SLO-aware Virtual Rebalancing for Edge Stream Processing

Peng Kang[1], Palden Lama[1], and Samee U. Khan[2]

[1]The University of Texas at San Antonio
[1]{peng.kang, palden.lama}@utsa.edu
[2]Mississippi State University
[2]skhan@ece.msstate.edu

Abstract—**The Internet of Things (IoT) has enabled an abundance of geographically distributed physical devices or "things" equipped with sensors and actuators to exchange information with the Cloud. However, this paradigm remains largely under-exploited for real-time analytic applications. The benefit of real-time data acquisition at the Edge becomes fruitless as it is not readily accessible to more powerful data analytic tools in the Cloud due to wide-area network delays. In this paper, we present VRebalance, a virtual resource orchestrator that provides an end-to-end performance guarantee for concurrent stream processing workloads at the Edge. VRebalance employs Bayesian Optimization $\mathcal{BO}$ to quickly identify near-optimal resource configurations. Experimental results with a real-time open-source IoT benchmark for Distributed Stream Processing Platforms (RIoTBench) and a representative stream processing engine (Apache Storm) demonstrate the superior performance, resource efficiency and adaptiveness of our $\mathcal{BO}$-based resource management system. VRebalance meets the performance SLO (service level objective) targets for stream processing workloads even in the presence of acute system dynamics. It decreases the SLO violation rate by at least 34% for static workloads and by 62.5% for dynamic workloads compared to a hill climbing method. Compared to Storm's default resource scaling mechanism, our method decreases the SLO violation rate by 83.7%.**

Index Terms—**Internet of Things, Stream Processing, Bayesian Optimization, Resource Management.**

I. Introduction

The Internet of Things (IoT) applications need to continually process data streams produced by devices for data integration and control operations within intelligent systems [1]–[3]. However, IoT systems that rely on the cloud for data processing can not reap the benefits of real-time data acquisition at the edge due to wide-area network delays and jitters [4], [5]. In most cases, such data needs to be processed with very strict time constraints to extract useful information for future actions. In some cases, data must be processed in almost real-time to attain any meaningful insights or detect patterns of interest. For example, making the power grid "smart" depends on the ability to wrangle the unprecedented influx of sensing data to identify critical events and automate the available controls to maintain grid stability [6]. Similarly, IoT based traffic control in smart cities will require real-time analysis of live video streams from cameras deployed at myriad of intersections within major cities [7]. Hence, it is desirable to perform data processing at the Edge, for instance using IoT gateways which are one-hop away from the IoT devices [8].

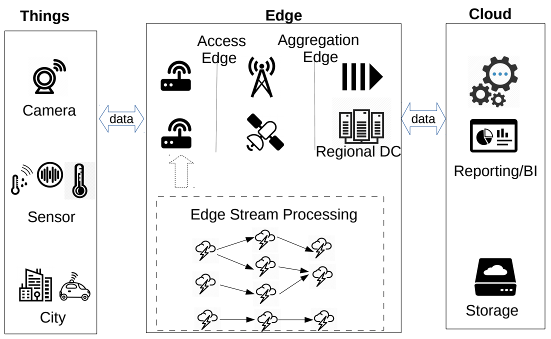


Fig. 1: IoT: Device to Cloud.

Stream processing has emerged as a dominant distributed computing paradigm for processing and analysis of high-volume, heterogeneous, and continuous data streams to extract insights and actionable results in real time [9]. A stream processing application is described by a directed acyclic graph, called a topology, whose vertices are data processing operations and edges indicate data flow as shown in Fig. 1. However, popular stream processing engines (SPEs), such as Apache Storm [1], Apache Spark [2] etc. are designed for resource-rich cloud environments. Hence, they may not perform optimally in a resource-constrained Edge environment. For instance, our motivational case study in Section II shows that Storm's default resource scaling mechanism for a running topology is very slow (takes several minutes to take effect) and can also lead to inefficient resource usage.

Recent studies [10]–[12] have shown that IoT workloads from different application domains can have diverse and highly dynamic characteristics in which the arrival rate of events can change drastically with time. Therefore, the need for adapting the resource configuration of a stream processing engine at the Edge in an agile and efficient manner becomes critical. Existing works [13], [14] on resource scaling for distributed stream processing mainly focus on removing bottlenecks in a single stream processing topology to maximize the average

[1]http://storm.apache.org/
[2]https://spark.apache.org/

performance. Furthermore, these techniques are often limited by the latency of the scaling mechanism of the SPEs. On the other hand, this paper aims to meet the end-to-end tail latency targets of concurrent stream processing workloads through agile and fine-grained resource control at the system level.

We present VRebalance, a virtual resource management system that provides an end-to-end performance guarantee for concurrent stream processing applications at the Edge. VRebalance employs Bayesian Optimization $\mathcal{BO}$ [15] to quickly identity near-optimal resource configurations that minimize the violation of performance SLO targets in terms of 95^{th}-percentile latency. It is noteworthy to mention that VRebalance does not require prior knowledge about the application or expensive workload profiling. Furthermore, it meets the performance SLO targets for stream processing workloads even for changing system dynamics. Experimental results with an open-source IoT benchmark (RIoTBench) [16] and a representative stream processing engine (Apache Storm), demonstrate the superior performance, resource efficiency and adaptiveness of our resource management system.

Our key contributions are:

- We analyze the limitations of existing SPEs (i.e Apache Storm) in quickly adapting the parallelism of a running stream processing topology and highlight its resource inefficiency in the Edge settings.
- We design and develop a SLO-aware virtual resource orchestrator based on $\mathcal{BO}$ that can quickly identify and deploy near-optimal resource configurations for stream processing applications at the Edge.
- We develop an algorithm for coordinated resource configuration of concurrent stream processing applications based on suggestions collected from multiple $\mathcal{BO}$ models. Furthermore, we handle dynamic workloads by efficiently utilizing $\mathcal{BO}$ models corresponding to different ranges of workload intensity.
- We implement and evaluate our system utilizing a laboratory-sized real testbed as an edge server using the RIoT benchmark. By comparing a hill-climbing method with $\mathcal{BO}$, experimental evaluations demonstrate that the latency SLO violations were reduced from 31.3% to 11.3% at the exploration phase of $\mathcal{BO}$ and to 8.5% at the exploitation phase of $\mathcal{BO}$. Compared to Storm's default resource scaling technique, our method decreases the SLO violation rate by at least 83.7%.

The remainder of the paper is organized as follows. Section II discusses the background of stream processing model and describes the challenges that we address. Section III elaborates the key design and implementation details. Section IV details the testbed setup and experimental results. In Section V, we present the related work. Finally, conclusions are drawn in Section VI.

II. BACKGROUND AND MOTIVATION

In this section, we provide background on stream processing models, limitations of existing SPEs in Edge settings, and the challenges of choosing the best resource configuration for a stream processing topology.

