Reducing End-to-End Latency of Trigger-Action IoT Programs on Containerized Edge Platforms

Wenzhao Zhang, Yixiao Teng, Yi Gao, *Member, IEEE,* and Wei Dong, *Member, IEEE*

Abstract—IoT rule engines are important middlewares that allow users to easily create custom trigger-action programs (TAPs) and interact with the physical world. Users expect their TAPs to give a timely response within a certain deadline. Existing works provide this support by boosting the process of trigger event identification. Many IoT rule engines now run in containerized environments, bringing about new challenges and opportunities. Prior solutions can no longer satisfy the need of mitigating the end-to-end latency of containerized TAPs. In this work, we propose EdgeRuler, which couples the IoT rule engine and the container runtime to assure the performance of latency-critical TAPs. To enable such capability, EdgeRuler precisely models the end-to-end latency by exploiting information from both the physical and the cyber world. EdgeRuler then enforces a deadline-aware life-cycle control and resource provision for meeting the TAP constraints in a lightweight and efficient way. We prototype and evaluate EdgeRuler on top of production-ready open-source components, which shows that EdgeRuler reduces the end-to-end latency by 28.6%-96.2% compared to existing scheduling algorithms and 68.4%-89.1% to that of the state-of-the-art IoT rule engines, incurring negligible runtime overhead.

Index Terms—Edge Computing, IoT Rule Engine, Real-Time

1 Introduction

The IoT rule engine allows users to seamlessly interact with increasingly pervasive IoT devices by writing their own custom services as a trigger-action program (TAP) with simple "If This Then That" syntax. As a fundamental middleware of edge cloud IoT platforms, many IoT rule engine solutions run in containerized environments [1]–[3]. Compared to previous bare-metal solutions, containerization help IoT rule engines achieve better portability, isolation, and manageability. Holding TAP in separate containers reliably prevent buggy TAPs from sabotaging others when they are running on the same machine [4]. Containerization is also an ongoing trend. According to Gartner, by 2025, more than 85% of global organizations will be running containerized applications in production [5].

It is crucial for the IoT rule engine to provide real-time response as high latency may lead to the failure to promptly respond to emergency scenarios, compromised system stability, and even potential safety hazards. This need has been recognized by both industry (e.g., IFTTT [6]) and research (e.g., RTX-IFTTT [7]) solutions. Specifically, IFTTT provides real-time API [8] for developers to inform the engine of new events available and ask for immediate polling. Based on the API, RTX-IFTTT further proposes a prediction method to facilitate the identification process of device trigger events (e.g., switch on/off). Although existing works perform well in non-containerized environments, they overlook the complex situations for containerized ones. Besides data or event polling, there are many subsequent procedures afterward that might impact the end-to-end (E2E) latency of containerized TAPs, including trigger event routing, action container provisioning, and execution, etc.

• W. Zhang, Y. Teng, Y. Gao, and W. Dong are with the College of Computer Science, Zhejiang University, Zhejiang 310027, China. E-mail: wz.zhang@zju.edu.cn, tengyixiao@zju.edu.cn, gaoyi@zju.edu.cn, dongw@zju.edu.cn.

Manuscript received xx xx xxxx; revised xx xx xxxx.
Date of publication x.xxxx; date of current version x.xxxx.
Corresponding author: Wei Dong.

To find out the bottleneck of E2E latency, we conduct experiments on representative edge benchmarks [9]. We find that TAPs experience a disappointing E2E response latency even for a simple rule. For example, the TAP "IF smoke >300ppm THEN set off the alarm" takes>2.97s. An in-depth investigation reveals the root reason, i.e., the loose-coupling design between the rule engine and the container runtime. On one hand, the rule engine does *not* know the instantaneous container provision dynamics. On the other hand, the container engine does *not* know the real TAP demand (e.g., how many resources are required and their deadline), and thus provisions an inappropriate amount of resources. Consequently, the two factors combined lead to excessive long response latency (§2).

To deal with the long E2E latency, we propose EdgeRuler, the first deadline-aware containerized IoT rule engine for edge cloud platforms. Specifically, EdgeRuler provides an enhanced IFTTT syntax that allows users to specify the TAP demand such as deadline and resource requirements. With these specifications, EdgeRuler designs a *holistic* scheduler that couples IoT rule engine and container runtime to minimize the E2E latency while meeting the deadline. The key idea of the scheduler is to integrate the physical-world information from the IoT rule engine (e.g., raw sensor value and the predicted event occurrence time) and cyber-world information from the container runtime (e.g., available resources and estimated execution time). Given the aforementioned information, the scheduler jointly considers container life-cycle control and resource allocation by coordinating both sides. §2 provides a motivating example to show that only considering one aspect can lead to sub-optimal or infeasible solutions.

Challenges. We face two unique challenges when realizing the scheduler. *Firstly, how to precisely understand the* *impact of the scheduling policies on E2E latency for containerized TAPs?* Our scheduler has two kinds of policies, i.e., when to start an action container (mainly affects T^{prov}) and how many resources to allocate (mainly affects T^{exec}). Traditionally, $T^{prov} \geq$ the container cold-start time T^{cold}. The IoT rule engine imposes unique opportunities to optimize T^{prov}. We can leverage sensor value prediction [10] to pre-warm container(s) [11] and thus reduce or even get rid of T^{prov}. However, IoT sensors have inherent data drift and bias due to imperfections in manufacturing and calibration, which makes the predicted TAP triggered time less reliable. On the other hand, the impact of allocated resources for T^{exec} of action container(s) is vague [12]. If we cannot judiciously understand both sides, it will lead to that: (i) over-shooting the TAP demand degrading resource usage; (ii) underestimation of its effects on response latency. *Secondly, how to efficiently derive a (near-)optimal scheduling plan when facing the exponential growth of decision space?* Once EdgeRuler learns the E2E latency, the real-time guarantee depends on the timely scheduling decisions. A straightforward way is to exhaustively enumerate all possible combinations. Nevertheless, the decision space increases exponentially as the number of triggered TAPs grows, which makes it challenging to obtain the optimal solution. Meanwhile, dynamically adjusting the decision according to different quality of experience further complicates the problem.

Our solution. For *challenge 1*, EdgeRuler advocates a streamified derivative based *online* sensor value prediction model to predict T^{prov}, which provides better performance than offline models. Besides, we propose a maximum normalized time gradient (MNTG) model to bound the estimation error. Moreover, we integrate a performance model to estimate T^{exec}, which can seamlessly work with MNTG and is easy to replace. For *challenge 2*, we propose a progressive online scheduling algorithm based on a dynamic merging cache. In essence, the algorithm uses the deadline and resource constraints to prune the search space, caches feasible plans and temporary optimal solutions for each TAP. Then it progressively goes through other TAPs in a first in first out manner and adjusts the results in the cache retrospectively.

We prototype EdgeRuler on top of a bunch of production-ready open-source components. Our evaluation shows that EdgeRuler achieves a remarkable performance, i.e., reduces an average of 68.4% and 89.1% E2E latency compared to edge and cloud baselines from both industry and literature solutions. Moreover, EdgeRuler incurs the negligible system overhead.

The contributions of this paper are:

- To our best knowledge, EdgeRuler is the first real-time IoT rule engine for containerized edge cloud platforms.
- With the extra information specified by an enhanced syntax, EdgeRuler proposes mathematical models to uncover the underlying relationship between specifications and scheduling policies. Based on the models, we further advocate a progressive scheduling algorithm based on a dynamic merging cache to handle the exponentially large decision space. Compared to the oracle solution, we have an increase of 1% deadline misses and 3% E2E latency but give timely decisions in 3.7ms.
- We prototype EdgeRuler on production-ready open-source software and validate significant performance gains using extensive experiments.

2 Motivating Example

A containerized IoT rule engine typically serves multiple TAPs with diverse computing resource requirements and time sensitivities. To find out the bottleneck of E2E latency, we conduct experiments on top of a containerized IoT rule engine we built in accordance with the state-of-the-art reference designs [2], [3], [10].

2.1 Experiment Setup

The containerized IoT rule engine. We use production-ready open-source software to build the engine. Specifically, we (1) use TDEngine to store time-serial IoT data; (2) leverage Redis to keep record of events, TAPs, and other runtime specifications; (3) exchange messages via RabbitMQ; (4) coordinate requests with Nginx; (5) encapsulate above modules and host them on the Docker container engine. Note that these components run in *separate* containers at runtime, which offers opportunities for auto-scaling mechanisms and prevents single-point failure.

The TAP execution model. Users appear online and produce TAP in arbitrary order and time. Each TAP is correlated with one or more IoT sensors, which are served as *triggers*. The sensor data of IoT devices are continuously streaming into the edge cloud server with fixed time intervals. Each TAP is also correlated with one or more IoT actuators or other services, which are served as *actions*. In general, each TAP is triggered and run in a serverless-like manner. Every time the trigger condition of a TAP is satisfied (e.g., the sensor reading is above an predefined threshold), the scheduler of EdgeRuler will decide whether to bootstrap the corresponding action container(s).

Workloads and host machine specifications. As the action applications can be quite diverse at the edge [13]–[15], we select three cases (i.e., threshold-based alarm, image, and audio processing) that cover both scalar and stream data from a representative edge benchmark [9] as workloads. We use the same hardware and software (including versions) settings in the evaluation (see §7.1).

The E2E latency definition. The action container images are already pulled from the centralized registry at the edge cloud platform. The E2E latency throughout the paper starts from the trigger event of a rule arrives and ends with its last action container finishes.

