Metronome: Differentiated Delay Scheduling for Serverless Functions

Zhuangbin Chen
Sun Yat-sen University
Zhuhai, China
chenzhb36@mail.sysu.edu.cn

Juzheng Zheng
Sun Yat-sen University
Zhuhai, China
zhengjzh8@mail2.sysu.edu.cn

Zibin Zheng*
Sun Yat-sen University
Zhuhai, China
zhzibin@mail.sysu.edu.cn

Abstract

Function-as-a-Service (FaaS) computing is an emerging cloud computing paradigm for its ease-of-management and elasticity. However, optimizing scheduling for serverless functions remains challenging due to their dynamic and event-driven nature. While data locality has been proven effective in traditional cluster computing systems through delay scheduling, its application in serverless platforms remains largely unexplored. In this paper, we systematically evaluate existing delay scheduling methods in serverless environments and identify three key observations: 1) delay scheduling benefits vary significantly based on function input characteristics; 2) serverless computing exhibits more complex locality patterns than cluster computing systems, encompassing both data locality and infrastructure locality; and 3) heterogeneous function execution times make rule-based delay thresholds ineffective. Based on these insights, we propose Metronome, a differentiated delay scheduling framework that employs predictive mechanisms to identify optimal locality-aware nodes for individual functions. Metronome leverages an online Random Forest Regression model to forecast function execution times across various nodes, enabling informed delay decisions while preventing SLA violations. Our implementation on OpenLambda shows that Metronome significantly outperforms baselines, achieving 64.88%–95.83% reduction in mean execution time for functions, while maintaining performance advantages under increased concurrency levels and ensuring SLA compliance.

CCS Concepts

• **Computer systems organization → Cloud computing**; • **Software and its engineering → Scheduling**.

Keywords

Serverless Computing, Delay Scheduling, Performance Prediction, Online Random Forest Regression

ACM Reference Format:
Zhuangbin Chen, Juzheng Zheng, and Zibin Zheng. 2026. Metronome: Differentiated Delay Scheduling for Serverless Functions. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3744916.3773239

*Zibin Zheng is the corresponding author.

ICSE '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2025-3/2026/04
https://doi.org/10.1145/3744916.3773239

1 Introduction

Function-as-a-Service (FaaS) computing offers key benefits including reduced operational overhead, automatic scalability, and cost efficiency via pay-per-use pricing [5, 8, 34]. However, the dynamic and event-driven nature of serverless workloads presents unique challenges for their scheduling [28, 46]. Recent studies [1, 7, 25, 33] have shown that *data locality* between data-intensive functions plays a critical role in optimizing resource utilization. For example, routing subsequent requests from the same user to the same function instance can significantly improve cache effectiveness, while scheduling data analytics tasks that consume each other's outputs on the same instance can minimize network data transfer overhead.

In traditional cluster computing systems such as Hadoop and Dryad, previous work exploits and enforces data locality through the strategy of *delay scheduling* [63]. The principle of delay scheduling involves intentionally postponing the execution of tasks until nodes that contain the required data become available. Specifically, when a node requests a task, if the head-of-line job cannot launch a local task, the scheduler will skip it and examine the subsequent jobs. If a job has been skipped long enough, it will be permitted to launch non-local tasks to avoid starvation. The key insight is that tasks typically complete quickly, so the local nodes with the required data are likely to be freed up within a few seconds. This technique has been shown to significantly minimize data transfer overhead and improve response times. Similar to the tasks in cluster computing systems, serverless functions also benefit from data locality and typically have short execution times (ranging from milliseconds to a few seconds). While Meta's XFaaS platform [43] employs "time-shifting" techniques to defer delay-tolerant functions to off-peak hours, the primary objective is capacity management and cost reduction, which require developers to manually annotate functions as non-critical or specify execution deadlines. In contrast, effective locality-aware scheduling requires automated, real-time decisions.

To fill this significant gap, in this work we systematically evaluate the performance of existing delay scheduling methods [38, 43, 58, 63] in serverless environments. Specifically, we deploy four widely-used serverless applications from FunctionBench [24] and generate workloads based on production traffic [22]. Through extensive experiments, we find that the unique computing paradigm and intricate execution patterns of serverless functions render existing approaches ineffective. Particularly, we have identified three key observations that serve as essential design principles for optimizing delay scheduling in serverless environments:

- *Delay scheduling is not universally beneficial, depending on the input characteristics of individual functions.* Specifically, functions with large input data sizes benefit significantly from data locality, as it minimizes substantial data transfer overhead,

while those with smaller inputs show minimal performance gains. Existing work fails to account for this critical factor, leaving a gap in identifying which functions are truly suitable candidates for delay scheduling techniques.

- *The locality characteristics in serverless computing exhibit greater complexity compared to traditional systems.* Unlike traditional cluster computing systems where data locality is the primary concern, we identify another critical type of locality, i.e., *infrastructure locality*, which refers to the reuse of warm containers and cached dependencies across function invocations on the same node. Moreover, functions with different characteristics benefit from different forms of locality.
- *The heterogeneity in function execution times makes it challenging to determine the appropriate delay thresholds.* While most serverless functions have short execution times, there exists a non-negligible portion of long-running functions that can take tens of seconds to minutes to complete. Employing a rule-based delay strategy without considering these variations could result in SLA violations, particularly for time-sensitive functions.

To address these limitations, we propose Metronome, which employs differentiated delay scheduling based on specific function characteristics, in contrast to existing one-size-fits-all solutions. At its core, Metronome leverages predictive delay mechanisms to identify optimal locality-aware nodes (data versus infrastructure) for individual functions (with varying input data sizes and dependency requirements). Specifically, Metronome employs an online Random Forest Regression (RFR) model that forecasts function execution times across various nodes based on comprehensive feature sets, including function attributes, node metrics, and historical execution data. This not only enables us to compare the performance benefits of different delay decisions but also helps determine the maximum allowable delay duration to prevent SLA violations. The online learning capability of the prediction model also facilitates it to rapidly adapt to new function characteristics and system dynamics. We have implemented Metronome on OpenLambda [18, 37] and compared its performance against traditional delay scheduling approaches. The results demonstrate that Metronome can significantly improve the performance of function execution over baselines, while avoiding SLA violations.

The major contributions of this work are summarized as follows:

- A comprehensive study that reveals the limitations of existing static, rule-based delay scheduling techniques in serverless environments, which establishes essential design principles for optimizing delay scheduling for serverless functions.
- A differentiated delay scheduling framework that determines whether a function invocation should be delayed and the optimal locality type. This is achieved by an execution time prediction model with online learning capabilities. It is able to adaptively adjust delay thresholds to optimize locality benefits while ensuring SLA compliance.
- A practical implementation[1] on OpenLambda [18, 37] that leverages gRPC and server-side streaming for efficient communication between components, enabling asynchronous model inference, online learning, and system metrics collection through server push notifications. These operations are offloaded from the critical path of function execution while maintaining low-latency scheduling and prediction accuracy.

[1] https://github.com/OpsPAI/Metronome

2 Background and Motivation

2.1 Delay Scheduling

Delay scheduling was first introduced by Zaharia et al. [63] to address the conflict between data locality and scheduling fairness in traditional big data processing systems like Hadoop. The core idea is to temporarily delay a task's scheduling when its preferred node (typically the one containing its input data) is unavailable, allowing other tasks to be executed first. Such an approach has proven highly effective in cluster computing environments, where data transfer costs often outweigh waiting time. In particular, this simple mechanism is able to achieve nearly optimal data locality while maintaining fair sharing in Hadoop clusters.

This delay scheduling strategy exhibits promising potential in serverless computing environments for several reasons. First, the majority of serverless functions have short execution times (typically milliseconds to seconds), which means even brief scheduling delays could create sufficient opportunities for better locality without significantly impacting overall latency. Second, serverless platforms often manifest high concurrency with multiple function instances competing for resources, making it feasible to find alternative tasks to execute during delay periods. Third, the pay-per-use pricing model of serverless computing makes performance optimization particularly valuable, as even small improvements in execution time can lead to significant cost savings at scale.

However, adapting delay scheduling to serverless environments presents unique challenges that warrant investigation. The fundamental differences between traditional batch processing systems and serverless platforms, such as the presence of cold starts, strict resource isolation requirements, and highly variable function characteristics, raise questions about the direct applicability of the original delay scheduling approach. To better understand these challenges and evaluate the potential benefits, we conduct a series of experiments examining how delay scheduling performs in serverless scenarios with different workload characteristics.

2.2 An Empirical Study of Delay Scheduling in Serverless Functions

To understand the effectiveness and limitations of traditional delay scheduling in serverless environments, we conduct a comprehensive study using four representative serverless applications from FunctionBench [24], a widely-used benchmark suite [15, 53, 65]. Table 1 provides a detailed overview, including the functions of each application, the number of their dependencies, and descriptions.

- *Video Processing*: An application that splits input videos, performs parallel transcoding of video segments, and merges the processed segments. Each step generates intermediate results that need to be transferred between functions.
- *Log Analysis*: A service that splits log files, performs parallel analysis to compute per-IP access frequencies from log segments, and aggregates the results. The workflow typically processes log files and generates intermediate analysis results between processing stages.

