figure 2: Contrary to the assumption of previous works, many common serverless functions do not exhibit linear relationships between size and execution time. See § 2.

to an online learning agent that predicts the required resource configuration (cores and memory) to satisfy the user's intent. Since delayed decision-making may increase cold starts as invocations to the same function may need differently-sized containers, we build **ALAP's Scheduler** to complement per-invocation resource allocations. If the desired size is unavailable, ALAP's scheduler uses a larger warm container and pre-launches the right-sized one in the background for future use.

We summarize our contributions as follows:

- Our measurement study, using a wide range of short- and long-running serverless functions, with varying thread-level parallelism and input types, shows that the user intent and function inputs dictate the optimal resource allocation.
- We introduce ALAP, the first serverless system that makes both input-aware and independent resource allocation decisions per resource type to greatly reduce resource waste while meeting diverse user intents.
- We implement ALAP atop OpenWhisk and show that with minimal overheads, ALAP can reduce latency by up to 17.5× (3× avg), cost by 13× (3.7× avg), or latency and cost service-level objective (SLO) violations by 1.3-2.3× and 1.6-2.2×, respectively, with minimal resource waste compared to several state-of-the-art systems, including Parrotfish [67], Aquatope [112], Cypress [14], and Bilal et al. [16].

2 Characterization & Design Principles

We study how different user intents influence the optimal resource allocation and the shortcomings of allocation policies used by existing serverless systems. We study 11 functions (Table 4, § 8), both short-lived and long-running (> 10s) [80, 84], including scientific applications, image/video processing, ML training/inference, and web services, commonly seen in literature and benchmark suites [14, 16, 21, 41, 45, 52, 67]. We use hundreds of unique inputs from open source datasets [24, 33, 36, 61, 64, 72]. We observe the functions on Apache OpenWhisk [29] with core allocations 1-28 (§ 8 describes our experimental setup). Our observations lead to four design principles for building a serverless system that adapts to the user's needs while improving the provider's resource utilization.

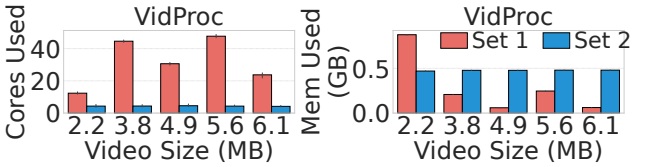


Figure 3: Resources used by two sets of inputs to VidProc. Both sets contain inputs of the same size, but different unique inputs. Properties other than size impact utilization (e.g., video resolution). See § 2.

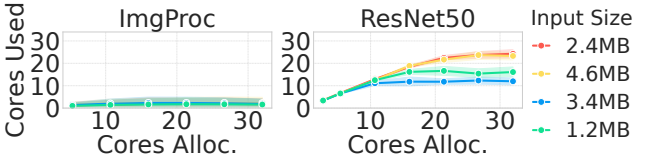


Figure 4: Core allocated versus used for two functions given the same inputs. Function semantics affect resource utilization. See § 2.

1. Support various user intents. Users have different needs, from meeting strict cost or latency SLOs to requiring best-effort allocations that minimize cost or latency. However, existing serverless systems do not expose a flexible interface to allow users to communicate their intent; instead, they internally optimize for either cost [12, 26, 67, 83] or latency [14, 38, 52, 102, 112]. Figure 1 shows that the intent dictates the required resource configuration (results for four functions with fixed inputs). *As no single resource configuration minimizes cost/latency or meets SLOs, serverless systems need an interface where users can express their intent and provide "hints" to the platform as to what allocation best suits the function.*

2. Delay allocation decisions until inputs are available. Commercial platforms [7, 13, 19] and recent works [16, 38, 52, 67, 69, 102, 110, 112] make input-agnostic allocation decisions. Cypress [14] uses the input's size to predict execution time. However, it assumes (a) the relationship between input size and function execution time is linear, and (b) properties beyond size do not affect function performance and resource utilization. However, we observe that for simple, common serverless functions, like video and image processing, there is no clear linear relationship between input size and execution time (Figure 2). Instead, input properties beyond just size greatly affect resource utilization for several functions. For example, Figure 3 shows that input videos of the same size (e.g., 5.6MB) differ by 4.78× in the number of cores used by a video processing function (converts an RGB video to grayscale [21, 45]). Videos in set-1 exhibit a seemingly unpredictable relationship between input size and cores used, while set-2 exhibits constant utilization regardless of the video size. We found that all videos in set-2 had the same resolution (1280 × 720), whereas resolution varied for set-1 videos, causing the wide variation in utilization. As properties beyond size (e.g., bit rate, resolution) affect performance and resource utilization, allocations that ignore inputs are sub-optimal. *Instead, resource managers should delay allocation decisions until inputs are available to be holistically input-aware.*

3. Account for function semantics. Serverless systems host functions with varying semantics, including differences in thread-level parallelism and arbitrary internal state/branching. Thus, input-aware allocations alone are insufficient, as identical inputs can demand vastly different resources across functions. For example, while both ImgProc and ResNet50 ingest the same images, ImgProc (single-threaded) uses only one core, whereas ResNet50 (multi-threaded) can use tens of cores to reduce execution time (Figure 4). Accounting for function semantics is imperative to make informed allocation decisions. However, learning the impact of a function's semantics on performance and cost is challenging without expensive profiling or statistical formal methods [100]. *We need lightweight techniques to account for the relationship between function semantics and resource allocations that meet user intents.*

4. Make independent allocations per resource type. Commercial systems [7, 19], open-source frameworks [29, 62], and several previous works [14, 38, 52, 67, 67, 69, 71, 88, 102, 110] couple resource types, as they assume serverless functions are both memory- and compute-intensive. We find this degrades resource utilization, like previous works [16, 112]. Figure 3 shows that VidProc can use 19 cores (set-1, 5.6 MB), but the function uses at most 0.8 GB of its memory. Systems that couple resource types would allocate > 10GB memory [7] with 19 cores, resulting in > 92% memory underutilization. Instead, in addition to being input-aware, *serverless systems should make independent allocations per resource type.*

3 Existing Approaches

We exhaustively review the limitations of the interfaces and resource management policies of previous systems. In Table 1, we detail the differences across the most closely related work to ALAP. No previous serverless system incorporates the four key design principles detailed in § 2. To the best of our knowledge, we are the first to build a serverless framework that makes both input-aware and decoupled resource allocation decisions to greatly reduce resource waste and meet diverse user intents.

Existing user interfaces. Commercial providers (AWS λ [7], Google Functions [19]), open-source (OpenWhisk [29] and OpenFaaS [62]), and several proposed systems (Cypress [14], Golgi [52], Kraken [15], OFC [69], EcoLife [38]) require users to specify memory limits for their functions. This forces users to reverse engineer the implications of their choices on performance and cost. Some systems allow users to specify a latency SLO (Aquatope [112], Golgi [52], Cypress [14], EcoFaaS [88], StepConf [102], GSight [110]), while others do not allow users to provide SLO (OFC [69], Parrotfish [67], EcoLife [38]). However, none expose a flexible interface for users to trade off latency versus cost in different forms. Bilal et al. [16] discuss potential interfaces for users to express their intent, but this work is an analysis of existing commercial platforms [13], not a prototyped serverless system. To the best of our knowledge, ALAP *is the first serverless system to expose and support a flexible interface that allows users to express different intents: a cost or latency SLO, or minimize cost or latency.*

Existing resource management approaches. To meet diverse user intents without wasting resources, ALAP makes input-aware, decoupled resource allocation decisions. Parrotfish [67] and AWS Lambda's Power Tuning [12] are offline profiling tools that select a memory limit for serverless functions. However, these works assume coupled resource types and pessimistically over-allocate resources with their input-agnostic allocations, leading to resource underutilization (Figures 8, 9). We show these inefficiencies in our comparison against Parrotfish (§ 9). EcoLife [38] adjusts container placement across heterogeneous hardware to reduce carbon emissions. GSight [110] makes scheduling decisions to reduce partial interference between serverless functions. StepConf [102] allocates a function's memory limit to improve SLO compliance. FaaSConf [99] deploys heavy-weight reinforcement machine learning agents per application to improve SLO compliance. Freyr [106] uses reinforcement learning to harvest idle resources. OFC [69] leverages spare memory across functions to dynamically scale in-memory caches, reducing external data retrieval time. However, all these systems end up wasting resources and degrade function performance as they make input-agnostic, coupled resource allocation decisions. Cypress [14], Golgi [52], and Kraken [15] pack multiple invocations into the same container. However, since these systems assume coupled resources, they degrade utilization under sparse arrival patterns. Moreover, Golgi and Kraken are input-agnostic. Cypress, along with ChunkFunc [74], assumes that input properties beyond size do not affect performance or resource utilization. We compare against Cypress in (§ 9) to show the inefficiencies of bin-packing and not being holistically input-aware (§ 2). EcoFaaS [88] reduces the energy footprint of serverless platforms with CPU frequency scaling, however, it couples resource types. Finally, Aquatope [112] and Bilal et al. [16] use Bayesian optimization to make decoupled allocation decisions. However, we show ALAP achieves superior SLO compliance and resource utilization than these systems with its lightweight modeling and input-aware decisions in § 9.

Other serverless optimizations. Several systems optimize other aspects of serverless computing: reducing cold-start overheads [2–4, 17, 25, 32, 47, 53, 55, 58, 59, 70, 78, 81, 82, 85, 91, 92, 97, 101, 103, 104, 111], efficient function-to-function communication [1, 3, 46, 48, 53, 56, 57, 63, 65, 73, 79, 89, 90, 95, 101, 102], and state management [5, 37, 42, 44, 49, 51, 63, 66, 69, 75, 77, 107, 108]. These optimizations are orthogonal to and compatible with ALAP.

Study	Intents Supported	Input Aware	Decoupled Resources
OFC [69]	Min. Latency	✗	✗
Bilal et al. [16]	Min. Latency or Cost	✗	✓
Aquatope [112]	Latency SLO	✗	✓
Golgi [52]	Latency SLO	✗	✗
Cypress [14]	Latency SLO	✗	✗
Parrotfish [67]	Min. Cost	✗	✗
EcoFaaS [88]	Latency SLO	✓	✗
StepConf [102]	Latency SLO	✗	✗
FaaSConf [99]	Latency SLO	✗	✗
Freyr [106]	Latency SLO	✗	✗
ChunkFunc [74]	Latency SLO	✗	✗
EcoLife [38]	Min. Latency	✗	✗
GSight [110]	Latency SLO	✗	✗
ALAP (our work)	Min. Latency or Cost, Latency or Cost SLO	✓	✓

Table 1: Summary of the interface and allocation policies of previous serverless systems. See § 3.

API	Parameters
`register_fxn`	fxnName, fxn, intent <minCost, minLat, latSLO, costSLO>
`register_fxn`	fxnName, fxn, cpuLimit, memLimit
`invoke_fxn`	fxnName, input(s)

Table 2: ALAP's declarative, intent-based API. See § 4.1.

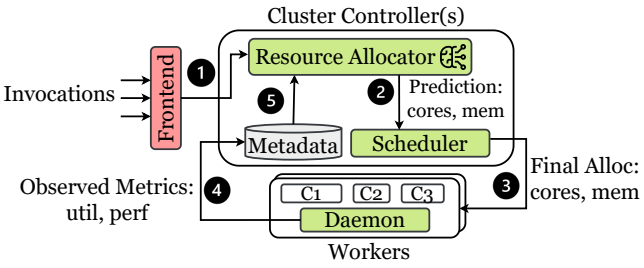


Figure 5: ALAP's workflow. See § 4.

