Jingle: IoT-Informed Autoscaling for Efficient Resource Management in Edge Computing

Yixuan Wang
University of Minnesota
Twin Cities, USA
yixua003@umn.edu

Abhishek Chandra
University of Minnesota
Twin Cities, USA
chandra@umn.edu

Jon Weissman
University of Minnesota
Twin Cities, USA
jon@cs.umn.edu

***Abstract*—Edge computing is increasingly applied to various systems for its proximity to end-users and data sources. To facilitate the deployment of diverse edge-native applications, container technology has emerged as a favored solution due to its simplicity in development and resource management. However, deploying edge applications at scale can quickly overwhelm edge resources, potentially leading to violations of service-level objectives (SLOs). Scheduling edge containerized applications to meet SLOs while efficiently managing resources is a significant challenge. In this paper, we introduce Jingle, an autoscaler for edge clusters designed to efficiently scale edge-native applications. Jingle utilizes application performance metrics and domain-specific insights collected from IoT devices to construct a hybrid model. This hybrid model combines a predictive-reactive module with a lightweight learning model. We demonstrate Jingle's effectiveness through a real-world deployment in a classroom setting, managing two edge-native applications across edge configurations. Our experimental results show that Jingle can fulfill SLO requirements while requiring up to 50% fewer containers than a state-of-the-art cloud scheduler, which highlights its resource management efficiency and SLO compliance.**

***Index Terms*—Resource Management, Autoscaling, Workload Prediction, Internet of Things, Edge Computing**

I. Introduction

Edge computing, characterized by its proximity to data sources and end-users, is increasingly needed in supporting a diverse set of latency-sensitive, and privacy-centric applications. In this environment, container technology has emerged as an attractive application deployment approach. Container technology is well-suited to the resource-constrained nature of edge servers due to its flexibility and elasticity.

Autoscaling refers to the dynamic allocation of containers to match workload demands. It was originally proposed to serve cloud environments [1]–[3]. However, directly applying existing autoscaling algorithms to edge environments encounters several challenges. First, the traditional rule-based methods, depending on static thresholds (e.g., CPU usage or memory utilization), struggle against the diverse application requirements at the edge because fine-tuning system metrics to align with the SLOs of a diverse application set is a complex task [4]. Secondly, cloud autoscalers that prioritize application performance may lead to resource over-allocation, thus making them unsuited for resource-limited edge scenarios. Thirdly, the sudden workload spikes typical in edge environments negatively impact the latency between the detection of SLO breaches and the execution of scaling decisions, resulting in SLO violations. Lastly, many works incorporate machine-learning models [1], [3], [4] and optimization techniques [2], [5] into autoscaling. However, their computational and storage requirements make them unsuitable for edge environments. Due to constrained resources and privacy concerns of edge applications, monitoring and collecting long-term deployment data is difficult.

This paper proposes Jingle, an edge-native autoscaler designed to address the challenges of edge-computing resource management. By harnessing the power of IoT devices, Jingle anticipates workload fluctuations caused by crowd movements, which allows for utilizing domain-specific knowledge. Our proposed solution outperforms the traditional rule-based algorithms by dynamically adapting to application performance. It also outperforms a machine learning based workload prediction model with restricted history data. Instead, it utilizes a lightweight, IoT sensing-based prediction mechanism, which avoids the pitfalls of intensive training and expensive computation that are inappropriate in edge servers.

Jingle achieves domain awareness and application scaling with a two-level autoscaling algorithm. During deployment, Jingle utilizes IoT sensors to track environmental variables, like crowd arrival and departure, translating this information into predictive workload fluctuations. Then, its hybrid autoscaling model combines a predictive-reactive module with a proactive learning model, ensuring SLOs with the minimum allocations.

We have implemented Jingle on a Kubernetes [6] edge cluster and present a detailed experimental evaluation of Jingle in the context of a classroom setting. We experimentally show that Jingle can manage resources efficiently, outperforming existing autoscaling methods while maintaining SLO compliance with efficient container allocations.

In summary, our contributions are:

- The design of a domain-aware autoscaler, Jingle, optimized for edge computing resource management.
- The novel incorporation of IoT devices as an indicator of crowd-driven workload spikes for workload prediction, deployed in a real-world classroom for the experiment.
- The deployment of Jingle within an edge computing cluster, showing easy integration with Kubernetes.

- An extensive experimental evaluation of Jingle with comparisons to state-of-the-art autoscalers

II. Background

A. Horizontal Autoscaling

Horizontal autoscaling refers to automatically scaling up or down the number of containers per application. Designing an edge-native autoscaler is challenging, and the existing autoscaling algorithms are ill-suited for the edge environment. In this section, we provide an overview of existing approaches and analyze why they commonly result in issues like delayed and inaccurate scaling decisions.

First, the traditional rule-based approach [7], [8] configures a static upper and lower threshold for certain system metrics, such as CPU and memory usage. This method lacks consideration for the application's specific requirements. Given the diverse requirements of edge native applications, it is challenging and labor-intensive to tune and evaluate the system metrics thresholds to meet each edge application's SLO. Further, application performance metrics can be collected from third-party monitoring tools [9] and integrated into the traditional rule-based approach to prioritize the application performance and requirements. One example is the K8s horizontal pod autoscaling (HPA) [10]. It calculates the ratio between the desired and current metric values and adjusts the container amount accordingly. This performance-driven rule-based method only adjusts container counts in response to SLO violations, such as when the current metrics significantly deviate from the desired metrics. Thus, this approach may lead to delayed decisions and SLO violations.

Additionally, ML-based approaches have emerged in recent years. For example, AWARE [4] encodes spatial and temporal characteristics of cloud systems to enable machine learning techniques. MagicScalar [11] proposed a Gaussian process-based predictor. Ernest [12] predicts performance under various resource configurations. However, these approaches often require hundreds of hours of data traces and intensive training, which is impractical in edge environments. Yet, insufficient training data can result in sub-optimal allocations.

B. Predictive Autoscaling

Predictive autoscaling is a technique to adjust the number of containers in advance of application usage. It is well-suited for a workload that shows typical recurring patterns or cyclical traffic [13]. This is because predictive scaling usually incorporates a workload prediction mechanism that could forecast workload. It potentially mitigates the delay of scaling decisions, compared to using only horizontal scaling that is reactive. Cilantro [2] predicts future workload and makes scheduling decisions accordingly. However, speculative autoscaling may lead to temporary over- or under-provisioning. This results in sub-optimal resource utilization [14]. Further, oscillations in this process may degrade performance due to frequent autoscaling adjustments [14].

III. Jingle Design

Jingle is an edge native autoscaler that utilizes edge domain knowledge to satisfy applications' SLOs and manage resources efficiently. This is achieved without the requirement of prior data gathering or profiling. In this section, we define the scaling problem, describe the Jingle design, and discuss the model assumptions.

A. Problem definition

Symbol	Description
p_i	Performance metrics of the i-th application.
$\hat{p}_i$	Predefined SLOs for the i-th application.
$a_{i,prev}$	Allocation for the i-th application in the preceding allocation round.
$\hat{a}_i$	Proposed allocation for the i-th application in the current allocation round.
l_i	Recorded workload experienced by the i-th application in the previous allocation round.
$\hat{l}_i$	Estimated workload for the i-th application projected for the current allocation round.
λ_i	Service quality ratio achieved by an instance of the i-th application during the last allocation round.