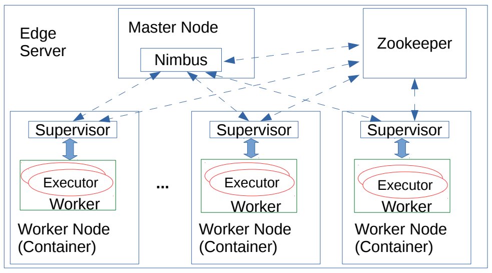


Fig. 2: Apache Storm Architecture.

A. Edge Stream Processing Model

1) Edge Data Stream Model: We consider edge SPEs on an IoT Gateway, similar to the model considered in a recent work [17]. These IoT Gateways have limited computing resources compared to the Cloud; however, are with more resources than those available to embedded, wireless sensor networks, etc. As shown in Fig. 1, data streams produced by IoT devices are processed by an IoT Gateway running multiple topologies. Stream processing follows the dataflow programming model [18], where each application is packaged as a directed acyclic graph (DAG) data structure, called a topology. Individual data points (tuples) flow through a topology from sources to sinks. Each inner node is an operation that performs arbitrary computation on the data, ranging from simple filtering to complex operations like Machine Learning (ML)-based classification algorithms. We assume that each application has an SLO target in terms of the 95^{th}-percentile end-to-end latency of data tuples flowing through the topology.

2) Stream Processing Engines: In this paper, we use Apache Storm as a representative SPE. Storm is a distributed real-time computation system to process unbounded streams of data. A Storm topology is a DAG of spouts and bolts, where a spout is a source of data streams and a bolt is a data processing unit. Storm is typically deployed on a cluster using the master-worker architecture as shown in Fig. 2. On the master node, a process called Nimbus is responsible for distributing code within the cluster, assigning tasks to workers, and monitoring for failures. A worker node consists of a supervisor process and some worker processes. The supervisor listens for work assigned to the worker node and starts and stops worker processes as necessary. A worker process belongs to a specific topology and may run one or more executors (threads) for one or more components of this topology. An executor may run one or more tasks belonging to a particular component.

3) Containerization of SPE: As shown in Fig. 2, we deploy a Storm cluster composed of Docker containers running on an IoT Gateway, where each container serves as a worker node. This deployment methodology benefits from the resource isolation and fine-grained resource control features provided by container-based virtualization while incurring minimal overheads as compared to hardware virtualization.

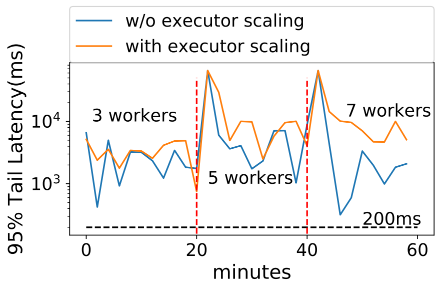


Fig. 3: Storm rebalance with and without executor scaling. Both case rebalanced from 3 to 5 workers at time 20 minutes and from 5 to 7 workers at time 40 minutes. ETL topology from RIoTBench benchmark suite is used to process CITY dataset. SLO target is 200ms. All workers is running the same node equipped with 4 CPU cores and 8GB memory.

B. Limitations of Existing SPEs in Edge Settings

Although existing SPEs provide the mechanism to change the parallelism of a running topology for dynamic resource allocation, they can be quite inefficient. Changing the parallelism of a running topology means that users can increase or decrease the number of worker processes and/or executors without a restart of the cluster or topology. This act is called rebalancing. Rebalancing a running topology is associated with significant latency as it involves checkpointing the dataflow, deactivating the topology for the certain duration, redistributing the worker processes evenly around the cluster, and restoring the topology from the checkpoint. The rebalancing latency can get even worse with increasing workload intensity. To demonstrate this limitation of existing SPEs, we conducted an experiment using a containerized Storm cluster running on a prototype testbed (more details in Section IV-A). We used the ETL(Extraction, Transform, and Load) topology from the RIoTBench benchmark suite [16] to process the "Sense your City" [19] dataset. The workload intensity was set to 100K tuples per minute.

First, we used Storm's default rebalancing mechanism to scale up the number of workers from 3 to 5 at time 20 minutes, and from 5 to 7 at time 40 minutes. Initially one executor was spawned for each component. The total number of executors was not changed. As shown in Fig. 3, Storm rebalancing was able to slowly improve the 95^{th}-percentile end-to-end latency of this application. However, it took several minutes to take effect. This is mainly due to the overheads associated with rebalancing the topology. Next, we repeated the experiment but this time we also doubled the number of executors for selected components with increase in the number of workers. In this case, the tail latency did not show any improvement. This is because having more executor threads per worker increases thread-scheduling overheads in a resource-constrained worker.

C. Challenges of Resource Allocation

Due to limited Edge resources, it is important to allocate resources efficiently while satisfying the SLO target of each application (topology) running on the Edge node. This is a challenging problem. Since each component of a topology performs a different operation and have different resource demands, arbitrary allocation of resources can easily violate the SLO target. There are strawman solutions for finding an optimal solution. One is to model the application performance and then pick the best configuration. However, this methodology has poor adaptivity. Building a model that works for a variety of applications and configurations can be difficult and tedious because prior knowledge of the internal structure of a specific application is needed to make the model effective. Another way is to exhaustively search for the best configuration without relying on an accurate performance model. However, this methodology has a high overhead. When the running environment changes (a new application is added or the workload is changed for one of the applications), we must search again to make sure that our SLO target is not violated.

III. METHODOLOGY

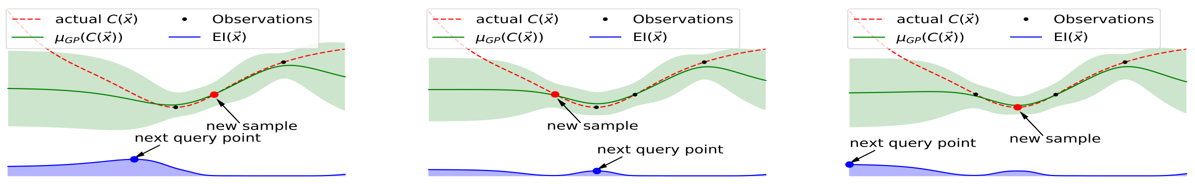
In this section, we present the key design principles and implementation of VRebalance — an adaptive SLO-aware system that aims to provide a strong performance guarantee for Edge stream processing.

A. VRebalance: Overview

VRebalance adopts the following design principles to overcome the limitations of existing SPEs and the challenges of Edge resource allocation. **(1) "Be frugal and be agile".** In a resource-constrained environment, it is important to allocate only the minimum resources required to meet the SLO target of each application. This will allow more applications to benefit from the limited resources. Furthermore, there should be minimum delay between resource allocation and performance improvement. VRebalance focuses on dynamically configuring resource usage limits of containerized SPE workers to meet the SLO targets instead of scaling the number of workers. This approach is not only resource-efficient but also agile since it completely avoids the latency of rebalancing a topology. **(2) "Use models that are just accurate enough to separate near-optimal solutions from the rest".** This strategy allows VRebalance to quickly identify good resource configurations for concurrent stream processing topologies without requiring a priori knowledge about them or conducting expensive workload profiling. In particular, it uses $\mathcal{BO}$ to intelligently explore efficient resource configurations while observing their respective performance.