4 ALAP Design

We leverage our design principles (§ 2) to build ALAP, a serverless system that dynamically allocates resources per function invocation *As Late As Possible* to meet diverse user-intents while improving resource efficiency for cloud providers.

4.1 ALAP's Intent-Based Interface

The interface of existing serverless systems is inflexible, forcing developers to specify either (1) a memory limit [7, 19, 29, 62] or (2) a latency SLO [14, 79, 88, 102, 112]. While (1) forces developers to reverse-engineer resource limits for their latency/cost needs, (2) continues to burden developers by forcing them to determine appropriate latency SLOs. Alternatively, platforms intrinsically optimize for latency [16, 38, 69] or cost [16, 67] without giving users control over their latency or cost requirements. ALAP is the first flexible intent-based serverless computing platform. Table 2 outlines ALAP's intent-based interface. Developers register functions via `register_fxn`, specifying the function name, serialized function, and optimization intent (row 1): minimize cost or latency, or meet a cost or latency SLO. Offering the intent to minimize latency/cost relieves developers from determining appropriate SLO values, however, ALAP also supports meeting SLOs, both latency *and* cost. Cost SLOs allow users to maximize performance (e.g., latency) within specified budget constraints without reverse engineering latency SLOs. Additionally, developers wanting direct resource control can specify cores and/or memory allocations for their function (row 2). Finally, users can execute functions via `invoke_fxn` (row 3), providing the function name and input(s) directly in the payload.

4.2 ALAP's Intent-Based Architecture

Figure 5 describes ALAP's workflow. ALAP's Frontend receives invocations specifying the function, input(s), and optionally, a user intent. ALAP defaults to minimizing cost if no intent is specified. ❶ The Resource Allocator receives invocations from ALAP's Frontend. ❷ To make input-aware allocation decisions, the Allocator extracts *readily-available* metadata (e.g., image resolution) specific to the input type in the payload (§ 5.2) and feeds this metadata as features to its function-specific online agents (implicitly accounting for function semantics, § 5.1) to predict the required resource allocation for the invocation's intent. ❸ The Scheduler uses this prediction to finalize the invocation's allocation. It prioritizes dispatching the invocation to a warm container of the exact or larger size than that specified by the prediction to avoid cold starts (§ 6). ❹ On each worker, ALAP deploys a lightweight daemon to collect performance and resource utilization metrics per invocation. When an invocation completes, the daemon sends this data to ❺ ALAP's Allocator to update its model and learn the evolving relationship between allocated resources and function cost/performance (§ 5.2).

Scalability of ALAP. We host ALAP's Allocator and Scheduler on the same server to enable efficient communication between them (ALAP's Scheduler consults with the online agents managed by the Allocator). Previous serverless systems scale their logical scheduler with physical replicas, each managing a sub-cluster of workers [17, 80, 84, 86]. To minimize data movement (e.g., container images), a hierarchy of load balancers routes invocations of the same function to the same sub-cluster, exploiting caching and locality. ALAP achieves scalability in a similar manner. We deploy a physical replica of the Allocator over each physical scheduler, ensuring each Allocator replica hosts online learning agents for only the functions executed in its sub-cluster. This ensures ALAP's Allocator is not a single point of contention.

ALAP's Scope. In this work, we target single-function applications, commonly used in recent works [3, 14, 16, 21, 41, 52, 67, 80, 105] and a high use case on commercial platforms (60% of applications on AWS Lambda contain only one function [23]). Previous works [48, 56, 102, 112] optimizing multi-function workflows are complementary to ALAP. Importantly, our proposed design principles generalize to workflows. Intermediate functions receive inputs derived from preceding functions, which can be used to guide resource allocation decisions. Asynchronous, non-blocking workflows [11] are naturally tolerant of ALAP's minimal per-invocation overheads (see § 9.6). Synchronous workflows require further optimization, however, our initial study opens opportunities to optimize more complex, workflow-based applications in future work.

5 Delayed Decisions for Intent-Based Allocation

ALAP's Resource Allocator delays allocation decisions until function inputs are available. It uses online learning to predict the resource configuration (core count, memory) that will meet the user intent.

Why use online machine learning? § 2 highlights the complexity of making accurate resource allocations. Input properties beyond just size (e.g., video resolution) affect resource utilization and performance. Hence, to make tight resource allocations that minimize resource waste, serverless systems need a mechanism that accounts for the relationship between function inputs, resources allocated, and function performance (e.g., latency, cost). To add to the complexity, this relationship needs to be understood per function, since functions with identical inputs can greatly differ in performance and utilization due to differences in function semantics. While we could devise a heuristic-based algorithm, this heuristic still needs to model the relationship between function inputs, resource allocations, and performance in some capacity, whether it be analytically or via ML. Moreover, the heuristic would require extensive manual tuning to be robust and generalize to all functions and edge cases. Instead, we choose to use lightweight ML algorithms to learn this relationship and predict the required resource configuration per

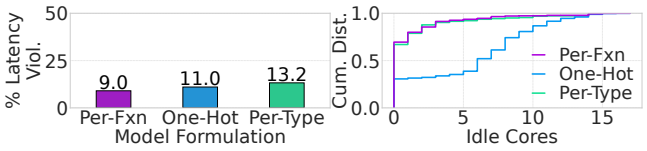


Figure 6: Design exploration of ALAP's online resource allocation predictor. A model per function provides the best SLO compliance and utilization. See § 5.1.

invocation in a data-driven manner. We further design ALAP's Resource Allocator to use *online* ML for five reasons. (1) Online ML enables our agent to observe and adapt to the dynamically changing runtime environments. (2) Online ML removes the overhead of extensive offline training. (3) Representative function inputs may not be available offline to train accurate models. (4) Models trained offline are susceptible to data drifts if the distribution of inputs, functions, or SLO requirements changes over time. (5) Training ML models offline may not generalize to new, previously unseen functions and inputs. We devise and compare against a heuristic-based algorithm that uses lightweight profiling per function in § 9.

5.1 Accounting for Function Semantics

Previous works use machine learning to make resource allocation decisions by building a model per function [14, 67, 88, 99, 112]. Ideally, we could build a single model that makes resource allocation decisions across all functions. However, it is unclear how to provide function semantics as features to the model to enable it to learn the differences between functions. In this section, we explore the efficacy of building such a model and compare it with building a model per function.

One model across functions. Learning one model across functions requires extracting meaningful features to distinguish between functions. For feasibility, we use simple function-level features: lines of code, function calls, libraries used, and external API calls. To generate same-length feature vectors for a model, we need to standardize features across functions, which is challenging since the input type dictates the number of features (Table 3). We evaluate (1) *one-hot encoding*, which concatenates function vectors and zeroes out irrelevant parts, and (2) *per input-type* models, eliminating the need for standardization. We also considered embeddings, which encode variable-sized vectors into a common-dimensional latent space. However, embeddings require learning a custom ML transformation engine per function [68], defeating the purpose.

A single model (one-hot encoding) lowers SLO violations but wastes resources (Figure 6): it cannot learn each function's requirements, allocating 9-13 cores to all functions. A model per input type reduces waste, but increases violations: ResNet50's violations grow, as the multi-threaded function consistently receives 1-2 cores. Because ImgProc completes faster, the common agent between the two functions first learns from the single-threaded ImgProc invocations. If ResNet50 dominated initial learning, violations would drop, but ImgProc would waste resources with unnecessarily larger allocations (> 2 cores). As neither approach optimizes SLO attainment *and* resource utilization, we explore per-function models.

One model per function. A model per function eliminates the challenges of creating a model across functions. Each function's model can ingest its own feature vector size dictated by input type.

Input Type	Extracted Metadata
video	w×h, duration, bit-rate, frame-rate, size
image	w×h, # channels, DPI, size
audio	# channels, duration, sample/bit-rate
matrix	# rows, # cols, density
string	string length
json	size, # objects
csv	size, # rows, # cols
text file	size

Table 3: Metadata extracted per input type. § 5.2.

Moreover, function-level features are the same across invocations and provide no meaningful information to the model, obviating the need to model function semantics. Instead, per-function models customize to functions with differing semantics (e.g., ImgProc vs. ResNet50) by inherently capturing semantics through observation, reducing SLO violations without resource waste (Figure 6). Ultimately, like prior work [14, 67, 88, 112], we design ALAP to build a model per function.

5.2 ALAP's Online Learning Model Formulation

We build a lightweight online model per function to predict the number of cores and amount of memory to allocate each invocation that satisfies the user intent. ALAP builds separate agents per resource type to make independent decisions. We present the formulation that the online learning agents for both resource types use, followed by details specific to core and memory predictions.

Inputs and outputs of online agent. ALAP's agents ingest an invocation's input and user intent. If the intent is to minimize latency or cost, ALAP sets a low SLO value (§ 5.3). The output is a predicted core count and memory amount that will meet the intent without wasting resources.

Using *readily-available* input metadata as features. To enable ALAP's agent to make input-aware allocations, we construct input-specific feature vectors consisting of *readily-available* metadata commonly used to describe input types. Using readily available metadata ensures fast and resource-efficient vector construction (§ 9.6). Moreover, we do not analyze the contents of the input as this is invasive and expensive to obtain. Table 3 lists the metadata ALAP extracts for common input types seen in commercial serverless applications [10] and research [3, 4, 14, 16, 45, 52, 67, 69, 96, 105]. This metadata helps the agent learn input features affecting performance, cost, and resource utilization. ALAP infers the input type from the file extension if not specified. If ALAP has not seen the input type (e.g., XML), developers can specify the input's descriptive features during function registration; otherwise, ALAP defaults to input size as the feature. ALAP concatenates the input's metadata to construct its feature vector for model prediction. We assume ALAP can access input metadata since inputs are embedded in an invocation's payload.

To support featurization of inputs stored in external datastores (e.g., S3), ALAP can leverage the datastore's existing API to retrieve already processed input metadata. For example, S3's HeadObject API [9] exposes object size via the *Content-Length* field. We propose extending the metadata object to include a few more lightweight metadata for common input types (Table 3), adding only a few bytes per object. This extension allows ALAP to retrieve necessary

features directly from S3 without fetching the full object, eliminating network overhead. ALAP's improvements in performance and utilization (§ 9) justify the minor changes to existing datastore APIs. However, even without the extension, ALAP can still exploit existing metadata to make input-size-aware, decoupled resource allocation decisions.

Online learning algorithm. We formulate the resource allocation prediction as a classification problem to match the incremental allocation offerings of commercial serverless systems [8, 20]. While we could use regression, continuous allocations inflate the number of unique container sizes for ALAP to provision, rendering it impractical. Underpredictions are more detrimental than overpredictions—an insufficient number of cores allocated may lead to SLO violations, while insufficient memory allocations triggers out-of-memory (OOM) invocation termination. To differentiate between the severity of different allocations, we use a *cost-sensitive multi-class classification* (CSMCC) algorithm to make resource allocation decisions, described next.

Let $C_r = \{c_1, c_2, \ldots, c_{N_r}\}$ denote the discrete set of allocation classes for a given resource type $r \in \{\text{CPU}, \text{Memory}\}$, where N_r is the number of supported configurations for that resource. For example, each CPU class corresponds to a core count (e.g., class c_4 = 4 cores), and each memory class represents a fixed 64 MB increment (e.g., class c_2 = 128 MB). Given a feature vector $\mathbf{x}$ representing the metadata vector of an invocation's input, our goal is to select the class $\hat{c}_r \in C_r$ that minimizes the expected cost of allocating the resources associated with the class[1]. To do so, we train a small linear regressor $f_c(\mathbf{x})$ for each class $c \in C_r$ to estimate the cost of allocating the resources associated with the class to the invocation. We select the class with the lowest cost as the invocation's allocation:

$$\hat{c}_r = \arg\min_{c \in C_r} f_c(\mathbf{x}).$$

This approach facilitates easy cost differentiation between allocations with lightweight, fast predictions (§ 9.6).