TABLE I: Notation employed in this paper.

We target the efficient autoscaling of containerized edge native applications to satisfy SLOs without over-provisioning resources. Edge environments are characterized by their limited resources and proximity to end-users, which imposes strict latency requirements. In such a context, the autoscaling mechanism must be responsive, accurate, and resource-efficient.

The Autoscaling Problem. For a collection of edge applications, each with specified Service Level Objective (SLO) requirements $\{r_1, r_2, \ldots, r_n\}$, determine the minimal set of container instances $\{a_1, a_2, \ldots, a_n\}$ required for each application such that all the SLOs are met.

B. Jingle Overview

The Jingle's design is informed by two key insights.

1. In many edge environments, the arrival of people can be used as an indicator of workload spikes. In this work, we focus on edge native applications that are specifically designed to operate at the edge due to performance, privacy, and management reasons. Notably, we observed scenarios where the arrival of people in a space served as the trigger for edge application deployment and/or access, such as students arriving at a large class, people arriving at a virtual meeting, or people arriving at a train station at rush hour. In these situations, edge applications encounter sudden workload spikes and require autoscaling actions to avoid SLO violations.

2. Edge-native autoscaling demands knowledge of both immediate and long-term allocation behavior. Our objective is to support edge servers that operate beyond the traditional data center. The heterogeneous nature of edge-native applications and edge resources, coupled with operational constraints, poses a significant challenge in collecting and preserving long-term accurate application knowledge. Consequently, an efficient autoscaler must exhibit responsiveness to immediate

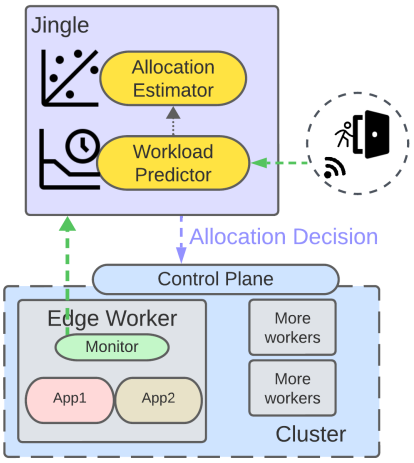


Fig. 1: Jingle Infrastructure.

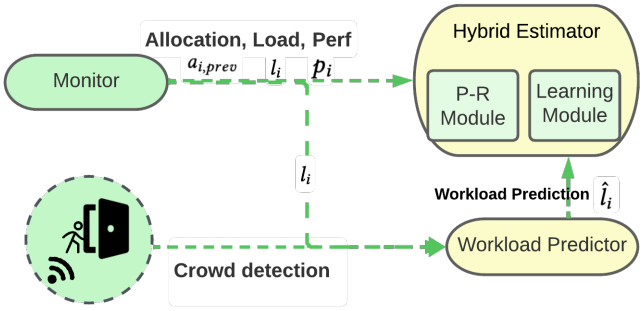


Fig. 2: Jingle Implementation.

performance metrics while concurrently cultivating a comprehensive long-term understanding of historical allocation decisions.

We leverage these observations to develop Jingle. Fig. 1 demonstrates the Jingle infrastructure. Jingle employs an online feedback loop to continuously monitor application performance by periodically pulling application-specific data from edge worker nodes. Jingle then decouples the autoscaling task into two components: 1) A workload predictor that develops a model to aggregate IoT sensing data of crowd presence and use this to make workload predictions for the next allocation round; 2) An allocation estimator that makes autoscaling decisions according to the workload prediction and application performance feedback. After the allocation adjustment is decided, the allocation decision is sent to the cluster control plane and actuated in the corresponding edge worker. Fig. 2 presents the workflow of Jingle's components. We describe each component in the following sections.

C. IoT-Enhanced Workload Predictor

Edge applications are characterized by their fluctuating workloads throughout their deployment, necessitating an adept workload prediction mechanism to make autoscaling decisions accurately. Traditional time-series models, while robust for steady-state analysis, often fall short of capturing the sudden workload spikes inherent to edge environments. To address this shortfall, we introduce a refined time-series model that considers historical workload patterns and integrates real-time crowd presence data, as provided by IoT sensors.

Traditional time-series forecasting methods, such as the Autoregressive Integrated Moving Average (ARIMA) and moving average (MA), typically depend on a recent window of historical data to predict future trends. These models, by their historical nature, are not predisposed to anticipate immediate, spike-driven changes in workload. However, accurate forecasting of spikes in such dynamic scenarios is critical for accurate predictive autoscaling.

Our approach is based on the observation that human traffic, as measured as arrivals and departures, correlates strongly with edge-native workload intensity. To harness this correlation, we implement a rudimentary yet effective IoT sensing solution for edge environments. This sensor logs the time and direction of human movement (arrivals or departures). It transmits a tiny data packet comprising the timestamp and binary movement indication to the workload predictor in real-time. This data is then used to derive a crowd movement frequency and the workload predictor continuously monitors it. When this frequency exceeds a predefined threshold, a signal denoting a crowded arrival/departure is generated accordingly.

To accommodate the limited computational resources at the edge, our workload predictor is based on a simple ARIMA(1, 1, 1) model. It operates on a data horizon encompassing only the 20 most recent data points (equivalent to a period of 10 minutes), chosen for its balance between prediction capability and computational simplicity.

The prediction mechanism is designed to be responsive to real-time sensing data inputs, particularly those indicative of sudden changes in workload. Upon detecting a signal that suggests a likely increase in workload (an uptick in crowd arrivals), the predictor proactively amplifies the forecasted demand by a factor of 1.5. This amplifying factor is chosen empirically. We leave a self-adaptive amplifying mechanism as future work. This amplification is a preemptive measure to mitigate potential SLO breaches due to sudden workload spikes. Conversely, the predictor takes a conservative approach in response to crowd departure signals to prevent oscillated scaling actions, which could lead to performance degradation. The predictor withholds immediate downward adjustments. A reduction in the predicted workload is deferred until the subsequent allocation cycle and is only enacted if the preceding cycle does not manifest a workload increase. This conservative strategy ensures a smoother scaling process and avoids scaling oscillations.

D. Hybrid Allocation Estimator

The allocation estimator utilizes a hybrid model that consists of a learning model and a predictive-reactive module as shown in Fig. 2. The learning model cumulatively learns from the recent application data and constructs a model that correlates

the application's performance metrics with workload and the number of containers. Simultaneously, the predictive-reactive module directly evaluates the allocations of the last allocation round and suggests adjustments. To perform autoscaling, Jingle first makes load predictions from the IoT-enhanced workload predictor. Then, the hybrid model makes autoscaling decisions aiming to satisfy SLOs with the minimum number of containers.

1) Predictive-Reactive Module: We now describe the predictive-reactive module (PRM) of the allocation estimator, which allocates resources utilizing anticipated workload and reacting to previous allocation actions (Algorithm 1).