B. Problem Formulation

Considering an Edge stream processing application and workload, we aim to find a near-optimal resource configuration that satisfies the specified SLO target while minimizing the amount of resource allocated. In this paper, resource configurations include the CPU usage limits of containerized SPE workers and the SLO target is set in terms of the end-to-end tail latency of an application. We did not include Memory as a candidate for dynamic resource configuration. This is because



(a) t = 3.

(b) t = 4.

(c) t = 5.

Fig. 4: An illustration of $\mathcal{BO}$ procedure over 3 iterations that minimize an objective function $C(\vec{x})$ with a 1-dimensional continuous input.

to achieve low latency an SPE must be able to perform message processing without having costly storage operations in the critical processing path [20]. Exploring Memory configuration online could lead to thrashing or other unpredictable behaviors.

We formulate the problem as follows:

$$\underset{\vec{x}}{minimize} C(\vec{x}) = \max\left(tail\_latency, target - tail\_latency\right) \times \sum \vec{x}$$
$$subject\, to \sum \vec{x} \leq CPU^{total}, \tag{1}$$

where $C(\vec{x})$ is the total cost associated with resource configuration $\vec{x}$ and CPU^{total} is the maximum available CPU resource at the edge server. $\vec{x}$ is a vector of normalized CPU usage limits of containerized SPE workers. For configurations that meet the SLO target, the cost function $C(\vec{x})$ accounts for possible over-provisioning of resources. If SLO target is not met, then $C(\vec{x})$ is dominated by the measured tail latency.

C. $\mathcal{BO}$

$\mathcal{BO}$ is a method of optimizing an expensive black-box function [21], which is well-suited for VRebalance. Here, black-box implies to the unknown relationship between the input and objective function; however, the relationship can be derived through observations (experiments, probing, monitoring, etc.). In the context of VRebalance, evaluation of the objective function involves applying a candidate resource configuration and measuring its impact on the end-to-end tail latency of an application. This is indeed an expensive process as exploring resource configurations iteratively can either lead to SLO violations due to under-provisioning or resource wastage due to over-provisioning. The strength of $\mathcal{BO}$ lies in finding near-optimal solutions in *only* few iterations.

As $\mathcal{BO}$ explores the search space by evaluating the objective function with different input samples (configurations), it builds a probabilistic model (aka *surrogate model*) for the objective function to estimate the performance of different configurations. This probabilistic model is iteratively updated based on subsequent evaluations of the objective function. As shown in Fig. 4a, the dashed red line is the actual function $C(\vec{x})$. $\mathcal{BO}$ computes a confidence interval, which is marked green, with three observed points. The green solid line shows the expected value of $C(\vec{x})$ and the value of $C(\vec{x})$ at each input point $\vec{x}$ falls within the confidence interval of 95% probability. The confidence interval is updated (posterior distribution in Bayesian Theorem) after new samples are taken in Fig. 4b and Fig. 4c. The estimate of $C(\vec{x})$ improves as the area of the confidence interval decreases.

Next, $\mathcal{BO}$ explores different neighborhoods of the search space with the help of a pre-defined acquisition function, which strikes a balance between "exploration" (exploring the areas of the search space with large uncertainty) and "exploitation" (exploiting the non-sampled areas where the predicted mean value of the surrogate model is high). In Fig. 4, the blue dash line with a blue area is the acquisition function. The point $\vec{x}$ with the highest value of the acquisition function is chosen as the next sampling point. Note that as more points are sampled, the surrogate model is updated and the acquisition function is reevaluated after each sample.

D. *Design Options and Optimizations*

Surrogate Model. We use Gaussian Process (GP) as the surrogate model to define the prior/posterior distribution over the objective function. GP is a non-parametric model that provides probability distribution over all the possible functions that are consistent with the observed data. Although the prior/posterior in $\mathcal{BO}$ can be defined by a variety of conjugate distributions, we pick GP because it is flexible enough to approach the actual function given enough data samples, and is also computationally tractable. Furthermore, it is broadly accepted as a surrogate model [22]–[24]. Based on our empirical results, we choose RationalQuadratic [21] kernel as the covariance kernel function with GP model. The covariance function determines the similarity between two resource configurations. We use the terms *surrogate model* and $\mathcal{BO}$ model interchangeably. **Acquisition function.** There are various strategies for designing an acquisition function. Popular methods include: probability of improvement (PI), expected improvement (EI) and upper confidence bounds (UCB) [24]. We use EI to design an acquisition function so that it picks the next sampling point with the aim of maximizing the expected improvement over the current best. EI is chosen for VRebalance as it provides practical balance between exploration vs. exploitation at a low evaluation cost. Other options such as PI often gets stuck in local optima, and UCB requires additional parametric tuning [24].

Expected improvement is defined as:

$$EI(\vec{x}) = \mathbb{E}\left[max\left\{f(\vec{x}) - f(\vec{x}^{+}), 0\right\}\right], \tag{2}$$

where $f(\vec{x}^{+})$ is the value of the best sample (until now) and $\vec{x}^{+}$ is the location of that sample i.e. $f(\vec{x}^{+}) = min_{\vec{x}}\left\{C(\vec{x}) | \vec{x} \in X_t\right\}$. Therefore, we can calculate EI as [25]:

$$EI(\vec{x}) = \begin{cases} (\mu(\vec{x}) - f(\vec{x}^{+}))\Phi(Z) + \sigma(\vec{x})\phi(Z) & if \sigma(\vec{x}) > 0 \\ 0 & if \sigma(\vec{x}) = 0, \end{cases} \tag{3}$$

where

$$Z = \begin{cases} \frac{(\mu(\vec{x}) - f(\vec{x}^{+}))}{\sigma(\vec{x})} & if\, \sigma(\vec{x}) > 0 \\ 0 & if\, \sigma(\vec{x}) = 0, \end{cases} \tag{4}$$

where $\mu(\vec{x})$ and $\sigma(\vec{x})$ are the mean and the standard deviation of GP posterior distribution at $\vec{x}$, respectively. $\Phi(Z)$ and $\phi(Z)$ are the cumulative distribution and probability density function of the (multivariate) standard normal distribution, respectively.

Reducing the Search Space. To facilitate faster $\mathcal{BO}$ convergence, we reduce the search space through quantization of resource configuration. We use a quantization step of 50 millicores for the CPU usage limit of a containerized SPE worker, where 1000 millicores is equivalent to 1 CPU core [3]. Furthermore, we set the minimum CPU limit as 150 millicores and maximum CPU limit as 4,000 millicores. The maximum CPU limit represents the total CPU resource available at the edge server. As a result of the quantization, the overall search space is reduced from $3,850 \times n$ configurations to $77 \times n$ configurations, where n is the number of SPE workers.