Table 1: Descriptions of benchmark functions

Application	Functions	Deps Count	Description
Video Processing	video-split	5	Splits video into segments
	video-transcode	4	Transcodes segments
	video-merge	4	Merges segments
Log Analysis	log-split	0	Splits log files
	log-analyze	3	Computes IP frequencies
	log-merge	2	Aggregates results
Doc Conversion	any2md-validate	2	Format validation & prep.
	any2md-process	34	Doc to markdown conv.
ML Inference	ml-normalize	4	Data normalization
	ml-process	17	Model prediction

- *Document Conversion*: A function that transforms various document formats to markdown. While the input and output files are relatively small, the function requires several document processing libraries and runtime dependencies.
- *ML inference*: A service that performs model-based prediction. It depends on multiple ML frameworks that need to be loaded into the runtime environment.

These applications are carefully selected to represent common serverless workload patterns observed in production scenarios, i.e., data processing pipelines (video processing and log analysis), standalone compute-intensive tasks (document conversion), and machine learning services (ML inference). Moreover, they exhibit diverse characteristics in terms of execution duration (ranging from milliseconds to minutes), resource requirements (CPU, memory, and I/O patterns), and dependency complexity (from lightweight libraries to heavy ML frameworks), making them ideal for evaluating scheduling strategies under different scenarios.

To ensure a realistic evaluation, we generate workload patterns based on production cloud serverless traces [22], which contain over 1.4 trillion invocations across more than 5,000 functions. Our workload generation process systematically extracts and synthesizes key characteristics from production environments. Specifically, we analyze the traces to extract key metrics including request volumes, invocation frequencies, inter-arrival time distributions, and function execution patterns. We calculate statistical properties such as arrival rate distributions (from 0.1 to 50 requests per second), function popularity patterns (following power-law distributions), and temporal correlations between dependent functions in workflows. The workload generator then synthesizes these characteristics to create realistic request streams that preserve the essential properties of production workloads while adapting to our experimental applications. For each application, we conduct 100 independent end-to-end executions and record their execution times.

In our experiments, we compare two scheduling strategies:

- *Basic Data-locality Scheduling*: A basic locality-aware scheduling strategy that considers data locality but without delay mechanisms. When a function reaches the head of the scheduling queue, the scheduler first checks available nodes that contain relevant input data (i.e., nodes where the predecessor functions in the workflow were executed). If such nodes exist, the scheduler selects one based on current load conditions. If no nodes with data locality are available, the function is scheduled immediately to non-local nodes.

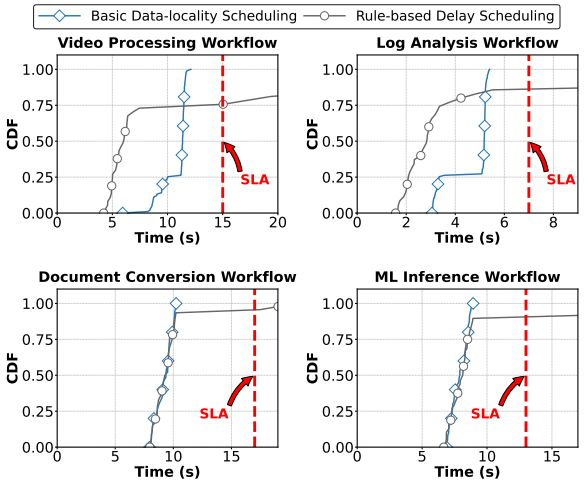


Figure 1: CDF of function execution time

- *Rule-based Delay Scheduling*: An implementation of traditional delay scheduling [38, 58, 63]. When a function reaches the head of the queue, if no data-locality nodes are available, instead of immediately scheduling to any available node, the function is marked for delay and remains in the queue. The scheduler records the delay start time and continues to check for nodes with data locality whenever resources are released. The delay timeout is calculated based on our serverless environment characteristics and workload patterns, following the rules in [63]. If the delay exceeds this calculated timeout, the function is scheduled to the next available node regardless of locality.

The results reveal interesting and sometimes counter-intuitive patterns, as illustrated in Figure 1. The cumulative distribution function (CDF) shows the proportion of function runs with an execution time less than or equal to a given value. A curve that rises more steeply and is shifted to the left indicates better performance. The video processing shows mixed performance results with a modest reduction (3.9%) in mean execution time but a more substantial reduction (47.1%) in median time. Similarly, the log analysis achieves a 2.7% reduction in mean execution time and a 45.3% in median time. However, this comes at the cost of significantly degraded tail latencies, i.e., the video processing's 95th and 99th percentile latencies increase by 142.9% and 151.7%, respectively, while the log analysis shows even more severe degradation with increases of 202.6% and 228.1%. Approximately 10.2% and 9.8% of function invocations result in SLA violations for video and log workflows, respectively. For the other two functions, the results are consistently negative. The document conversion shows increased latencies across all metrics, i.e., 8.5% higher mean, 2.1% higher median, and 51.3% higher 95th percentile, with 6% SLA violations. The ML inference exhibits even more severe degradation, i.e., 17.6% higher mean, 4.4% higher median, and 118.4% higher 95th percentile, with 10% SLA violations.

While we observe some potential benefits in median cases, the significant tail latency degradation and SLA violations across all function types indicate that a simple delay scheduling is insufficient. To better understand the results, we conduct a deeper analysis of the underlying patterns and identify three key observations.

Observation 1: The effectiveness of data-locality-based delay scheduling is highly dependent on the input data size. Data-intensive workloads tend to benefit from data locality optimization, while smaller workloads often demonstrate minimal improvement or even performance degradation.

Our results show that the effectiveness of data-locality scheduling directly correlates with function input size. Workflows such as video processing and log analysis generally involve larger input data sizes, with median sizes ranging from tens of megabytes to several gigabytes. These workflows exhibit notable benefits from data locality optimization. In contrast, functions like document conversion and ML inference typically process smaller inputs, with median sizes of only a few megabytes. For these functions, delay scheduling often yields near-zero improvement or even performance degradation. This pattern persists even in outlier scenarios. When video and log processing functions occasionally handle smaller input sizes, they also gain minimal benefits from delay scheduling, similar to the other applications. This consistent correlation between input size and scheduling effectiveness confirms that when data movement overhead is small, the potential benefits from data locality optimization become negligible or are outweighed by other factors. That is, the delay scheduling introduces unnecessary waiting time without compensating benefits in reduced data transfer overhead.

The observed correlation between input data size and delay scheduling effectiveness can be explained by examining the underlying data movement mechanisms. As illustrated in Figure 2, when functions are scheduled across different nodes, data transfer between workflow stages must occur over the network, introducing significant latency overhead for intermediate results. This can become a major performance bottleneck, especially for data-intensive applications. In contrast, Figure 3 demonstrates how co-locating functions on the same node minimizes data transfer overhead through efficient local communication channels such as shared memory [32, 33], Unix domain sockets [21, 27], eBPF-based architectures [40], and local filesystem access [21]. This optimization is particularly beneficial for applications that process and transfer substantial amounts of intermediate data.

Based on these observations, we conclude that serverless functions may not benefit from data locality-aware scheduling in all cases. Therefore, before applying delay scheduling, we need to carefully assess whether it would actually benefit the function's performance, given its characteristics and current system conditions. This evaluation requires considering factors like input data size and potential performance gains from data locality.

Observation 2: In serverless computing, while data locality remains crucial for workflow-based applications, infrastructure locality emerges as another critical dimension due to the container-based execution model and package dependency management.

Our experiments with document conversion and ML inference reveal a pattern that challenges the traditional focus on data locality in distributed systems. Despite having minimal data transfer requirements, these functions consistently show degraded performance under data locality-focused delay strategies. Further investigation reveals a critical dimension of locality unique to serverless

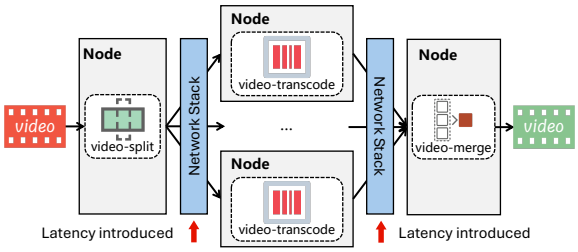


Figure 2: Video processing workflow without data locality

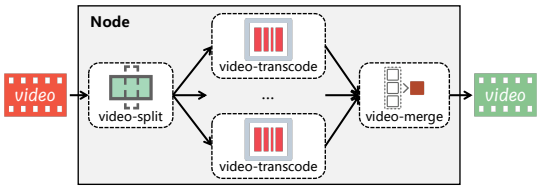


Figure 3: Video processing workflow with data locality

computing. The performance degradation can be attributed to the container-based execution model and complex package dependency management inherent in serverless platforms. The document conversion, requiring 34 packages, shows increased latencies across all metrics, while the ML inference, with 17 dependencies, exhibits even more severe degradation. When these dependency-heavy functions are scheduled purely based on data locality considerations, they may miss opportunities to reuse existing infrastructure components that could significantly improve their performance.

This infrastructure reuse (which we term infrastructure locality) can manifest in multiple forms in serverless environments, e.g., warm containers, zygote processes [13, 29], and pre-installed package dependencies. Previous studies like SOCK [37] and RainbowCake [62] have demonstrated that serverless functions can benefit from different levels of such infrastructure reuse. The most beneficial form is the reuse of warm containers that have already been initialized with the function code and runtime environments [26, 44, 48, 49]. When warm containers are not available, nodes with cached package dependencies can still offer significant benefits by avoiding time-consuming package installations.

The importance of infrastructure reuse is particularly evident in functions like document conversion and ML inference, where the overhead of container initialization and package importing can dominate the execution time. These functions, with their numerous package dependencies but relatively small data transfer requirements, demonstrate that optimal scheduling decisions in serverless environments must consider more than just data placement. In contrast, video processing and log analysis functions, which have significantly fewer dependencies in our experimental setup (as shown in Table 1), do not suffer the same performance degradation under delay scheduling, even when processing smaller input sizes.