ALAP bootstraps its online agent by first observing invocations with intentionally oversized allocations, helping it learn the relationship between inputs and resource needs. A confidence threshold (§ 9.5) determines how many invocations to observe before switching to predictive allocations. ALAP updates its agent online using a cost function based on observed performance (latency or cost) and resource utilization, detailed per resource type next.

Online updates to the core count prediction agent. After a retired invocation, we update our agent using the invocation's observed performance and resource consumption. We rank the classes (core count) to inform the agent of suitable allocations for the invocation's input and intent. We first determine the best class (§ 5.3) and assign the class a cost of one. Then, the costs of the remaining classes grow linearly based on their distance from the class deemed best. We penalize underpredictions further compared to overpredictions using an asymmetric scaling factor $\alpha > 1$. Formally, for a given class c and the best class c^*, the cost is defined as:

$$\text{assignCost}(c) = \begin{cases} 1, & \text{if } c = c^* \\ 1 + \alpha \cdot (c^* - c), & \text{if } c < c^* \quad \text{(underprediction)} \\ 1 + (c - c^*), & \text{if } c > c^* \quad \text{(overprediction).} \end{cases}$$

The best-ranked class guides the agent between exploiting its current allocation (i.e., the core count the invocation received is the best class) or exploring new ones. In § 5.3, we describe how we assign the lowest cost to adapt to user intent.

Algorithm 1: ALAP's logic for updating its online agent with the best core allocation.

```
Function determineBestCoreAlloc():
    if latency SLO then
        if SLO is met then
            bestCoreAlloc = maxCoresUsed - ⌊(SLO − execTime) / (α · SLO)⌋
        else if p90Util > 90% then
            bestCoreAlloc = maxCoresUsed + ⌈(execTime − SLO) / (β · SLO)⌉
        else
            bestCoreAlloc = maxCoresUsed
    else if cost SLO then
        if SLO is met and p50Util > 90% then
            bestCoreAlloc = maxCoresUsed + ⌊(SLO − execCost) / (α · SLO)⌋
        else if SLO is not met and p50Util > 90% then
            bestCoreAlloc = maxCoresUsed + ⌈(execCost − SLO) / (β · SLO)⌉
        else
            bestCoreAlloc = maxCoresUsed - ⌈(execCost − SLO) / (β · slo)⌉
```

Online updates to the memory prediction agent. Unlike core allocations, fluctuating memory does not affect latency since traditionally, serverless lacks swap space [7, 13, 19, 29]. We simply allocate enough memory to prevent invocations from being terminated by the Linux OOM daemon. This allocation is a fixed user cost. Therefore, our algorithm to update the online agent is much simpler than core predictions: we rank the best class as the one corresponding to the observed memory utilization, without exploring limits based on execution time or cost. Similar to core count predictions, we penalize underpredictions more heavily than overpredictions.

We reduce the risk of OOM exceptions with two safeguards. (1) We set the confidence threshold for memory predictions to 2× the core threshold. This allows the model to learn from a diverse range of inputs, as the execution of both smaller and larger inputs completes within the threshold. (2) We ensure the predicted memory amount exceeds the maximum memory used by the function previously plus the input's size; if not, we default to this value with some additional memory for slack. This is rarely invoked, however, with these safeguards, we greatly reduce invocations killed due to OOM exceptions: previous works produce 3-8% [69, 112], whereas ALAP only has 0.47% OOM errors (§ 9.5).

5.3 Aligning with Various User Intents

We next describe how we determine the best class (core allocation) for a retired invocation. While latency is a monotonically decreasing function of the number of cores allocated, a function's execution cost is not, since it depends on both latency and resource usage. Thus, the optimal allocation varies with user intent (§ 2).

[1]The "cost" associated with a class is different than the price paid by a user of the serverless platform. We assign costs to each class after invocations retire using a cost function. This cost function is synonymous with a reward function used in traditional reinforcement learning.

We therefore construct separate algorithms for latency and cost constraints, both described in Algorithm 1.

Adapting to latency constraints. We first describe our algorithm with a user-specified latency SLO (lines 2-6, Algorithm 1). If the SLO is met, we inform the agent to explore lowering the core allocation size to reduce the cost of meeting the SLO (lines 3-4). When the SLO is not met, we inform the agent to increase the allocation size, *only if* the invocation had high utilization (> 90%), indicating that it likely requires more cores to meet SLOs (lines 5-6). Otherwise, we attribute the violation to factors beyond ALAP's scope (e.g., environment noise, unrealistic SLO) and inform the agent to lower the allocation to the maximum number of cores the invocation used (lines 7-8). The slack (difference between execution time and SLO) determines how much we update the allocated number of cores (lines 3-6). We explored two techniques for how to use slack. (1) *Absolute*: for every α% the latency is below the SLO, decrease the core count by one (line 4); for every β% above, increase by one (line 6), and (2) *Proportional*: increase or decrease the cores by the ratio of an invocation's SLO to execution time. Figure 7a shows SLO violations under both methods. We tune α to 15% and β to 5% (§ 9.5 presents a sensitivity analysis). *Absolute* reduces SLO violations with slightly more aggressive adjustments, enabling the agent to quickly learn by observing the efficacy of different allocations in meeting SLOs. Though its p95 CPU waste is one core higher, it is a modest tradeoff for better SLO adherence.

ALAP also uses this algorithm when the intent is to minimize latency. To find the amount of resources needed to attain the minimum latency, we set the latency SLO to a small, unobtainable value (0.001), prompting ALAP to continue exploring larger core allocations until additional resources are wasted.

Adapting to cost constraints. We follow a similar algorithm as latency to adapt to cost constraints (lines 9-15). If a cost SLO is met, we increase the core count to reduce invocation latency further if the slack is large enough and utilization is high for a prolonged period of its duration: p50 core util > 90% (lines 10-11). However, correcting cost SLO violations is more nuanced than latency, as cost is linearly proportional to both latency and allocation [7, 19]. We can either increase the core count to reduce latency or decrease the core count at the expense of latency. If utilization is high for a prolonged period, we increase the core allocation in an attempt to reduce the latency considerably with more resources (lines 12-13). Otherwise, we lower the core count from the maximum number of cores used (lines 14-15). If the intent is to minimize cost, we set the cost SLO to an unobtainable low value (0.001) to trigger ALAP to respond to "cost violations" and converge to an allocation that minimizes cost without wasting resources. As a safeguard, if increasing the core count deviates further from the SLO, we inform the agent to explore lowering core allocations, as larger allocations outweigh the cost savings from reducing latency.

Scalability and robustness. ALAP's model formulation is lightweight (evaluated in § 9.6) and independent of the cluster size, ensuring it can scale to millions of workers. Through observation, ALAP promptly discerns the thread-level parallelism of functions and restricts allocation sizes to mitigate resource waste. Moreover, we gracefully adjust allocations to not "overreact" to SLO violations (§ 9.2) that may be caused by external factors (e.g., noise from multi-tenancy). Finally, similar to commercial providers [7, 19], we cap allocation sizes to curb malicious users from hogging resources. These design decisions ensure ALAP's agents are robust to one-off SLO violations due to noise [112] and malicious functions.

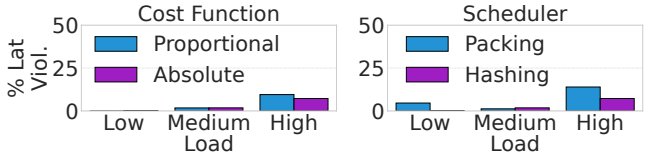


Figure 7: Design exploration of ALAP's (a) cost function and (b) scheduler algorithm. ALAP incurs fewer SLO violations using the *absolute* cost function in its Resource Allocator and a *hashing*-based scheme in its Scheduler. See § 5.3 & 6.

6 Delayed Container Provisioning

We design a new scheduler to complement ALAP's Resource Allocator for three reasons. (1) Delayed, input-aware allocations per invocation may increase cold starts due to varying container sizes required to meet user intents. Existing schedulers pre-warm containers of *fixed size* to reduce cold starts, but do not account for the distribution of incoming inputs, as they assume allocations are static per function [1, 14, 52, 112]. This exacerbates resource waste, as invocations to the same function with smaller inputs get scheduled on larger containers. While ALAP's Allocator is compatible with previous pre-warming techniques, we find our simple, delayed provisioning technique greatly reduces resource waste per container. (2) Previous serverless schedulers optimize how to schedule or pack invocations to containers [14, 52, 86, 93], how many containers to create [14, 52], and when to create/destroy them [14, 112]. Other than Hermod [41], previous schedulers do not optimize container-to-server placement, as they do not consider the load and exact resource usage per invocation when choosing the server to place a container on. We empirically show Hermod's limitations with realistic functions in a few paragraphs. Finally, Kubernetes-based schedulers [27, 28, 62, 96] consider utilization but do not vertically scale (i.e., add more resources to containers) on a per-invocation basis like ALAP. (3) OpenWhisk's scheduler is memory-centric: it only considers the memory availability of a server, not CPU resources, when making placement and load balancing decisions. This scheduling policy falls short when making independent allocation decisions per resource type; we observe memory-centric schedulers lead to severe CPU oversubscription (§ 9.1.1). We describe ALAP's scheduling algorithm next.

ALAP's scheduling algorithm. The Scheduler (1) prioritizes routing an invocation to a warm container of the exact size predicted by ALAP's online learning agent, (2) routes an invocation to a warm container larger but closest to the size predicted, and in the worst case (3) creates a new cold container of the exact size if no warm containers are suitable. If the Scheduler routes to a larger container, it also pre-launches a container of the requested size in the background to accelerate future invocations. This simple delayed provisioning feature greatly reduces underutilized resources for future invocations (§ 9.4). ALAP's Scheduler tracks the CPU and memory load per server to make intelligent load-balancing decisions: when routing an invocation to a container (whether it is cold or warm), the Scheduler ensures the server has enough available

resources. To avoid severe interference across functions sharing a server, we limit the overall utilization of each server to be between 70-80%, following commercial providers [18, 40, 94, 109].

Upon a cold start, the Scheduler must choose a server to create the new container on. We empirically evaluate two policies. (1) Hashing: similar to OpenWhisk, our implementation attempts to reduce cache contention and improve locality by assigning each function a "home server" via a hashing algorithm [29, 105]. If the home server has capacity, we create a new container on it. If not, we select the next available server. If no server has capacity, we randomly pick a server. This policy spreads invocations across servers. (2) Packing: we implement Hermod's policy [41], which packs invocations one server at time until the server reaches capacity.

At high loads, Hermod's packing algorithm violates more latency SLOs (Figure 7b). Several functions (e.g., MLTrain, VidProc) retrieve inputs from an external database, requiring network bandwidth, which Hermod overlooks in its study [41]. Packing invocations until the server reaches peak CPU capacity (70-80%) makes network bandwidth the bottleneck of the server, thereby inflating latency. Hence, ALAP uses the hashing-based algorithm in its final design.