2.2 Observations and Challenges

During the experiments, we run each workload at a time with no other interference. We measure the provisioning time (i.e., the cold-start time), execution time, and others such as message exchange time by keeping track of essential timestamps. More importantly, we manually vary the allocated resources (e.g., CPU cores) by altering the typical resource control flags (e.g., `--cpus`) to measure the range of execution time. Each case is executed several times and we present the average values. Fig. 1 shows the overall results.

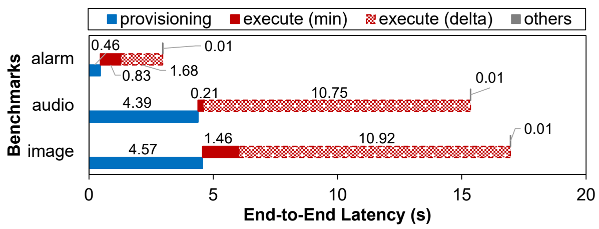


Fig. 1: Latency breakdown of containerized TAPs. The provisioning time is the duration from the TAP being triggered to the action container being ready; the execute(min) + execute(delta) indicates the execution latency range when the container is allocated different resources.

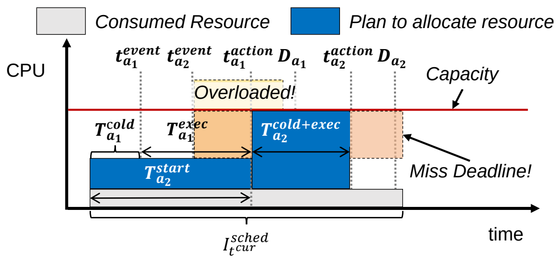
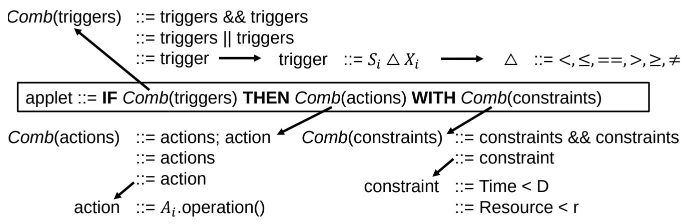


Fig. 2: Timeline of two example TAPs, each of which is assumed to have single action container, c_{11} and c_{12}.

Observations and possible solutions. We find that TAPs experience a disappointing E2E response latency even for a simple rule. We notice that the time of container provision (T^{prov}) and TAP execution (T^{exec}) dominate the latency (takes up 99.88%). Moreover, the range of T^{exec} is quite large when the TAP is allocated different amount of resources. One line of possible solutions is to alleviate the container cold-start [16]–[20] by reusing resources, optimize virtualization techniques, and control the life-cycle of containers. They make decisions only with cyber-world information (e.g., request arrival pattern and available resources), which leads to sub-optimal solutions. There are also mechanisms like admission control, offloading, and batching for serverless or containerized services [21], [22] to tune the QoS, which is orthogonal to our work.

Challenges. More importantly, it becomes a harder nut to crack when the cold-start problem runs into the multi-container resource allocation problem. Fig. 2 illustrates an example. Within a scheduling interval, the predictors predict that two TAPs will be triggered. The scheduler should decide when to start c_{11} and c_{12} and how many computing resources (e.g., CPU) are supposed to allocate. However, the decisions are difficult to make. If we follow the straightforward on-demand execution manner, c_{11} will be invoked at $t_{a_1}^{event}$ and the required resource will be assigned to a_1. When it is $t_{a_2}^{event}$, the resource left for a_2 is not enough for getting a timely response (the orange and red blocks). Fortunately, as our scheduler has already *foreseen* the two trigger events, it can decide to pre-start the c_{11} and postpone c_{12} for $T_{a_2}^{start}$ so that c_{12} can have enough resources to finish on-time (the blue blocks). Situations will be much more complicated when $I_{t_{cur}}^{sched}$ and the number of triggered TAPs are dynamically changed.



Comb(triggers) ::= triggers && triggers
::= triggers || triggers
::= trigger
trigger ::= $S_i \triangle X_i$
$\triangle$ ::= $<, \leq, ==, >, \geq, \neq$
applet ::= IF Comb(triggers) THEN Comb(actions) WITH Comb(constraints)
Comb(actions) ::= actions; action
::= actions
::= action
action ::= A_i.operation()
Comb(constraints) ::= constraints && constraints
::= constraint
constraint ::= Time < D
::= Resource < r

Fig. 3: EdgeRuler syntax. Here, X_i, D, and r are constants.

3 Design of EdgeRuler Framework

3.1 Design Goals

We aim to design a framework that can provide the *real-time* response of IoT TAPs yet is resource friendly to seamlessly work on edge cloud platforms. To balance responsiveness and resource utilization, we want EdgeRuler to be:

- **Configurable** that allows users to precisely specify their distinct needs for different TAPs, e.g., deadline and maximum resource consumption.
- **Lightweight** that does not interfere with other modules and can provide responsive decisions.
- **Adaptive** that can handle diverse data modalities and complex computing resource situations at runtime.

3.2 Usage of EdgeRuler

Existing industrial solutions only offer implicit real-time interfaces and thus cannot be *configurable*. Fortunately, some recent research works [10], [23] extend the existing syntax by adding the `constraints` segment. EdgeRuler further extends the resource syntax in this segment.

Fig. 3 illustrates the overall syntax of the EdgeRuler language. The syntax consists of three segments, i.e., `triggers`, `actions`, and `constraints`. Like the syntax of IFTTT, `actions` will be actuated when `triggers` are all triggered in a TAP. The `trigger` can recursively consist of && or || combinations of `triggers`. Each `trigger` consists of a sensing value S_i, a target value X_i, and a relational operator $\triangle$. Similarly, `action` can recursively consist of a series of parallel or sequential executed `actions`, each of which describes an operation of a containerized actuator.

Unlike the syntax of IFTTT, EdgeRuler considers specific requirements in edge cloud platforms and has `constraints` in its syntax. The `constraints` allow users to specify the deadline D and resource quota r for a TAP. With the information, the EdgeRuler framework can then adaptively tune the underlying system-level parameters at runtime, making the best effort to ensure TAPs meet their deadlines and balance the overall resource utilization.

3.3 EdgeRuler Workflow

Traditional IoT rule engine follows a relatively simple workflow [24] with the following three steps. (1) The event processors continuously take in IoT data and generate (trigger) events. (2) Given the trigger events, the rule evaluator determines if any rules are triggered. (3) When a rule is triggered, the rule engine executes the corresponding action(s).

To provide *real-time* support in a *containerized* environment, EdgeRuler follows a quite different workflow as

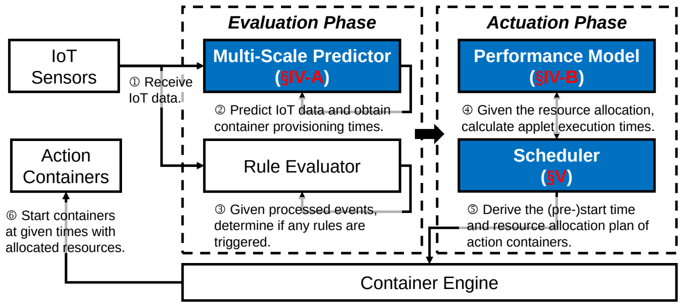

Evaluation Phase
Actuation Phase
IoT Sensors
① Receive IoT data.
Multi-Scale Predictor (§IV-A)
② Predict IoT data and obtain container provisioning times.
Rule Evaluator
③ Given processed events, determine if any rules are triggered.
Performance Model (§IV-B)
④ Given the resource allocation, calculate applet execution times.
Scheduler (§V)
⑤ Derive the (pre-)start time and resource allocation plan of action containers.
Action Containers
⑥ Start containers at given times with allocated resources.
Container Engine

Fig. 4: The overall workflow of the EdgeRuler framework.

TABLE 1: Main Notations.

Notation	Definition
a_i	The ith TAP
c_{ij}	The jth action container of a_i
$R_{a_i,t}$	Resource allocation for a_i at t
D_{a_i}	The daedline of a_i
$I_{t_{cur}}^{sched}$	The scheduling interval at current time point
$t_{a_i}^{start}$	The time point of starting action container(s) of a_i
$t_{a_i}^{event}$	The occurrence time point of a_i's trigger event
$T_{a_i}^{cold}$	The cold start time of a_i
$T_{a_i}^{prov}$	Action container(s) provisioning time of a_i
$T_{a_i}^{exec}$	Action execution time of a_i
$T_{a_i}^{pred}$	The predicted to be occurred time of a_i's trigger event
$\Delta T_{a_i}^{pred}$	The error of the predicted occurance time of a_i
$T_{a_i}^{e2e}$	The E2E latency of a_i
$\mathbb{T}^{pred'}$	The set of (T^{pred}-ΔT^{pred})
$\mathbb{D}$	The deadline set
$\mathbb{M}$	The performance model set
Obj	The objective function
τ	The minimum scheduling time unit
ι	The minimum resource unit
Λ	A set of parameters that can tune the search space

shown in Fig. 4. Specifically, the IoT data will also feed to predictors (detailed in §5.1), where we predict the event occurrence time and estimate T^{prov}. The predicted trigger event will in turn enforce a pre-flight rule evaluation. With the help of performance models (detailed in §5.2) that estimates T^{exec}, the scheduler (detailed in §6) (1) collects about to actuate rules, (2) derives the (pre-)start time together with the resource allocation plan of their action containers, and (3) informs the container engine.

4 Problem Statement

The scheduler in the EdgeRuler framework dynamically tunes the multi-scale predictor and calculates the appropriate resource allocation plan for action containers. This section formally defines the real-time guarantee problem that the scheduler is trying to address.