Observation 3: Function execution times exhibit high variability and heterogeneity, rendering rule-based delay thresholds impractical. Optimal delay decisions must consider both current system conditions and historical execution patterns, and should be dynamically determined in a predictive way.

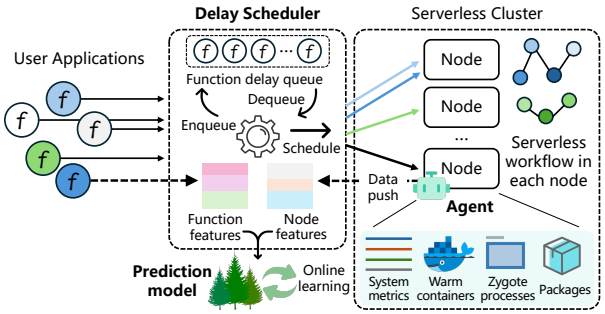


Figure 4: Overall architecture of Metronome

Our results with rule-based delay scheduling reveal significant challenges in determining appropriate delay thresholds. The execution times of our benchmark functions are highly variable due to multiple dynamic factors. Node-level conditions such as CPU utilization, memory pressure, and I/O contention significantly impact function performance. Function-specific characteristics like input data size, container warmth state, and package dependency availability introduce further variability, as evident from the previous two observations. For instance, the video processing application's execution time could vary by up to 151.7% at the 99th percentile, while the ML inference service shows even greater variation of up to 174.9%. Furthermore, our experiments show that the time required for target nodes to free up resources is also highly unpredictable. While nodes often release resources quickly, improving median execution times, excessive waiting times can occur. This results in substantial SLA violations, i.e., 6% for document conversion and around 10% for other applications. This is because the heuristic delay strategy waits for an inappropriate time for preferred nodes to become available, without considering either the variable function execution times or the unpredictable resource release patterns.

This dual variability makes it impractical to use traditional, rule-based delay thresholds based on the simple setting in [63]. This calls for an adaptive delay duration that needs to be dynamically determined based on both factors, i.e., how long the function is likely to take to execute (which varies based on current system conditions and function characteristics) and how long we might need to wait for resources to become available on the preferred nodes. A heuristic threshold [63] cannot capture this complex interplay of factors and may lead to either pre-mature scheduling (missing locality benefits) or excessive delays (causing SLA violations).

3 Methodology

3.1 Overview

Based on our observations, we identify three key design principles. First, effective delay scheduling should consider the unique input characteristics, as smaller inputs may compromise data-locality benefits. Second, functions exhibit varying sensitivities to different locality types, i.e., functions that transfer large data volumes benefit from data locality, whereas dependency-heavy functions gain more from infrastructure locality. Third, the substantial heterogeneity in function execution times renders rule-based delay thresholds ineffective. We need to balance the tradeoff between waiting for preferred nodes and meeting SLA compliance.

Building on these insights, we propose Metronome, a differentiated scheduling framework that uses per-function execution time predictions to guide delay decisions. The prediction capability enables us to assess whether delaying a function is beneficial, identify the most suitable locality type, and dynamically determine optimal delay thresholds. As shown in Figure 4, Metronome integrates *function and node profiling*, *execution time prediction*, and *differentiated delay scheduling*, performed by three components: 1) distributed data agents deployed on each node that continuously collect and push system metrics, warm container status, zygote process information, and package dependency data to the scheduler; 2) an online learning-enabled prediction model that estimates execution times across different nodes to adaptively determine optimal delay thresholds; and 3) a delay scheduler that maintains function characteristics, manages function queuing, and scheduling delay decisions. The scheduler identifies the most suitable locality strategy for each function and determines whether delaying for locality benefits outweighs costs under current system conditions.

3.2 Function and Node Profiling

To enable accurate execution time prediction, Metronome implements a comprehensive profiling system that collects both function characteristics and node-level metrics. The goal is to capture all relevant factors that may impact function execution times, including both data locality and infrastructure locality [5, 34, 56]. While node-level metrics are gathered by distributed agents on each worker node and streamed to the scheduler, function-specific metrics are collected by the scheduler itself when functions arrive for scheduling, providing a complete picture for real-time decision making.

For node-level profiling, distributed agents continuously collect metrics that are essential for capturing infrastructure locality potential and its impact on cold start latency [3, 4, 37]. Specifically, we track the number of warm containers available for each function type, which eliminates container initialization overhead when reused [37, 56]. The agents also monitor active zygote processes, which serve as pre-initialized runtime environments that can be quickly forked to create new function instances, substantially reducing startup times. Additionally, we collect package dependency cache status and pre-loaded library information, as these directly affect dependency resolution time, which can dominate execution time for dependency-heavy functions [14, 47, 55]. Besides container and dependency state metrics, agents also report system resource metrics to provide context on each node's current operational environment [61]. For example, CPU utilization, load averages, and scheduling latencies help assess potential contention [50, 64]. Network metrics, including bandwidth utilization and packet rates, are important for functions that perform significant data transfers or API calls [33, 55]. Disk I/O metrics help predict performance for storage-intensive operations [33, 67]. These system metrics are collected at regular intervals and streamed to the scheduler.

For function-level characteristics, the scheduler itself collects and maintains relevant information when functions arrive for scheduling. Input data size is recorded as it directly correlates with data transfer times and often with processing requirements [24]. The scheduler maintains a historical record of elapsed times for key execution phases (data transfer, container initialization, and actual

function execution) from previous invocations, which serves as valuable training data for our prediction model. For functions that are part of larger workflows, we track dependency information to enable co-location of related functions when beneficial. The scheduler also records required package dependencies for each function, which helps identify nodes with matching pre-loaded dependencies.

All these metrics are continuously updated in the scheduler's state cache, providing a real-time view of both infrastructure conditions and function characteristics. This comprehensive profiling enables our prediction model to accurately estimate execution times across different nodes, considering both the current system state and function-specific requirements.

3.3 Function Execution Time Prediction

Accurate prediction of function execution time is essential for effective delay scheduling. We model this problem as a regression task, aiming to learn a mapping from the feature space to the function execution time. In designing the prediction model, we address two key challenges: 1) capturing the diverse factors influencing execution time, such as node status, function characteristics, and locality features; and 2) ensuring the model is both efficient and adaptable to evolving workload patterns and system dynamics.

To this end, we employ online Random Forest Regression (RFR) to build our model [45, 67]. RFR provides robust estimates by averaging predictions from multiple decision trees built with bootstrap sampling and random feature selection [19]. Unlike linear regression, which assumes linear relationships between features and execution times, RFR automatically captures complex, non-linear performance dynamics inherent to serverless environments without manual feature engineering. This involves intricate relationships between system metrics, function characteristics, and performance outcomes. The choice of RFR over deep learning methods is based on the following considerations: while deep learning may offer higher accuracy, it typically demands more training samples and significantly more computational resources, making it less efficient for real-time environments; RFR is not only simple to implement but also suitable for handling high-dimensional, multi-type continuous variables and is highly robust against overfitting [17, 41, 51]. Our experimental results demonstrate that RFR outperforms other representative machine learning algorithms and deep learning models in terms of prediction accuracy and computational efficiency.

Our prediction model can be formally represented as learning a mapping function $f : \mathcal{X} \rightarrow \mathbb{R}^+$ from the feature space $\mathcal{X}$ to the positive real-valued execution time. Given n observed samples $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$, where $\mathbf{x}_i \in \mathcal{X}$ represents the feature vector and $y_i \in \mathbb{R}^+$ represents the corresponding execution time, our goal is to minimize the prediction error:

$$\min_f \sum_{i=1}^{n} \mathcal{L}(f(\mathbf{x}_i), y_i) + \lambda\varphi(f) \tag{1}$$

where $\mathcal{L}(\cdot,\cdot)$ is the loss function measuring prediction accuracy, and $\varphi(f)$ is the regularization term with hyperparameter λ used to prevent overfitting. The feature space $\mathcal{X}$ contains a comprehensive feature set that captures system status and function characteristics:

$$\mathbf{x}_i = [\mathbf{s}_i, \mathbf{c}_i, \mathbf{n}_i, \mathbf{d}_i] \tag{2}$$

where $\mathbf{s}_i$, $\mathbf{c}_i$, $\mathbf{n}_i$, and $\mathbf{d}_i$ represent system resource metrics, container status metrics, network metrics, and function characteristic metrics, respectively. Specifically, $\mathbf{s}_i$ includes metrics such as CPU utilization, memory usage, disk I/O, and system load, which directly affect computing performance and the degree of resource contention. $\mathbf{c}_i$ captures the status of containers and dependencies, including the number of available warm containers, active zygote processes, and cached dependencies. These metrics are crucial for assessing infrastructure locality, as they directly impact cold start latency and dependency resolution time. $\mathbf{n}_i$ records metrics such as network bandwidth utilization, packet rate, and network latency, which are particularly important for data transfer-intensive functions. $\mathbf{d}_i$ includes function-specific features such as input data size, the number of required dependencies, and historical execution time. These features help the model distinguish between different types of functions and their sensitivity to various types of locality.

Our RFR model constructs an ensemble of decision trees $\{h_k\}_{k=1}^K$, with the final prediction obtained by averaging:

$$f(\mathbf{x}) = \frac{1}{K} \sum_{k=1}^{K} h_k(\mathbf{x}) \tag{3}$$

For each scheduling decision, the model provides an execution time estimate for each candidate node by processing the combined feature vector. These predictions serve as crucial inputs to the scheduler's delay decision-making process, allowing it to compare expected execution times across different nodes.