Provisioning idle containers in the background. Creating right-sized containers in the background introduces more idle warm containers to provision. Several systems [14, 52] use bin-packing solutions to decrease the number of containers provisioned to improve resource utilization. We argue that the container number is the wrong resource utilization metric, as containers are simply an abstraction over the actual hardware resources. We find our approach introduces no overhead to the system: while idle, containers do not consume cores and only consume small amounts of memory, as previous works heavily optimize container image sizes [17, 25, 55, 86, 92, 97, 111]. Moreover, ALAP's scheduler only provisions background containers if a server has enough unallocated memory available; it does not harm/evict currently running containers. Thus, by launching idle containers, the Scheduler does not increase the current load of a server. In ALAP, we use the default OpenWhisk keep-alive policy for these idle containers, but our scheduler is compatible with more advanced approaches [84].

7 Implementation

We implement ALAP on Apache OpenWhisk (OW) [29].

ALAP's Resource Allocator. ALAP's Allocator is a Python shim layer (2052 LoC) atop OW. Developers register and invoke functions through ALAP's API via gRPC. The Allocator forwards requests to OW. ALAP's agent uses Vowpal Wabbit's lightweight, efficient (§ 9.6) cost-sensitive multi-class classification algorithm [50]. We write simple C++ functions to obtain input metadata (Table 4) within μs. For initial invocations (10 for predicting core count, 20 for memory § 9.5), we default allocations while the agent learns. For scalability, we maintain a pool of 40 threads (a configurable knob) to extract metadata and predict resource configurations.

Decoupled resource allocations. OW couples resource types. We decouple resources in OW to enable independent allocations per resource type. We introduce a new resource limit class, `CPULimit()`, that can be set for each invocation and update the OW API to include this limit in the API call header.

ALAP's Scheduler. We implement ALAP's Scheduler in OW (794 LoC Scala). The Scheduler maintains per-worker maps that track warm containers and their sizes. The map has at most two entries per invocation if a container of the predicted size is not found. The Scheduler's delayed container provisioning policy and OW's keep-alive policy reduce the need to maintain new containers over time. Hence, this map is at most a few MBs, ensuring its scalability. **ALAP's Daemon.** ALAP's per-worker daemon (326 LoC of C code) collects container-level utilization metrics every 25 ms using Linux cgroups and aggregates this data per invocation to send it (via gRPC) along with the invocation's performance metrics (exec time, cold start latency) to ALAP's Allocator to update its agent online.

Functions	# Threads	# Inputs	Input Sizes	Latency
Encrypt	single	string (36)	2-24KB	93-847 ms
SentAnal	single	json (23)	63KB-5MB	0.08-3 sec
ImgProc	single	image (64)	184KB-4.5MB	0.5-3 sec
ResNet50	multiple	image (64)	184KB-4.5MB	2-5 sec
MobileNet	single	image (64)	184KB-4.5MB	2-13 sec
VidProc	multiple	video (28)	2.2-6.1MB	2-20 sec
Linpack	multiple	matrix (22)	5.3MB-1.7GB	6-74 sec
Speech2Txt	single	audio (32)	48KB-12MB	10-42 sec
Compress	single	text file (28)	64MB-2GB	10-85 sec
Matmult	multiple	matrix (25)	5.3MB-1.7GB	20-89 sec
MLTrain	multiple	csv (17)	10-100MB	0.5-4 min

Table 4: Functions used in ALAP's evaluation. See § 8.

8 Evaluation Methodology

Testbed. We deploy ALAP on 17 servers in Chameleon Cloud [43] connected via a NetXtreme-E 10Gb/25Gb RDMA Ethernet Controller. Each server has two Intel Xeon Gold 6240R CPUs (96 cores) at 2.40 GHz, 192GB memory, and runs Ubuntu LTS 20.0.4. One server hosts OpenWhisk's API gateway, Controller, CouchDB, and ALAP's central components (allocator, scheduler, and metadata store). The other servers host an OpenWhisk Invoker and ALAP's daemon.

Functions. Serverless production traces [40, 80, 84] conceal the details about actual functions executed. Thus, we use representative functions (Table 4) from several serverless benchmark suites [21, 45, 96] commonly used in recent works [3, 4, 14, 16, 30, 38, 52, 54, 67, 69, 88, 109]. Our functions include scientific apps, image/video processing, ML training/inference, and web services, with varying semantics (single- vs. multi-threaded) and inputs with multiple descriptive features (e.g., size, resolution, bit-rate). Azure [84] and Meta [80] report that 70% of functions are short-lived, running for 100ms-10s. We show ALAP's efficacy in optimizing these functions' cost and resource utilization (e.g., Encrypt, SentAnal, ImgProc) without harming latency. For longer-running functions (> 10s) that dominate resource usage (25% of functions in Azure, 20 – 40% of functions in Meta), we optimize these functions across latency, cost, and utilization to maximize the gains of the serverless platform.

Workload. We follow the methodology of previous works [14, 41, 52, 87, 88] to create our workload. We use Azure's production serverless traces [84] to mimic real-world invocation patterns while executing functions from Table 4. The traces capture the bursty behavior of serverless workloads. We analyze the trace and find that 74.65% of invocations are made to the same 11 functions. As the trace only provides timestamps of arriving invocations without detailing the function, we randomly assign 11 of the functions in Table 4 to

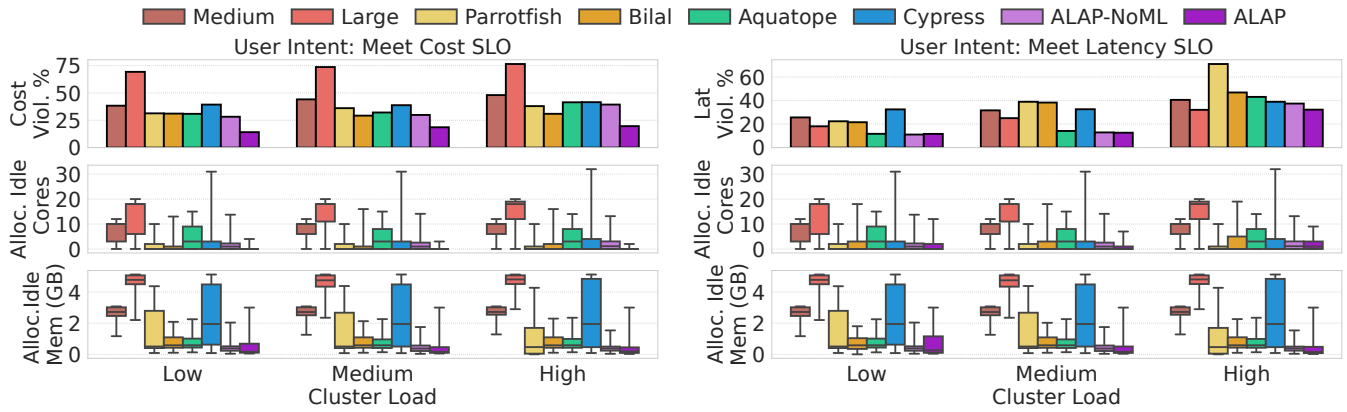


Figure 8: Across different SLO intents, ALAP consistently reduces (1) cost SLO violations (left column) and (2) latency SLO violations (right column) across loads while minimizing resource waste (rows 2 and 3). See § 9.1.1.

the corresponding invocations from the trace (we show that ALAP scales as the number of functions it has to manage grows in § 9.7). We set every function's SLO to 1.4× its median observed execution time and cost in isolation; we demonstrate ALAP's efficacy across looser SLOs in § 9.5. Because we do not have thousands of servers to support the full trace, we scale the trace down: we randomly choose a 4-hour window and select timestamps to match our target requests per second (RPS) while maintaining the burstiness of the trace. Like commercial providers [18, 22, 31, 60, 76, 94, 109], we generate low, medium, and high load to reflect cluster utilization of 25%, 50%, and 75%, respectively. As our functions execute for 30 seconds and utilize 7 cores on average, our 1536 core cluster can support $1536/7/30 = 7.3$ requests per second (RPS) at peak load before SLO violations surge. We generate high load by scaling the trace to six RPS (SLO violations grow > 50% at RPS > 7, but ALAP still outperforms baselines). This RPS is not a limitation of ALAP but our cluster size: we demonstrate ALAP scales to 940 RPS in § 9.7.

Cost model. A serverless invocation's cost is $\gamma\times$ GB $\times$ latency (s), where γ is set by the serverless provider [7, 13, 19]. This model assumes tight coupling of resources (i.e., the cost of cores is included in γ). With ALAP's independent allocations, we need to associate a separate cost multiplier for each resource type. Analyzing AWS EC2 offerings [6], we find doubling memory or both memory and cores increases γ by 2×, but the effect of doubling cores alone is unclear. For simplicity, we set each resource type's cost multiplier to $\gamma/2$ in our evaluation, however, even with greater emphasis on cores or memory, our cost improvements still surpass baselines.

Baselines. We compare ALAP to seven baselines. Our static baselines provide *Medium* (12 cores, 3GB) or *Large* (20 cores, 5GB) allocations per function, representing how users request static allocations on commercial platforms [7, 19].

Parrotfish [67] is a state-of-the-art developer tool that profiles functions offline with representative inputs to recommend a function-level memory limit. We provide Parrotfish three inputs (small, medium, large) per function and use its recommendation for all invocations to the function.

Bilal et al. [16] and *Aquatope* [112] both build Bayesian Optimization neural networks per function to make function-level, decoupled allocations that are input agnostic. Unlike Aquatope, Bilal et al. adjust allocations to minimize cost or latency.

Cypress [14] is a state-of-the-art system that represents the works that bin-pack multiple invocations into a single container in an attempt to reduce container provisioning [14, 15, 52]. Cypress builds regressors per function to predict an invocation's latency given its input size (ignoring other input properties) and assigns a batch size per invocation accordingly. It then packs similarly-sized batches into one container to minimize container provisioning. We compare against Cypress specifically since it uses an input's size to predict latency; we show that not being holistically input-aware leads to poor allocation decisions that either inflate latency or waste resources.

ALAP-NoML showcases a simpler profiling- (rather than ML-) based resource allocator. It offline profiles a function with user-provided inputs (e.g., small, medium, large inputs). For each input, it observes the latency and resource utilization across all allocations to determine the optimal allocation that minimizes latency without wasting resources. For incoming inputs online, it selects the appropriate allocation by comparing the input size to the profiled inputs.

9 Evaluation

We evaluate ALAP's efficacy in meeting various user intents while minimizing resource waste. We then analyze ALAP's Resource Allocator and ALAP's Scheduler efficacy. Finally, we empirically demonstrate ALAP's robustness, overheads, and scalability.

9.1 ALAP's Efficacy in Meeting User Intents

9.1.1 SLO Compliance. We first show ALAP's efficacy in meeting user-specified SLOs. Figure 8 shows the percent of invocations with cost (left column) and latency (right column) SLO violations, and the amount of idle allocated resources.

