Bayesian-Driven Automated Scaling in Stream Computing With Multiple QoS Targets

Liang Zhang, *Student Member, IEEE*, Wenli Zheng, *Member, IEEE*, Kuangyu Zheng, *Member, IEEE*, Hongzi Zhu, *Senior Member, IEEE*, Chao Li, *Senior Member, IEEE*, and Minyi Guo, *Fellow, IEEE*

***Abstract*—Stream processing systems commonly work with auto-scaling to ensure resource efficiency and quality of service (QoS). Existing auto-scaling solutions lack accuracy in resource allocation because they rely on static QoS-resource models that fail to account for high workload variability and use indirect metrics with much distractive information. Moreover, different types of QoS metrics present different characteristics and thus need individual auto-scaling methods. In this paper, we propose a versatile auto-scaling solution for operator-level parallelism configuration, called AuTraScale+, to meet the throughput, processing-time latency, and event-time latency targets. AuTraScale+ follows the Bayesian optimization framework to make scaling decisions. First, it uses the Gaussian process model to eliminate the negative influence of uncertain factors on the performance model accuracy. Second, it leverages the expected improvement-based (EI-based) acquisition function to search and recommend the optimal configuration quickly. Besides, to make a more accurate scaling decision when the new model is not ready, AuTraScale+ proposes a transfer learning algorithm to estimate the benefits of all configurations at a new rate based on existing models and then recommend the optimal one. We implement and evaluate AuTraScale+ on the Flink platform. The experimental results on three representative workloads demonstrate that compared with the state-of-the-art methods, AuTraScale+ can reduce 66.6% and 36.7% resource consumption, respectively, in the scale-down and scale-up scenarios while achieving their throughput and processing-time latency targets. Compared with other methods of optimizing event-time latency, AuTraScale+ saves 26.9% of resources on average.**

***Index Terms*—Auto-scaling, Bayesian optimization, streaming system.**

I. Introduction

STREAM processing systems (SPSs), such as Flink [1], Spark [2], and Storm [3], have been widely used in anomaly detection, ad-hoc analysis, real-time index building, and many other scenarios. The data from these scenarios is continuously generated at a fast and time-varying rate, then cached by some messaging system like Kafka, and finally analyzed by a job program in stream processing systems (see Fig. 1). The job program can be modeled into a directed acyclic graph (DAG) composed of several operators (like *map* and *filter*). Each operator can be executed by multiple instances, whose quantity is called the operator's parallelism. When the input data rate changes, the system needs to adjust these operator-level parallelism parameters in a timely manner to guarantee the quality of service (QoS) and minimize resource usage. Traditionally, those parameters are set by manual tuning, which takes too much time and easily results in sub-optimal configurations. Therefore, automated on-demand scaling technologies have become necessary to find optimized configurations avoiding both resource waste and QoS violation.

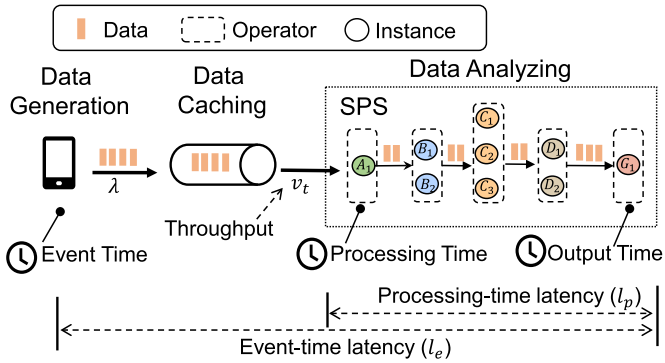


Fig. 1. An example of the stream processing scenario. The workflow of a streaming job. A, B, C, D, E represent five operators in the logical graph of this job. X_i ($X \in \{A, B, C, D, E\}$) denotes the ith instance of operator X.

A practical auto-scaling solution in stream processing systems should fulfill three crucial criteria as follows. *1) Versatility.* The solution should accurately model complex and unknown relationships between parallelism parameters and diversified QoS metrics (i.e., throughput, processing-time latency, and event-time latency mentioned in Fig. 1). *2) Efficiency.* Considering frequent changes in data rates and expensive reconfiguration overheads, the optimization algorithms used by the solution should experience as few iterations and time as possible to find the optimal parallelism configuration to meet diversified QoS requirements. *3) Robustness.* The solution should be robust to data rate fluctuations, i.e., to make scaling decisions rapidly and precisely even when a new performance model is not trained well in the initial phase of the rate changes.