To support the real-time execution of a containerized TAP, the EdgeRuler framework should execute various tasks such as polling sensors, predicting the occurrence time of events, evaluating trigger conditions with (predicted) events, deciding when to (pre-)start the action container(s) and how many resources to allocate, and forwarding action requests to the container(s) before its deadline.

4.1 Problem Formulation

Before introducing the problem, we first list important notations in TABLE 1 for clarity. For the framework to meet the deadline of the TAP, its action container(s) provisioning time $T_{a_i}^{prov}$ and action execution time $T_{a_i}^{exec}$ should be shorter than the deadline D_{a_i}.

$$T_{a_i}^{e2e} = T_{a_i}^{prov} + T_{a_i}^{exec} = t_{a_i}^{complete} - t_{a_i}^{event} \leq D_{a_i}, \tag{1}$$

where $T_{a_i}^{prov}$ and $T_{a_i}^{exec}$ are determined by the longest c_{ij} (or the critical path for concurrent actions [12]) of the TAP; $t_{a_i}^{event}$ and $t_{a_i}^{complete}$ specify the time point of the trigger event happen and all c_{ij} are completed, respectively.

On the other hand, the framework is also supposed to guarantee that the computing resources of the started action containers should not surpass the currently available ones of the system. So we can have:

$$\sum_{i}^{|A|} r_{a_ik,t}^{req} < r_{k,t}^{avail}, \forall t \in Set(I_{t_{cur}}^{sched}), \forall k \in \mathbb{K}. \tag{2}$$

Here, $r_{a_ik,t}^{req}$ and $r_{k,t}^{avail}$ indicate the resource requirement and capacity at each t for a_i; t^{cur} represents the current time point and $I_{t_{cur}}^{sched}$ denotes the scheduling period at t^{cur}; $Set(I_{t_{cur}}^{sched})$ is defined as $[t^{cur}, t^{cur} + I_{t_{cur}}^{sched}]$. the number of t is determined by a tunable parameter τ to adjust the time granularity of the framework (e.g., 10 ms), which is another design choice that reflects the *configurable* requirement.

To strike a good balance between E2E latency and resource overhead, this work formulates a problem as follows:

Find the values of $T_{a_i}^{start}, R_{a_i,t}\ \forall a_i \in \mathbb{A}, \forall t \in Set(I_{t_{cur}}^{sched})$

$$\min \sum_{i}^{|A|} T_{a_i,t}^{e2e}, \forall t \in Set(I_{t_{cur}}^{sched}). \tag{3}$$

$$s.t. \begin{cases} Time\ Constraint\ (See\ Eq.1), \\ Resource\ Constraint\ (See\ Eq.2). \end{cases} \tag{4}$$

The $R_{a_i,t}$ here denotes the resource allocation for a_i at t.

However, it is possible that the latency-resource tradeoff problem (Eq. 3) has no feasible solution. In this case, EdgeRuler advocates to relax *Time Constraint* (See Eq. 1) and obtain a best-effort solution. The relaxed problem is defined as follows.

$$\min \sum_{i}^{|A|} \Delta D_{a_i,t}, \forall t \in Set(I_{t_{cur}}^{sched}). \tag{5}$$

Here, $\Delta D_{a_i,t}$ is the relative deviation between $T_{a_i,t}$ and D_{a_i}, which is defined as follows.

$$\Delta D_{a_i,t} = \begin{cases} 0, if\ T_{a_i,t}^{e2e} \leq t + D_{a_i}, \\ T_{a_i,t}^{e2e} - D, if\ T_{a_i,t}^{e2e} > t + D_{a_i}. \end{cases} \tag{6}$$

As our scheduler is highly *configurable*, system operators are allowed to substitute the optimization goals or add constraints such as energy consumption of IoT sensors [10], [23] with the help of EdgeRuler interfaces.

5 Mathematical Models for Time Estimation with Probability Error Bound

To solve the aforementioned problem effectively and efficiently, EdgeRuler should be able to accurately estimate $T_{a_i}^{prov}$ and $T_{a_i}^{exec}$ with of latency and performance models.

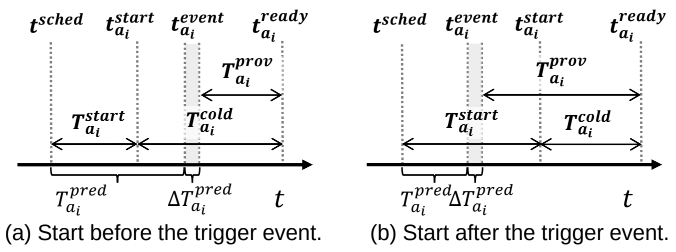


Fig. 5: Timeline illustration for modeling $T^{prov}_{a_i}$.

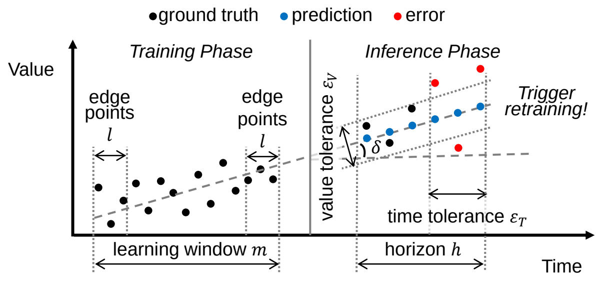


Fig. 6: The essence of the DBP model.

5.1 Modeling the container provisioning time $T^{prov}_{a_i}$

Mathematical formulation. $T^{prov}_{a_i}$ specifies the action container provisioning time for a_i, which can either be a positive value or zero. Specifically, if we start the action container(s) on demand (e.g., start c_{11} at $t^{event}_{a_1}$ as in Fig. 2), $T^{prov}_{a_i}$ equals to the cold-start time of the action container(s). On the other hand, with the help of sensor value predictors, our scheduler is able to "foresee" the arrival time of a_i. Ideally, $T^{prov}_{a_i}$ can be reduced to zero as we can pre-start the action container(s).

To formally model $T^{prov}_{a_i}$, as shown in Fig. 5, there are two cases, i.e., the action container (c_{ij}) are started before and after the estimated trigger event. Within a typical I^{sched}_t that begins at t^{sched}, the predictor reports that the trigger event will happen at $t^{event}_{a_i}$ in $T^{pred}_{a_i}$ with an error of $\Delta T^{pred}_{a_i}$. EdgeRuler scheduler decides to start c_{ij} after $T^{start}_{a_i}$ at $t^{start}_{a_i}$. c_{ij} are ready to execute actions (i.e., actually run the IoT program) after $T^{cold}_{a_i}$ at $t^{ready}_{a_i}$. Given the above description, we can derive directly from Fig. 5 that:

$$T^{prov}_{a_i} = \max[T^{start}_{a_i} + T^{cold}_{a_i} - (T^{pred}_{a_i} + \Delta T^{pred}_{a_i}), 0]. \quad (7)$$

Here, the zero value is taken only when it is the first case and $T^{prov}_{a_i}$ is less than or equal to zero. Note that Eq. 7 holds regardless of whether $\Delta T^{pred}_{a_i}$ is positive or negative.

Nevertheless, the above estimation of $T^{prov}_{a_i}$ relies heavily on the accuracy of sensor value predictors, which in turn asks for an accurate estimation of $\Delta T^{pred}_{a_i}$.

Online sensor value prediction based on streamified DBP. Before estimating the prediction error, we first need to decide the prediction models to use in EdgeRuler. Sensor value prediction has been a well-explored area since the early days of wireless sensor networks [25]. There are many common techniques to support predictive-based scheduling [26]–[29], [29], [30]. To meet our design goals, we choose to use the Derivative Based Prediction (DBP) model [31], [32]. DBP is an online prediction model (*adaptive*) that has many parameters to tune (*configurable*) and more importantly, its retraining and inference overhead is very small (*lightweight*).

Fig. 6 shows the essence of the DBP model. During the training phase, the model takes in m samples and uses the average of l edge points at the beginning and the end to calculate the slope δ. During the inference phase, the model keeps generating estimated values until there are consecutive ε_T samples that deviate from the ground truth by more than ε_V, which will then trigger a retraining process.

However, in our scenario, DBP is not directly applicable for the following three reasons. (1) The original DBP is used to predict sensor values instead of the time interval for the next trigger event (i.e., $T^{pred}_{a_i}$) as we need. (2) We can not control the prediction horizon. (3) The brute-force retraining limits the scalability especially when the frequency scales up.

Accordingly, we make the following **three modifications** based on the original DBP model. (1) We use δ to estimate $T^{pred}_{a_i}$, i.e., $T^{pred}_{a_i} = |cur\_value - target|/\delta$. (2) To provide more adaptability, we add the prediction horizon h in DBP (see Fig. 6). This is especially helpful for EdgeRuler scheduler to tune $\Delta T^{pred}_{a_i}$ and bootstrap the process of $T^{pred}_{a_i}$ estimation as we can use traditional divide and conquer method such as binary search to reduce the number of inference times. (3) Traditionally, when retraining is triggered, DBP just recalculates δ with the updated $2 \cdot l$ samples, which may cause much redundant computation. Inspired by the idea of stream computing [33], we build a suffix sum array over the m samples and keep updating it at runtime. When retraining, we can reuse the suffix sum in the array and reduce the overhead.

Modeling the prediction error $\Delta T^{pred}_{a_i}$. Even online updated predictors can hardly be 100% accurate. It is important that we have the right method to model $\Delta T^{pred}_{a_i}$. One popular method is to use probabilistic analysis [10], [34], i.e., instead of giving precise values of the time estimation error, they use estimations with a probability. One line of work builds mathematical models with thorough proof. But they are not directly applicable as they usually pose hard assumptions on the task and jobs [34]. Heo et al. [10], on the other hand, design a maximum normalized sensor value gradient (MNSVG) model that builds distributions with history IoT data, which is more suitable in our scenario.