To capture the dynamics of the serverless environment, we employ an online learning mechanism to incrementally update the model. The online RFR model follows a two-phase training approach. It is initially trained with a basic set of historical execution data (including function/system features and the corresponding function execution times) to establish baseline prediction capabilities. During system operation, Metronome continuously collects new observations to incrementally update the model through online learning [67]. The update process is automatically triggered when the number of new observations reaches a predefined threshold. This adaptive approach enables the model to effectively handle the dynamic characteristics of the serverless environment without the overhead of complete retraining. Together, these techniques, i.e., function and system profiling, ensemble learning, and online model updating, yield accurate execution time predictions, which are used to guide Metronome's delay scheduling decisions.

3.4 Differentiated Delay Scheduling

Based on our analysis of locality benefits and SLA constraints, we now present Metronome's differentiated delay scheduling (Algorithm 1). The core principle is to evaluate whether delay scheduling is beneficial for each function by considering both locality types and execution time predictions. Our algorithm combines continuous delay monitoring with event-driven scheduling, which is triggered by function arrivals, resource releases, and SLA violation signals from the delay monitor. Metronome implements these principles through systematic *node classification* and *adaptive delay decision-making* that maximize locality benefits with SLA compliance.

Algorithm 1: Differentiated Delay Scheduling

Input: Function γ, Node set $\mathcal{N}$, SLA threshold θ
Output: Scheduling decision for function γ

1 **Function** `DelayScheduling`(γ, $\mathcal{N}$, θ)**:**
2 $\quad\mathcal{N}_l, \mathcal{N}_f \leftarrow$ ClassifyNodes(γ, $\mathcal{N}$);
3 $\quad\mathcal{T} \leftarrow$ PredictExecutionTime(γ, $\mathcal{N}$);
4 $\quad\mathcal{T}_l, \mathcal{T}_f \leftarrow \min_{node\in\mathcal{N}_l} \mathcal{T}[node], \min_{node\in\mathcal{N}_f} \mathcal{T}[node]$
5 $\quad$**if** $\mathcal{N}_l = \emptyset$ **or** $\mathcal{T}_l \geq \mathcal{T}_f \times \alpha$ **then**
6 $\quad\quad\eta_f \leftarrow \arg\min_{node\in\mathcal{N}_f} \mathcal{T}[node]$;
7 $\quad\quad$Immediately schedule γ to η_f;
8 $\quad\quad$**return**;
9 $\quad$**end**
10 $\quad\eta_l \leftarrow \arg\min_{node\in\mathcal{N}_l} \mathcal{T}[node]$;
11 $\quad$**Start** DelayMonitoring(γ, η_l, $\mathcal{N}_f$, θ);
12 **Function** `DelayMonitoring`(γ, η_l, $\mathcal{N}_f$, θ)**:**
13 $\quad$**while** *True* **do**
14 $\quad\quad$**if** η_l *has available resource* **then**
15 $\quad\quad\quad$Immediately schedule γ to η_l;
16 $\quad\quad\quad$**return**;
17 $\quad\quad$**end**
18 $\quad\quad\mathcal{T} \leftarrow$ PredictExecutionTime(γ, $\mathcal{N}_f$);
19 $\quad\quad violation \leftarrow$ True;
20 $\quad\quad$**foreach** $node \in \mathcal{N}_f$ **do**
21 $\quad\quad\quad delay \leftarrow \theta \times (1-\beta) - \mathcal{T}[node]$;
22 $\quad\quad\quad$**if** $delay > 0$ **then**
23 $\quad\quad\quad\quad violation \leftarrow$ False;
24 $\quad\quad\quad\quad$**break**;
25 $\quad\quad\quad$**end**
26 $\quad\quad$**end**
27 $\quad\quad$**if** $violation$ **then**
28 $\quad\quad\quad\eta_f \leftarrow \arg\min_{node\in\mathcal{N}_f} T[node]$;
29 $\quad\quad\quad$Immediately schedule γ to η_f;
30 $\quad\quad\quad$**return**;
31 $\quad\quad$**end**
32 $\quad\quad$Sleep($\mathcal{I}$);
33 $\quad$**end**

Node Classification serves as a fundamental mechanism in Metronome to enable intelligent scheduling decisions by systematically categorizing computational resources according to their current state and locality characteristics. For each function awaiting scheduling, Metronome dynamically classifies available nodes into two distinct categories (line 2), i.e., *local nodes* and *fallback nodes*. This is done by analyzing both workflow dependencies and infrastructure characteristics. For data locality, the system examines the function's position within the workflow and identifies nodes that contain intermediate data produced by predecessor functions. For infrastructure locality, the system queries each node's cached state to determine the availability of warm containers matching the function's runtime requirements and the presence of pre-installed package dependencies. The fundamental purpose of this step is to distinguish which nodes possess locality characteristics that can enhance function execution performance. These classifications are inherently function-specific rather than static designations, meaning a node's category may vary considerably across different functions and evolve temporally as underlying node conditions change. This dynamic classification approach allows the scheduler to make nuanced, context-aware decisions that appropriately balance the potential benefits of locality against the costs of scheduling delays.

- *Local nodes ($\mathcal{N}_l$):* Nodes that currently offer locality benefits for the function through either data locality or infrastructure locality. Rather than using discrete binary classifications as simply "data-local" or "infrastructure-local," Metronome employs a continuous methodology that quantifies the degree of locality advantage each node offers for the current function. Specifically, a node delivers data-locality benefits when it contains relevant input data from predecessor functions in the application workflow, particularly for functions with data-intensive workloads. Infrastructure-locality benefits manifest when a node hosts warm containers or cached dependencies that can accelerate the startup of heavy-dependency functions. A node can simultaneously exhibit both types of locality benefits with varying degrees of effectiveness. This continuous approach enables Metronome to automatically prioritize nodes based on their locality characteristics. In particular, a node's locality status can change dynamically as data are processed or containers expire. Multiple local nodes may exist, and the system only needs to wait for one suitable node to become available.
- *Fallback nodes ($\mathcal{N}_f$):* Nodes with suboptimal locality conditions that provide little performance advantages for function execution. They tend to have minimal queueing time (typically less than the *safe delay threshold* to be introduced later), rendering them effectively available for rapid scheduling. While they may not offer locality benefits, they serve as reliable fallback options to prevent excessive delays and ensure SLA compliance. In line 5-9, if the function lacks local nodes (e.g., if it is newly created), it will be immediately scheduled to the fallback node with the smallest predicted execution time. A node may transition between local and fallback status as its resource availability and locality characteristics change over time.

Adaptive Delay Decision-making comprises the following two key steps to schedule a function:

(1) *Locality target selection:* Metronome determines the optimal locality target by comparing predicted execution times across all available local nodes for the function, selecting the one that yields the lowest predicted execution time.
(2) *Delay benefit validation:* The system validates whether delaying is worthwhile by comparing the predicted execution time on the target local node ($\mathcal{T}_l$) with the minimum predicted time on fallback nodes ($\mathcal{T}_f$). Delay scheduling proceeds (line 10-11) only if $\mathcal{T}_l < \mathcal{T}_f \times \alpha$, where α ($\alpha < 1$) is a configurable threshold balancing locality benefits against delays (line 5). This condition ensures that delay scheduling is only applied when the execution time on a local node is significantly better than on fallback nodes, providing a quantifiable performance advantage that justifies the additional waiting time. Without this validation, the system might delay execution for minimal performance gains, potentially wasting time that could be better spent executing the function immediately on a fallback node.

When a function enters delay state, a dedicated thread monitors the scheduling decision every $\mathcal{I}$ ms (default 100 ms, line 32) until either a local node becomes available (line 14-17), or the safe delay

threshold approaches zero (line 20-26). This safe delay threshold represents the maximum time a function can safely wait for a local node without risking SLA violation. It can be determined by subtracting the predicted execution time from the product of the SLA and a safety margin factor $(1-\beta)$ (line 18). Both the monitoring interval $\mathcal{I}$ and safety margin β are configurable parameters: shorter intervals provide more responsive scheduling at increased system overhead, while larger safety margins better protect against SLA violations at the cost of reduced locality opportunities. This approach ensures dynamic delay decisions based on both function characteristics and current system conditions while maintaining SLA compliance through continuous monitoring and adaptive thresholds.

4 Implementation

Metronome is implemented as a distributed system on OpenLambda [18, 37]. We leverage OpenLambda's inherent capability to distribute serverless functions across a cluster of worker nodes while extending it with locality-aware and delay scheduling capabilities. The system architecture consists of four main components, i.e., a *gateway*, a *scheduler*, a *prediction model*, and *worker agents*. Metronome uses different programming languages for its components to optimize their respective performance. The gateway handles function invocation requests and load balancing, serving as the primary entry point for function invocations. The scheduler operates centrally and makes global scheduling decisions based on cluster-wide information. The prediction model runs as a separate service to provide execution time forecasts. Worker agents on each worker node execute functions and continuously stream local metrics (CPU, memory, network I/O, container status) to the scheduler at configurable intervals (default 50 ms), maintaining an up-to-date view of cluster-wide system state.