Exploration, Exploitation, and Stopping Condition. The stopping condition for $\mathcal{BO}$ search in VRebalance is defined as follows. After observing N_{min} resource configurations, VRebalance switches from exploration to exploitation phase. In the exploitation phase, VRebalance starts to select the best configuration observed so far. However, if there are two consecutive SLO violations, VRebalance starts $\mathcal{BO}$ search again until a total of N_{max} resource configurations have been observed. After observing N_{max} resource configurations, VRebalance will stay in the exploitation phase even if SLO violations occur. Such a situation indicates that there is not enough resources available, and further exploration of resource configuration can be counter-productive. In current setup, we empirically choose N_{min} and N_{max} as 5 and 16 respectively.

Handling workload changes. The overheads associated with $\mathcal{BO}$ exploration can be significant if the workload changes very often. This issue is partly addressed by the fact that VRebalance switches from exploration to exploitation phase after observing N_{min} resource configurations, and remains in this phase until it observes two consecutive SLO violations. Hence, small fluctuations in the workload will not trigger unnecessary $\mathcal{BO}$ search. Instead, when the workload intensity changes drastically, the $\mathcal{BO}$ model gets updated with new observations. This enables $\mathcal{BO}$ to find good resource configuration that meets the SLO target in the face of a dynamic workload. For further speeding up $\mathcal{BO}$ adaptation to changing workload, we use separate $\mathcal{BO}$ models for different ranges of workload intensity. Each range of workload intensity is considered as a workload level (l). VRebalance keeps track of the number of resource configurations observed at each l. When the workload changes back to a previously seen level, VRebalance reuses the corresponding $\mathcal{BO}$ model.

[3]https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/

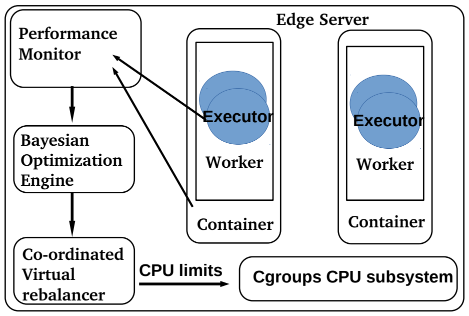


Fig. 5: The Architecture of VRebalance.

E. Putting It All Together

Fig. 5 shows the interaction between various components of VRebalance and containerized SPE workers running on an Edge node. Apache Storm is used as a representative SPE in our work. We consider multiple stream processing applications sharing the Edge node, where each application (topology) consists of a group of SPE workers. We assume that each container hosts only one SPE worker. We will refer to various line numbers in Algorithm 1 to describe the overall operation of VRebalance. The *Performance Monitor* collects data (line 7) for each application running on the Edge server at a sampling interval of one minute. The performance data includes the average throughput and the end-to-end tail latency of concurrent stream processing workloads. The throughput of a topology is measured by counting the number of tuples produced by the farthest downstream components. The end-to-end latency of a topology is measured by assigning each tuple a unique ID and calculating the difference between their timestamps recorded at the spout and the sink. It is possible for VRebalance to operate in the exploration phase for some applications, while operating in the exploitation phase for others. This depends on the number of resource configurations or SLO violations observed so far (lines 8-12). In the exploration phase (lines 9-10), the *Bayesian Optimization Engine* updates the workload-specific $\mathcal{BO}$ model for an application a based on observed data, and uses its acquisition function to fetch a resource configuration candidate c_i^a to be evaluated next. In the exploitation phase (lines 11-12), VRebalance bypasses the $\mathcal{BO}$ models and selects the best configuration observed so far for an application at the particular workload level. It also keeps track of the set of applications A_i' for which it is operating in the exploitation phase (line 12).

The *Co-ordinated Virtual Rebalancer* calculates the total CPU resources to be allocated $CPU^{finalized}$ for the set of applications A_i' (line 14). These applications are given

Algorithm 1: VRebalance Algorithm.

```
1: Let $A'_i$ denote a set of apps whose resource config(CPU
   limit for each container) at
   interval $i$ has been finalized.
2: for $i = 1$ to $infinity$ do
3:     $A_i \leftarrow \{a|app\ running\ on\ the\ edge\ node\}$;
4:     $A'_i \leftarrow \emptyset$;
5:     for all $a \in A_i$ do
6:         $l \leftarrow$ workload level(tuple per minute) of app $a$;
7:         $p \leftarrow$ performance score based on latency and
           throughput of app $a$;
8:         if $\mathcal{BO}\_iteration^l_a < N_{min}$ or
           ($\mathcal{BO}\_iteration^l_a < N_{max}$ and
           two consecutive SLO violations occur) then
9:             use p and $c^{i-1}_a$ to update $BO\_model^l_a$;
10:            $c^i_a \leftarrow$ ask next config from $\mathcal{BO}\_model^l_a$
   based on Eq.1;
           else
11:            $c^i_a \leftarrow$ best config observed for workload $l$;
12:            $A'_i \leftarrow A'_i \cup \{a\}$;
           end
13:    end for
14:    $CPU^{finalized} \leftarrow \sum_{a \in A'_i} c^i_a$;
15:    $CPU^{avail} \leftarrow CPU^{total} - CPU^{finalized}$;
16:    $CPU^{need} \leftarrow \sum_{a \in A_i - A'_i} c^i_a$;
17:    for all $a \in A_i$ do
18:        if $CPU^{need} > CPU^{avail}$ and $a \notin A'_i$ then
19:            $c \leftarrow (c^i_a / CPU^{need}) \times CPU^{avail}$;
           else
20:            $c \leftarrow c^i_a$;
           end
21:        use c to update the resource config of app $a$;
22:    end for
23: end for
```

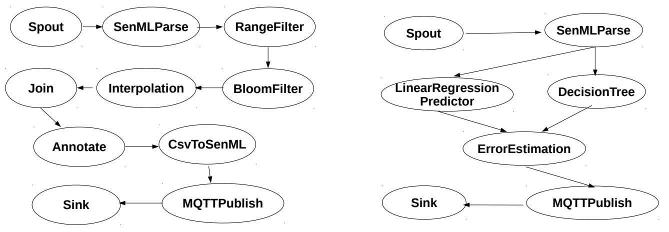
a higher priority for resource allocation than others. This is because good-enough resource configurations are already found for these applications. Then, it calculates the amount of CPU resources that would still be available after allocating $CPU^{finalized}$, and the total CPU resources needed CPU^{need} by the remaining applications (lines 15-16). If CPU^{need} is greater than the available resources, resources are allocated to each remaining application in proportion to its need (lines 18-19). If SLO violations continue to occur over multiple intervals, adding more nodes may be required. We explore resource management with multiple edge nodes in the future work. The resource configuration of each application(/sys/fs/cgroup/cpu,cpuacct/kubepods/burstable/pod/cpu.cfs_quota_us) is updated through cgroup CPU subsystem[4].