Manuscript received 8 February 2023; revised 2 April 2024; accepted 8 May 2024. Date of publication 13 May 2024; date of current version 24 May 2024. This work was supported in part by the National Key Research and Development Program of China under Grant 2022YFB4501400, and in part by the National Natural Science Foundation of China under Grant 61972247, Grant 62172270, and Grant U2333201. Recommended for acceptance by A. Randles. *(Corresponding authors: Wenli Zheng; Kuangyu Zheng; Hongzi Zhu.)*

Liang Zhang, Wenli Zheng, Hongzi Zhu, Chao Li, and Minyi Guo are with the Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai 200240, China (e-mail: zhangliang@sjtu.edu.cn; zheng-wl@cs.sjtu.edu.cn; hongzi@cs.sjtu.edu.cn; lichao@cs.sjtu.edu.cn; guo-my@cs.sjtu.edu.cn).

Kuangyu Zheng is with the School of Electronic and Information Engineering, Beihang University, Beijing 100191, China (e-mail: zhengky@buaa.edu.cn).

Digital Object Identifier 10.1109/TPDS.2024.3399834

TABLE I
COMPARISON OF EXISTING AUTO-SCALING SOLUTIONS IN STREAM PROCESSING SYSTEMS

Auto-scaling solution	Versatility			Efficiency		Robustness
	Throughput	Processing-time latency	Event-time latency	Accuracy	Overhead	
Threshold-based [4], [5], [6], [10], [11], [12], [13], [26], [27]	√	√	×	Low	Low	Medium
Queuing theory model [16], [17], [18], [19], [20], [21], [22]	×	√	×	Medium	Medium	Low
Rule-based [9] [14] [7], [8]	√	√	×	Medium	Low	Low
Dataflow-based [15] [25] [28]	√	×	×	High	Medium	High
Reinforcement learning [23], [24]	√	√	×	High	High	Low
AuTraScale+ (ours)	√	√	√	**High**	**Low**	**High**

The bold values highlight the strengths of our method AuTraScale+.

In the literature, some auto-scaling solutions in stream processing systems make resource scaling decisions based on QoS thresholds or specific rules [4], [5], [6], [7], [8] depending on simple yet indirect metrics, such as backpressure (or congestion) [9], [10], queue size [9], [11] and observed rate [10], [12], [13], [14], [15]. These metrics can not indicate the true task processing rate (i.e., the throughput of operator instances), which may lead to more configuration iterations before requirements are met and even resource over- or under-provisioning. Some studies rely on queueing theory models [16], [17], [18], [19], [20], [21], [22] to model the relationship between the average total sojourn time (i.e., processing latency) of input data in stream processing system and parallelism parameters, then make reconfiguration decision for minimizing latency based on these models' estimation. However, their estimation accuracies heavily rely on strong assumptions about the logical structure of jobs and data rate distribution. Once these assumptions can not be satisfied, their decision-making process will converge slowly and even end up with poor scaling schemes. Reinforcement learning solutions [23], [24] for resource scaling in the cloud are too expensive to be suitable for dynamic stream processing systems. The popular dataflow-based auto-scaling solutions like DS2 [25] efficiently optimize throughput but do not guarantee latency requirements. In summary, none of the existing solutions can fulfill the three crucial criteria–versatility, efficiency, and robustness–for resource auto-scaling in stream processing systems, as outlined in Table I.

In this paper, we propose a versatile auto-scaling solution, AuTraScale+, tailored for the streaming job with varying data rates to guarantee its throughput, processing-time latency, and event-time latency requirements. Driven by the Bayesian optimization framework, AuTraScale+ accurately models the intricate relationships between QoS metrics and parallelism configurations, efficiently identifying optimal configurations that guarantee quality of service while conserving resources. Additionally, considering the interplay among three QoS metrics, AuTraScale+ follows a specific optimization order. Initially, it determines the minimum parallelisms for all operators to align throughput with the input data rate, ensuring a steady state where accumulated data ceases to increase. Subsequently, AuTraScale+ constructs a comprehensive benefit model across various parallelism configurations, seeking an optimal parallelism configuration to fulfill processing-time latency requirements while minimizing resource consumption. Lastly, within the steady-state system and with known processing-time latency, AuTraScale+ aims to identify optimal parallelisms capable of processing redundant accumulated data within a specified timeframe, thereby maintaining event-time latency below its threshold.

We tackle three main challenges when designing AuTraScale+. *First*, building an accurate performance model for throughput and latency is difficult due to the diverse system environment and job structure. As discussed earlier, existing throughput metrics and queueing model-based latency models have their limitations. In AuTraScale+, a new throughput metric for the operator instance, named the true processing rate [25], is used to guide parallelism configuration. Moreover, latency performance models are built based on the Gaussian process, which can avoid defining model structures in advance and filter out the interference of underlying uncertainties. *Second*, vast operator numbers and parallelism knobs challenge the algorithm to rapidly find the optimal configuration that meets the QoS target and minimizes resource usage. To tackle it, AuTraScale+ designs a comprehensive benefit scoring function for evaluating all configurations and selects the optimal one by comparing their maximum expected increment (EI). Meanwhile, we adopt the Bayesian optimization (BO) framework to integrate GP-based model training and EI-based heuristic searching policy to improve scaling decisions iteratively. *Third*, it is unacceptable to train performance models slowly when the input data rate rapidly changes, which may cause inaccurate or outdated scaling decisions and additional reconfiguration costs. Considering the potential relationship between performance models corresponding to different rates, AuTraScale+ proposes a transfer learning algorithm to fully use trained models at the old rate to estimate new samples at the new rate and explore the optimal parallelism configuration quickly based on these estimations.