Inspired by MNSVG, we propose a maximum normalized time gradient (MNTG) model that calculates a probability of a maximum normalized ΔT^{pred} after a given predictor horizon using the history data. It takes four steps to build an MNTG model for a sensor. First, we group the history data into subsets according to the parameters ψ of the predictors, where $\psi = \langle m, l, \varepsilon_T, \varepsilon_V, \delta \rangle$. The intuition here is that the online updated DBP models can fit in for highly dynamic sensor values. This grouping helps to characterize the data fluctuations for a typical time period. Traditional MNSVG uses all the history data, which can impair the estimation when there is a relatively large $\Delta\delta$ within a time period.

Second, for each subset of history data, we calculate maximum normalized time gradients over typical predictor horizons at a certain time point, t_0, by

$$\Delta T^{pred}(h = t - t_0) = \max_{t_0 \le t' \le t} \frac{|T^{event}_{t'} - T^{pred}_{t_0}|}{|T^{pred}_{t_0}|}.$$

Third, we generate a multiset, $S_{\psi,h}$, that represents a distribution of maximum normalized time gradients over different time points from t_s to t_e for a certain prediction horizon, h.

$$S_{\psi,h} = \{\Delta T^{pred}(\Delta t_f = (t_f + h) - t_f) | t_s \leq t_f \leq t_e\}.$$

Finally, we generate a probability function about a maximum normalized time gradient after a given predictor horizon, $p(\Delta T^{pred}|h)$.

$$p(\Delta T^{pred}|h) = \frac{|\{T^{pred}|T^{pred} = \Delta T^{pred}, T^{pred} \in S_{\psi,h}\}|}{|S_{\psi,h}|}.$$

The probability function will be used by the scheduler to analyze the relationship between the horizon and the performance of the predictor. Note that our scheduler uses the prediction and error modeling methods as a black box, EdgeRuler can seamlessly use other ones [35]–[38].

5.2 Modeling the action execution time $T^{exec}_{a_i}$

Another important part in Eq. 1 is $T^{exec}_{a_i}$, which is highly correlated with the allocated computing resources. Existing works employ many well-established methods [39]–[41]. However, the above works require tons of offline sampling and thus can hardly be *adaptable* to highly dynamic context changes (e.g., hardware and computational intensity).

EdgeRuler, on the other hand, prefers white-box modeling methods to reduce the overhead of sampling, (re-)fitting, inference, and model parameters. Specifically, we adopt a recent solution proposed in StepConf [12], which leverages an exponential function and an inverse proportional function to model the relationship between computing resources and execution latency.

$$T^{exec}_{c_{ij}} = \begin{cases} (\alpha_{j,1} \cdot \bar{\gamma}_j + \beta_{j,1}) e^{-\alpha_{j,2} \cdot \min(b_{jk}, r_{jk})} + \phi_{j,1}, case\ 1, \\ \frac{\alpha_{j,2} \cdot \bar{\gamma}_j}{\theta_s \cdot \min(b_{jk}, r_{jk}) + \beta_{j,2}} + \phi_{j,2}, case\ 2. \end{cases} \quad (8)$$

Here, $\alpha_{j,1}$, $\alpha_{j,2}$, $\beta_{j,1}$, $\beta_{j,1}$, $\phi_{j,1}$, $\phi_{j,2}$ are model parameters. $\min(b_{jk}, r_{jk})$ indicates the speed up ratio, where b_{jk} is the upper bound and r_{jk} is the input variable for a resource type. Case 1 explicitly considers multi-core execution, while Case 2 is more appropriate for single-core execution. Taking CPU as an example, if the program running in the action container is multi-core friendly, the scheduler will choose to follow case 1 and otherwise use the formula in case 2.

Similar to §5.1, it is hard to build 100% accurate performance models. We adopt the same method in the previous subsection to build a probability model of ΔT^{est}.

As the accuracy of the estimation depends on the underlying workload, hardware situations, and other computation contexts, there is no such universal method that works perfectly in all cases. Therefore, EdgeRuler exposes standard interfaces through which developers can implement their own time estimation methods that best fit their applications. They can either use more precise algorithms like sparse polynomial regression [42] or more sophisticated infrastructure like Mantis [43] that can automatically extract performance-relevant features through program analysis.

Algorithm 1: Optimal scheduling algorithm

```
   Input: $I^{sched}, R^{avail}, \mathbb{T}^{pred'}, \mathbb{C}^{triggered}, \mathbb{D}, \mathbb{M}, Obj, \tau, \iota$
   Output: $\mathbb{T}^{start*}, \mathbb{R}^{alloc*}$
1  begin
2      $sort\ \mathbb{C}^{triggered}\ by\ \mathbb{T}^{pred'}\ in\ asc.\ order$
3      Cascade $loops\ of\ all\ c \in \mathbb{C}^{triggered}$:
4          $initialize\ obj, T^{start*}_c, \mathbb{R}^{alloc*}$
5          $t \leftarrow 0$
6          while $t \leq I^{sched}$ do
7              $T^{start}_c \leftarrow t$
8              $Cal\ T^{prov}_c\ as\ Eq.7$
9              $r \leftarrow 0$
10             while $r \leq R^{avail}$ do
11                 $T^{exec}_c \leftarrow M_c(r)$
12                 if $T^{prov}_c + T^{exec}_c < D_c\ \&\&\ \sum \mathbb{R}^{alloc} <$
                     $R^{avail}\ \&\&\ obj > Obj(\mathbb{R}^{alloc})$ then
13                     $obj \leftarrow Obj(\mathbb{R}^{alloc})$
14                     $\mathbb{T}^{start*} \leftarrow \mathbb{T}^{start} \cup T^{start*}_c$
15                     $\mathbb{R}^{alloc*} \leftarrow \mathbb{R}^{alloc} \cup R^{alloc*}_c$
16                 $r \leftarrow r + \iota$
17             $t \leftarrow t + \tau$
18     return $\mathbb{T}^{start*}, \mathbb{R}^{alloc*}$
```

6 Progressive Online Scheduling Algorithm Based on a Dynamic Merging Cache

Within a I^{sched}, given the estimation of $T^{pred}_{a_i}$, $T^{exec}_{a_i}$, and their errors $\Delta T^{pred}_{a_i}$, $\Delta T^{exec}_{a_i}$, EdgeRuler scheduler should then decide $T^{start}_{a_i}$ and R_{a_i}. In this section, we describe two algorithms to give optimal or optimized scheduling results.

6.1 Brute-force optimal scheduling.

The basic idea of the optimal scheduling algorithm is to exhaustively enumerate all possible combinations within a I^{sched}. For each triggered container c, the algorithm will propose a possible value of T^{start}_c and R^{alloc}_c the scheduler will use the proposal value to calculate the objective value. After traversing all combinations of these proposal values, the algorithm is able to find the optimal one. Algorithm 1 describes the overall workflow. EdgeRuler scheduler works in a hybrid working mode, i.e., it will execute the scheduling algorithm periodically every I^{sched} and whenever the prediction result in the previous intervals is updated.

Here, $\mathbb{T}^{pred'}$ indicates the set of (T^{pred}-ΔT^{pred}) for each triggered TAP. $\mathbb{M}$ is the set of performance model defined as Eq. 8. Obj is the objective function defined in Eq. 3.

The algorithm largely consists of two steps, which are executed cascadingly for all containers within a I^{sched} (Line 3). First, given the t and r (Line 5, 9), we calculate the corresponding T^{prov} and T^{exec} (Line 8, 11). Second, if the solution meets the latency and resource constraints and the resulting object value is smaller than the current optimal one, we update the current solution (Lines 13-15).

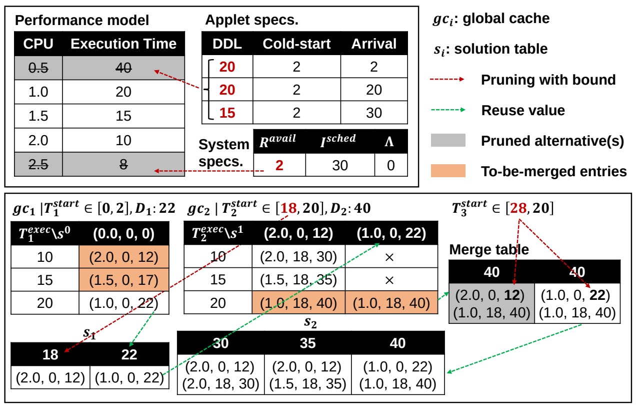


Fig. 7: Illustration of EdgeRuler data structures.

6.2 Optimized scheduling with a dynamic merged cache.

Although Algorithm 1 can derive the optimal solution for each I^{sched}, its time complexity is prohibitively high, which prevents it from giving timely feedback. Fortunately, not all the combinations are necessary in our scenario. For example, when iterating over time (Line 6 in Algorithm 1), there is no need to search $[0, I^{sched}]$. We can use a much smaller time window $[\min(0, T^{pred'} - T^{cold}), \min(I^{sched}, D - \min(T^{exec}) - T^{cold})]$. Any other time before the left bound will only be a waste of resources and time after the right bound will miss the deadline. We can apply a similar idea to prune the search space when iterating over resources (Line 10 in Algorithm 1).

The basic idea of the proposed algorithm is iterative pruning and reusing. Concretely, inspired by POP [44], we first divide the original problem into sub-problems, i.e., dealing with one container at a time and optimizing the solution progressively. To reduce the enumeration, we use the deadline and the available resources as pre-flight bound. During the progressive process, similar to the idea of dynamic programming, we build a global cache to keep track of feasible solutions to avoid redundant calculations.