IV. Evaluation

A. Experimental Testbed

1) Edge node Configuration.: We setup a prototype testbed to represent an IoT Gateway by using a Ubuntu 16.04 machine equipped with 4 CPU cores and 8 GB RAM. We used Docker container engine (Version 18.06.2-ce) and Kubernetes (Version 1.18.2) container orchestration system to deploy an Apache Storm cluster composed of 12 containers. Two of the containers were used for Nimbus and Zookeeper, while the remaining containers were running as worker nodes. In our experiments, the default CPU limit of each container was set to 400 millicore (equal to 0.4 CPU core) and the requested CPU was 200 millicore.

[4]https://www.kernel.org/doc/Documentation/scheduler/sched-bwc.txt



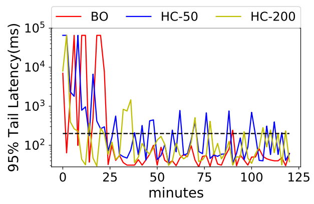
(a) Layout of ETL topology. (b) Layout of PRED topology.

Fig. 6: DAG adapted from RIOTBench.

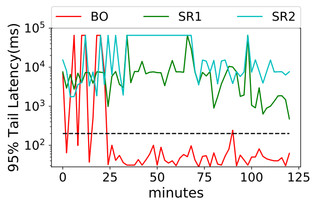
2) Datasets.: We used two IoT datasets including Sense your City [19] and NY city taxi trips [26]. **Sense your City (CITY)** is a real-world Smart Cities data stream that was collected from crowd-sourced sensors deployed across seven cities in three continents, with about 12 sensors per city [16]. Six timestamped observations including temperature, humidity, ambient light, sound, dust and air quality, are reported every minute by each sensor along with metadata on sensor ID and geolocation. **The New York City taxi trips (TAXI)** data offers a stream of smart transportation messages that arrive from 2M trips taken in 2013 on 20,355 New York city taxis equipped with GPS. Each trip provides the pickup and drop-off dates, the taxi and license details, the start and end coordinates and timestamp, the metered distance records by the taximeter, the taxes, and tolls paid. We used aggregated data from January 2013 for our benchmark runs [11], [27]. Throughout the paper, we used CITY dataset to generate a static workload and TAXI dataset to generate a dynamic workload.

3) Benchmarks.: We used the RIoTBench benchmark suite [16], which includes four IoT applications based on common IoT patterns for data pre-processing, statistical summarization, and predictive analytics. We choose two applications ETL (Extraction, Transform, and Load) and PRED (Predictive Analytics) to analyze our datasets as shown in Fig. 6. We do not use the other two applications (STATS and TRAIN) because they are integrated with public cloud services and are unsuitable for low-latency Edge stream processing. To produce a dynamic workload in our experiments, we modified RIoTBench's input generator as follows. At each one minute interval, the input generator feeds multiple fixed-sized batches (10 tuples) of data into Storm's source task (Spout). These data batches are interspersed with random delays that follow a Poisson distribution [28]. We change the total number of data batches at each interval to change the workload intensity.

4) Measuring tail latency: The simplest way to calculate the 95^{th}-percentile latency is to sort all the measured latencies



(a) $\mathcal{BO}$ vs Hill climbing.



(b) $\mathcal{BO}$ vs Storm rebalance.

Fig. 7: Tail latency comparison under static workload.

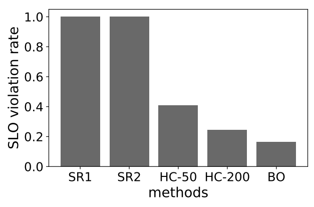


Fig. 8: SLO violation rates under static workload.

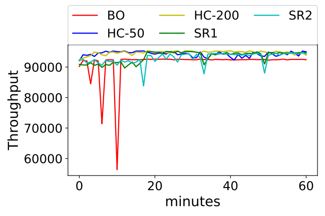


Fig. 9: Throughput comparison under static workload.

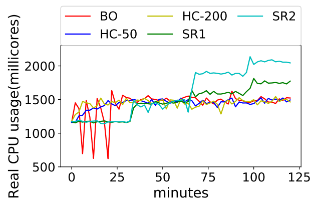
per tuple, and take the $95/100^{th}$ value. However, this method has a significant overhead when the workload arrival rate is high. We address this issue by using a histogram approximation method [29]. Instead of storing and sorting all values, we bin them into groups. Our sampling interval is one minute so latency values will be in the range of 0 to 60,000ms. We can use histogram bins that double in size from 1ms to 60,000ms, for example(0-1ms], (1,2ms], (2,4ms], ..., (512, 1024ms], etc. Extending to 65536ms (2^{16}) would give us 18 bins. Each bin will record the count of values that belong to its range. Thus we only need to store 18 counts, instead of the unbounded latency values. Table I illustrates this method. Here, the first column represents the range of the bin, and the second column is the count of values within that bin. The third column shows a running total of counts seen until that row. Finally, the eCDF(x) represents the empirical cumulative distribution function, which is calculated as the running total divided by the sum of all counts. In this example, the 95^{th}-percentile latency lies between 128ms and 256ms. We use linear interpolation to find its exact position (the value is 136.7ms) in the bin. Since the accuracy of this method relies on the number of bins, we doubled the number of bins (from 18 to 36). This method gives us a good trade-off between speed and accuracy.

B. *Performance Comparison under Static Workload*

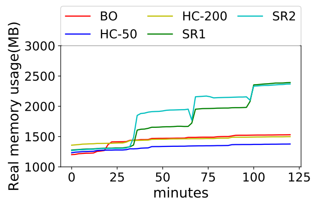
We evaluate the performance and resource efficiency of our $\mathcal{BO}$-based resource management system under a static workload of 100k tuples per minute using CITY dataset and

TABLE I: Example for histogram bin.

Bin Range	Count	Total	eCDF(x)
...	...	...	...
(16, 32ms]	2026	5435	58.8%
(32, 64ms]	1269	6704	72.5%
(64, 128ms]	2061	8765	94.7%
(128, 256ms]	403	9168	99.1%
...	...	...	...



(a) Total CPU usage.



(b) Total memory usage.

Fig. 10: Resource usage comparison under static workload.

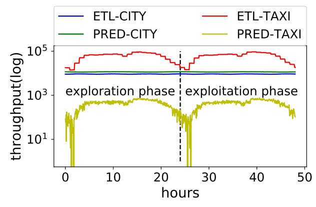
ETL topology. The SLO target for the end-to-end tail latency is set to be 200 ms. For performance comparison, we use the hill climbing-based search method with different step sizes (HC-50 and HC-200) to find the optimal resource configuration. We also use Storm Rebalance without executor scaling (SR1) and with executor scaling (SR2). Both SR1 and SR2 adds one worker to the Storm cluster every 30 minutes.