Service Quality Ratio. In PRM, the service quality ratio is a pivotal metric that informs the allocation adjustments for containerized applications. We observed that the service quality of a containerized application is influenced not only by performance metrics but also by the application's workload and the predefined SLOs. Therefore, we introduce a metric, service quality ratio, that calculates the proportionality between the application's load, the predefined SLOs, and the empirically observed performance metrics. Consequently, we employ a heuristic reactive approach to adjust the allocation decisions from the previous allocation round, guided by the calculated service quality ratio. The equations that underpin this process are as follows:

$$\lambda_i = \frac{l_i \cdot \hat{p}_i}{p_i} \quad (1) \qquad a_i = \frac{\hat{l}_i \cdot a_{i,\text{prev}}}{\lambda_i} \quad (2)$$

Equation (1) defines the service quality ratio, λ_i, as the ratio of the SLO of the i-th application ($\hat{p}_i$) and the performance metrics (p_i), scaled by the workload (l_i). This expression encapsulates the desired service quality by correlating the actual performance with the expected performance (SLOs).

Utilizing the previously calculated service quality ratio and the predicted workload, we can adjust the allocation for the current round as per Equation (2). This adjustment aims to provision resources in a manner that anticipates the predicted demand while reactively correcting any over- or under-allocation from the previous round.

Equation (2) outlines the reactive scaling strategy for the current scaling action. It scales the last allocation ($a_{i,\text{prev}}$) in accordance with the predicted workload ($\hat{l}_i$) and inversely to the service rate (λ_i). This ensures that the allocation is responsive to changes in the workload and performance, intending to continuously align resource provision with the SLO targets.

2) Learning Model: In this section, we introduce the second component of our hybrid allocation estimator: a lightweight learning model that incorporates knowledge from historical allocation rounds. This model continuously assesses the relationship between allocation decisions and their consequent impact on actual performance metrics. It incrementally refines the model through periodic data polling and model updates. Unlike the universal PRM applicable to all applications, this model grants each application its specialized learning instance, enabling a long-term determination of application-specific allocation behaviors.

Algorithm 1 Predictive-Reactive Algorithm

Input: Workload l_i, SLO $\hat{p}_i$, Performance metrics p_i, Previous allocation ($a_{i,\text{prev}}$) for all applications
Output: Adjusted allocation a_i for all applications
procedure PREDICTIVE-REACTIVE ALLOCATION
  for each application i **do**
    Calculate service rate λ_i using $\lambda_i = \frac{l_i \cdot \hat{p}_i}{p_i}$
    Predict next workload $\hat{l}_i$
    Adjust allocation a_i using $a_i = \frac{\hat{l}_i \cdot a_{i-1}}{\lambda_i}$
    Return new allocation a_i
  end for
  Monitor actual performance and iterate the procedure
end procedure

Allocation Function. At the core of the learning model lies the allocation function that is designed to explore the complex correlations between allocations, actual performance outcomes, and workloads. Our empirical experiment has revealed that the relationship is not merely linear; doubling the resources does not guarantee optimally addressing a double increase in workload. Furthermore, the variability of service quality offered by edge servers adds a layer of complexity, as identical allocations may yield fluctuating performance levels. To address the fluctuating nature of edge server performance and to limit allocation volatility, the SLO is scaled by a coefficient 0.9. This coefficient is chosen empirically and this multiplication offers a margin to tolerate noise from some edge-case scenarios. The code interfaces are as follows:

```
def model_update(latest_data)
def get_allocation(perf_goal=SLO *
    scaling_coeff, predict_load=load)
```

Listing 1: Learning Model Interfaces

The core idea of our allocation function is monotonicity — assuming that an increase in the number of instances per application will not decrease performance. The architecture of our learning model is plug-and-play and provides an interface to generate allocation recommendations a_i, based on the specified SLOs and forecasted workloads. This flexible approach allows the integration and adaptation of various learning algorithms as needed by the evolving SLOs.

Furthermore, we acknowledge the initial 'warm-up' phase inherent to the learning model. During this phase, the learning model does not make correct allocation suggestions until it reaches a threshold level of confidence in its decisions. This confidence is quantified by an internal loss function that evaluates the effectiveness of its recommendations, using feedback from the system to compare its suggestions with proceeding allocations and performance.

3) Decision Making: Jingle periodically collects real-time performance metrics and workload data from deployed applications to continuously assess and monitor SLO adherence. However, we consider the latency inherent in starting a new container and its subsequent impact on application perfor-

mance. Therefore, we define an allocation round that is a designated time interval during which new allocation decisions are made and executed.

The final allocation decision for each application is an aggregation of insights pulled from both the PRM and the learning model. During the learning model's warm-up phase, the P-R module's decision dominates. As the learning model matures and its confidence in allocation recommendations increases, its influence over the final decision increases. Specifically, the final allocation is computed as a ceiling of the average of the outputs from the learning model and PRM.

IV. Implementation

We next describe the implementation details of Jingle.

A. Integration with Kubernetes

Jingle has been implemented to work seamlessly with containerized edge-native applications. These applications are containerized using Docker and managed through Kubernetes. It employs two approaches to monitor application performance metrics. The first method allows applications to record their performance data in a log file, which Jingle then periodically reads to update performance metrics. This approach is compatible with applications that use third-party monitoring tools like Prometheus, enabling seamless integration.

Additionally, Jingle provides an online profiling client within the edge nodes. This client can send profiling requests during each allocation round to collect up-to-date performance metrics. We have designed these methods to be generally applicable across a diverse range of applications and to integrate with existing orchestration systems easily.

Jingle functions as a standalone scheduler within the edge cluster, interfacing directly with Kubernetes. It pulls container allocation information from Kubernetes' worker monitors and posts its scaling decisions to Kubernetes' control plane to replace the default scheduler's decisions. This integration allows for straightforward deployment and operation within the Kubernetes systems.

B. Learning Model of Allocation Estimator

The hybrid allocation estimator of Jingle utilizes two distinct methods to adapt to applications. We have designed these methods to be modular, allowing for a pluggable interface that can be tailored to specific application needs. In our current implementation, we have implemented two types of learning models, which we will describe below.

The first model leverages a classic structure known as an interval binary tree (IBT) [2]. For a given performance metric, it assumes that each request occupies a certain range of allocations, thereby recording the highest and lowest allocation points for every workload request. Thus, it stores the maximum and minimum points of allocations per workload request. When making allocation adjustments, the IBT can quickly find the allocation intervals with the workload predictions and given SLO value. It is well-suited to storing and managing historical performance data associated with different container configurations. Its key advantage is to perform efficient insertion and search operations, typically within tens of milliseconds. This makes it an ideal structure for maintaining a historical record of performance and allocations, enabling it to perform near real-time adjustments for upcoming allocation rounds. This model is particularly effective with a single SLO parameter and operates under the assumption that the allocation scales linearly with the workload.

The second model is constructed using a lightweight Support Vector Regressor (SVR) that benefits from online feedback. Unlike the interval binary tree with a linear assumption of workload and allocations, the SVR can explore more complex relationships between application performance, workload, and allocations. By considering the performance metrics, workload, and previous allocations as input data, the SVR model can accommodate multi-objective SLOs, representing them as vectors, and develop a model to map (performance, workload) to allocations. SVR allows for a more nuanced understanding of the interplay between various factors influencing application performance. However, it is important to note that the SVR model generally requires more training when compared to the interval binary tree approach.