Apart from the search space pruning, there are two enablers of the algorithm: (1) a set of data structures and (2) a dynamic merging policy. Fig. 7 shows an illustration. The data structures are typically two dictionaries, i.e., a dynamically changed global cache dictionary gc and a temporary solution dictionary s. Within a gc, an entry can be uniquely indexed by a pair of s and r. Each entry is a tuple of consumed resource r, start and end timestamp t^{start}, $t^{start} + T^{cold} + T^{exec}$ of a triggered action container. Within a s, we use possible ending timestamps within gc entries as index keys and keep the corresponding s as values.

As for the merging policy (the red dashed arrows in the bottom part), the key point here is that if two containers do not overlap, i.e., the latest end timestamp of the former is smaller or equal to the earliest start timestamp of the latter, we can safely merge the entries and only keep the ones with better objective values. We define the policy formally as follows.

$$isOverlapped(m, s_1, s_2) = \begin{cases} \max_{j=1}^{m} T_j^{exec} < T_{i+1}^{next}, \\ \bigcap_{j=m}^{i} s_1[j] = s_2[j]. \end{cases} \tag{9}$$

Algorithm 2: Optimized scheduling algorithm

```
Input: $I^{sched}, \mathbb{T}^{pred'}, \mathbb{T}^{cold}, \mathbb{C}^{triggered}, \mathbb{D}, \mathbb{M}, Obj, \tau, \iota, \Lambda$
Output: $S$
1  begin
2      Func find($r, s, T^{pred'}, T^{cold}, \tau, \lambda$):
3          $t \leftarrow T^{pred'} - T^{cold}$
4          while $t \leq T^{pred} + \lambda$ do
5              $iteratively\ derive\ R^{avail}\ with\ s$
6              if $r < R^{avail}$ then
7                  return $t$
8              $t \leftarrow t + \tau$
9          return $-1$
10     Func merge($\mathbb{S}, t^{next}$):
11         if $\exists_{m=1}^{i} \exists_{s_1, s_2 \in \mathbb{S} \wedge s_1 \neq s_2} (isOverlapped)$ then
12             $\mathbb{S} \leftarrow \mathbb{S} - \{s_1, s_2\} + \{\arg\min(T_{s1}^{e2e}, T_{s2}^{e2e})\}$
13         return $\mathbb{S}$
14     $sort\ \mathbb{C}^{triggered}\ by\ \mathbb{T}_*^{pred} - \mathbb{T}^{cold}\ in\ asc.\ order$
15     $S = [(0, 0, 0)], gc = \{\}$
16     for $i \leftarrow 0$ to $len(\mathbb{C}^{triggered})$ do
17         for $s \in S$ do
18             $r \leftarrow 0$
19             while $r \leq R^{avail}$ do
20                 $T_i^{exec} \leftarrow M_i(r)$
21                 if $T_i^{exec} < D_i$ then
22                     $t_i^{start} \leftarrow$
                         $find(r, s, T_i^{pred'}, T_i^{cold}, \tau, \Lambda[i])$
23                     $gc_i[s][T_i^{exec}] \leftarrow$
                         $(r, t_i^{start}, t_i^{start} + T_i^{exec})$
24                 $r \leftarrow r + \iota$
25             $S \leftarrow merge(Set < gc_i >, T_{i+1}^{pred'} - T_{i+1}^{cold})$
26     return $S$
```

where T_{i+1}^{next} is the earliest start timestamp of the $i+1$th container, which is $T_{i+1}^{pred'} - T_{i+1}^{cold}$. If $isOverlapped$ is true, it means that in two different solutions s_1, s_2, the first m containers are all finished before T_{i+1}^{next} and the last $i - m$ containers are the same.

Algorithm 2 shows the overall workflow. Here, S represents the final decisions for each $c \in \mathbb{C}^{triggered}$ that are comprised of T_c^{start} and R_c^{alloc}. Λ is a set of *configurable* parameter that can tune the search efficiency of EdgeRuler scheduler. For a typical action container c the corresponding entry in Λ, λ has a range of $[0, D_c - \min(T_c^{exec}) - T_c^{cold}]$.

The algorithm mainly has four steps. First, given the T_i^{exec} (Line 21), we use the deadline to filter out part of unnecessary combinations (i.e., $T_i^{exec} > \mathbb{D}[i]$). Second, we use the intuitive bound of t_i^{start} to shrink the time searching steps from $\lfloor I^{sched}/\tau \rfloor$ down to $\left\lfloor (T_i^{pred'} + \Lambda[i] - T_i^{cold})/\tau \right\rfloor$ (Line 3-8). During this process, we will iteratively check if the resource consumption is still under control by going through table s (Line 5). Third, we cache the feasible solutions into gc (Line 23). Finally, we dynamically merge the existing solutions with t_{i+1}^{next} and Eq. 3 to further reduce the enumerations (Line 11).

Recall the example in Fig. 7. There are predicted to be three TAPs in the coming scheduling interval. They share the same performance model (i.e., the table in the top left) and the cold-start time. While they have different (relative) deadlines and arrival times. At the beginning of the interval, the system has 2 CPUs available. According to Algorithm 2, we first process a_1. We can use its deadline and R^{avail} to eliminate impossible alternatives in its performance model (the gray entries shown in the top left of Fig. 7). Then we construct gc_1 with feasible solutions. Specifically, we apply Eq. 9 to derive s_1 from gc_1. We use the lower bound of T_2^{start} to merge the first two entries of gc_1. We calculate the objective value of $gc_1[s^0][10]$ and $gc_1[s^0][15]$ according to Eq. 3 and keep the better one in s_1. After that, we progressively process a_2, a_3, and retrospectively update gc_i and s_i before getting the final decision.

6.3 Competitive Analysis

In this subsection, we thoroughly compare the optimal and our optimized scheduling algorithm in terms of computational complexity and search space. The latter help to justify the benefits of our algorithm.

Computational Time Complexity. We use n to refer the number of triggered TAPs $|A|$. Let $\sqcup$ be $\lfloor I^{sched}/\tau \rfloor$ and ∇ be $\lfloor R^{avail}/\iota \rfloor$. The time complexity of Algorithm 1 is $O(\sqcup^n \cdot \nabla^n)$. Let $\sqcup'$ be $\lfloor (T^{cold}+\lambda)/\tau \rfloor$ and ∇' be $\lfloor \max\{r|T_r^{exec} \leq D_i\}/\iota \rfloor$. The time complexity of Algorithm 2 is $O(\sqcup'^n \cdot \nabla'^n)$. Obviously, $\sqcup'$ is much smaller than $\sqcup$, and ∇' is not larger than ∇. Therefore, EdgeRuler greatly reduces the computational complexity compared to the optimal algorithm.

Search Space. To find $\mathbb{T}^{start}$ and $\mathbb{R}^{alloc}$, the total search space is $[0, I^{sched}]$ and $[0, R^{avail}]$, respectively. Without prior knowledge, Algorithm 1 has to exhaustively go over the whole space. EdgeRuler, on the other hand, takes in abundant information from both cyber and physical world can thus properly reduce the search space.

(1) Space that can be safely pruned. For finding $\mathbb{T}^{start}$, the search space can be safely reduced to $[\min(0, T^{pred'} - T^{cold}), D - \min(T^{exec}) - T^{cold}]$. Here, $\min(0, T^{pred'} - T^{cold})$ indicates the earliest time point a container can be started, which is quite possible that it takes a larger-than-zero value. $\min(I^{sched}, D - \min(T^{exec}) - T^{cold})$ refer to the latest time point a container has to be started, or it will miss its deadline otherwise. Similarly, we can safely prune the search space of $\mathbb{R}^{alloc}$ with the lower/upper bound obtained from the performance models (see Fig. 7).

(2) Space that are configurable to be pruned. As EdgeRuler is designed to be *configurable* (see §3.1), we further provide the Λ parameter. When Λ is set to maximum values, EdgeRuler shares the same performance with the optimal algorithm with a tiny speed-up in terms of search efficiency. If Λ is set to other values, EdgeRuler opts to trade the optimality to the search efficiency by skipping unlikely-to-happen cases. In current version of EdgeRuler, we do not provide a *configurable* parameter to tune the search space of $\mathbb{R}^{alloc}$. The main reason is that the original space is usually much smaller than that of the time. We can easily derive and integrate a similar mechanism to further reduce the search space if necessary.

Final Remarks. In summary, EdgeRuler offers a *configurable* scheduling algorithm, which is optimized according to the previous analysis of search space.

7 Evaluation

We design our evaluation to answer the following four questions: (1) Can EdgeRuler achieve low E2E latency and deadline miss ratio and have better resource efficiency compared to existing works (§7.2)? (2) Can EdgeRuler time predictor accurately predict T^{pred} for different IoT data modalities (§7.3)? (3) What are the impacts of EdgeRuler configurations (§7.3)? (4) What is the system overhead of EdgeRuler components (§7.3)?

7.1 Methodology

Baselines. EdgeRuler includes a sensor value predictor, a performance model, and a scheduling algorithm.

1) Predictor baselines. We compare with three baselines: ML-based models, RT-IFTTT [10], and original DBP [31].