Fig. 7 shows that $\mathcal{BO}$ outperforms all other methods in quickly meeting the SLO target. As shown in Fig. 8, it decreases the SLO violation rate by at least 34% compared to hill climbing method. This is because hill climbing method suffers from decision making based on local behavior of the objective function. On the other hand, $\mathcal{BO}$ makes more informed decisions by modeling the global behavior of the objective function based on sampled data. Compared to the Storm Rebalancing method, $\mathcal{BO}$ decreases the SLO violation rate by 83.7%. SR1 and SR2 are too slow in reducing the tail latency due to overheads associated with rebalancing the ETL topology whenever new workers are added. It is also worth mentioning that increasing the number of workers increases the inter-process communication cost as well. Unlike Storm Rebalance, our method which updates Container CPU limits has an immediate impact on the tail latency.

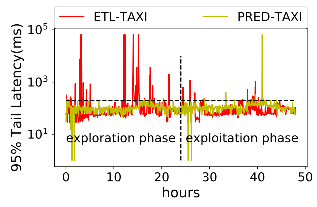
Fig. 9 shows that the throughput achieved with the various methods are similar for most of the experiment duration. However, in the beginning of the experiment, $\mathcal{BO}$ causes the throughput to drop on few sampling intervals. This is due to some aggressive configuration changes made by $\mathcal{BO}$ during initial exploration phase. Fig. 10 shows that $\mathcal{BO}$ is comparable to hill climbing method and significantly better than Storm Rebalancing method in terms of resource efficiency. This is because unlike Storm Rebalancing, both $\mathcal{BO}$ and hill climbing methods perform fine-grained resource allocation instead of scaling the number of workers.

C. *Performance Comparison under Dynamic Workload in the presence of Multiple Applications*

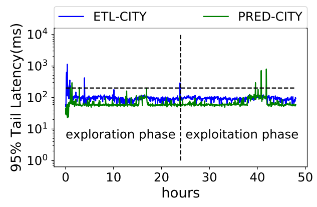
Next, we evaluate $\mathcal{BO}$'s ability to meet the SLO targets of multiple stream processing applications hosted on an Edge node under dynamic workload conditions. We run four applications (ETL-TAXI, ETL-CITY, PRED-CITY, PRED-TAXI) from the RIoTBench benchmark suite using different topology and dataset combinations. As shown in Fig. 11 (a), ETL-TAXI and PRED-TAXI face a dynamic workload, whereas ETL-CITY and PRED-CITY face a static workload. The total workload duration is 48 hours with repeating workload patterns.



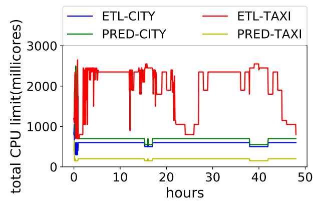
(a) Throughput achieved.



(b) Latency with TAXI dataset.

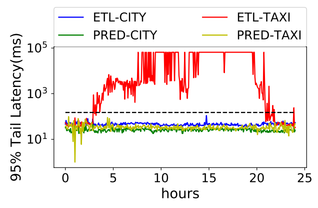


(c) Latency with CITY dataset.

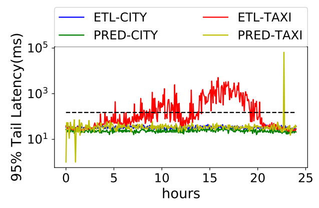


(d) Total CPU limits for each app.

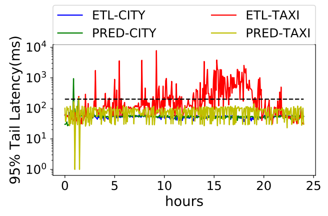
Fig. 11: Using $\mathcal{BO}$ to find near-optimal resource configurations under dynamic workloads.



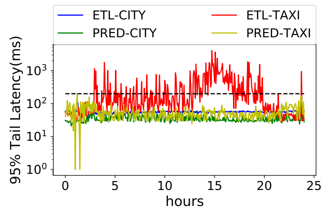
(a) CPU-400.



(b) CPU-4000.



(c) HC-200.



(d) HC-50.

Fig. 12: Tail latency under dynamic workload with static resource allocation and hill-climbing methods.

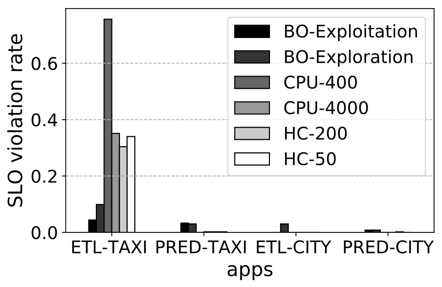
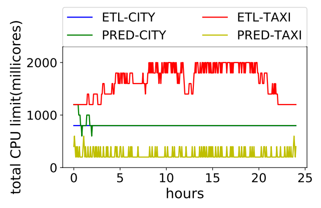


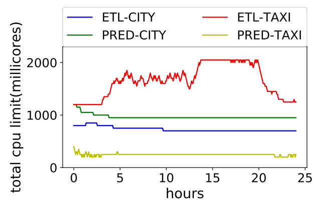
Fig. 13: SLO violation rate.

The SLO target latency is set to 200 ms for each application. For performance comparison, we use the hill climbing methods HC-50, HC-200 and static resource allocation methods that set the CPU limit of each worker to 400 millicores (CPU-400) and 4000 millicores (CPU-4000) respectively.

Fig. 11 (b) and (d) show that $\mathcal{BO}$ is able to adapt the resource configuration of ETL-TAXI and PRED-TAXI in response to dynamic workload variations. Although there are some SLO violations during the exploration phase, the exploitation phase provides very strong performance guarantee. In the cases of ETL-CITY and PRED-CITY shown in Fig. 11 (c), there are minimal SLO violations in both exploration and exploitation phases. This is because it is easier to quickly find near-optimal configurations under static workloads. Note that some of the SLO violations could not be avoided due to limited resources available on the Edge node. Such cases can be potentially addressed by using multiple Edge nodes and increasing the number of workers when needed. This will be explored in our future work. Due to space limitation, Fig. 11 (d) only shows the total CPU limits allocated to each application instead of showing the breakdown of CPU limits for its individual workers.



(a) Hill climbing with 200 step.



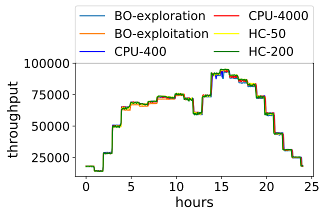
(b) Hill climbing with 50 step.

Fig. 14: Total CPU limit for each app with hill climbing.