We implement and evaluate AuTraScale+ based on the Flink framework, and the experimental results on three representative workloads demonstrate its efficacy. Specifically, it can achieve the throughput target within four iterations at most. On the premise of ensuring the processing-time latency requirement, it reduces 66.6% and 36.7% resource usage, respectively, in the scale-down and scale-up scenarios compared with the state-of-the-art methods. When event-time latency targets are met on Yahoo Streaming jobs, AuTraScale+ saves an average of 26.9% parallelism resources compared with the other three

scaling methods. Compared with our earlier version's traditional transfer learning method, the more lightweight one achieves 1.17-1.9x speedup and saves 20-84% resource on average at optimal parallelism configuration.

The main contributions in this work are as follows:

- We abstract the relationship between the parallelism and QoS metric in streaming systems to a Gaussian process model, which is trained by the data containing the interference due to resource contention, so the prediction results can be more accurate.
- We propose an automated scaling solution based on Bayesian optimization to update Gaussian process models and recommend better parallelisms iteratively for meeting different QoS targets. The acquisition function is used to improve the search efficiency of optimal solutions and reduce reconfiguration costs.
- We propose a lightweight transfer learning algorithm to make full use of the trained model to find the optimal configuration quickly at a new data rate, significantly reducing model retraining costs.

The rest of the paper is organized as follows. We elaborate system model and problem definitions in Section II. Then, we analyze key challenges and the potential of Bayesian optimization in Section III. The design details of AuTraScale+ with performance models and optimization algorithms are presented in Section IV. Experimental results are presented and analyzed in Section V. We discuss related work in Section VI and conclude the paper in Section VII.

II. System Model and Problem Definition

This section describes our system model, including node roles, architecture components, and some system assumptions. Then, we define a general resource scaling problem and decompose it into three optimization subproblems based on different QoS requirements.

A. System Model

We consider a cluster consisting of multiple physical nodes to run a stream processing job and achieve resource auto-scaling, as shown in Fig. 2. The logic of a stream processing job includes several operators, and each can be executed by multiple instances like Fig. 1.

1) Node Roles. Master: It is a manager responsible for task scheduling, resource management, and scaling decisions. The Job Manager component of the stream processing system provides the first two functions. Among them, Task Scheduler splits the job into several tasks and allocates them into different execution nodes (called Workers). Resource Manager manages the status of all nodes in this cluster to schedule idle nodes or release used nodes according to Task Scheduler's requirements. The master node also incorporates components associated with auto-scaling, such as the Metric Aggregator, Scaling Manager, and Policy Controller. These components are responsible for gathering QoS metrics, determining whether scaling should be initiated, and making configuration decisions accordingly.

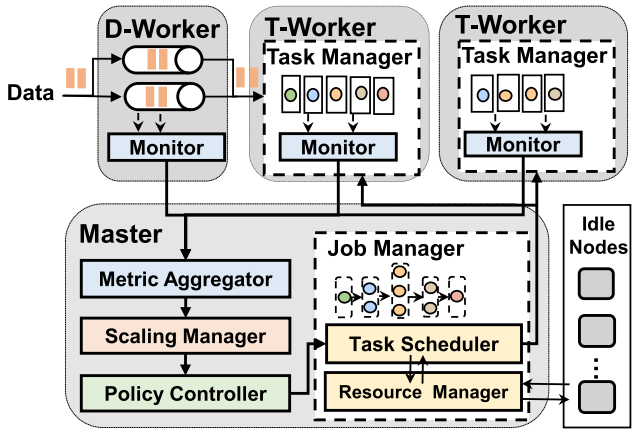


Fig. 2. The system model of AuTraScale+.

Task Worker (T-Worker): It maintains a Task Manager component to receive and execute one or more tasks allocated by Job Manager, then feedback task status and metric information to the Master node. The resource on the Worker Node is divided into multiple units (called slots in Flink), and each unit is responsible for executing a task. Here a task represents a single operator's instance or a set of instances from different operators, which depends on the Task Scheduler's decision as much as the mapping between tasks and nodes. The scheduling decision affects the quality of service of the whole job, but it is beyond the scope of this paper. We discuss the relationship between resource scaling and QoS metrics under the same scheduling decision (using Task Scheduler's default setting).