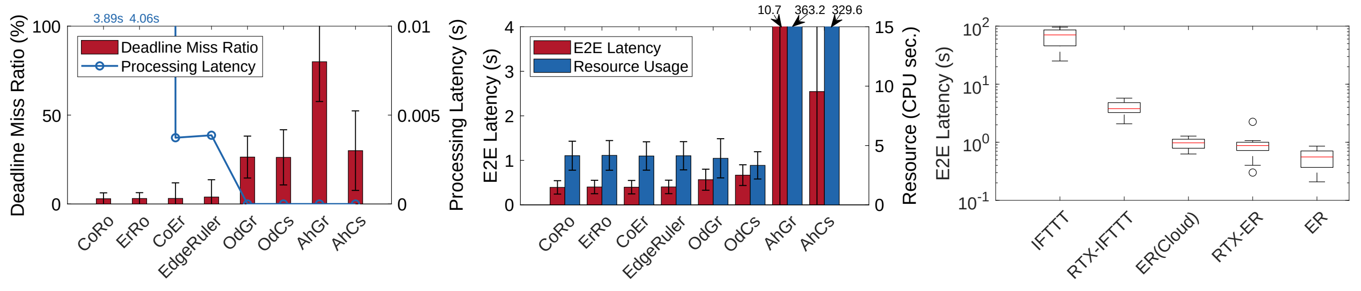
2) Scheduling algorithm baselines. There are two scheduling policies, each of which has three baselines.

(1) Container life-cycle control. (i) On-demand [45]: start the container(s) when triggered. (ii) Always hot [4]: keep the container(s) always alive. (iii) COracle: assume the predictor is 100% accurate and use a brute-force method to find the optimal start times. (2) Resource allocation. (i) Greedy [46]: allocate maximum resources for the action container that comes first. (ii) Conservative [47]: allocate minimum resources for the action container that comes first. (iii) ROracle: use a brute-force method to find the optimal resources.

3) E2E baselines. To give an effective E2E comparison, we combine existing solutions and get the following two sets of baselines: (1) (Partial) Oracle, i.e., COracle + ROracle (CoRo), EdgeRuler life-cycle control + ROracle (ErRo), and COracle + EdgeRuler resource allocation (CoEr). (2) Responsive heuristics, i.e., On demand + Greedy (OdGr), On demand + Conservative (OdCs), Always hot + Greedy (AhGr), and Always hot + Conservative (AhCs).

Datasets, workloads, and benchmarks. We leverage the IoT sensor data collected in RT-IFTTT [10], which installs 10 physical sensors that are sampled at 1 Hz for 10 days. Given the data, we generate TAPs with random triggers, cold-start times, and resource requirements. We simulate container workloads with Stress-NG [48] by varying the number of bogo ops using the `--cpu-ops` flag, which is randomly sampled in {1000, 2000, 3000, 4000, 5000}. We use $R_{CPU}^{avail} = 8$, $R_{MEM}^{avail} = 16$, $I^{sched} = 0.1$s. The events are obtained from raw IoT data for the macro-benchmark and arbitrarily simulated for the micro-benchmark. The average number of events per I^{sched} is 2. The ratio of generated action container specifications (i.e., T^{cold} and T^{exec}) is approximately 1:1:1 for $T^{cold} < T^{exec}$, $T^{cold} \approx T^{exec}$, and $T^{cold} > T^{exec}$. The deadline of each TAP is set to 90% of its longest T^{exec}.

Evaluation setup. Our experiments are conducted on a PC that is equipped with an Intel Core i7-7700 CPU @ 3.60 GHz, 16 GB DDR4 RAM, and runs on Ubuntu 22.04. The open source components and versions we use are as follows: Docker 20.10.21, TDengine 3.0.1.8, RabbitMQ 3.9.11, Redis 6.2.6, and Nginx 1.23.3.



(a) Deadline miss ratio comparison.

(b) E2E latency and resource usage comparison.

(c) Empirical study of E2E latency between IFTTT, variants of RTX-IFTTT and EdgeRuler.

Fig. 8: Main results of macro-benchmark.

7.2 Macro-benchmark

Deadline miss ratio. Fig. 8a compares the deadline miss ratio of EdgeRuler with the baselines. Note that we run oracle solutions under a simulated environment, making sure that they can generate decisions. Results show that EdgeRuler has a comparable performance with oracle solutions but has a dramatically lower overhead than baselines. Here, the processing latency of ErRo and EdgeRuler is slightly higher than that of CoRo and CoEr because the inaccurate prediction of EdgeRuler will lead to extra trigger events. Moreover, EdgeRuler predictor and scheduling policies are effective, which incur marginally lower the deadline miss ratio ($\sim$4%) compared to CoRo. As for responsive heuristics, they have apparently low runtime overhead (less than 1.8ms) but poor performance. This is because they suffer long cold-start times (Od-based), consume too many resources (Ah- and Gr-based), or easily miss deadlines (Cs-based) because of resource shortage. Intuitively, Ah-based solutions should have better performance compared to Od-based ones because the former can totally get rid of cold-start time. As R^{avail} is limited, if the number of trigger events within a I^{sched} increases, Ah-based solutions will drain R^{avail} for the TAPs that triggered first without releasing, leaving others to miss their deadlines. Od-based ones, though suffer from full cold-start time, can still release resources for TAPs that are triggered afterward and thus have a lower deadline miss ratio.

E2E latency. Fig. 8b compares the E2E latency of EdgeRuler with the baselines. We have the following observations. (1) EdgeRuler has a comparable performance with oracle solutions. (2) Ah-based solutions have counter-intuitively poor performance due to the resource shortage. When there are multiple rules, those triggered first will consume the resources without releasing, and those triggered afterward suffer from high execution time. (3) Cs-based solutions have longer latency because they allocate just enough resources for execution. (4) Od-based solutions suffer from the cold-start time, resulting in higher average E2E latency.

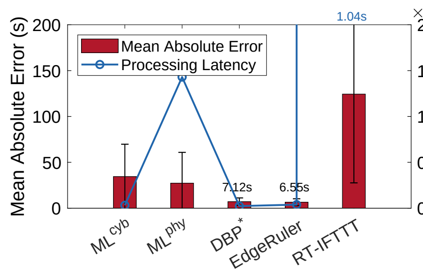
Empirical study. We conduct an empirical study to evaluate EdgeRuler performance in real-world settings with IFTTT and RTX-IFTTT [7]. To strike a fair comparison, we: (1) deploy EdgeRuler to the cloud, i.e., ER(Cloud), and connect RTX to EdgeRuler backend without the scheduler support (RTX-ER); (2) use a simple lamp rule: "IF lamp A is turned on THEN turn on lamp B" as the benchmark because RTX can only facilitate device trigger event identification. Fig. 8c shows the results. EdgeRuler and its variant can reduce an average of 89.1% and 68.4% latency compared to cloud and edge baselines. The main reason for the high latency of IFTTT is its long polling interval. RTX-IFTTT alleviates this part by using the real-time trigger update API. RTX-ER can reduce the E2E latency as the backend locates at the edge. EdgeRuler variants can further reduce the latency by appropriately allocating resources.

Resource usage. We use the consumed CPU volume, i.e., CPU core(s) $\cdot$ time as the metric for evaluating resource usage. E.g., using 4 CPUs for 10 seconds result in 40 CPU seconds consumption. Fig. 8b compares the resource usage of EdgeRuler with the baselines. Results show that EdgeRuler has a comparable performance with the oracle solutions. Although OdGr and OdCs consume slightly smaller resources, they suffer from higher E2E latency and deadline miss ratio. Ah-based solutions, as expected, consume a much higher volume of resources as they keep the container always alive.

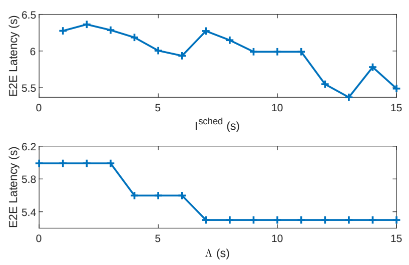
7.3 Micro-benchmark

Predictor performance. Predictors play an important role in container life-cycle control. We compare EdgeRuler solution with predictor baselines in §7.1. We use ML^{cyb} and ML^{phy} to denote machine learning models that predict event occurrence time or raw sensor value. The DBP* here has been applied the *three modifications* and EdgeRuler further adds the error model. Fig. 9a shows the results. We have three main observations. (1) ML^{phy} performs consistently better than ML^{cyb}, the reason of which is that ML^{phy} can extract more information from the physical world. (2) Online predictors (DBP*, EdgeRuler) perform consistently better than offline ones (others). This is an obvious trade-off between accuracy and retraining overhead. We show later that this overhead is negligible. (3) RT-IFTTT performs poorly. The high mean absolute error comes from its conservative strategy (with an event miss ratio of 0.1), which leads to a too-early T^{start}. The high prediction latency is due to its large distribution, which requires exhaustive searching before getting a result.

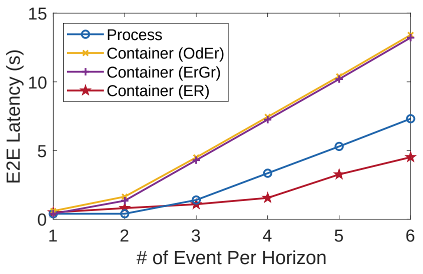
Impacts of hyper-parameters. As EdgeRuler is designed to be *configurable*, there are a bunch of hyper-parameters for users to tune. We cannot go through all the parameters for the page limit. Fortunately, the impact of some parameters



(a) Predictor performance comparison.



(b) Impact of hyper-parameters.



(c) Scalability comparison.

Fig. 9: Main results of micro-benchmark.