Our architecture emphasizes efficient data collection and processing through several key mechanisms that improve distributed system scalability. Worker agents extend OpenLambda with lightweight monitoring capabilities for serverless-specific metrics like warm container counts and package information, which are cached in memory for rapid access. The system uses gRPC with server-side streaming, enabling worker agents to push state updates to the scheduler without polling overhead, which ensures efficient cluster-wide coordination. The scheduler maintains the system states in its local cache and performs its scheduling algorithms exclusively on this local cache, eliminating network round-trips during critical decision-making processes. Moreover, function execution predictions are requested from the model service via gRPC only when necessary. To enhance responsiveness, time-consuming operations are moved off the critical path of function execution and scheduling decisions. Specifically, delay monitoring runs in separate goroutines to prevent blocking the main scheduling loop, while model inference and metric collection operate asynchronously. These architectural decisions enable Metronome to maintain low-latency scheduling decisions even at high concurrency levels.

5 Experimental Evaluation

In this section, we conduct extensive experiments to evaluate the performance of Metronome. Our evaluation aims to answer the following research questions:

- **RQ1**: How effective is Metronome's differential delay scheduling in improving function execution performance compared to baseline methods?
- **RQ2**: How do the performance advantages of Metronome change as the concurrency level increases (which lead to heavy node loads and resource contention)?
- **RQ3**: How does Metronome's prediction model perform in terms of accuracy and adaptability. Additionally, how does the scheduler itself perform regarding overhead and scalability?

Experimental Setup: We implement and deploy Metronome on a cluster of ten nodes, each equipped with an Intel(R) Xeon(R) Gold 5318Y CPU @ 2.10GHz (16 cores), 8GB RAM, 500GB storage, and connected via 1Gbps networking. The experiments are conducted using OpenLambda [18, 37] as the underlying serverless platform. We compare the following scheduling strategies:

- *Basic Scheduling (BS)*: A conventional round-robin scheduling approach that distributes function invocations across available nodes without considering locality characteristics. This strategy represents typical serverless platforms that prioritize load balancing and immediate execution over locality benefits.
- *Naive Locality Scheduling (NLS)*: A locality-aware scheduling approach that considers both data and infrastructure locality when making placement decisions. It executes functions immediately on local nodes without delay, but falls back to nodes without locality when local resources are unavailable. This strategy attempts to leverage locality benefits when possible but does not wait for optimal nodes to become available.
- *Rule-based Delay Scheduling (RDS)*: The same strategy employed in our empirical study (Section 2.2), which follows the rules in [63] to determine the delay thresholds.
- *XFaaS*: The scheduling approach of XFaaS [43] focuses on infrastructure locality optimization. It partitions functions and workers into locality groups to maximize warm container reuse and cached dependency benefits. XFaaS employs a heuristic delay scheduling mechanism specifically for functions marked as non-critical (requiring manual annotations), allowing these functions to be delayed during high-load periods to preserve resources for critical functions. For each application, we annotate user-facing and latency-sensitive workflow stages as critical and tune the remaining non-critical annotations to minimize mean end-to-end latency without violating the SLA.
- *AQUATOPE*: A QoS-and-uncertainty-aware resource management system [68] that improves serverless performance through dynamic pre-warmed container pool management. AQUATOPE employs a hybrid Bayesian neural network to predict future function invocation rates and proactively adjusts the number of warm containers to eliminate cold starts. For each application, we conduct a training phase, allowing its Bayesian model and Bayesian optimization engine to process sufficient workflow executions until the recommended resource configurations and end-to-end latency stabilize. Icebreaker [42] follows a similar principle to reduce cold start latency by predicting function invocation and proactively warming containers.

Beyond the selected baselines, there are many other approaches proposed for serverless performance optimization. However, these

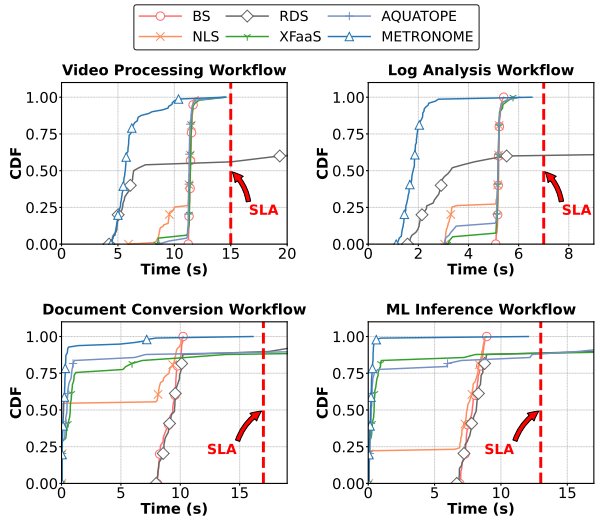


Figure 5: CDF of function execution time

works focus on different layers of the system stack that are orthogonal to Metronome. For instance, approaches targeting cold start reduction through custom runtimes [37, 62] or kernel-level modifications [13, 49], efficient data transfer mechanisms [2, 33], and workflow orchestration systems [1] represent infrastructure-level optimizations that could potentially be combined with Metronome's scheduling algorithms. Moreover, our baselines capture the fundamental scheduling principles employed by many existing approaches in both serverless computing and distributed systems, e.g., Palette [1], Netherite [6]. Therefore, our baseline selection follows established practices in serverless scheduling evaluation and enables focused assessment of our algorithmic contributions.

We evaluate these strategies using the same four representative serverless applications as in the pre-experiments (Section 2.2). Each experiment is repeated multiple times to ensure statistical significance. We measure execution times at various percentiles to understand both average-case and tail-latency performance.

5.1 RQ1: The Effectiveness of Metronome

Figure 5 presents the CDFs of application execution times under different approaches. The results show that Metronome achieves substantial performance improvements across various function types, with patterns varying according to function characteristics.

For applications involving substantial data transfer, Metronome yields considerable improvements in both average and median execution times. Specifically, in video processing, it reduces mean execution time by 47.1% and 90th percentile latency by 31.2% compared to BS, with similar gains over NLS, XFaaS and AQUATOPE. When compared to RDS, the mean and 90th percentile reductions are 74.8% and 87.2%, respectively. In log analysis, Metronome demonstrates even more significant gains compared to all baselines but RDS, i.e., with approximately 64% reduction in mean and 58% improvement in 90th percentile. When compared to RDS, both metrics realize over 90% enhancement. These advances can be attributed to Metronome's prioritization of data locality, reducing data transfer overhead between function stages. For dependency-heavy applications, Metronome's infrastructure locality optimization is highly effective. Document conversion achieves around 93% and 95% reductions in mean execution time and 90th percentile latency, respectively, across most baselines. In particular, Metronome demonstrates an 83.9% reduction in mean execution time compared to AQUATOPE. ML inference demonstrates similar benefits.

Overall, the enhancements over RDS are more substantial, which highlights the limitations of traditional rule-based delay scheduling approaches in serverless environments. For document conversion and ML inference, RDS performs even worse than NLS as the delay overhead outweighs any potential data locality benefits. This is because NLS considers infrastructure locality while RDS does not, which confirms the benefits of this new dimension of locality as revealed in our study. The improvements over XFaaS demonstrate the value of an automated and intelligent delay scheduler. XFaaS's delay scheduling requires manual annotation of functions as "non-critical" and heuristic, time-shifting rules to defer these functions to off-peak hours (e.g., overnight). The lack of data locality consideration also limits its effectiveness for data-intensive workflows. The gains over AQUATOPE can be attributed to its aggressive warm container provisioning strategy. While reducing cold starts, it can lead to resource contention and interference between co-located functions, ultimately degrading end-to-end performance. Additionally, AQUATOPE's complex Bayesian optimization process introduces significant computational overhead in the decision-making process. Data locality is also not considered in AQUATOPE.

Despite these improvements, we observe some increases in tail latencies compared to BS and NLS, which, however, are minimal and affect only a small percentage of executions. Nevertheless, Metronome still delivers positive improvements in 99th percentile latency over BS and NLS, i.e., roughly 6% for video processing, 30% for log analysis, 19% for document conversion, and 79% for ML inference. Gains are even more pronounced relative to RDS, XFaaS, and AQUATOPE. This demonstrates that even in extreme cases, Metronome maintains performance advantages. Furthermore, our adaptive delay mechanism successfully prevents any SLA violations across all applications, demonstrating an effective balance between waiting for optimal nodes and meeting latency requirements.

5.2 RQ2: Scalability Evaluation

We evaluate Metronome's performance under varying concurrency levels (1, 3, and 5 concurrent executions) to assess its scalability. This evaluation is particularly important because in production, serverless functions are typically invoked concurrently by multiple users, creating resource contention and potentially affecting locality benefits. Figure 6 presents the mean execution times across these concurrency levels for all applications, providing insights into how Metronome performs under realistic multi-user scenarios.