ALAP vs. static baselines. Allocating more resources (Medium to Large) improves latency SLO compliance (right column) at the expense of wasted resources and cost SLO compliance (left column). However, Large still violates latency SLOs. Large runs atop

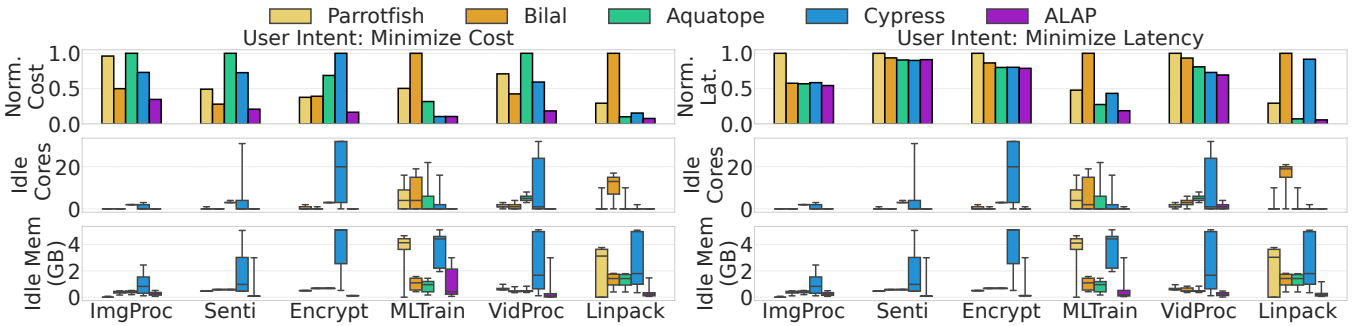


Figure 9: ALAP reduces cost and resource waste for all functions. It considerably reduces latency for multi-threaded functions without harming single-threaded functions. Values per function show the max. See § 9.1.2.

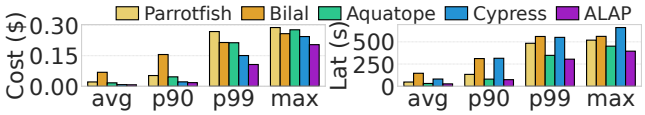


Figure 10: Breakdown of ALAP's cost and latency across functions at high load. See § 9.1.2.

OpenWhisk, whose memory-centric scheduling policy heavily oversubscribes a few servers with most invocations since it only considers the memory availability of a server, not CPU resources, when making scheduling decisions. This increases CPU contention and degrades performance despite larger allocation sizes.

ALAP vs. Parrotfish [67]. Parrotfish provides tighter allocations than the static baselines. However, as Parrotfish also runs atop OW, its tight allocations enable OW's scheduler to greatly overpack servers, exacerbating core contention and latency SLO violations (e.g., Parrotfish violates VidProc's SLOs 21.1× more than ALAP). Moreover, Parrotfish unnecessarily overprovisions memory to handle the worst-case input size (input-agnostic decisions) or obtain more cores (coupled resource types). Meanwhile, ALAP's input-aware and decoupled allocations, along with load-balancing decisions that account for both resource types, reduce memory waste (3.8×) and SLO violations (latency by 2.3×, cost by 2×).

ALAP vs. Bilal [16] & Aquatope [112]. Bilal's decoupled allocations reduce latency SLO violations at high load compared to Parrotfish. Both Bilal and Aquatope provide nearly identical allocations (both use similar Bayesian Optimization formulations), but Bilal runs atop OW's memory-centric scheduler which increases CPU contention and latency SLO violations compared to Aquatope. ALAP reduces latency SLO violations by 1.4× compared to Aquatope at high load. Although not required to meet SLOs, several inputs can well-utilize more cores if allocated. Aquatope provides these inputs with larger-than-needed allocations, thereby reducing the number of available cores and causing increased SLO violations for other invocations. This exemplifies the importance of not simply deploying larger containers. ALAP provides just enough resources to each invocation. This reduces contention, increases core availability for other invocations, and improves overall SLO compliance.

ALAP's input-aware allocations greatly reduce resource waste: at high load, ALAP reduces the median and p75 idle memory by 4× and 2×, respectively, and the p95 idle cores by 3× compared to Aquatope. Moreover, by being intent-aware, ALAP adjusts its allocations (i.e., ALAP allocates MLTrain 20 cores to meet latency SLOs versus 14 cores to meet cost SLOs) to reduce cost SLO violations by 2.1× compared to Aquatope. Bilal also adjusts allocations to minimize either cost or latency, however, ALAP's input-aware decisions reduce cost SLO violations by 1.6× compared to Bilal.

ALAP vs. Cypress [14]. Cypress provisions large containers to bin-pack multiple invocations in one container. Under frequent arrival, Cypress allocates 1-2 cores to every invocation, including multi-threaded ones: Cypress assumes functions are single-threaded and makes poor allocation decisions for certain functions (e.g., VidProc, ImgProc) since it is not holistically input-aware (only input-size aware). Thus, Cypress inflates latency SLO violations by 1.3× compared to ALAP. Under sparse arrival, the resources of its large containers remain underutilized: Cypress wastes memory, allocating invocations multiples of their needs. Cypress violates 2.1× the cost SLOs as ALAP across functions (13.2× more violations for Encrypt), as invocations are either given too many or too few resources.

ALAP vs. ALAP-NoML. ALAP-NoML outperforms Aquatope in cost with its "relative" input-aware decisions that reduce resource waste. However, with online ML, ALAP performs better in both intents (2× reduction in cost violations, 1.2× in latency), as it makes custom allocation decisions per input. Moreover, ALAP does not incur any overhead of offline profiling. For example, profiling VidProc offline took 117 seconds, a 500% overhead for this function alone. Hence, we proceed with ALAP for the remainder of this evaluation.

Overall, ALAP reduces latency and cost SLO violations by 1.3-2.3× and 1.6-2.2×, respectively, compared to the advanced baselines. While meeting latency (cost) SLOs, ALAP also reduces the secondary metric of cost (latency) by 1.2-1.8x and nearly eliminates resource waste with its input-aware, decoupled resource allocations.

9.1.2 Minimizing Cost or Latency. We next evaluate ALAP when users simply ask to minimize cost or latency. Figure 9 presents the total cost and latency to serve the requests made to six functions at high load (we omit the poor-performing static baselines for brevity). We normalize each function's cost and latency by its maximum across the systems. We also report summary statistics of ALAP's cost and latency across the entire high-load trace in Figure 10. ALAP adjusts its allocations to adapt to the intent. For example, it allocates the multi-threaded VidProc 4 versus 12 cores to minimize cost versus latency, respectively. Cypress allocates fewer resources (1-2

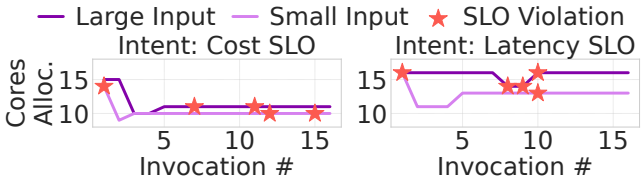


Figure 11: Timeline of ALAP's core allocations to Matmul. ALAP adapts its allocations in response to different inputs (shown by large and small), user intents, and SLO violations. See § 9.2.

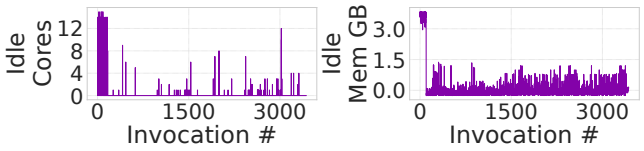


Figure 12: ALAP's online models quickly converge and make accurate allocations that reduce resource waste. See § 9.3.

cores) to all invocations regardless of the intent and packs invocations at high loads, inflating latency for multi-threaded functions (e.g., MLTrain) by > 2× compared to ALAP. Aquatope is comparable in latency to ALAP, however, ALAP reduces resource waste and cost. Both Parrotfish and Bilal make input-agnostic allocations without improving scheduling decisions; hence, functions under both systems severely contend for resources, disallowing either to greatly reduce cost or latency across all functions (e.g., ImgProc's latency is 2× greater under Parrotfish than ALAP, Linpack's cost and latency under Bilal is 13× and 17.5× ALAP's).

For all functions, ALAP minimizes resource waste and reduces cost by up to 13× (3.7× avg) compared to the advanced baselines. Moreover, it reduces latency for multi-threaded functions by up to 17.5× (5.98× avg) without harming latency for single-threaded functions compared to the baselines.

9.2 ALAP's Response to Different Intents, Inputs, and SLO Violations

We next analyze ALAP's response to diverse user-intents, function inputs, and SLO violations. Figure 11 shows ALAP's core allocations made to two Matmult inputs (large and small) under cost and latency constraints. ALAP learns to allocate fewer cores to smaller inputs that cannot exploit additional resources. Further, it adjusts its allocations to the user intent: it learns to provide Matmult fewer cores to meet cost SLOs compared to latency SLOs. Under latency constraints (right column), ALAP lowers the allocation for the large input (invocation #7) to lower user cost. But after observing violations, it quickly reverts to 16 cores to meet the SLO. Under cost constraints, ALAP slightly increases large input's allocation at invocation #3 to reduce latency while meeting the cost SLO. Despite a few violations, it avoids changing its decisions, as most invocations continue to meet the SLOs. ALAP is robust and adapts to varying user intent, inputs, and SLO violations.

9.3 ALAP's Online Model Converges Quickly

We show ALAP quickly converges and makes accurate allocation decisions. Figure 12 shows ALAP's idle resources over time for a high, stable workload. ALAP defaults the first 10-20 invocations of a function with large allocations to train its online agent during this initial learning phase. The cost of this learning phase is the initial spikes in idle cores/memory (10-12 invocations). After, CPU core underutilization is rare. The few remaining spikes occur because ALAP's scheduler routes invocations to larger, warm containers to mitigate cold starts. Similar reasoning applies to idle memory spikes (> 512MB). We note that ALAP produces slightly higher memory waste, allocating 64MB more than predicted as a safeguard to prevent OOM exceptions. We justify this with the significant reduction in OOM exceptions (< 0.47%, § 5.2) compared to previous work (6-8% [69]). Moreover, ALAP consistently reduces memory waste compared to the advanced baselines (Figures 8 and 9).

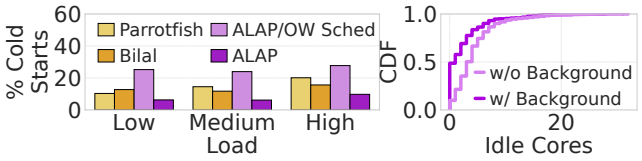


Figure 13: ALAP's Scheduler reduces (a) cold starts by using warm, predicted-sized or larger containers (left) and (b) resource waste by provisioning right-sized containers in the background (right). See § 9.4.

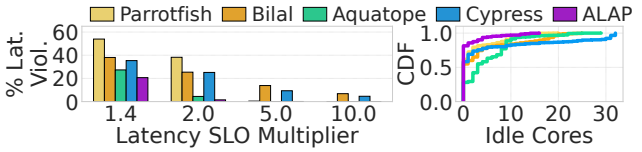


Figure 14: ALAP is robust to a range of SLOs, providing the lowest violations while minimizing resource waste. See § 9.5.

9.4 Efficacy of ALAP's Scheduler

Figure 13 evaluates ALAP Scheduler's efficacy by comparing (1) the percent of invocations with cold starts between ALAP (left plot), and (2) the resource waste reduction when using ALAP's Scheduler with and without background container creation (right plot). We omit Cypress and Aquatope, as we do not implement their pre-warming policies. ALAP dispatches invocations to perfectly sized or slightly larger warm containers, reducing cold starts by 50-56% compared to the baselines. Moreover, the Scheduler provisions right-sized containers in the background, further reducing resource waste (27% reduction in idle cores), as future invocations can utilize these perfectly sized containers without incurring cold starts. Provisioning these extra containers has minimal overheads: on average, we create at most 22 background containers per server, which consume no CPU cycles when idle and only 1.8GB memory (< 0.9% of server memory). Meanwhile, this feature reduces aggregate memory waste by > 24GB per server, further exemplifying its efficacy. ALAP's Scheduler is crucial in enabling delayed allocations that efficiently meet user intents.