C. Integration with IoT Devices

Jingle enhances its workload predictor and decision-making quality by utilizing real-time crowd movement from IoT devices. For instance, we configured a entry sensor in a room, connected to a Raspberry Pi. This setup is designed to detect and record entry and exit events. Whenever someone enters or exits the room, the sensor logs the event with a precise timestamp and sends this data to the Raspberry Pi. The Raspberry Pi processes this information and stores it as a CSV file. Jingle then accesses this file at five-second intervals to update its understanding of room occupancy patterns in real-time. To guarantee ease of deployment, Jingle does not require any complex IoT data pre-processing.

D. Assumptions

Jingle makes the following assumptions:

No Request Imbalance. Jingle assumes that a load balancer [15] evenly distributes requests across all instances. It does not account for the potential skewness of request distribution that might lead to unrepresentative performance metrics. Such an imbalance, which could negatively affect performance evaluation, is not within the scope of this model to solve by scaling adjustments. Additionally, we assume that all requests are uniform, meaning the I/O and computation demands of each request are considered to be the same.

No distribution of pipelined applications. For pipelined applications that consist of multiple containers, we do not distribute these containers across various servers.

Basic IoT Sensing Capabilities. The model presumes that IoT sensing capabilities are fundamentally coarse-grained, equipped solely with the functionality to detect and timestamp human motions. The system is not required to perform granular tracking, such as counting each individual's movements

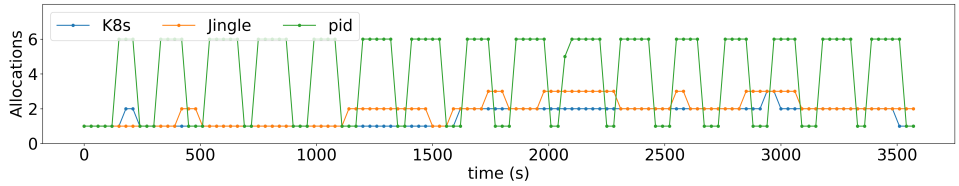


Fig. 3: Allocations over time: Jingle vs. baselines in assignment distribution application with multi-SLOs.

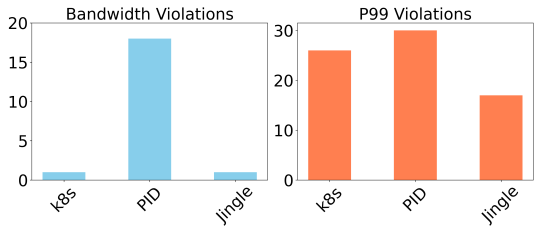


Fig. 4: Comparing jingle and baselines in assignment distribution application with multi-SLOs

explicitly. This assumption allows for a simplification of the sensing mechanism with minimal operational requirements. Moreover, we assume that IoT devices maintain consistent connectivity with the edge server. In scenarios where IoT sensing data is unavailable or delayed, Jingle would default to using the ARIMA model for workload prediction. This is to ensure that the system can still perform properly in the absence of real-time sensing data.

V. Evaluation

Our evaluation of Jingle is structured around three core questions:

1) How does Jingle perform across various edge-native applications, compared to other state-of-the-art baselines?
2) How does Jingle scale to address increasing users and workload?
3) What is the performance impact of each individual component within Jingle?

In the following sections, we begin by describing the configuration of our experimental testbed. Subsequently, we assessed Jingle regarding each of the questions outlined above.

A. Experimental Setup

Testbed Configuration. We test on a single-node Kubernetes cluster. For our edge server testbed, we have selected hardware equipped with an Apple M1 Pro chip, complemented by 16 GB of RAM. This setup also includes an Aqara Motion Sensor P1 and a Raspberry Pi that enable the collection of crowd-moving data in real-time. Jingle monitors application performance metrics, workload (over the previous 30 seconds), and allocations, updating every 30 seconds. Furthermore, Jingle is configured to adjust allocations every 2 minutes. This adjustment frequency was chosen based on the time required to prepare a container on our test server.

To demonstrate Jingle's ability to support applications across diverse environments, particularly in resource-constrained situations, we have limited the scheduler to use only 200 milliCPU units (20% of a single CPU core). This constraint ensures that our findings apply to servers with various resource limitations.

Application Test Scenarios. We simulated a realistic edge environment within a university lab classroom. We evaluated the performance of several edge-native applications:

1) An assignment distribution application, powered by Nginx, facilitates the process for students to access and download their assignments. It is I/O intensive, meaning that it heavily relies on managing a large number of concurrent connections and the ability to transmit files of significant size efficiently. It has two SLOs: the capacity of concurrent connections and the bandwidth of transmitted file size.
2) A coding assignment application, designed to handle the submission of students' C programming assignments. It processes submissions by compiling the code and validating it against pre-defined test cases to check correctness, making it a computation-intensive application. Its SLO is the latency to process the submissions.

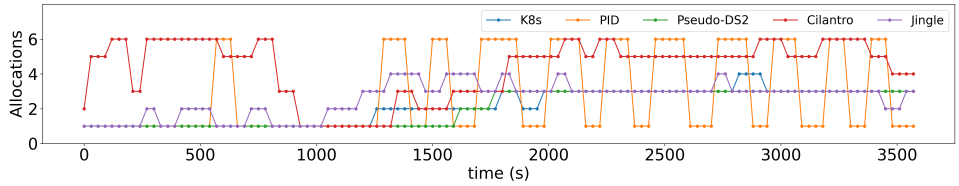
Both applications are deployed without any prior data.

Data Traces. We synthesized request traces based on actual classroom activity data from 2023 Oct. 16th, between 12:56 pm and 1:56 pm, which is a university recitation section with 30 students. We recorded room student occupancy, generating synthetic traces assuming each student sent 20 requests following a normal distribution during their stay in the class. Nginx ingress [15] is utilized to balance requests across containers.

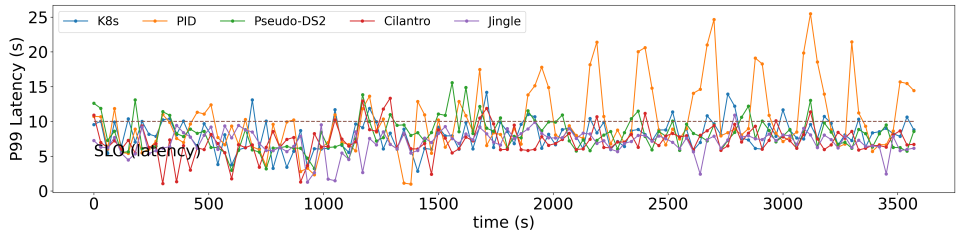
B. Comparing with Baselines

This section focuses on comparing Jingle with state-of-the-art autoscalers across various applications. It is divided into two parts: the first part evaluates Jingle's ability to manage applications with multiple SLOs, while the second part examines its performance with single-SLO applications.

1) Multiple Performance Objectives: We assess Jingle to handle the application with multiple SLO parameters in this section. The SLOs for our assignment distribution application consist of various performance metrics. The first SLO is that each container should not exceed 4 concurrent connections for downloading assignment files to ensure capacity. The second SLO parameter is the throughput of file size transmission, stipulating the need for scaling when the data transfer rate



(a) Allocations over time in coding assignment.



(b) P99 latency over time in coding assignment.

Fig. 5: Time-series analysis of Jingle versus baselines in coding assignment.