The results reveal distinct scaling patterns across application types. For data-intensive applications (video processing and log analysis), both BS and NLS exhibit substantial performance degradation as concurrency increases, likely due to increased network contention when transferring intermediate data between workflow stages. RDS shows some improvements at higher concurrency levels. However, its rule-based delay threshold mechanism leads to significant performance degradation in certain scenarios, even worse than both BS and NLS due to excessive tail latencies. XFaaS

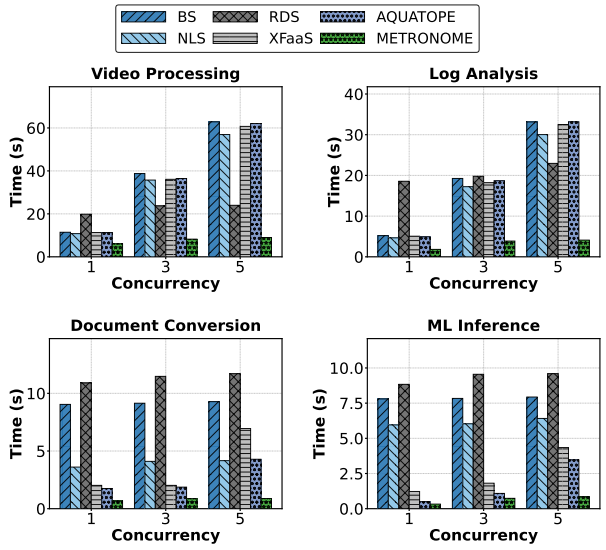


Figure 6: Mean execution times under diff. concurrency levels

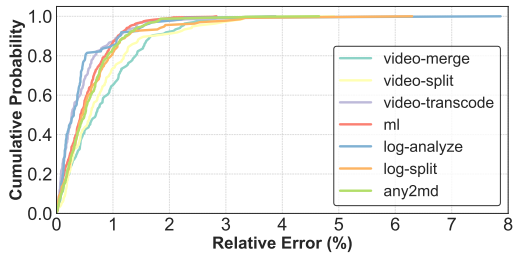


Figure 7: CDF of prediction errors across different functions

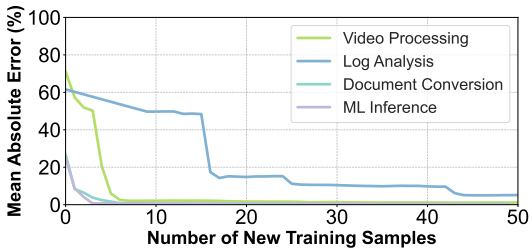


Figure 8: Model prediction error recovery with new samples

and AQUATOPE exhibit mixed scaling behavior. They achieve moderate improvements for dependency-heavy applications but suffer noticeable performance degradation in data-intensive functions. This is due to increased resource contention when multiple instances compete for limited bandwidth. In contrast, Metronome demonstrates significantly better scalability with only modest execution time increases at higher concurrency levels. This improved scalability can be attributed to Metronome's ability to maintain data locality through its differentiated delay scheduling mechanism, effectively reducing network contention under higher load.

For dependency-heavy applications like document conversion and ML inference, the impact of increased concurrency is less pronounced across all strategies. Their primary overhead stems from package initialization and dependency loading rather than data transfer. Once dependencies are cached, they can be reused across multiple function instances with minimal additional overhead. RDS performs significantly worse in this case because it only focuses on data locality. XFaaS exhibits scalability limitations under high concurrency due to its static locality grouping and heuristic scheduling strategy, which can lead to load imbalances as workload patterns evolve. AQUATOPE's performance degrades under increased concurrency as its resource allocation operates at coarse temporal granularity, while its container-centric optimization approach may not address all locality requirements for diverse workload types. Additionally, AQUATOPE exclusively focuses on infrastructure locality and neglects data locality consideration, which becomes particularly critical for data-intensive workflows under high load.

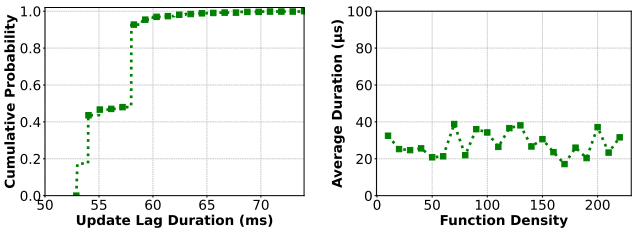
5.3 RQ3: Model and Scheduler Performance

We evaluate our prediction model's effectiveness across various function types. Figure 7 illustrates the CDF of relative prediction errors, demonstrating high accuracy with a median error of 0.44% and a 90th percentile error of 1.28%. The model maintains reasonable accuracy even at the tail, with 95th and 99th percentile errors of 1.63% and 2.98%, respectively.

To assess practical feasibility, we analyze the model's computational overhead by calculating the prediction latency distribution across 550 samples. Our model demonstrates efficient inference with mean latencies of 37.3 ms, respectively, while maintaining reasonable tail performance with 95th and 99th percentile latencies of 45 ms and 54 ms. These values suggest negligible prediction overhead compared to typical function execution times, making real-time integration feasible.

We further examine the model's adaptability to emerging workload patterns. Figure 8 depicts error recovery curves as the model encounters previously unseen function characteristics. The model demonstrates rapid adaptation capabilities. Specifically, in data-intensive applications, video processing errors drop below 10% after just 5 samples, while log analysis requires approximately 30 samples to achieve similar accuracy. Dependency-heavy applications converge even faster, with document conversion and ML Inference stabilizing below 5% error after only 2-3 samples. All application types eventually stabilize at 1-5% error rates after sufficient training samples. This differentiated adaptation pattern aligns with our earlier findings on application characteristics and demonstrates that our online learning mechanism effectively adapts to diverse workload patterns without extensive retraining.

For scheduler performance, we evaluate its overhead by analyzing system metrics collection update lag and scheduling decision latency, both critical for system's real-time capabilities. Figure 9a presents the CDF of update lag durations for system metrics collection. Our implementation uses gRPC server-side streaming to push metrics updates every 50 ms. Results show consistent metrics updates with approximately 45% completing within 55 ms and another 45% between 55-60 ms. The 95th percentile lag remains at 60 ms, ensuring the scheduler maintains an up-to-date system state view without polling. Similarly, scheduling decision latency (Figure 9b) is remarkably low. By moving network operations off the critical path, our scheduler achieves microsecond-level efficiency with 45% of decisions made within 55μs and most others



(a) CDF of system metric update lag duration

(b) Metronome's scheduling decision latency

Figure 9: Performance of the delay scheduler

between 55-60μs. The 95th percentile latency of 60μs indicates consistently fast and predictable core scheduling logic even in worst-case scenarios. These metrics demonstrate that our implementation achieves minimal-overhead scheduling while maintaining sophisticated locality-aware placement capabilities. The sub-60μs latency represents less than 0.1% of typical function execution times (hundreds of milliseconds to seconds), ensuring our locality-aware scheduling operates efficiently without becoming a system bottleneck.

6 Related Work

Scheduling for Serverless Computing: Scheduling optimization in serverless computing has attracted significant attention. For example, Hermod [23] and Palette [1] investigate key scheduling decisions around timing, node selection, and intra-node task scheduling. For QoS-aware resource management, FaaSConf [57] and scalable serverless architectures [20] focus on resource configuration optimization. Workload characterization studies [22, 47] have revealed significant heterogeneity in serverless function characteristics. General datacenter scheduling approaches from systems like Hawk [10], Eagle [9], Sparrow [38], and Pigeon [58] have proposed various techniques to balance efficiency and scalability in distributed computing environments. Building upon them, Metronome focuses on adaptive scheduling decisions that consider function execution characteristics and system state, enabling more efficient resource utilization while maintaining SLA compliance.

Another closely related domain is microservice scheduling [35]. However, microservices typically operate as long-running services with persistent state and continuous execution, whereas serverless functions are ephemeral, stateless, and event-driven. These fundamental differences highlight the need for specialized scheduling mechanisms that account for the unique characteristics of serverless computing, such as cold start latency, container warm-up overhead, and short function lifetime.

Locality Optimization: Locality optimization in serverless computing spans two dimensions: data and infrastructure. For data locality, Palette [1] improves the locality between functions in workflows through scheduling optimization. SONIC [33] explores efficient data passing mechanisms for chained serverless applications. Pocket [25], SAND [2], and the work by Pu et al. [39] address efficient data handling. Netherite [6] addresses data locality by co-locating functions and their data within the same partitions. For infrastructure locality, SOCK [37] introduces serverless-optimized containers, RainbowCake [62] proposes hierarchical container caching, Pagurus [29] explores container-sharing approaches, and FaaSLight [31] focuses on cold-start latency optimization. XFaaS [43] groups functions and workers to maximize warm container reuse benefits while employing delay scheduling for non-critical functions. AQUATOPE [68] addresses infrastructure locality through dynamic pre-warmed container pool management, using predictive models to maintain optimal container pool sizes. Icebreaker [42] is similar to AQUATOPE, which also considers server heterogeneity to balance performance and cost. Metronome uniquely combines both data and infrastructure locality considerations into a unified scheduling framework that automatically determines the most beneficial locality type for each function.

Performance Prediction: Accurate performance prediction is essential for informed scheduling. Zhao et al. [67] studied inter-function interference in serverless environments and proposed a machine learning approach to predict interference levels. Jiagu [30] decouples prediction and decision making to achieve efficient and fast scheduling. Recent work has also focused on performance variance analysis [59] and performance testing methodologies [60] for serverless computing. For the general cloud computing field, Quasar [12], Paragon [11], Cuanta [16], ASM [52], Ernest [54], and other approaches [36, 66] employ various techniques to predict performance under different resource allocations. Metronome extends this line of research by introducing a serverless-specific prediction model that incorporates function features and system metrics. Our approach leverages random forest regression for its computational efficiency and interpretability, while employing online learning to adapt to evolving workload patterns. The prediction results are directly integrated into our delay scheduling framework, enabling effective balancing of locality benefits with SLA compliance.

7 Conclusion

This paper presents Metronome, a differentiated delay scheduling framework for serverless computing that addresses the varying locality requirements across different functions. Our key contributions include: 1) a differentiated scheduling approach that automatically identifies and prioritizes the most appropriate locality type for each function; 2) an adaptive delay threshold mechanism with online learning capabilities that balances locality benefits with SLA compliance; and 3) an efficient implementation architecture that maintains low-latency scheduling decisions while collecting comprehensive system metrics. Our experimental evaluation demonstrates that Metronome achieves significant performance improvements, i.e., up to 64.88% reduction in mean execution time for data-intensive workflows and up to 95.83% for dependency-heavy functions, while successfully preventing SLA violations. The system maintains these performance advantages under increased concurrency levels where traditional approaches experience substantial degradation. The prediction model demonstrates high accuracy and rapid adaptation to new workload patterns, while the scheduling mechanism operates with negligible overhead.