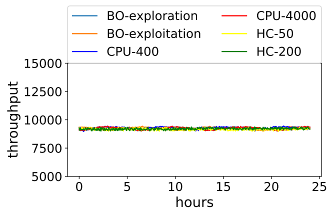
Fig. 12 shows the impact of hill climbing and static resource allocation methods on the end-to-end tail latency of the four applications. The SLO violation rates for all methods are compared in Fig. 13. Table II shows the improvement in SLO violation rate due to $\mathcal{BO}$ as compared with HC-200, HC-50, CPU-4000 and CPU-400 respectively. $\mathcal{BO}$ decreases the SLO violation rate by 62.5% to 75% when compared to the hill climbing method, and by 64.7% to 88.6% when compared to static resource allocation method. For CPU-400, ETL-taxi shows severe SLO violations since it is under-provisioned in the face of increasing workload. In the case of CPU-4000, each application has unrestricted access to use the total CPU resource available in the Edge server when possible. As a result, the competition for CPU resources and interference between the applications when facing high workloads cause several SLO violations. When compared to $\mathcal{BO}$, the hill climbing methods (HC-200 and HC-50) underperform due to the lack of a global model for the objective function behavior. Fig. 14 (a) and (b) show the total CPU limits allocated to the four applications by HC-200 and HC-50 respectively.

TABLE II: Improvement in SLO violation rate.

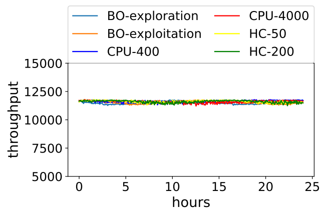
vs.	HC-200	HC-50	CPU-4000	CPU-400
BO-exploration	63.3%	67%	64.7%	84.3%
BO-exploitation	62.5%	75%	73.4%	88.6%



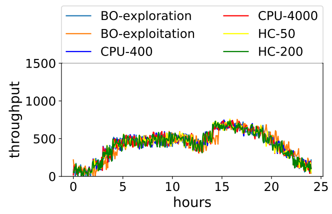
(a) ETL-TAXI.



(b) ETL-CITY.



(c) PRED-CITY.



(d) PRED-TAXI.

Fig. 15: Throughput achieved with different methods under dynamic workloads for ETL-TAXI and PRED-TAXI, and static workloads for ETL-CITY and PRED-CITY.

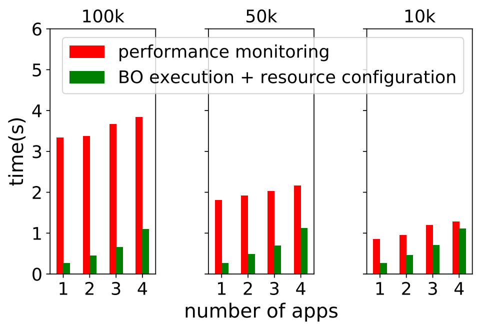


Fig. 16: VRebalance Computational Overhead at various workload intensity(50k, 100k, 200k tuples per minute).

Figure 15 shows that each method including $\mathcal{BO}$, HC-200, HC-50, CPU-4000 and CPU-400 achieve similar throughput for the four applications.

D. *Computational Overhead Analysis*

We analyze the computational overhead of our VRebalance system, which runs as a daemon process on the Edge node and interacts with containerized SPE workers as illustrated by Fig. 5 and Algorithm 1. Fig. 16 shows the running time for performance monitoring, $\mathcal{BO}$ execution and resource reconfiguration at each sampling interval. These overheads increase with the increase in workload intensity and the number of applications running on the Edge node. The running time for performance monitoring is dominated by data collection time and the time taken to calculate the end-to-end tail latency for each application. Nevertheless, the overall computational overhead of VRebalance remains much smaller than its sampling interval even at a high workload intensity. In terms of resource usage, VRebalance brings an additional 4% CPU overhead and its memory usage is less than 200Mb.

V. Related Work

A. *Resource Management in Edge Computing*

Xu et al. [30] proposed an auction-based mechanism for resource contract establishment, and a latency-aware scheduling technique that maximizes the utility for both Edge computing infrastructures and the service providers. In a recent work, Araldo et al. [31] implemented a polynomial-time resource allocation algorithm that allows the Edge network operator to maximize its utility, which can be inter-domain traffic savings, improved QoE (Quality of Experience) for users, or other metrics of interest. Wang et al. [32] explored client workload reduction and server resource allocation to manage application quality of service in the face of contention for cloudlet resources. Unlike these works that focus on resource allocation in a set of small-scale micro-datacenters (cloudlets), this paper addresses the challenges of meeting the performance SLO targets for real-time stream processing in a resource-constrained Edge node (e.g IoT Gateway), which are one-hop away from IoT devices.

B. *Auto-scaling Techniques for Stream Processing*

These works focused on auto-scaling distributed stream processing systems by monitoring performance model of streaming dataflows [13], [33]. Kalavri et al. [14] present an automatic scaling controller that relies on a general performance model of streaming dataflows and lightweight instrumentation to estimate the true processing and output rates of individual dataflow operators. Unlike these works that mainly focus on removing bottlenecks in a single stream processing topology to maximize the average performance, we develop a $\mathcal{BO}$-based resource management system that meets the end-to-end tail latency targets of concurrent stream processing topologies through agile and fine-grained system-level resource control.

C. *Task-scheduling for Stream Processing*

R-Storm [34] handles the problem of task assignment in Apache Storm by providing custom resource-aware scheduling schemes. Shieh et at. [35] propose a topology-based scaling mechanism for Apache Storm. Khare et al. [36] present an algorithm that first transforms any arbitrary stream processing DAG into an approximate set of linear chains to predict the placement of operators. These studies are complementary to our work where we focus on the resource management of stream processing workloads while being agnostic to the underlying task scheduling policies.

VI. Conclusions and Future Work

In this paper, we present VRebalance, a virtual resource orchestrator that provides an end-to-end performance guarantee for concurrent stream processing workloads in a resource-constrained Edge computing environment. Our main contributions are in the design and implementation of an agile and efficient resource management system based on $\mathcal{BO}$.

VRebalance is able to coordinate resource configuration of concurrent workloads and meet their performance SLO targets even for changing system dynamics. Experimental results with RIoT benchmark and Apache Storm deployed on a Edge node, demonstrate the superior performance, resource efficiency and adaptiveness of our $\mathcal{BO}$-based resource management system. In the future, we will extend this work to include co-operation between multiple Edge nodes and explore various stream processing engines apart from Apache Storm.

Acknowledgments

The research is supported by NSF CNS 1911012 grant and NSF CNS 2124908 grant.

References

[1] S. Nastic, T. Rausch, O. Scekic, S. Dustdar, M. Gusev, B. Koteska, M. Kostoska, B. Jakimovski, S. Ristov, and R. Prodan, "A serverless real-time data analytics platform for edge computing," *IEEE Internet Computing*, vol. 21, pp. 64–71, 01 2017.

[2] M. Satyanarayanan, "The emergence of edge computing," *Computer*, vol. 50, no. 1, pp. 30–39, 2017.

[3] C. Qin, H. Eichelberger, and K. Schmid, "Enactment of adaptation in data stream processing with latency implications—a systematic literature review," *Information and Software Technology*, vol. 111, pp. 1 – 21, 2019.