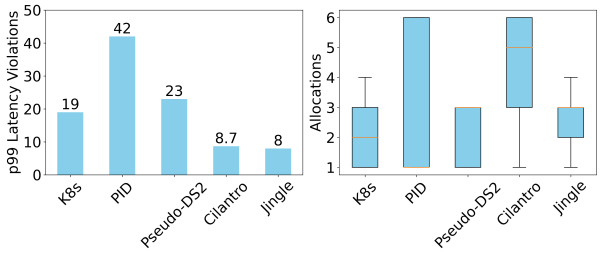


Fig. 6: Comparing jingle and baselines in coding assignment application

surpasses 1024 bytes per container. To evaluate this, we simulated the application using synthesized request traces, shown in Fig. 8. We compare Jingle against Kubernetes Horizontal Pod Autoscaler (K8s) and Proportional–Integral–Derivative Controller (PID). They are two classic autoscaling algorithms that can be applied to scenarios in which multiple SLO parameters are required.

Our evaluation indicates that Jingle outperforms the Kubernetes HPA in achieving throughput SLO over time. Notably, Jingle exhibits a 34% reduction in throughput violations in Fig. 4, while the number of containers utilized remains comparable, with an allocation of only 11% more containers. These results suggest that our method is a promising approach to handling multiple SLOs without excessive allocations.

Furthermore, Jingle demonstrates a smoother allocation curve in Fig. 3 when comparing to all baselines, with 35% fewer fluctuations compared to Kubernetes HPA. This stability is particularly beneficial for handling dynamic workloads, where frequent reallocations can degrade performance.

2) Single Performance Objective: We conducted a comparative evaluation of Jingle against several state-of-the-art autoscalers, utilizing the coding assignment application as our test case. In this case, the application has only one performance objective, and this is well-aligned with the capabilities of baselines. For this analysis, we utilized data traces as depicted in Fig. 8. Below, we present an overview of the baselines:

1) Kubernetes Default Horizontal Pod Autoscaling (K8s): This method scales resources based on the ratio between the desired and current metric values.
2) PID Controller: This approach uses three components (proportional, integral, and derivative) to compute and adjust resources to meet the desired metric value.
3) Pseudo-DS2 [14]: A reactive mechanism that adjusts allocations based on the difference between the desired processing rate and the true processing rate.
4) Cilantro [2]: Utilizes the ARIMA model for workload prediction, aiming to minimize the difference between p99 latency and SLO requirement.

	Load Prediction	Performance Awareness	Domain Awareness
K8s	N	Y	N
PID	N	Y	N
DS2	N	Y	N
Cilantro	Y	Y	N
Jingle	Y	Y	Y

TABLE II: Jingle and baselines comparisons.

In terms of their operational patterns, the Kubernetes default method, the PID controller, and Pseudo-DS2 are reactive systems. They adjust allocations solely based on the performance metrics from the last allocation round, without predicting future workload. In contrast, Cilantro [2] is a predictive autoscaler, making allocation decisions based on anticipated workload demands. While the original design of Cilantro was focused on minimizing the P99 latency, we adapted it to minimize the discrepancy between actual latency and the SLO requirement. This adaption is intended to alleviate the issue of resource over-provisioning.

We conducted three iterations of experiments and the aggregated results of which are shown in Fig. 5a and Fig. 5b. Jingle

exhibits a more consistent allocation trajectory that mirrors the demand of assignment distribution application. Regarding allocation efficiency, Jingle, and the Kubernetes HPA deployed a similar number of containers, whereas Cilantro [2] required nearly double the allocations made by Jingle. This underscores Jingle's ability to avoid resource over-provisioning effectively.

Moreover, Jingle recorded the fewest P99 latency violations, reducing them by 9% compared to Cilantro—even though Cilantro allocated twice as many containers as shown in Fig. 6. These results highlight Jingle's proficiency in meeting latency SLO while optimizing container utilization, striking an ideal balance between resource efficiency and SLO compliance.

C. Scalability

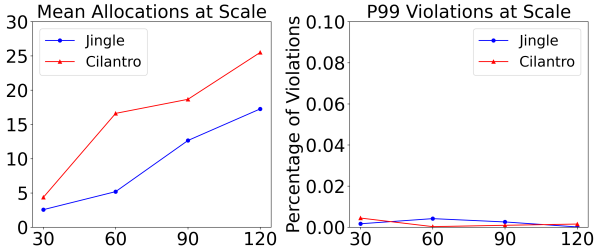
We also evaluated the scalability of Jingle with more users. The initial benchmark, as shown in Fig. 8, was set in a classroom environment with 30 students. To assess Jingle's performance under growing workload conditions, we incrementally increased the number of students from 30 to 120. This scenario simulates a larger classroom setting with more students. The outcomes of these tests are presented in Fig. 10. Notably, larger classrooms might require additional IoT sensors for several entrances. Jingle can seamlessly accommodate more IoT sensors because it operates on the frequency of human movements.

Our results demonstrate that Jingle effectively scales its mean allocations in response to the growing users, exhibiting a near-linear trend as depicted in Fig. 10. Furthermore, in terms of SLO compliance, Jingle consistently maintained a constant level of application violations across all scenarios. This consistency underscores Jingle's robustness and its capacity to handle growing users without compromising performance.

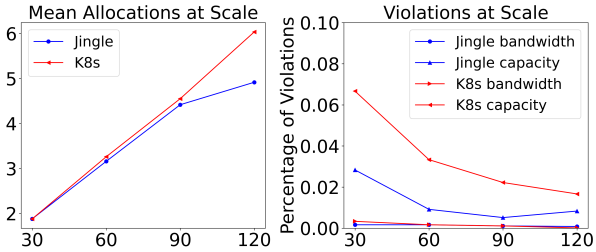
Fig. 10 also illustrates the scalability of Jingle in comparison to the second-best baseline models for both applications. Notably, Jingle is able to allocate a smaller number of containers even as it scales up. Simultaneously, it consistently maintains a constant fraction of violations, underlining its efficiency and stability in handling increasing workloads.

D. Assessment of Jingle's Individual Components

1) Workload Prediction Enhanced by IoT Sensing: We compare our IoT-enhanced workload predictor against common prediction models to demonstrate its effectiveness. In this case, we introduce another evaluation metric: the Critical Success Index (CSI), a metric traditionally employed in meteorology. This adaptation of the CSI specifically measures the predictor's ability to accurately forecast workload spikes. A key aspect of this adaptation is the consideration of the time required for preparing containers, necessitating that the predicted peak values precede the actual spikes by 1 to 2 rounds. The CSI calculation involves counting 'hits' (accurate predictions of peaks in advance), 'misses' (instances where spikes are not predicted), and 'false alarms' (incorrect predictions of peaks). The CSI is then calculated as the ratio of hits to the aggregate of hits, misses, and false alarms. This approach provides a



(a) Comparing Jingle with Cilantro in a larger scale of users for coding assignment application.



(b) Comparing Jingle with K8s HPA in a larger scale of users for assignment distribution application.

Fig. 7: Jingle's scalability comparisons.

detailed and precise evaluation of the predictor's capability to anticipate and manage peak workloads.

IoT Sensor Deployment. We installed an Aqara Motion Sensor P1 at the classroom entrance to detect student movement during the specified hour on Oct. 16th, 2023. This sensor could only identify entry and exit events without an exact count of individuals.