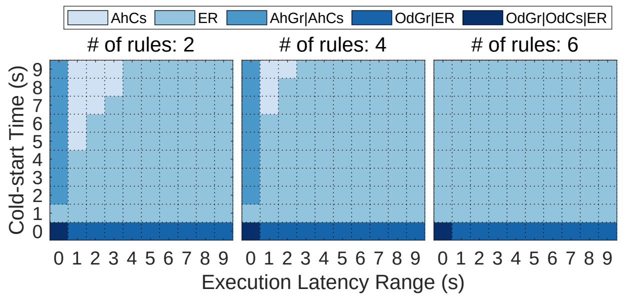


Fig. 10: Impacts of cold-start time, execution latency range, and the number of rules.

can be straightforward to derive, e.g., a smaller τ will obtain more accurate results at a cost of longer solving time. Moreover, the DBP paper has covered the discussion of its parameters [31], [32]. So we focus on the analysis of I^{sched} and Λ. Fig. 9b shows the results. A larger I^{sched} can generally yield a better performance because of the inclusion of more information. However, sometimes the latency counter-intuitively increases with the I^{sched} because of the cross-interval cases. The latency decreases as Λ increases. It is interesting that sometimes the latency remains unchanged as Λ increases, which is because the spared time is not enough to include more TAPs. Note that the computation overhead of the scheduler will go up with the Λ, we should find a sweet point in between.

Scalability analysis. EdgeRuler uses containers as the basic execution unit instead of processes for scalability. Fig. 9c shows the comparison results. We can see that EdgeRuler generally outperforms the process-based solution. This is because TAP execution will queue up after a certain threshold for the latter solution, resulting in a higher E2E latency. To further evaluate the effectiveness of EdgeRuler, we also implement OdEr and ErGr. We can see that only part of the EdgeRuler solution is not enough to alleviate the overhead of using containers.

Impacts of the cold-start time (CT), the execution latency range (ELR), and the number of rules. The evaluation results are highly correlated with the real-world workloads/traces, which are quite scarce in the containerized rule engine scenario. To evaluate EdgeRuler and the baselines more comprehensively, we vary the CT, ELR, as well as the number of rules and run simulations to unveil the performance. Fig. 10 shows the results. The color of each cell indicates the best solution(s) under the (ELR, CT) setting.

TABLE 2: The overhead of EdgeRuler.

Module	Resource Consumption		Latency (ms)
	CPU (Core)	Mem. (MB)	
Predictor	0.18	0.46	<0.01
Perf. Model	0.06	<<0.01	0.06
Scheduler	0.42	79.4	3.7
Runtime	0.02	453.4	NaN

We have the following observations. (1) Ah-based solutions perform better as the CT increases when the ELR is small. This is because they can totally get rid of CT but will suffer when the TAP requires more resources (larger ELR). (2) Od-based solutions perform well when the CT is small and can handle various ELR conditions. (3) EdgeRuler performs well in most cases and is capable of handling more situations.

System overhead. TABLE 2 shows the overhead of EdgeRuler. When idle, the EdgeRuler runtime consumes about 1.6% CPU and ∼450 MB memory, which we believe is acceptable for modern edge cloud platforms. With workloads, it appears that CPU consumption can rise to 0.6%, but with negligible time spent (<3.7ms). Overall, EdgeRuler is a lightweight solution that can be deployed in large-scale containerized IoT edge cloud platforms readily.

8 Related Work

Optimization for IoT rule engines: As an important infrastructure for IoT systems, there exist a wide range of research works that optimize rule engines. One important aspect is the real-time guarantee. IFTTT [6] provides real-time APIs for users to force instance notification of sensing value changes. RT-IFTTT [10] and TinyLink 2.0 [23] offer an enhanced IFTTT syntax and compiler that allow users to write their own IoT TAPs with real-time constraints and save the energy from IoT sensors with flexible polling intervals. RTX-IFTTT [7], on the other hand, offloads the trigger event monitoring task from the cloud to the edge to give real-time service. We have already shown in §7.3 that EdgeRuler predictor performs better than that of RT-IFTTT and TinyLink 2.0. The latter two solutions cannot be applied in containerized IoT rule engines to improve E2E latency for the lack of sophisticated latency models and scheduling algorithms. RTX-IFTTT is orthogonal to EdgeRuler as it focuses on triggers that are not based on raw IoT data.

Container cold start elimination techniques: Container cold start is a well-known problem and there is a large body of work to alleviate, which can roughly be classified into three categories: (1) *Resource sharing*, i.e., accelerate start process by reusing function runtime [16], instance [49], packages [50], and so on. (2) *Optimize virtualization*, i.e., alleviate cold-start by optimizing Docker containers [17] or designing more lightweight virtualization techniques. (3) *Life-cycle control*, i.e., circumvent cold-start by intelligently pre-warming containers [11] or keeping hot ones alive [19]. EdgeRuler solution can be regarded as the third category but gives better results in cold-start elimination by leveraging real-world information and fine-grained data sampling. The other two techniques can also be applied in EdgeRuler to further improve the performance of cold-start elimination.

Resource planning for containers: Existing works on resource management for containerized workloads typically scale resources at a coarse granularity. Some works focus on adjusting the number of replica containers to meet service level objectives (SLOs) or reduce costs [30], [51]–[53]. Other works attempt to allocate CPU cores to containers but still at an integer level [12], [54], [55]. Existing container orchestrators like K8s support fine-grained resource allocation. However, its built-in allocator (i.e., HPA and VPA) fails to determine the right amount of resources for each container to optimize resource efficiency while meeting SLOs. EdgeRulercollectively considers the container life-cycle control and fine-grained resource allocation to strike a good balance between E2E rule execution latency and resource overhead.

9 Conclusion

Motivated by the measurement findings, we present EdgeRuler which couples the IoT rule engine and the container runtime to understand the action container provisioning time and execution time to shorten the service response latency by proactive life-cycle control and fine-grained resource grants. We implement and evaluate EdgeRuler on top of production-ready open-source software according to reference design. Moreover, EdgeRuler follows a lightweight and standard-compatible design principle, which is friendly to deploy in edge cloud platforms for IoT applications.

Acknowledgements

We thank all the reviewers for their valuable comments and helpful suggestions. This work is supported by the National Natural Science Foundation of China under grant no. 62072396 and no. 62272407, the "Pioneer" and "Leading Goose" R&D Program of Zhejiang under grant No. 2023C01033, the National Youth Talent Support Program, the Information Technology Center and State Key Lab of CAD&CG, Zhejiang University. Wei Dong is the corresponding author.

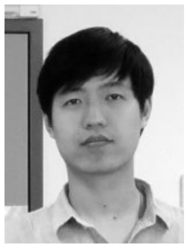
References

[1] SiteWhere LLC., "Rule processing microservice." 2023, https://sitewhere.io/docs/2.1.0/guide/microservices/rule-processing/.
[2] ThingsBoard., "What is thingsboard rule engine?" 2023, https://thingsboard.io/docs/user-guide/rule-engine-2-0/re-getting-started/.
[3] Kaa Enterprise., "Action automation." 2023, https://docs.kaaiot.io/KAA/docs/current/Features/Automation/AA/.
[4] S. Fu and S. Ratnasamy, "dspace: Composable abstractions for smart spaces," in *Proc. of ACM SOSP*, 2021, pp. 295–310.
[5] Gartner, Inc., "Predicts 2021: Building on cloud computing as the new normal." 2020, https://www.gartner.com/en/documents/3994453/.
[6] IFTTT., "Ifttt - connect your apps." 2023, https://ifttt.com/.
[7] K. Dong, Y. Zhang, Y. Zhao, D. Li, Z. Ling, W. Wu, and X. Zhu, "Real-time execution of trigger-action connection for home internet-of-things," in *Proc. of IEEE INFOCOM*, 2022, pp. 1489–1498.
[8] "Ifttt realtime api." 2023, https://ifttt.com/docs/api_reference#realtime-api.
[9] J. McChesney, N. Wang, A. Tanwer, E. De Lara, and B. Varghese, "Defog: fog computing benchmarks," in *Proc. of ACM/IEEE SEC*, 2019, pp. 47–58.
[10] S. Heo, S. Song, J. Kim, and H. Kim, "Rt-ifttt: Real-time iot framework with trigger condition-aware flexible polling intervals," in *Proc. of IEEE RTSS*, 2017, pp. 266–276.
[11] A. Singhvi, A. Balasubramanian, K. Houck, M. D. Shaikh, S. Venkataraman, and A. Akella, "Atoll: A scalable low-latency serverless platform," in *Proc. of ACM SoCC*, 2021, pp. 138–152.
[12] Z. Wen, Y. Wang, and F. Liu, "Stepconf: Slo-aware dynamic resource configuration for serverless function workflows," in *Proc. of IEEE INFOCOM*, 2022, pp. 1868–1877.
[13] B. Hou, S. Yang, F. A. Kuipers, L. Jiao, and X. Fu, "Eavs: Edge-assisted adaptive video streaming with fine-grained serverless pipelines," in *Proc. of IEEE INFOCOM*, 2023.
[14] K. Apicharttrisorn, J. Chen, V. Sekar, A. Rowe, and S. V. Krishnamurthy, "Breaking edge shackles: Infrastructure-free collaborative mobile augmented reality," in *Proc. of ACM SenSys*, 2022, pp. 1–15.
[15] Y. Chen, H. Inaltekin, and M. Gorlatova, "Adaptslam: Edge-assisted adaptive slam with resource constraints via uncertainty minimization," in *Proc. of IEEE INFOCOM*, 2023.
[16] D. Du, T. Yu, Y. Xia, B. Zang, G. Yan, C. Qin, Q. Wu, and H. Chen, "Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting," in *Proc. of ACM ASPLOS*, 2020, pp. 467–481.
[17] J. Thalheim, P. Bhatotia, P. Fonseca, and B. Kasikci, "Cntr: Lightweight OS containers," in *Proc. of USENIX ATC*, 2018, pp. 199–212.
[18] S. Shillaker and P. Pietzuch, "Faasm: lightweight isolation for efficient stateful serverless computing," in *Proc. of USENIX ATC*, 2020, pp. 419–433.
[19] R. B. Roy, T. Patel, and D. Tiwari, "Icebreaker: Warming serverless functions better with heterogeneity," in *Proc. of ACM ASPLOS*, 2022, pp. 753–767.
[20] L. Pan, L. Wang, S. Chen, and F. Liu, "Retention-aware container caching for serverless edge computing," in *Proc. of IEEE INFOCOM*, 2022, pp. 1069–1078.
[21] S. Chen, L. Wang, and F. Liu, "Optimal admission control mechanism design for time-sensitive services in edge computing," in *Proc. of IEEE INFOCOM*, 2022, pp. 1169–1178.
[22] W. Zhang, Y. Gao, and W. Dong, "Providing realtime support for containerized edge services," *ACM Transactions on Internet Technology*, vol. 23, no. 4, pp. 1–25, 2023.
[23] G. Guan, B. Li, Y. Gao, Y. Zhang, J. Bu, and W. Dong, "Tinylink 2.0: integrating device, cloud, and client development for iot applications," in *Proc. of ACM MobiCom*, 2020, pp. 1–13.
[24] Y. Sun, T.-Y. Wu, G. Zhao, and M. Guizani, "Efficient rule engine for smart building systems," *IEEE Transactions on Computers*, vol. 64, no. 6, pp. 1658–1669, 2014.
[25] G. M. Dias, B. Bellalta, and S. Oechsner, "A survey about prediction-based data reduction in wireless sensor networks," *ACM Computing Surveys*, vol. 49, no. 3, pp. 1–35, 2016.
[26] B. Li, W. Dong, G. Guan, J. Zhang, T. Gu, J. Bu, and Y. Gao, "Queec: Qoe-aware edge computing for iot devices under dynamic workloads," *ACM Transactions on Sensor Networks*, vol. 17, no. 3, pp. 1–23, 2021.
[27] N. Roy, A. Dubey, and A. Gokhale, "Efficient autoscaling in the cloud using predictive models for workload forecasting," in *Proc. of IEEE CLOUD*, 2011, pp. 500–507.
[28] A. Zhao, Q. Huang, Y. Huang, L. Zou, Z. Chen, and J. Song, "Research on resource prediction model based on kubernetes container auto-scaling technology," in *IOP Conference Series: Materials Science and Engineering*, vol. 569, no. 5, 2019, p. 052092.