[4] S. Chakravarthy and Q. Jiang, *Stream Data Processing: A Quality of Service Perspective Modeling, Scheduling, Load Shedding, and Complex Event Processing*, 1st ed. Springer Publishing Company, Incorporated, 2009.

[5] M. Kleppmann, *Designing Data-Intensive Applications*. Beijing: O'Reilly, 2017.

[6] Y. Simmhan, V. Prasanna, S. Aman, A. Kumbhare, R. Liu, S. Stevens, and Q. Zhao, "Cloud-based software platform for big data analytics in smart grids," *Computing in Science and Engg.*, vol. 15, no. 4, p. 38–47, Jul. 2013.

[7] C.-C. Hung, G. Ananthanarayanan, P. Bodík, L. Golubchik, M. Yu, V. Bahl, and M. Philipose, "Videoedge: Processing camera streams using hierarchical clusters," in *ACM/IEEE Symposium on Edge Computing (SEC)*, October 2018.

[8] A. Abhishek, C. Adeniyi-Jones, E. Van Hensbergen, and M. Balmakhtar, "Accounting and resource scheduling at the edge." HotEdge'20, July 2020.

[9] H. C. M. Andrade, B. Gedik, and D. S. Turaga, *Fundamentals of Stream Processing: Application Design, Systems, and Analytics*, 1st ed. USA: Cambridge University Press, 2014.

[10] S. Di, D. Kondo, and W. Cirne, "Characterization and comparison of cloud versus grid workloads," in *2012 IEEE International Conference on Cluster Computing*, 2012, pp. 230–238.

[11] U. Tadakamalla and D. A. Menascé, "Characterization of iot workloads," in *Edge Computing – EDGE 2019*, T. Zhang, J. Wei, and L.-J. Zhang, Eds. Cham: Springer International Publishing, 2019, pp. 1–15.

[12] F. Metzger, T. Hoßfeld, A. Bauer, S. Kounev, and P. E. Heegaard, "Modeling of aggregated iot traffic and its application to an iot cloud," *Proceedings of the IEEE*, vol. 107, no. 4, pp. 679–694, 2019.

[13] A. Floratou, A. Agrawal, B. Graham, S. Rao, and K. Ramasamy, "Dhalion: Self-regulating stream processing in heron," *Proc. VLDB Endow.*, vol. 10, no. 12, p. 1825–1836, Aug. 2017.

[14] V. Kalavri, J. Liagouris, M. Hoffmann, D. Dimitrova, M. Forshaw, and T. Roscoe, "Three steps is all you need: Fast, accurate, automatic scaling decisions for distributed streaming dataflows," in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'18. USA: USENIX Association, 2018, p. 783–798.

[15] J. Mockus, *Bayesian approach to global optimization: theory and applications*. Springer Science & Business Media, 2012.

[16] A. Shukla, S. Chaturvedi, and Y. Simmhan, "Riotbench: An iot benchmark for distributed stream processing systems," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 21, p. e4257, 2017, e4257 cpe.4257.

[17] X. Fu, T. Ghaffar, J. C. Davis, and D. Lee, "Edgewise: A better stream processing engine for the edge," in *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '19. USA: USENIX Association, 2019, p. 929–945.

[18] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik, "Monitoring streams: A new class of data management applications," in *Proceedings of the 28th International Conference on Very Large Data Bases*, ser. VLDB '02. VLDB Endowment, 2002, p. 215–226.

[19] *Data Canvas. Sense your city: Data art challenge.* [Online]. Available: http://datacanvas.org/sense-your-city/

[20] M. Stonebraker, U. Çetintemel, and S. Zdonik, "The 8 requirements of real-time stream processing," *SIGMOD Rec.*, vol. 34, no. 4, p. 42–47, Dec. 2005.

[21] C. E. Rasmussen and C. K. I. Williams, *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2005.

[22] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang, "Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics," in *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'17. USA: USENIX Association, 2017, p. 469–482.

[23] T. Patel and D. Tiwari, "Clite: Efficient and qos-aware co-location of multiple latency-critical jobs for warehouse scale computers," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 193–206.

[24] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," in *Advances in Neural Information Processing Systems*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds., vol. 25, 2012.

[25] D. Jones, M. Schonlau, and W. Welch, "Efficient global optimization of expensive black-box functions," *Journal of Global Optimization*, vol. 13, pp. 455–492, 12 1998.

[26] B. Donovan and D. Work, *New York City Taxi Trip Data (2010-2013)*, 2016. [Online]. Available: https://doi.org/10.13012/J8PN93H8

[27] S. Khare, H. Sun, K. Zhang, J. Gascon-Samson, A. Gokhale, and X. Koutsoukos, "Poster abstract: Ensuring low-latency and scalable data dissemination for smart-city applications," 04 2018, pp. 283–284.

[28] A. C. Cameron and P. K. Trivedi, *Regression Analysis of Count Data*, 2nd ed., ser. Econometric Society Monographs. Cambridge University Press, 2013.

[29] A. Brampton. (2018) Measuring percentile latency. [Online]. Available: https://blog.bramp.net/post/2018/01/16/measuring-percentile-latency/

[30] J. Xu, B. Palanisamy, H. Ludwig, and Q. Wang, "Zenith: Utility-aware resource allocation for edge computing," in *2017 IEEE International Conference on Edge Computing (EDGE)*, 2017, pp. 47–54.

[31] A. Araldo, A. D. Stefano, and A. D. Stefano, "Resource allocation for edge computing with multiple tenant configurations," in *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, ser. SAC '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1190–1199.

[32] J. Wang, Z. Feng, S. George, R. Iyengar, P. Pillai, and M. Satyanarayanan, "Towards scalable edge-native applications," in *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, ser. SEC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 152–165.

[33] J. S. Van Der Veen, B. Van Der Waaij, E. Lazovik, W. Wijbrandi, and R. J. Meijer, "Dynamically scaling apache storm for the analysis of streaming data," in *2015 IEEE First International Conference on Big Data Computing Service and Applications*, 2015, pp. 154–161.

[34] B. Peng, M. Hosseini, Z. Hong, R. Farivar, and R. Campbell, "R-storm: Resource-aware scheduling in storm," in *Proceedings of the 16th Annual Middleware Conference*, ser. Middleware '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 149–161.

[35] C.-K. Shieh, S.-W. Huang, L.-D. Sun, M.-F. Tsai, and N. Chilamkurti, "A topology-based scaling mechanism for a pache s torm," *International Journal of Network Management*, vol. 27, no. 3, p. e1933, 2017.

[36] S. Khare, H. Sun, J. Gascon-Samson, K. Zhang, A. Gokhale, Y. Barve, A. Bhattacharjee, and X. Koutsoukos, "Linearize, predict and place: Minimizing the makespan for edge-based stream processing of directed acyclic graphs," in *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, ser. SEC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1–14.