Baselines. We compare Jingle's workload predictor, which employs a lightweight ARIMA model augmented with IoT data (ARIMA-IoT), to several baselines:

1) Moving average (MA), which utilizes recent workload data within a specified window for predictions;
2) A recurrent Neural Network (RNN) model, which learns from past workload patterns;
3) A standard ARIMA model without IoT data;
4) An ARIMA model supplemented with static class schedules (ARIMA-static), enabling workload adjustments around the class start and end times.

We assessed the MAE and CSI of the evaluated models in Fig. 9. The results indicate that the ARIMA model with IoT sensing data is the most effective in forecasting upcoming workload spikes, as evidenced by its outstanding CSI metric. Notably, the ARIMA-IoT model exhibits a marginally higher MAE compared to the standard ARIMA model. This increase in MAE is an expected consequence of advanced spike prediction, which inherently involves discrepancies between predicted and actual workloads. Importantly, CSI is arguably a more important metric, presenting the model's proficiency in preemptively identifying workload surges, thereby providing valuable insights for subsequent autoscaling algorithms.

2) Assessment of Jingle's Components: We conducted an in-depth analysis of various underlying components of Jingle,

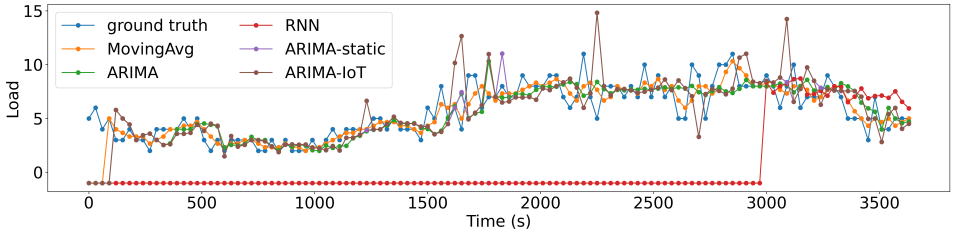


Fig. 8: Workload predictor simulation: comparing IoT-enhanced ARIMA model with time-series forecasters.

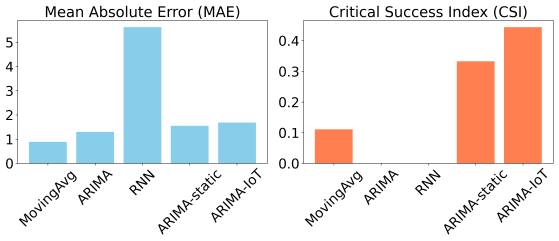


Fig. 9: Comparing workload predictors with MAE and CSI.

which include:

- PRM: The Predictive-Reactive Module (PRM) with the standard ARIMA workload predictor.
- PRM+IBT: A hybrid model that combines PRM with the Interval-Binary Tree (IBT) learning module, also utilizing the standard ARIMA predictor.
- PRM+SVR: A hybrid model that combines PRM with the Support Vector Regressor (SVR) learning module, also utilizing the standard ARIMA predictor.
- Jingle: An advanced integrated approach that merges PRM, the IBT module, and an IoT-enhanced ARIMA workload predictor.

	IoT-Info	Immediate Response	Long-term Learning
PRM	N	Y	N
PRM+IBT	N	Y	Y
PRM+SVR	N	Y	Y
Jingle	Y	Y	Y

TABLE III: Four combinations of Jingle's components.

It's crucial to highlight that PRM was a consistent element in all these evaluations. This strategic inclusion is due to the learning module's requirement for an initial 'warm-up' phase, which is necessary to build a robust profile of application deployment patterns, especially in scenarios lacking pre-existing historical data. PRM's role is thus fundamental across all configurations. Table.III illustrates the characteristics of the four combinations of Jingle's components. Our evaluations centered on these Jingle sub-components, with the coding assignment application, where the SLO is the end-to-end latency.

Fig. 10a and Fig. 10b illustrate the allocations and P99 latency over time for the coding assignment application. Additionally, Fig. 11 evaluates average allocations and total SLO violations. Overall, Jingle leverages long-term learning and domain awareness with IoT sensing to reduce SLO violations, at the cost of utilizing a slightly higher number of containers. Additionally, among the allocation modules with learning modules, the PRM+IBT is better than PRM+SVR, demonstrating the fewest P99 latency violations. This is due to IBT's effective recording of performance and allocation history with a linear relationship, enabling faster 'warming up'. The SVR model, learning through a non-linear relationship, requires more extensive training data for optimal allocation decisions. Significantly, Jingle, compared to the PRM module, allocates 30% more containers but achieves a 50 times reduction in P99 latency, showcasing the benefits of IoT-sensing domain awareness and long-term learning mechanism.

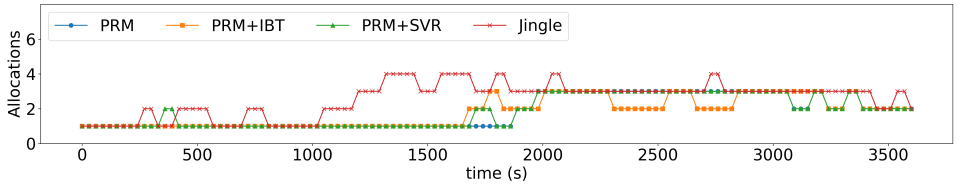
VI. Related Work

In this section, we compare Jingle with prior work. Table.II summarizes the key differences between Jingle with baselines.

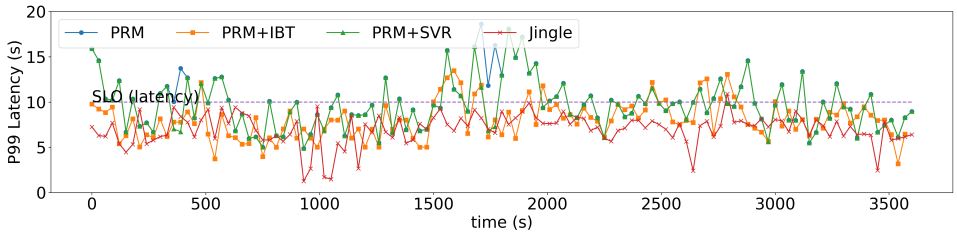
Containerization in Edge Computing Integrating container technology in edge computing has gained significant attention. Numerous studies [16] have developed containerized edge applications. For instance, DeathStarBench [17] offers a suite of pipelined containerized applications, that serve as a benchmarking tool. Similarly, Comb [18] presents a containerized video analytics application tailored for edge environments. These developments are complementary to Jingle, which seamlessly integrates with Kubernetes, facilitating its compatibility with a broad range of containerized applications.

Performance Monitoring Accurately gathering online performance metrics presents a notable challenge. Cilantro [2] introduces an online profiling client for real-time performance assessment, whereas Quasar [19] leverages offline profiling to gather proxy metrics. Paragon [20] addresses resource heterogeneity and inter-job interference to ensure performance consistency. AGILE [21], on the other hand, models resource pressure to minimize SLO violations. These methodologies demonstrate diverse strategies for performance measurement and adaptation. [22] develops an interface and APIs to report metrics for the scheduler for scheduling workflow. [23] focused on power metrics such as CPU and GPU consumption. Jingle distinguishes itself by directly interfacing with container logs to extract real-time performance and workload data, offering an adaptable solution applicable to various configurations.