[29] D. Janardhanan and E. Barrett, “Cpu workload forecasting of machines in data centers using lstm recurrent neural networks and arima models,” in *Proc. of IEEE ICITST*, 2017, pp. 55–60.
[30] C. Zhang, M. Yu, W. Wang, and F. Yan, “Mark: Exploiting cloud services for cost-effective, slo-aware machine learning inference serving,” in *Proc. of USNIX ATC*, 2019, pp. 1049–1062.
[31] U. Raza, A. Camerra, A. L. Murphy, T. Palpanas, and G. P. Picco, “What does model-driven data acquisition really achieve in wireless sensor networks?” in *Proc. of IEEE PerCom*, 2012, pp. 85–94.
[32] ——, “Practical data prediction for real-world wireless sensor networks,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 27, no. 8, pp. 2231–2244, 2015.
[33] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt *et al.*, “The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing,” *Proceedings of the VLDB Endowment*, vol. 8, no. 12, 2015.
[34] F. Marković, T. Nolte, and A. V. Papadopoulos, “Analytical approximations in probabilistic analysis of real-time systems,” in *Proc. of IEEE RTSS*, 2022, pp. 158–171.
[35] T. B. Matos, A. Brayner, and J. E. B. Maia, “Towards in-network data prediction in wireless sensor networks,” in *Proc. of ACM SAC*, 2010, pp. 592–596.
[36] H. Jiang, S. Jin, and C. Wang, “Prediction or not? an energy-efficient framework for clustering-based data collection in wireless sensor networks,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 6, pp. 1064–1071, 2010.
[37] Y.-A. Le Borgne, S. Santini, and G. Bontempi, “Adaptive model selection for time series prediction in wireless sensor networks,” *Elsevier Signal Processing*, vol. 87, no. 12, pp. 3010–3020, 2007.
[38] Y. Gao, W. Dong, H. Huang, J. Bu, C. Chen, M. Xia, and X. Liu, “Whom to blame? automatic diagnosis of performance bottlenecks on smartphones,” *IEEE Transactions on Mobile Computing*, vol. 16, no. 6, pp. 1773–1785, 2016.
[39] M. Zhang, J. Cao, L. Yang, L. Zhang, Y. Sahni, and S. Jiang, “Ents: An edge-native task scheduling system for collaborative edge computing,” in *Proc. of IEEE/ACM SEC*, 2022, pp. 149–161.
[40] K. Ye, Y. Kou, C. Lu, Y. Wang, and C.-Z. Xu, “Modeling application performance in docker containers using machine learning techniques,” in *Proc. of IEEE ICPADS*, 2018, pp. 1–6.
[41] Y. Zhang, W. Hua, Z. Zhou, G. E. Suh, and C. Delimitrou, “Sinan: Ml-based and qos-aware resource management for cloud microservices,” in *Proc. of ACM ASPLOS*, 2021, pp. 167–181.
[42] L. Huang, J. Jia, B. Yu, B.-G. Chun, P. Maniatis, and M. Naik, “Predicting execution time of computer programs using sparse polynomial regression,” in *Proc. of NeurIPS*, 2010, pp. 883–891.
[43] B.-G. Chun, L. Huang, S. Lee, P. Maniatis, and M. Naik, “Mantis: Predicting system performance through program analysis and modeling,” *arXiv preprint arXiv:1010.0019*, 2010.
[44] D. Narayanan, F. Kazhamiaka, F. Abuzaid, P. Kraft, A. Agrawal, S. Kandula, S. Boyd, and M. Zaharia, “Solving large-scale granular resource allocation problems efficiently with pop,” in *Proc. of ACM SOSP*, 2021, pp. 521–537.
[45] T. Leesatapornwongsa, A. Sengupta, M. S. Ardekani, G. Petri, and C. A. Stuardo, “Transactuations: Where transactions meet the physical world,” *ACM Transactions on Computer Systems*, vol. 36, no. 4, pp. 1–31, 2020.
[46] “Assign cpu resources to containers and pods.” 2023, https://kubernetes.io/docs/tasks/configure-pod-container/assign-cpu-resource/#if-you-do-not-specify-a-cpu-limit.
[47] X. Chen, Q. Shi, L. Yang, and J. Xu, “Thriftyedge: Resource-efficient edge computing for intelligent iot applications,” *IEEE Network*, vol. 32, no. 1, pp. 61–65, 2018.
[48] C. King, “stress-ng.” 2023, https://github.com/ColinIanKing/stress-ng/tree/master.
[49] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, “SAND: Towards high-performance serverless computing,” in *Proc. of USENIX ATC*, 2018, pp. 923–935.
[50] Z. Li, L. Guo, Q. Chen, J. Cheng, C. Xu, D. Zeng, Z. Song, T. Ma, Y. Yang, C. Li *et al.*, “Help rather than recycle: Alleviating cold startup in serverless computing through Inter-Function container sharing,” in *Proc. of USENIX ATC*, 2022, pp. 69–84.
[51] G. Yu, P. Chen, and Z. Zheng, “Microscaler: Automatic scaling for microservices with an online learning approach,” in *Proc. of IEEE ICWS*, 2019, pp. 68–75.
[52] J. R. Gunasekaran, P. Thinakaran, N. C. Nachiappan, M. T. Kandemir, and C. R. Das, “Fifer: Tackling resource underutilization in the serverless era,” in *Proc. of ACM/IFIP/USENIX Middleware*, 2020, pp. 280–295.
[53] X. Shang, Y. Mao, Y. Liu, Y. Huang, Z. Liu, and Y. Yang, “Online container scheduling for data-intensive applications in serverless edge computing,” in *Proc. of IEEE INFOCOM*, 2023.
[54] Y. Han, S. Shen, X. Wang, S. Wang, and V. C. Leung, “Tailored learning-based scheduling for kubernetes-oriented edge-cloud system,” in *Proc. of IEEE INFOCOM*, 2021, pp. 1–10.
[55] T. Ouyang, K. Zhao, X. Zhang, Z. Zhou, and X. Chen, “Dynamic edge-centric resource provisioning for online and offline services co-location,” in *Proc. of IEEE INFOCOM*, 2023.

Wenzhao Zhang received the BS degree from the college of Computer Science and Engineering, Northeastern University of China in 2018. He received the PhD degree from the College of Computer Science and Technology, Zhejiang University in 2023. His research interests include Internet of Things and edge computing.

Yixiao Teng received the BS degree from the college of Computer Science and Engineering, Nanjing University of Science and Technology in 2022. She is currently working toward a Master degree in the College of Computer Science and Technology, Zhejiang University. Her research interests include edge computing and blockchain.



Yi Gao received the BS and PhD degrees from Zhejiang University, in 2009 and 2014, respectively. He is currently a research assistant professor with Zhejiang University, China. From 2015 to 2016, he visited McGill University as a visiting scholar. His research interests include network measurement, sensor networks, and Internet of Things.

Wei Dong received the BS and PhD degrees from the College of Computer Science, Zhejiang University, China, in 2005 and 2011, respectively. He is currently a full professor in the College of Computer Science, Zhejiang University, China. His research interests include Internet of Things and sensor networks, wireless and mobile computing, and network measurement.