Acknowledgments

This work was supported in part by the National Key Research and Development Program of China (No. 2023YFB2703600) and the National Natural Science Foundation of China (No. 62402536).

References

[1] Mania Abdi, Samuel Ginzburg, Xiayue Charles Lin, Jose M. Faleiro, Gohar Irfan Chaudhry, Iñigo 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, EuroSys 2023, Rome, Italy, May 8-12, 2023*. ACM, 365–380. doi:10.1145/3552326.3567496

[2] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. SAND: Towards High-Performance Serverless Computing. In *Proceedings of the 2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*. USENIX Association, 923–935. https://www.usenix.org/conference/atc18/presentation/akkus

[3] Carlos Arango, Rémy Dernat, and John Sanabria. 2017. Performance Evaluation of Container-based Virtualization for High Performance Computing Environments. *CoRR* abs/1709.10140 (2017). arXiv:1709.10140 http://arxiv.org/abs/1709.10140

[4] Naylor G. Bachiega, Paulo S. L. Souza, Sarita Mazzini Bruschi, and Simone do Rócio Senger de Souza. 2018. Container-Based Performance Evaluation: A Survey and Challenges. In *2018 IEEE International Conference on Cloud Engineering, IC2E 2018, Orlando, FL, USA, April 17-20, 2018*. IEEE Computer Society, 398–403. doi:10.1109/IC2E.2018.00075

[5] Ioana Baldini, Paul C. Castro, Kerry Shih-Ping Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, and Philippe Suter. 2017. Serverless Computing: Current Trends and Open Problems. In *Research Advances in Cloud Computing*. Springer, 1–20. doi:10.1007/978-981-10-5026-8_1

[6] Sebastian Burckhardt, Badrish Chandramouli, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, Christopher Meiklejohn, and Xiangfeng Zhu. 2022. Netherite: Efficient Execution of Serverless Workflows. *Proc. VLDB Endow.* 15, 8 (2022), 1591–1604. doi:10.14778/3529337.3529344

[7] Benjamin Carver, Jingyuan Zhang, Ao Wang, Ali Anwar, Panruo Wu, and Yue Cheng. 2020. Wukong: a scalable and locality-enhanced framework for serverless parallel computing. In *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020*. ACM, 1–15. doi:10.1145/3419111.3421286

[8] Matt Crane and Jimmy Lin. 2017. An Exploration of Serverless Architectures for Information Retrieval. In *Proceedings of the ACM SIGIR International Conference on Theory of Information Retrieval, ICTIR 2017, Amsterdam, The Netherlands, October 1-4, 2017*. ACM, 241–244. doi:10.1145/3121050.3121086

[9] Pamela Delgado, Diego Didona, Florin Dinu, and Willy Zwaenepoel. 2016. Job-aware Scheduling in Eagle: Divide and Stick to Your Probes. In *Proceedings of the Seventh ACM Symposium on Cloud Computing, Santa Clara, CA, USA, October 5-7, 2016*. ACM, 497–509. doi:10.1145/2987550.2987563

[10] Pamela Delgado, Florin Dinu, Anne-Marie Kermarrec, and Willy Zwaenepoel. 2015. Hawk: Hybrid Datacenter Scheduling. In *Proceedings of the 2015 USENIX Annual Technical Conference, USENIX ATC 2015, July 8-10, Santa Clara, CA, USA*. USENIX Association, 499–510. https://www.usenix.org/conference/atc15/technical-session/presentation/delgado

[11] Christina Delimitrou and Christos Kozyrakis. 2013. QoS-Aware scheduling in heterogeneous datacenters with paragon. *ACM Trans. Comput. Syst.* 31, 4 (2013), 12. doi:10.1145/2556583

[12] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: resource-efficient and QoS-aware cluster management. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2014, Salt Lake City, UT, USA, March 1-5, 2014*. ACM, 127–144. doi:10.1145/2541940.2541941

[13] 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 *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*. ACM, 467–481. doi:10.1145/3373376.3378512

[14] Tarek Elgamal. 2018. Costless: Optimizing Cost of Serverless Computing through Function Fusion and Placement. In *2018 IEEE/ACM Symposium on Edge Computing, SEC 2018, Seattle, WA, USA, October 25-27, 2018*. IEEE, 300–312. doi:10.1109/SEC.2018.00029

[15] Alexander Fuerst and Prateek Sharma. 2021. FaasCache: keeping serverless computing alive with greedy-dual caching. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*. ACM, 386–400. doi:10.1145/3445814.3446757

[16] Sriram Govindan, Jie Liu, Aman Kansal, and Anand Sivasubramaniam. 2011. Cuanta: quantifying effects of shared on-chip resource interference for consolidated virtual machines. In *ACM Symposium on Cloud Computing in conjunction with SOSP 2011, SOCC '11, Cascais, Portugal, October 26-28, 2011*. ACM, 22. doi:10.1145/2038916.2038938

[17] Ulrike Grömping. 2009. Variable importance assessment in regression: linear regression versus random forest. *The American Statistician* 63, 4 (2009), 308–319.

[18] Scott Hendrickson, Stephen Sturdevant, Edward Oakes, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. Serverless Computation with OpenLambda. *login Usenix Mag.* 41, 4. https://www.usenix.org/publications/login/winter2016/hendrickson

[19] Hemant Ishwaran, Udaya B Kogalur, Eugene H Blackstone, and Michael S Lauer. 2008. Random survival forests. (2008).

[20] Juntao Ji, Yubao Fu, Rongtong Jin, and Qingshan Lin. 2025. Designing for Scalability: Building a Universal Serverless Messaging Architecture with Apache RocketMQ. In *Proceedings of the 33rd ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE '25)* (Trondheim, Norway) *(FSE 2025)*. ACM. Industry Papers Track.

[21] Zhipeng Jia and Emmett Witchel. 2021. Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*. ACM, 152–166. doi:10.1145/3445814.3446701

[22] Artjom Joosen, Ahmed Hassan, Martin Asenov, Rajkarn Singh, Luke Nicholas 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, SoCC 2023, Santa Cruz, CA, USA, 30 October 2023 - 1 November 2023*. ACM, 443–458. doi:10.1145/3620678.3624783

[23] 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, SoCC 2022, San Francisco, California, November 7-11, 2022*. ACM, 289–305. doi:10.1145/3542929.3563468

[24] Jeongchul Kim and Kyungyong Lee. 2019. FunctionBench: A Suite of Workloads for Serverless Cloud Function Service. In *12th IEEE International Conference on Cloud Computing, CLOUD 2019, Milan, Italy, July 8-13, 2019*. IEEE, 502–504. doi:10.1109/CLOUD.2019.00091

[25] 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 2018, Carlsbad, CA, USA, October 8-10, 2018*. USENIX Association, 427–444. https://www.usenix.org/conference/osdi18/presentation/klimovic

[26] Sumer Kohli, Shreyas Kharbanda, Rodrigo Bruno, João Carreira, and Pedro Fonseca. 2024. Pronghorn: Effective Checkpoint Orchestration for Serverless Hot-Starts. In *Proceedings of the Nineteenth European Conference on Computer Systems, EuroSys 2024, Athens, Greece, April 22-25, 2024*. ACM, 298–316. doi:10.1145/3627703.3629556

[27] Bojie Li, Tianyi Cui, Zibo Wang, Wei Bai, and Lintao Zhang. 2019. Socksdirect: datacenter sockets can be fast and compatible. In *Proceedings of the ACM Special Interest Group on Data Communication, SIGCOMM 2019, Beijing, China, August 19-23, 2019*. ACM, 90–103. doi:10.1145/3341302.3342071

[28] Yongkang Li, Yanying Lin, Yang Wang, Kejiang Ye, and Cheng-Zhong Xu. 2023. Serverless Computing: State-of-the-Art, Challenges and Opportunities. *IEEE Trans. Serv. Comput.* 16, 2 (2023), 1522–1539. doi:10.1109/TSC.2022.3166553

[29] Zijun Li, Linsong Guo, Quan Chen, Jiagan Cheng, Chuhao Xu, Deze Zeng, Zhuo Song, Tao Ma, Yong Yang, Chao Li, and Minyi Guo. 2022. Help Rather Than Recycle: Alleviating Cold Startup in Serverless Computing Through Inter-Function Container Sharing. In *Proceedings of the 2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022*, Jiri Schindler and Noa Zilberman (Eds.). USENIX Association, 69–84. https://www.usenix.org/conference/atc22/presentation/li-zijun-help

[30] 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 *Proceedings of the 2024 USENIX Annual Technical Conference, USENIX ATC 2024, Santa Clara, CA, USA, July 10-12, 2024*, Saurabh Bagchi and Yiying Zhang (Eds.). USENIX Association, 1–17. https://www.usenix.org/conference/atc24/presentation/liu-qingyuan

[31] Xuanzhe Liu, Jinfeng Wen, Zhenpeng Chen, Ding Li, Junkai Chen, Yi Liu, Haoyu Wang, and Xin Jin. 2023. *FaaSLight*: General Application-level Cold-start Latency Optimization for Function-as-a-Service in Serverless Computing. *ACM Trans. Softw. Eng. Methodol.* 32, 5 (2023), 119:1–119:29. doi:10.1145/3585007