Reactive Autoscaling Reactive approaches, such as those employed by Kubernetes HPA [10] and PID [24] primarily



(a) Analysis of allocations over time.



(b) Analysis of P99 latency over time.

Fig. 10: Time-series analysis of comparing four combinations of Jingle's components

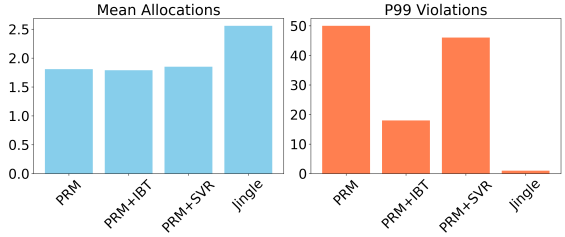


Fig. 11: Comparing four combinations of Jingle's components

involve scaling up in response to observed conditions. However, such reactive adjustments are triggered only upon violation occurrences, which is suboptimal for edge environments characterized by sudden workload fluctuations. DS2c [14] and Henge [25] are primarily designed for stream processing systems and are reactive to SLO violations. They did not include workload prediction since the stream processing prediction is very complex and inaccurate [25].

Predictive Autoscaling Predictive strategies typically utilize forecasting models to preemptively adjust resources before potential violations. Cilantro [2], for example, employs an ARIMA model, while MagicScalar [11] utilizes a more sophisticated Guassian process-based predictor. AWARE [4] and [26] are based on reinforcement learning. Ernest predicts performance under various resource configurations [12]. These systems rely on significant computational resources and extensive data for model training. Lack of training can lead to inaccurate prediction and over-provision. Consequently, they are not suitable for edge environments, where resources are constrained and training data may be insufficient.

In addition, [27] proposed a tool to predict the performance-cost tradeoff of applications. The Wing dependency profiler [28] focused on uncovering the job dependencies for better scheduling. Narya [29] predicts failures of cloud machines to make better migration decisions. These prediction and profiling mechanisms are orthogonal to Jingle and can potentially collaborate with Jingle.

VII. Conclusion and Future Work

This paper introduced Jingle, an autoscaler specifically designed for edge-native applications. By leveraging a hybrid model that combines a heuristic predictive-reactive module with a lightweight learning model, Jingle is able to provide near-optimal allocations for containerized applications. Our real-world deployment in a classroom setting demonstrated Jingle's capability to meet stringent SLOs while efficiently managing resources. Significantly outperforming traditional and state-of-the-art cloud schedulers, Jingle required up to 50% fewer containers, underscoring its efficiency in resource management and minimizing the number of violations in all experiments. Unlike existing autoscalers that either focus on reactive or predictive methods, Jingle adeptly navigates the challenges unique to edge computing, such as limited resources and the need for real-time performance metrics. Our study contributes to the evolving field of edge computing by offering a solution that is both resource-efficient and adaptable to a range of containerized applications.

In the future, we plan to extend Jingle to co-schedule multiple applications. This task is challenging as it involves allocating limited resources among various competing applications. In addition, we may also extend Jingle to support multi-component applications whose containers can be distributed across heterogeneous edge nodes.

VIII. Acknowledgement

We would like to thank the anonymous reviewers and the shepherd for their helpful comments and insights during the review process of this paper. We would also like to thank the Artifact Evaluation Committee for their meticulous examination and efforts to reproduce our results. This research was supported by the NSF Grant CNS-1908566 and the Cisco Research Grant.

REFERENCES

[1] K. Rzadca *et al.*, "Autopilot: Workload autoscaling at google," in *2020 Fifteenth European Conference on Computer Systems*, 2020.

[2] R. Bhardwaj *et al.*, "Cilantro: Performance-Aware resource allocation for general objectives via online feedback," in *Proc. 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, 2023.

[3] Y. Zhang *et al.*, "Sinan: Ml-based and qos-aware resource management for cloud microservices," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021.

[4] H. Qiu *et al.*, "AWARE: Automate workload autoscaling with reinforcement learning in production cloud systems," in *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, 2023.

[5] M. Bilal *et al.*, "With great freedom comes great opportunity: Rethinking resource allocation for serverless functions," in *Proc. Eighteenth European Conf. Comput. Syst.*, 2023.

[6] T. K. Authors, "Kubernetes," [Online]. Available: https://kubernetes.io/, 2023, accessed: Dec. 10, 2023.

[7] I. Amazon Web Services, "Aws auto scaling documentation," [Online]. Available: https://docs.aws.amazon.com/autoscaling/index.html, 2023, accessed: Nov. 23, 2023.

[8] M. Azure, "Azure autoscale," [Online]. Available: https://azure.microsoft.com/en-us/features/autoscale/, 2023, accessed: Nov. 23, 2022.

[9] "Prometheus documentation," Grafana support for Prometheus, 2023, [Online]. Available: https://prometheus.io/docs/visualization/grafana/.

[10] "Horizontal pod autoscaler," Kubernetes Documentation, 2023, [Online]. Available: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.

[11] Z. Pan *et al.*, "Magicscaler: Uncertainty-aware, predictive autoscaling," in *Proceedings of the VLDB Endowment*, vol. 16, no. 12, 2023, pp. 3808–3821.

[12] S. Venkataraman *et al.*, "Ernest: Efficient performance prediction for large-scale advanced analytics," in *Proc. 13th USENIX Symp. Networked Syst. Des. Implement. (NSDI 16)*, 2016.

[13] "Predictive scaling for amazon ec2 auto scaling," in Amazon EC2 Auto Scaling User Guide, 2023, [Online]. Available: https://docs.aws.amazon.com/autoscaling/ec2/userguide/ec2-auto-scaling-predictive-scaling.html.

[14] V. Kalavri *et al.*, "Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows," in *Proc. 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018.

[15] "Nginx ingress controller documentation," NGINX, 2023, [Online]. Available: https://docs.nginx.com/nginx-ingress-controller/.

[16] J. Wang *et al.*, "Towards scalable edge-native applications," in *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, 2019.

[17] Y. Gan *et al.*, "An open-source benchmark suite for microservices and their hardware-software implications for cloud edge systems," in *Proc. of the Twenty-Fourth Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2019.

[18] S. Bäurle and N. Mohan, "Comb: A flexible, application-oriented benchmark for edge computing," in *Proceedings of the 5th International Workshop on Edge Systems, Analytics and Networking*, 2022.

[19] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-efficient and qos-aware cluster management," *ACM SIGPLAN Notices*, vol. 49, no. 4, pp. 127–144, 2014.

[20] ——, "Paragon: Qos-aware scheduling for heterogeneous datacenters," *ACM SIGPLAN Notices*, vol. 48, no. 4, pp. 77–88, 2013.

[21] H. C. Nguyen *et al.*, "Agile: Elastic distributed resource scaling for infrastructure-as-a-service," in *Proc. International Conference on Automation and Computing*, 2013.

[22] F. Lehmann *et al.*, "How workflow engines should talk to resource managers: A proposal for a common workflow scheduling interface," in *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, 2023.

[23] M. Jay *et al.*, "An experimental comparison of software-based power meters: focus on cpu and gpu," in *CCGrid 2023-23rd IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing*, 2023.