9.5 Sensitivity Analysis of ALAP

Varying SLOs. We use tight SLOs in our E2E evaluation. We next evaluate ALAP's efficacy under increasingly looser SLOs. Figure 14 shows that ALAP has fewer violations than the baselines across SLOs. When SLOs are easy to meet (ALAP meets all and Aquatope

nearly meets all SLOs when the SLO multiplier is >2×), ALAP offers better resource utilization: ALAP only wastes at most 6 cores in the p90 compared to Aquatope's 13 idle allocated cores. Thus, ALAP is robust to tight and loose SLOs.

Confidence threshold. The confidence threshold is the number of invocations our online model must observe as it trains before using its predictions. We discuss latency violations and OOM exceptions under varying thresholds. Increasing the core threshold from 5 to 10 reduces latency violations. A larger threshold (15) increases them due to CPU contention, as multi-threaded invocations can consume more cores than needed to meet SLOs. Although the learning phase length depends on the complexity of the function and inputs, a confidence threshold of 8-12 invocations sufficed for core predictions in our observed cases. Increasing the memory prediction threshold ($\geq$ 20 invocations) drastically reduces OOM exceptions (to 0.47%), albeit memory utilization may worsen, as more invocations receive the default, large allocations.

Adaptive core allocation hyperparameters. ALAP adjusts its core allocations in response to SLO compliance or violations with α and β (both $\in$ [0%, 100%]). When SLOs are met, α dictates how aggressively ALAP adjusts allocations to optimize for the secondary metric. Larger α values ensure conservative adjustments to continue meeting SLOs. Meanwhile, β controls how aggressively ALAP adjusts core allocations in response to SLO violations. Smaller β values ensure aggressive adjustments to meet SLOs. Through a standard grid search (5–50%, step = 5%) with three functions under high-load traces, we find that α = 15–35% and β = 5–15% work well across all functions.

9.6 ALAP Incurs Minimal Overheads

ALAP's input featurization and model prediction are on the invocations' critical path. But our design minimizes these overheads.

Execution time overheads. Overall, ALAP introduces minimal execution time overheads. For the shortest running function (Encrypt), ALAP's overheads are <0.053% its latency; for the longest running function (MLTrain), overheads are <0.00062% its latency. These overheads break down as follows. Regardless of the function and input type, model prediction (<57μs) and ALAP's Scheduler (0.4-0.7 ms) contribute little to the overall overheads. Model updates take 0.2-0.4 ms, however, updates are not on an invocation's critical path. Input featurization overheads are also minimal (20 – 400μs).

Resource overheads. ALAP's ML models and input feature vectors are extremely lightweight. The weights comprising each function's model are at most 1.6KB (less than one-millionth of a server's memory). ALAP's input feature vectors are 40-70 bytes; the feature vectors of the 403 function inputs used in our evaluation only consume 20.14KB. The CPU time for input featurization and model prediction/updates is < 400μs [50, 98].

9.7 ALAP Scales with Load and Functions

Scalability with load. To demonstrate ALAP's scalability with load, we follow the methodology of Hermod [41] and Sparrow [71]. We use zero-work functions to make ALAP's Allocator/Scheduler the bottleneck, as workers never exceed 37% utilization. Then, we compare ALAP's throughput with OpenWhisk's as RPS scales. This methodology enables us to compare the scalability of ALAP's resource allocation models and scheduling algorithm with a widely used open-source serverless platform [14, 35, 87–89, 112]. Figure 15 column 1 shows ALAP's throughput under two extreme scenarios: with and without inputs (and hence featurization) per invocation. In both cases, ALAP's throughput closely matches OW (940 RPS); its throughput is slightly lower when featurizing inputs (922 RPS). We argue the benefits of meeting user intents while reducing resource waste outweigh this minor throughput reduction.

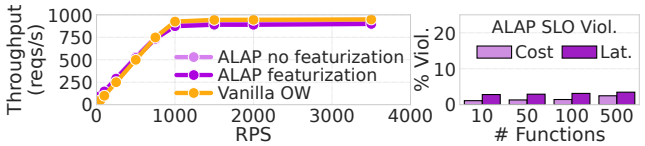


Figure 15: ALAP scales with (a) load (RPS) and (b) the number of functions it has to manage. See § 9.7.

Scalability with the number of functions. We show ALAP scales with the number of functions. We register an increasing number of functions with ALAP under our high load trace and use the same tight SLOs set in § 9.1.1. Figure 15 column 2 shows the cost and latency SLO violations as ALAP manages hundreds of functions. As the number of functions grows, ALAP has to provision more containers across workers. However, ALAP's SLO violations remain the same as the number of functions scales. Moreover, ALAP maintains peak throughput regardless of the number of functions it has to manage (920). Finally, ALAP's lightweight models have little overhead as the number of functions grows: the models consume merely 800KB for all 500 functions, < 0.00043% the server's memory capacity.

10 Conclusion

We introduce ALAP, an *intent-based* serverless computing system that provides an interface for users to communicate their intents. To meet user intents, ALAP delays resource allocation decisions until function inputs are available and makes independent decisions for each resource type. ALAP uses online learning to predict the required amount of resources to meet user intent. Across a diverse set of serverless functions and user-intents, ALAP outperforms several state-of-the-art systems while improving resource efficiency.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Dr. Hao Wang, for their invaluable feedback. We thank the UT-SysML research group for their insightful discussions. We also thank Vivek Chawda for his helpful feedback on the design and the introduction. This work was supported by the UT ECE junior faculty start-up fund, UT iMAGiNE consortium and its industrial affiliates, an award from the UT Machine Learning Lab (MLL), the AMD Chair Endowment, and the Amazon Research Award.

References

[1] Mania Abdi, Samuel Ginzburg, Xiayue Charles Lin, Jose Faleiro, Gohar Irfan Chaudhry, Inigo Goiri, Ricardo Bianchini, Daniel S Berger, and Rodrigo Fonseca. 2023. Palette Load Balancing: Locality Hints for Serverless Functions. In *Proceedings of the Eighteenth European Conference on Computer Systems* (Rome, Italy) *(EuroSys '23)*. Association for Computing Machinery, New York, NY, USA, 365–380. doi:10.1145/3552326.3567496

[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] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. SAND: Towards High-Performance Serverless Computing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 923–935. https://www.usenix.org/conference/atc18/presentation/akkus

[4] Lixiang Ao, George Porter, and Geoffrey M. Voelker. 2022. FaaSnap: FaaS Made Fast Using Snapshot-Based VMs. In *Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys '22)*. Association for Computing Machinery, Rennes, France, 730–746. doi:10.1145/3492321.3524270

[5] Moiz Arif, Kevin Assogba, and M. Mustafa Rafique. 2022. Canary: fault-tolerant FaaS for stateful time-sensitive applications. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Dallas, Texas) *(SC '22)*. IEEE Press, Article 41, 16 pages.

[6] AWS. 2025. Amazon EC2 On-Demand Pricing. https://aws.amazon.com/ec2/pricing/on-demand/.

[7] AWS. 2025. AWS Lambda. https://aws.amazon.com/lambda/.

[8] AWS. 2025. AWS Lambda Memory and Computing Power . https://docs.aws.amazon.com/lambda/latest/operatorguide/computing-power.html.

[9] AWS. 2025. AWS S3 HeadObject . https://docs.aws.amazon.com/AmazonS3/latest/API/API_HeadObject.html.

[10] AWS. 2025. AWS Serverless Application Repository . https://aws.amazon.com/serverless/serverlessrepo/.

[11] AWS. 2025. Managing Asynchronous Workflows with a REST API . https://aws.amazon.com/blogs/architecture/managing-asynchronous-workflows-with-a-rest-api/.

[12] AWS. 2025. Profiling functions with AWS Lambda Power Tuning. https://docs.aws.amazon.com/lambda/latest/operatorguide/profile-functions.html.

[13] Azure. 2025. Microsoft Azure Functions. https://azure.microsoft.com/en-us/services/functions/.

[14] Vivek M. Bhasi, Jashwant Raj Gunasekaran, Aakash Sharma, Mahmut Taylan Kandemir, and Chita Das. 2022. Cypress: Input Size-Sensitive Container Provisioning and Request Scheduling for Serverless Platforms. In *Proceedings of the 13th Symposium on Cloud Computing* (San Francisco, California) *(SoCC '22)*. Association for Computing Machinery, New York, NY, USA, 257–272. doi:10.1145/3542929.3563464

[15] Vivek M. Bhasi, Jashwant Raj Gunasekaran, Prashanth Thinakaran, Cyan Subhra Mishra, Mahmut Taylan Kandemir, and Chita Das. 2021. Kraken: Adaptive Container Provisioning for Deploying Dynamic DAGs in Serverless Platforms. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SoCC '21)*. Association for Computing Machinery, New York, NY, USA, 153–167. doi:10.1145/3472883.3486992

[16] Muhammad Bilal, Marco Canini, Rodrigo Fonseca, and Rodrigo Rodrigues. 2023. With Great Freedom Comes Great Opportunity: Rethinking Resource Allocation for Serverless Functions. In *Proceedings of the Eighteenth European Conference on Computer Systems* (Rome, Italy) *(EuroSys '23)*. Association for Computing Machinery, New York, NY, USA, 381–397. doi:10.1145/3552326.3567506

[17] Marc Brooker, Mike Danilov, Chris Greenwood, and Phil Piwonka. 2023. On-demand Container Loading in AWS Lambda. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, Boston, MA, 315–328. https://www.usenix.org/conference/atc23/presentation/brooker

[18] Alibaba Cloud. 2025. AliCloud Severless Function. https://www.alibabacloud.com/product/function-compute.

[19] Google Cloud. 2025. Google Cloud Functions. https://cloud.google.com/functions/.

[20] Google Cloud. 2025. Google Cloud Functions Pricing. https://cloud.google.com/functions/pricing.

[21] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. 2021. SeBS: A Serverless Benchmark Suite for Function-as-a-Service Computing. In *Proceedings of the 22nd International Middleware Conference* (Québec city, Canada) *(Middleware '21)*. Association for Computing Machinery, New York, NY, USA, 64–78. doi:10.1145/3464298.3476133

[22] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 153–167. doi:10.1145/3132747.3132772

[23] datadog. 2021. The state of Serverless 2021. https://www.datadoghq.com/state-of-serverless-2021/.

[24] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*. 248–255. doi:10.1109/CVPR.2009.5206848

[25] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. Catalyzer: Sub-Millisecond Startup for Serverless Computing with Initialization-Less Booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. Association for Computing Machinery, Lausanne, Switzerland, 467–481. doi:10.1145/3373376.3378512

[26] 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* (Québec city, Canada) *(Middleware '21)*. Association for Computing Machinery, New York, NY, USA, 248–259. doi:10.1145/3464298.3493398

[27] Cloud Native Computing Foundation. 2025. Knative. https://knative.dev/docs/.

[28] Cloud Native Computing Foundation. 2025. Kubernetes. https://kubernetes.io/.

[29] The Apache Software Foundation. 2025. Apache OpenWhisk. https://openwhisk.apache.org/.

[30] 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* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 386–400. doi:10.1145/3445814.3446757

[31] Jing Guo, Zihao Chang, Sa Wang, Haiyang Ding, Yihui Feng, Liang Mao, and Yungang Bao. 2019. Who Limits the Resource Efficiency of My Datacenter: An Analysis of Alibaba Datacenter Traces. In *2019 IEEE/ACM 27th International Symposium on Quality of Service (IWQoS)*. 1–10. doi:10.1145/3326285.3329074

[32] Tyler Harter, Brandon Salmon, Rose Liu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. Slacker: Fast Distribution with Lazy Docker Containers. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. USENIX Association, Santa Clara, CA, 181–195. https://www.usenix.org/conference/fast16/technical-sessions/presentation/harter

[33] https://www.openslr.org/12 2025. Open Speech and Language Resources . https://www.openslr.org/12.