[32] Fangming Lu, Xingda Wei, Zhuobin Huang, Rong Chen, Mingyu Wu, and Haibo Chen. 2024. Serialization/Deserialization-free State Transfer in Serverless Workflows. In *Proceedings of the Nineteenth European Conference on Computer Systems, EuroSys 2024, Athens, Greece, April 22-25, 2024*. ACM, 132–147. doi:10.1145/3627703.3629568

[33] Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. 2021. SONIC: Application-aware Data Passing for Chained Serverless Applications. In *Proceedings of the 2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*. USENIX Association, 285–301. https://www.usenix.org/conference/atc21/presentation/mahgoub

[34] M. Garrett McGrath and Paul R. Brenner. 2017. Serverless Computing: Design, Implementation, and Performance. In *37th IEEE International Conference on Distributed Computing Systems Workshops, ICDCS Workshops 2017, Atlanta, GA, USA, June 5-8, 2017*. IEEE Computer Society, 405–410. doi:10.1109/ICDCSW.2017.36

[35] Chunyang Meng, Shijie Song, Haogang Tong, Maolin Pan, and Yang Yu. 2023. DeepScaler: Holistic Autoscaling for Microservices Based on Spatiotemporal GNN with Adaptive Graph Learning. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE, 53–65. doi:10.1109/ASE56229.2023.00038

[36] Nikita Mishra, John D. Lafferty, and Henry Hoffmann. 2017. ESP: A Machine Learning Approach to Predicting Application Interference. In *2017 IEEE International Conference on Autonomic Computing, ICAC 2017, Columbus, OH, USA, July 17-21, 2017*. IEEE Computer Society, 125–134. doi:10.1109/ICAC.2017.29

[37] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2018. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In *Proceedings of the 2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*. USENIX Association, 57–70. https://www.usenix.org/conference/atc18/presentation/oakes

[38] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. 2013. Sparrow: distributed, low latency scheduling. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*. ACM, 69–84. doi:10.1145/2517349.2522716

[39] 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 2019, Boston, MA, February 26-28, 2019*. USENIX Association, 193–206. https://www.usenix.org/conference/nsdi19/presentation/pu

[40] Shixiong Qi, Leslie Monis, Ziteng Zeng, Ian-Chin Wang, and K. K. Ramakrishnan. 2022. SPRIGHT: extracting the server from serverless computing! high-performance eBPF-based event-driven, shared-memory processing. In *SIGCOMM '22: ACM SIGCOMM 2022 Conference, Amsterdam, The Netherlands, August 22 - 26, 2022*. ACM, 780–794. doi:10.1145/3544216.3544259

[41] Victor Rodriguez-Galiano, Manuel Sanchez-Castillo, M Chica-Olmo, and MJOGR Chica-Rivas. 2015. Machine learning predictive models for mineral prospectivity: An evaluation of neural networks, random forest, regression trees and support vector machines. *Ore geology reviews* 71 (2015), 804–818.

[42] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. 2022. IceBreaker: warming serverless functions better with heterogeneity. In *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*. ACM, 753–767. doi:10.1145/3503222.3507750

[43] 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, SOSP 2023, Koblenz, Germany, October 23-26, 2023*. ACM, 231–246. doi:10.1145/3600006.3613155

[44] Divyanshu Saxena, Tao Ji, Arjun Singhvi, Junaid Khalid, and Aditya Akella. 2022. Memory deduplication for serverless computing with Medes. In *EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022*. ACM, 714–729. doi:10.1145/3492321.3524272

[45] Mark R Segal. 2004. Machine learning benchmarks and random forest regression. (2004).

[46] Hossein Shafiei, Ahmad Khonsari, and Payam Mousavi. 2022. Serverless Computing: A Survey of Opportunities, Challenges, and Applications. *ACM Comput. Surv.* 54, 11s (2022), 239:1–239:32. doi:10.1145/3510611

[47] Mohammad Shahrad, Rodrigo Fonseca, Iñigo Goiri, Gohar Irfan 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 *Proceedings of the 2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, Ada Gavrilovska and Erez Zadok (Eds.). USENIX Association, 205–218. https://www.usenix.org/conference/atc20/presentation/shahrad

[48] Jiacheng Shen, Tianyi Yang, Yuxin Su, Yangfan Zhou, and Michael R. Lyu. 2021. Defuse: A Dependency-Guided Function Scheduler to Mitigate Cold Starts on FaaS Platforms. In *41st IEEE International Conference on Distributed Computing Systems, ICDCS 2021, Washington DC, USA, July 7-10, 2021*. IEEE, 194–204. doi:10.1109/ICDCS51616.2021.00027

[49] Simon Shillaker and Peter R. Pietzuch. 2020. Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing. In *Proceedings of the 2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*. USENIX Association, 419–433. https://www.usenix.org/conference/atc20/presentation/shillaker

[50] David Shue, Michael J. Freedman, and Anees Shaikh. 2012. Performance Isolation and Fairness for Multi-Tenant Cloud Storage. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*. USENIX Association, 349–362. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/shue

[51] Paul F Smith, Siva Ganesh, and Ping Liu. 2013. A comparison of random forest regression and multiple linear regression for prediction in neuroscience. *Journal of neuroscience methods* 220, 1 (2013), 85–91.

[52] Lavanya Subramanian, Vivek Seshadri, Arnab Ghosh, Samira Manabi Khan, and Onur Mutlu. 2015. The application slowdown model: quantifying and controlling the impact of inter-application interference at shared caches and main memory. In *Proceedings of the 48th International Symposium on Microarchitecture, MICRO 2015, Waikiki, HI, USA, December 5-9, 2015*. ACM, 62–75. doi:10.1145/2830772.2830803

[53] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. 2021. Benchmarking, analysis, and optimization of serverless function snapshots. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*. ACM, 559–572. doi:10.1145/3445814.3446714

[54] Shivaram Venkataraman, Zongheng Yang, Michael J. Franklin, Benjamin Recht, and Ion Stoica. 2016. Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2016, Santa Clara, CA, USA, March 16-18, 2016*. USENIX Association, 363–378. https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/venkataraman

[55] 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 *Proceedings of the 2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*. USENIX Association, 443–457. https://www.usenix.org/conference/atc21/presentation/wang-ao

[56] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael M. Swift. 2018. Peeking Behind the Curtains of Serverless Platforms. In *Proceedings of the 2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*. USENIX Association, 133–146. https://www.usenix.org/conference/atc18/presentation/wang-liang

[57] 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, ASE 2024, Sacramento, CA, USA, October 27 - November 1, 2024*. ACM, 957–969. doi:10.1145/3691620.3695477

[58] Zhijun Wang, Huiyang Li, Zhongwei Li, Xiaocui Sun, Jia Rao, Hao Che, and Hong Jiang. 2019. Pigeon: an Effective Distributed, Hierarchical Datacenter Job Scheduler. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2019, Santa Cruz, CA, USA, November 20-23, 2019*. ACM, 246–258. doi:10.1145/3357223.3362728

[59] Jinfeng Wen, Zhenpeng Chen, Federica Sarro, and Shangguang Wang. 2025. Unveiling overlooked performance variance in serverless computing. *Empir. Softw. Eng.* 30, 2 (2025), 59. doi:10.1007/S10664-025-10615-3

[60] Jinfeng Wen, Zhenpeng Chen, Jianshu Zhao, Federica Sarro, Haodi Ping, Ying Zhang, Shangguang Wang, and Xuanzhe Liu. 2025. *SCOPE*: Performance Testing for Serverless Computing. *ACM Trans. Softw. Eng. Methodol.* 34, 8 (2025), 227:1–227:30. doi:10.1145/3717609

[61] Timothy Wood, Prashant J. Shenoy, Arun Venkataramani, and Mazin S. Yousif. 2009. Sandpiper: Black-box and gray-box resource management for virtual machines. *Comput. Networks* 53, 17 (2009), 2923–2938. doi:10.1016/J.COMNET.2009.04.014

[62] 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, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024- 1 May 2024*. ACM, 335–350. doi:10.1145/3617232.3624871

[63] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. 2010. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *European Conference on Computer Systems, Proceedings of the 5th European conference on Computer systems, EuroSys 2010, Paris, France, April 13-16, 2010*. ACM, 265–278. doi:10.1145/1755913.1755940

[64] Jiangtao Zhang, Hejiao Huang, and Xuan Wang. 2016. Resource provision algorithms in cloud computing: A survey. *J. Netw. Comput. Appl.* 64 (2016), 23–42. doi:10.1016/J.JNCA.2015.12.018

[65] Yanqi Zhang, Iñigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. 2021. Faster and Cheaper Serverless Computing on Harvested Resources. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*. ACM, 724–739. doi:10.1145/3477132.3483580

[66] Jiacheng Zhao, Huimin Cui, Jingling Xue, and Xiaobing Feng. 2016. Predicting Cross-Core Performance Interference on Multicore Processors with Regression Analysis. *IEEE Trans. Parallel Distributed Syst.* 27, 5 (2016), 1443–1456. doi:10.1109/TPDS.2015.2442983

[67] Laiping Zhao, Yanan Yang, Yiming Li, Xian Zhou, and Keqiu Li. 2021. Understanding, predicting and scheduling serverless workloads under partial interference. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2021, St. Louis, Missouri, USA, November 14-19, 2021*. ACM, 22. doi:10.1145/3458817.3476215

[68] Zhuangzhuang Zhou, Yanqi Zhang, and Christina Delimitrou. 2023. AQUATOPE: QoS-and-Uncertainty-Aware Resource Management for Multi-stage Serverless Workflows. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*. ACM, 1–14. doi:10.1145/3567955.3567960