[24] T. Hahn and M. Hahn, "Control theory for sre," in *Proceedings of the 2019 USENIX Conference, Dublin, Ireland*. USENIX Association, 2019.

[25] F. Kalim *et al.*, "Henge: Intent-driven multi-tenant stream processing," in *Proceedings of the ACM Symposium on Cloud Computing*, 2018.

[26] H.-H. Sung *et al.*, "Decentralized application-level adaptive scheduling for multi-instance dnns on open mobile devices," in *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, 2023.

[27] A. Nassereldine *et al.*, "Predicting the performance-cost trade-off of applications across multiple systems," in *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, 2023.

[28] A. Chung *et al.*, "Unearthing inter-job dependencies for better cluster scheduling," in *Proc. 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020.

[29] S. Levy *et al.*, "Predictive and adaptive failure mitigation to avert production cloud vm interruptions," in *Proc. 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020.

APPENDIX

A. Introduction

Jingle is an autonomous autoscaler for edge clusters designed to efficiently scale edge-native applications. Jingle utilizes application performance metrics and domain-specific insights collected from IoT devices to construct a hybrid model. To evaluate the effectiveness of Jingle, we implemented two edge-native applications and deployed Jingle on a Kind Kubernetes cluster with a synthetic workload that is adapted from real-world classroom activities. This artifact is designed to reproduce the experiments with coding-assignment application and comparisons to the state-of-the-art autoscalers. However, we also provide the implementation of all other necessary objects to completely reproduce the experiments. This project is available through Zenodo (DOI: 10.5281/zenodo.10668807) and GitHub (https://github.com/YixuanJiujiu/Jingle).

B. System Requirements

1) Hardware: The primary experiments are performed on a personal workstation equipped with an Apple M1 Pro chip, complemented by 16 GB of RAM. However, thanks to the container technology, this hardware is not critical when reproducing the project. We still suggest a machine with at least 6 CPU cores to demonstrate the autoscaling results.

2) Software: Docker and Kubernetes are required. We also deploy Kind cluster to perform single edge node experiments.

C. Test Applications

This Jingle experiments focus on two edge native applications:

1) An assignment distribution application, powered by Nginx, allowing students to access and download assignments, where the capacity of concurrent connections and bandwidth of transmitted file size are the SLOs. It takes a GET HTTP request as a standard workload request. The worker container shall return with the desired file.

2) A coding assignment application, where students submit code and validate it with test cases, with latency as SLO; It takes a POST HTTP request as a standard workload request. The worker container shall compile the assignment code, run a couple of test cases, and return with test results.

```
,load,allocs,reward,event_start_time,event_end_time
1699255115.5617023,5.0,{'root--coding-assign': 1.0},7.253870964050293,1699255115.5617023,1699255122.8252356
1699255145.5903232,6.0,{'root--coding-assign': 1.0},6.304728984832764,1699255145.5903232,1699255151.9175904
1699255175.5977402,4.0,{'root--coding-assign': 1.0},6.204707145690918,1699255175.5977402,1699255181.819025
1699255205.6120207,5.0,{'root--coding-assign': 1.0},7.699618816375732,1699255205.6120207,1699255213.3240066
1699255235.623446,3.0,{'root--coding-assign': 1.0},5.402840614318848,1699255235.623446,1699255241.0262864
1699255265.656369,3.0,{'root--coding-assign': 1.0},4.460423946380615,1699255265.656369,1699255270.1200378
```

Fig. 12: Expected file content of root–coding-assign.csv.

The workload generators for the applications are different. Switching the workload generator requires re-modifying the images of the Jingle scheduler, thus we provide the guide to run experiments of coding assignments. You can simply modify the worker/workload/driver.py to switch the workload generator and re-build the images of Jingle scheduler. Please note that re-building and updating the image shall require a Docker account.

D. Data Traces

1) Workload Traces: We synthesized request traces based on actual classroom activity data from 2023 Oct. 16th, between 12:56 pm and 1:56 pm, which is a university recitation section with 30 students. We have implemented the workload generator and it is deployed by the workload generator.

2) IoT Sensing Traces: Jingle utilizes an Aqara Motion Sensor P1 and a Raspberry Pi that collect crowd-moving data. To enable reproducibility, we kindly provide the real-world students-moving data as sensing trace in the file Jingle/scheduler/allocation_policies/raw_sensing.csv. The Jingle scheduler will process it automatically to emulate the room crowd-moving activities.

E. Experiment Guide

In this section, we will introduce the steps to reproduce the experiments with the coding assignment application and do comparisons with significant baselines. We provide the shell script to deploy all baselines. Before running the experiments, please make sure that the port 10000, 6000, and 8000 are available. It shall take at least 5 hours to finish the script.

```
cd Jingle/real_experiments/ca

cd starter; ./create_kind_cluster.sh #
    this creates a kind cluster

cd ds2; ./deploy_all.sh
```

Listing 2: Commands to test ALL Algorithms

In addition, to run a single round for a specific autoscaling algorithm, you need to follow these commands.

```
cd Jingle/real_experiments/ca

cd starter; ./create_kind_cluster.sh

cd ds2; ./deploy.sh
```

Listing 3: Commands to test a single algorithm

In this scenario, the default autoscaling algorithm is Jingle. To deploy another algorithm, you will need to modify line 9 of the file Jingle/real_experiments/ca/ds2/deploy.sh. Specifically, you can replace the config file with another algorithm that you tend to test. You shall allow for about one hour to finish the experiment.

During the experiments, you can view the container logs through:

```
./starter/view_logs.sh
```

Listing 4: Commands to check live logs

After finishing the script, you need to fetch the experimental results from the containers:

```
cd ds2; ./fetch_results.sh
```

Listing 5: Commands to fetch results

After fetching results, the results of the experiments are stored in Jinle/real_experiments/ca/ds2/workdirs_kind/.

Then you can clean the cluster:

```
./starter/clean_kind_cluster.sh
```

Listing 6: Commands to fetch results

F. Expected Output

The raw experimental results should be stored in the folder Jinle/real_experiments/ca/ds2/workdirs_kind/. Each algorithm result owns an unique folder that contains four files: 1) env.txt; 2) info.txt; 3) root–coding-assign.csv; 4) Jinglescheduler.log. An example of root–coding-assign.csv is shown as in Fig. 12

Plot Results. We also provide the plotting functions in Jingle/real_experiments/ca/plot_demo.py.

To plot the experimental results, you need to modify the folder names in plot_demo.py from lines 41-65. Replacing it with your evaluation results that are stored in the folder Jingle/real_experiments/ca/ds2/workdirs_kind/.

After replacing the directory names, you can plot the results by:

```
python plot_demo.py
```

Listing 7: Commands to plot results

G. Konw Issues

Performance Fluctuations. Due to network conditions and various readiness time of containers, the same allocation decisions may result in different performance metrics. Consequently, the experiments of the same algorithm may result in different evaluation results regarding the number of SLO violations and the number of containers. We recommend running a couple of rounds of experiments to alleviate this bias.

Nginx Ingress. We applied a patch to the nginx-ingress to achieve load balance between containers. However, applying the patch at the first time may result in some errors. We recommend cleaning the cluster and re-start the experiment after a few minutes.