[34] Huawei. 2025. Huawei Cloud Functions. https://developer.huawei.com/consumer/en/agconnect/cloud-function/.

[35] IBM. 2025. IBM Cloud Functions. https://cloud.ibm.com/functions/.

[36] Haroon Idrees, Amir R. Zamir, Yu-Gang Jiang, Alex Gorban, Ivan Laptev, Rahul Sukthankar, and Mubarak Shah. 2016. The THUMOS Challenge on Action Recognition for Videos "in the Wild". *CoRR* abs/1604.06182 (2016). arXiv:1604.06182 http://arxiv.org/abs/1604.06182

[37] Zhipeng Jia and Emmett Witchel. 2021. Boki: Stateful Serverless Computing with Shared Logs. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)*. Association for Computing Machinery, Virtual Event, Germany, 691–707. doi:10.1145/3477132.3483541

[38] Yankai Jiang, Rohan Basu Roy, Baolin Li, and Devesh Tiwari. 2024. EcoLife: Carbon-Aware Serverless Function Scheduling for Sustainable Computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis* (Atlanta, GA, USA) *(SC '24)*. IEEE Press, Article 12, 15 pages. doi:10.1109/SC41406.2024.00018

[39] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Menezes Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. 2019. *Cloud Programming Simplified: A Berkeley View on Serverless Computing*. Technical Report UCB/EECS-2019-3. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-3.html

[40] 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)*. Association for Computing Machinery, New York, NY, USA, 443–458. doi:10.1145/3620678.3624783

[41] 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)*. Association for Computing Machinery, New York, NY, USA, 289–305. doi:10.1145/3542929.3563468

[42] Konstantinos Kallas, Haoran Zhang, Rajeev Alur, Sebastian Angel, and Vincent Liu. 2023. Executing Microservice Applications on Serverless, Correctly. *Proc. ACM Program. Lang.* 7, POPL, Article 13 (jan 2023), 29 pages. doi:10.1145/3571206

[43] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S. Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, François Halbach, Alex Rocha, and Joe Stubbs. 2020. Lessons Learned from the Chameleon Testbed. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association.

[44] Anurag Khandelwal, Yupeng Tang, Rachit Agarwal, Aditya Akella, and Ion Stoica. 2022. Jiffy: Elastic Far-Memory for Stateful Serverless Analytics *(EuroSys '22)*. Association for Computing Machinery, Rennes, France, 697–713. doi:10.1145/3492321.3527539

[45] Jeongchul Kim and Kyungyong Lee. 2019. FunctionBench: A Suite of Workloads for Serverless Cloud Function Service. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. 502–504. doi:10.1109/CLOUD.2019.00091

[46] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 427–444. https://www.usenix.org/conference/osdi18/presentation/klimovic

[47] Sumer Kohli, Shreyas Kharbanda, Rodrigo Bruno, Joao Carreira, and Pedro Fonseca. 2024. Pronghorn: Effective Checkpoint Orchestration for Serverless Hot-Starts. In *Proceedings of the Nineteenth European Conference on Computer Systems* (Athens, Greece) *(EuroSys '24)*. Association for Computing Machinery, New York, NY, USA, 298–316. doi:10.1145/3627703.3629556

[48] Swaroop Kotni, Ajay Nayak, Vinod Ganapathy, and Arkaprava Basu. 2021. Faastlane: Accelerating Function-as-a-Service Workflows. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 805–820. https://www.usenix.org/conference/atc21/presentation/kotni

[49] Peter Kraft, Qian Li, Kostis Kaffes, Athinagoras Skiadopoulos, Deeptaanshu Kumar, Danny Cho, Jason Li, Robert Redmond, Nathan Weckwerth, Brian Xia, Peter Bailis, Michael Cafarella, Goetz Graefe, Jeremy Kepner, Christos Kozyrakis, Michael Stonebraker, Lalith Suresh, Xiangyao Yu, and Matei Zaharia. 2023. Apiary: A DBMS-Integrated Transactional Function-as-a-Service Framework. arXiv:2208.13068 [cs.DB]

[50] Langford, John and the Vowpal Wabbit developers. 2025. Vowpal Wabbit. https://vowpalwabbit.org/index.html.

[51] Qiang Li, Lulu Chen, Xiaoliang Wang, Shuo Huang, Qiao Xiang, Yuanyuan Dong, Wenhui Yao, Minfei Huang, Puyuan Yang, Shanyang Liu, Zhaosheng Zhu, Huayong Wang, Haonan Qiu, Derui Liu, Shaozong Liu, Yujie Zhou, Yaohui Wu, Zhiwu Wu, Shang Gao, Chao Han, Zicheng Luo, Yuchao Shao, Gexiao Tian, Zhongjie Wu, Zheng Cao, Jinbo Wu, Jiwu Shu, Jie Wu, and Jiesheng Wu. 2023. Fisc: A Large-scale Cloud-native-oriented File System. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. USENIX Association, Santa Clara, CA, 231–246. https://www.usenix.org/conference/fast23/presentation/li-qiang-fisc

[52] Suyi Li, Wei Wang, Jun Yang, Guangzhen Chen, and Daohe Lu. 2023. Golgi: Performance-Aware, Resource-Efficient Function Scheduling for Serverless Computing. In *Proceedings of the 2023 ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '23)*. Association for Computing Machinery, New York, NY, USA, 32–47. doi:10.1145/3620678.3624645

[53] Yiming Li, Laiping Zhao, Yanan Yang, and Wenyu Qu. 2023. Rethinking Deployment for Serverless Functions: A Performance-First Perspective. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, CO, USA) *(SC '23)*. Association for Computing Machinery, New York, NY, USA, Article 67, 14 pages. doi:10.1145/3581784.3613211

[54] Zijun Li, Quan Chen, Shuai Xue, Tao Ma, Yong Yang, Zhuo Song, and Minyi Guo. 2020. Amoeba: QoS-Awareness and Reduced Resource Usage of Microservices with Serverless Computing. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 399–408. doi:10.1109/IPDPS47924.2020.00049

[55] Zijun Li, Jiagan Cheng, Quan Chen, Eryu Guan, Zizheng Bian, Yi Tao, Bin Zha, Qiang Wang, Weidong Han, and Minyi Guo. 2022. RunD: A Lightweight Secure Container Runtime for High-density Deployment and High-concurrency Startup in Serverless Computing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 53–68. https://www.usenix.org/conference/atc22/presentation/li-zijun-rund

[56] Zijun Li, Yushi Liu, Linsong Guo, Quan Chen, Jiagan Cheng, Wenli Zheng, and Minyi Guo. 2022. FaaSFlow: Enable Efficient Workflow Execution for Function-as-a-Service. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*. Association for Computing Machinery, Lausanne, Switzerland, 782–796. doi:10.1145/3503222.3507717

[57] Guowei Liu, Laiping Zhao, Yiming Li, Zhaolin Duan, Sheng Chen, Yitao Hu, Zhiyuan Su, and Wenyu Qu. 2024. FUYAO: DPU-enabled Direct Data Transfer for Serverless Computing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 431–447. doi:10.1145/3620666.3651327

[58] Haifeng Liu, Wei Ding, Yuan Chen, Weilong Guo, Shuoran Liu, Tianpeng Li, Mofei Zhang, Jianxing Zhao, Hongyin Zhu, and Zhengyi Zhu. 2019. CFS: A Distributed File System for Large Scale Container Platforms. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19)*. Association for Computing Machinery, Amsterdam, Netherlands, 1729–1742. doi:10.1145/3299869.3314046

[59] Qingyuan Liu, Yanning Yang, Dong Du, Yubin Xia, Ping Zhang, Jia Feng, James R. Larus, and Haibo Chen. 2024. Harmonizing Efficiency and Practicability: Optimizing Resource Utilization in Serverless Computing with Jiagu. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. USENIX Association, Santa Clara, CA, 1–17. https://www.usenix.org/conference/atc24/presentation/liu-qingyuan

[60] Qixiao Liu and Zhibin Yu. 2018. The Elasticity and Plasticity in Semi-Containerized Co-locating Cloud Workload: a View from Alibaba Trace. In *Proceedings of the ACM Symposium on Cloud Computing* (Carlsbad, CA, USA) *(SoCC '18)*. Association for Computing Machinery, New York, NY, USA, 347–360. doi:10.1145/3267809.3267830

[61] lr-train-dataset 2025. Airline Customer Satisfaction . https://www.kaggle.com/datasets?tags=13404-Logistic+Regression.

[62] Alex Ellis / OpenFaaS Ltd. 2025. OpenFaaS. https://www.openfaas.com/.

[63] Fangming Lu, Xingda Wei, Zhuobin Huang, Rong Chen, Minyu Wu, and Haibo Chen. 2024. Serialization/Deserialization-free State Transfer in Serverless Workflows. In *Proceedings of the Nineteenth European Conference on Computer Systems* (Athens, Greece) *(EuroSys '24)*. Association for Computing Machinery, New York, NY, USA, 132–147. doi:10.1145/3627703.3629568

[64] Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. 2011. Learning Word Vectors for Sentiment Analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, Dekang Lin, Yuji Matsumoto, and Rada Mihalcea (Eds.). Association for Computational Linguistics, Portland, Oregon, USA, 142–150. https://aclanthology.org/P11-1015

[65] Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. 2021. SONIC: Application-aware Data Passing for Chained Serverless Applications. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 285–301. https://www.usenix.org/conference/atc21/presentation/mahgoub

[66] Kai Mast, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2022. LambdaObjects: Re-Aggregating Storage and Execution for Cloud Computing. In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage '22)*. Association for Computing Machinery, Virtual Event, 15–22. doi:10.1145/3538643.3539751

[67] 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)*. Association for Computing Machinery, New York, NY, USA, 177–192. doi:10.1145/3620678.3624654

[68] Kevin P. Murphy. 2022. *Probabilistic Machine Learning: An introduction.* MIT Press. probml.ai

[69] Djob Mvondo, Mathieu Bacou, Kevin Nguetchouang, Lucien Ngale, Stéphane Pouget, Josiane Kouam, Renaud Lachaize, Jinho Hwang, Tim Wood, Daniel Hagimont, Noël De Palma, Bernabé Batchakui, and Alain Tchana. 2021. OFC: An Opportunistic Caching System for FaaS Platforms. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 228–244. doi:10.1145/3447786.3456239

[70] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 57–70. https://www.usenix.org/conference/atc18/presentation/oakes

[71] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. 2013. Sparrow: distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 69–84. doi:10.1145/2517349.2522716

[72] Jordi Pont-Tuset, Federico Perazzi, Sergi Caelles, Pablo Arbeláez, Alexander Sorkine-Hornung, and Luc Van Gool. 2017. The 2017 DAVIS Challenge on Video Object Segmentation. *CoRR* abs/1704.00675 (2017). arXiv:1704.00675 http://arxiv.org/abs/1704.00675

[73] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. 2019. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 193–206. https://www.usenix.org/conference/nsdi19/presentation/pu

[74] Thomas Pusztai and Stefan Nastic. 2025. ChunkFunc: Dynamic SLO-Aware Configuration of Serverless Functions. *IEEE Transactions on Parallel and Distributed Systems* 36, 6 (2025), 1237–1252. doi:10.1109/TPDS.2025.3559021

[75] Sheng Qi, Xuanzhe Liu, and Xin Jin. 2023. Halfmoon: Log-Optimal Fault-Tolerant Stateful Serverless Computing. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP '23)*. Association for Computing Machinery, Koblenz, Germany, 314–330. doi:10.1145/3600006.3613154

[76] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. 2012. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing* (San Jose, California) *(SoCC '12)*. Association for Computing Machinery, New York, NY, USA, Article 7, 13 pages. doi:10.1145/2391229.2391236

[77] Francisco Romero, Gohar Irfan Chaudhry, Íñigo Goiri, Pragna Gopa, Paul Batum, Neeraja J. Yadwadkar, Rodrigo Fonseca, Christos Kozyrakis, and Ricardo Bianchini. 2021. Faa$T: A Transparent Auto-Scaling Cache for Serverless Applications. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '21)*. Association for Computing Machinery, Seattle, WA, USA, 122–137. doi:10.1145/3472883.3486974

[78] 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 (ASPLOS '22). Association for Computing Machinery, Lausanne, Switzerland, 753–767. doi:10.1145/3503222.3507750

[79] G. Safaryan, A. Jindal, M. Chadha, and M. Gerndt. 2022. SLAM: SLO-Aware Memory Optimization for Serverless Applications. In *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*. IEEE Computer Society, Los Alamitos, CA, USA, 30–39. doi:10.1109/CLOUD55607.2022.00019

[80] Alireza Sahraei, Soteris Demetriou, Amirali Sobhgol, Haoran Zhang, Abhigna Nagaraja, Neeraj Pathak, Girish Joshi, Carla Souza, Bo Huang, Wyatt Cook, Andrii Golovei, Pradeep Venkat, Andrew Mcfague, Dimitrios Skarlatos, Vipul Patel, Ravinder Thind, Ernesto Gonzalez, Yun Jin, and Chunqiang Tang. 2023. XFaaS: Hyperscale and Low Cost Serverless Functions at Meta. In *Proceedings of the 29th Symposium on Operating Systems Principles* (, Koblenz, Germany,) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 231–246. doi:10.1145/3600006.3613155

[81] Divyanshu Saxena, Tao Ji, Arjun Singhvi, Junaid Khalid, and Aditya Akella. 2022. Memory deduplication for serverless computing with Medes. In *Proceedings of the Seventeenth European Conference on Computer Systems* (Rennes, France) *(EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 714–729. doi:10.1145/3492321.3524272

[82] David Schall, Artemiy Margaritov, Dmitrii Ustiugov, Andreas Sandberg, and Boris Grot. 2022. Lukewarm Serverless Functions: Characterization and Optimization. In *Proceedings of the 49th Annual International Symposium on Computer Architecture (ISCA '22)*. Association for Computing Machinery, New York, New York, 757–770. doi:10.1145/3470496.3527390

[83] Ozgur Sedefoglu and Hasan Sozer. 2021. Cost minimization for deploying serverless functions. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing* (Virtual Event, Republic of Korea) *(SAC '21)*. Association for Computing Machinery, New York, NY, USA, 83–85. doi:10.1145/3412841.3442069

[84] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 205–218. https://www.usenix.org/conference/atc20/presentation/shahrad

[85] Wonseok Shin, Wook-Hee Kim, and Changwoo Min. 2022. Fireworks: A Fast, Efficient, and Safe Serverless Framework Using VM-Level Post-JIT Snapshot. In *Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys '22)*. Association for Computing Machinery, Rennes, France, 663–677. doi:10.1145/3492321.3519581

[86] Arjun Singhvi, Arjun Balasubramanian, Kevin Houck, Mohammed Danish Shaikh, Shivaram Venkataraman, and Aditya Akella. 2021. Atoll: A Scalable Low-Latency Serverless Platform. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SoCC '21)*. Association for Computing Machinery, New York, NY, USA, 138–152. doi:10.1145/3472883.3486981

[87] Jovan Stojkovic, Chloe Alverti, Alan Andrade, Nikoleta Iliakopoulou, Hubertus Franke, Tianyin Xu, and Josep Torrellas. 2025. Concord: Rethinking Distributed Coherence for Software Caches in Serverless Environments. In *Proceedings of the 31th IEEE International Symposium on High-Performance Computer Architecture (HPCA-31)*.

[88] Jovan Stojkovic, Nikoleta Iliakopoulou, Tianyin Xu, Hubertus Franke, and Josep Torrellas. 2024. EcoFaaS: Rethinking the Design of Serverless Environments for Energy Efficiency. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 471–486. doi:10.1109/ISCA59077.2024.00042

[89] Jovan Stojkovic, Tianyin Xu, Hubertus Franke, and Josep Torrellas. 2023. MXFaaS: Resource Sharing in Serverless Environments for Parallelism and Efficiency. In *Proceedings of the 50th Annual International Symposium on Computer Architecture* (Orlando, FL, USA) *(ISCA '23)*. Association for Computing Machinery, New York, NY, USA, Article 34, 15 pages. doi:10.1145/3579371.3589069

[90] Jovan Stojkovic, Tianyin Xu, Hubertus Franke, and Josep Torrellas. 2023. SpecFaaS: Accelerating Serverless Applications with Speculative Function Execution. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 814–827. doi:10.1109/HPCA56546.2023.10071120

[91] Bo Tan, Haikun Liu, Jia Rao, Xiaofei Liao, Hai Jin, and Yu Zhang. 2020. Towards Lightweight Serverless Computing via Unikernel as a Function. In *2020 IEEE/ACM 28th International Symposium on Quality of Service (IWQoS)*. 1–10. doi:10.1109/IWQoS49365.2020.9213020

[92] Jörg Thalheim, Pramod Bhatotia, Pedro Fonseca, and Baris Kasikci. 2018. Cntr: Lightweight OS Containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 199–212. https://www.usenix.org/conference/atc18/presentation/thalheim

[93] Huangshi Tian, Suyi Li, Ao Wang, Wei Wang, Tianlong Wu, and Haoran Yang. 2022. Owl: Performance-Aware Scheduling for Resource-Efficient Function-as-a-Service Cloud. In *Proceedings of the 13th Symposium on Cloud Computing* (San Francisco, California) *(SoCC '22)*. Association for Computing Machinery, New York, NY, USA, 78–93. doi:10.1145/3542929.3563470

[94] Muhammad Tirmazi, Adam Barker, Nan Deng, Md Ehtesam Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. 2020. Borg: the Next Generation. In *EuroSys'20*. Heraklion, Crete.

[95] Dmitrii Ustiugov, Shyam Jesalpura, Mert Bora Alper, Michal Baczun, Rustem Feyzkhanov, Edouard Bugnion, Boris Grot, and Marios Kogias. 2023. Expedited Data Transfers for Serverless Clouds. arXiv:2309.14821 [cs.DC]

[96] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. 2021. Benchmarking, Analysis, and Optimization of Serverless Function Snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'21)*. ACM. doi:10.1145/3445814.3446714

[97] Ao Wang, Shuai Chang, Huangshi Tian, Hongqi Wang, Haoran Yang, Huiba Li, Rui Du, and Yue Cheng. 2021. FaaSNet: Scalable and Fast Provisioning of Custom Serverless Container Runtimes at Alibaba Cloud Function Compute. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 443–457. https://www.usenix.org/conference/atc21/presentation/wang-ao

[98] Yawen Wang, Kapil Arya, Marios Kogias, Manohar Vanga, Aditya Bhandari, Neeraja J. Yadwadkar, Siddhartha Sen, Sameh Elnikety, Christos Kozyrakis, and Ricardo Bianchini. 2021. SmartHarvest: Harvesting Idle CPUs Safely and Efficiently in the Cloud. In *Proceedings of the 16th European Conference on Computer Systems (EuroSys)*. https://www.microsoft.com/en-us/research/publication/smartharvest-harvesting-idle-cpus-safely-and-efficiently-in-the-cloud/

[99] Yilun Wang, Pengfei Chen, Hui Dou, Yiwen Zhang, Guangba Yu, Zilong He, and Haiyu Huang. 2024. FaaSConf: QoS-aware Hybrid Resources Configuration for Serverless Workflows. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering* (Sacramento, CA, USA) *(ASE '24)*. Association for Computing Machinery, New York, NY, USA, 957–969. doi:10.1145/3691620.3695477

[100] Ben Wegbreit. 1975. Mechanical Program Analysis. *Commun. ACM* 18, 9 (sep 1975), 528–539. doi:10.1145/361002.361016

[101] Xingda Wei, Fangming Lu, Tianxia Wang, Jinyu Gu, Yuhan Yang, Rong Chen, and Haibo Chen. 2023. No Provisioned Concurrency: Fast RDMA-codesigned Remote Fork for Serverless Computing. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 497–517. https://www.usenix.org/conference/osdi23/presentation/wei-rdma

[102] Zhaojie Wen, Yishuo Wang, and Fangming Liu. 2022. StepConf: SLO-Aware Dynamic Resource Configuration for Serverless Function Workflows. In *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications*. 1868–1877. doi:10.1109/INFOCOM48880.2022.9796962

[103] Yanan Yang, Laiping Zhao, Yiming Li, Shihao Wu, Yuechan Hao, Yuchi Ma, and Keqiu Li. 2024. Flame: A Centralized Cache Controller for Serverless Computing. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4* (Vancouver, BC, Canada) *(ASPLOS '23)*. Association for Computing Machinery, New York, NY, USA, 153–168. doi:10.1145/3623278.3624769

[104] 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. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 335–350. doi:10.1145/3617232.3624871

[105] Hanfei Yu, Athirai A. Irissappane, Hao Wang, and Wes J. Lloyd. 2021. FaaSRank: Learning to Schedule Functions in Serverless Platforms. In *2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. 31–40. doi:10.1109/ACSOS52086.2021.00023

[106] Hanfei Yu, Hao Wang, Jian Li, Xu Yuan, and Seung-Jong Park. 2024. Freyr $^+$+: Harvesting Idle Resources in Serverless Computing via Deep Reinforcement Learning . *IEEE Transactions on Parallel & Distributed Systems* 35, 11 (Nov. 2024), 2254–2269. doi:10.1109/TPDS.2024.3462294

[107] Haoran Zhang, Shuai Mu, Sebastian Angel, and Vincent Liu. 2025. CausalMesh: A Causal Cache for Stateful Serverless Computing. *Proceedings of the VLDB Endowment* (2025). https://par.nsf.gov/biblio/10534194

[108] Hong Zhang, Yupeng Tang, Anurag Khandelwal, Jingrong Chen, and Ion Stoica. 2021. Caerus: NIMBLE Task Scheduling for Serverless Analytics. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 653–669. https://www.usenix.org/conference/nsdi21/presentation/zhang-hong

[109] 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)*. Association for Computing Machinery, New York, NY, USA, 724–739. doi:10.1145/3477132.3483580

[110] Laiping Zhao, Yanan Yang, Yiming Li, Xian Zhou, and Keqiu Li. 2021. Understanding, predicting and scheduling serverless workloads under partial interference. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 22, 15 pages.

doi:10.1145/3458817.3476215

[111] Chao Zheng, Lukas Rupprecht, Vasily Tarasov, Douglas Thain, Mohamed Mohamed, Dimitrios Skourtis, Amit S. Warke, and Dean Hildebrand. 2018. Wharf: Sharing Docker Images in a Distributed File System. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '18)*. Association for Computing Machinery, Carlsbad, CA, USA, 174–185. doi:10.1145/3267809.3267836

[112] 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* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 1–14. doi:10.1145/3567955.3567960