Data Worker (D-Worker): It runs the messaging system (like Kafka or Redis) to cache data from users and then sends them into Worker nodes following the first-come-first-served rule. Data Worker is used to relieving the pressure on the Task Workers when the input data rate is much higher than the processing rate of the system such that tasks can run smoothly and correctly. However, it also brings a waiting delay to the input data and affects the event-time latency users focus on. Therefore, we also set a metric monitor on it and feedback its status to the Master Node.

2) Assumptions: We make the following assumptions about the above system architecture.

QoS threshold: Users can choose single or multiple optimization problems at the same time and set their corresponding thresholds. When a user wants to optimize both types of latency, we specify that the event-time latency is greater than or equal to the processing-time latency.

Balanced load: The streaming systems encapsulate resources like CPU and memory in the form of slots, which are fixed subsets each worker's resources are divided into. Data arriving at an operator is assigned to its instances, each in a different slot by specific rules. We assume each instance of the same operator has the same amount of data, like [17], [25].

No bandwidth limitation: We assume that the network does not bottleneck the system. In practice, if the network traffic to a server is too high, meeting the latency requirement for high responsiveness is usually impossible.

TABLE II
THE DEFINITION OF THREE QOS METRICS

Name	Definition
Throughput (v_t)	The total data processing rate of a job, i.e., the number of tuples processed by the job per second.
Processing-time latency (l_p)	The difference between the time that a tuple reaches the input operator and that it reaches the output operator.
Event-time latency (l_e)	The difference between the time a tuple is generated and the time it reaches the output operator.

Adequate resources: We assume that the cluster resources are enough to support the QoS requirements of the workload. It means there is no such situation where all resources are exhausted, but the QoS requirement is still unable to meet. For QoS optimization with limited resources, some task scheduling methods can be considered, such as [29] and [30].

Control period: AuTraScale+ targets for workloads whose variation period of input data rate is larger than its convergence time like other controllers [25]. For those workloads that change on a shorter period, other solutions like backpressure mechanism [1], [31], data buffering or load shedding [32], [33], [34] may outperform auto-scaling techniques.

B. Problem Definition

We focus on the resource scaling problem on the above system model when the input data rate changes, which targets minimizing resource usage and QoS violation. Here, a streaming job can be modeled as a directed acyclic graph $G = (V, E)$, where the set of vertices V denote operators and the set of edges E denote data dependencies between different operators. This graph will be transformed into a physical execution plan which maps operators to compute resources according to parallelism configuration and scheduling strategy (as shown in Fig. 2). Under a specific scheduling strategy, we discuss the following resource scaling problem.

Resource Scaling Problem: Given a streaming job (a static and known DAG) with operators $o_1, o_2, \ldots, o_N$ and input data rate λ, to find out the minimum parallelism π_i for each operator o_i such that some QoS metrics satisfy the user-specific requirements.

In this paper, we mainly focus on three critical QoS metrics: throughput, processing-time latency, and event-time latency. Their definitions are shown in Table II. Such that, we decompose the above resource scaling problem into three optimization subproblems to meet their requirements.

- *Throughput optimization (TO):* This problem targets to find the minimum parallelism parameter π_i^t for each operator o_i (denote as $\boldsymbol{\pi}^t$) that allows the throughput to catch up with input data rate (λ).
- *Processing-time latency optimization (PLO):* This problem targets to find minimum parallelism parameters π_i^p for each operator o_i (denote as $\boldsymbol{\pi}^p$) such that the average processing-time latency (l_p) is less than the threshold L_p provided by the user.

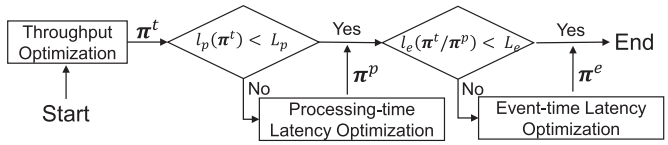


Fig. 3. The logical relation of three optimization subproblems.

- *Event-time latency optimization (ELO):* According to the definition, event-time latency is the sum of processing-time latency and waiting latency ($l_e = l_p + l_w$). Given a fixed l_p, the objective of this problem is to find minimum parallelism parameters π_i^e for each operator o_i (denote as $\boldsymbol{\pi}^e$) so that the system can consume a proper amount of accumulated data within a given time limit to decrease l_w until $l_e \leq L_e$ (L_e is the threshold).

Discussion: The solution to **TO** ($\boldsymbol{\pi}^t$) gives a lower bound on the parallelism configuration to ensure the system's processing rate can catch up with the input data rate. $\boldsymbol{\pi}^t$ can guarantee the amount of accumulated data in input/output network buffers of each operator instance on Task Worker and caching queues on Data Worker is no longer continuously increasing; hence, two latency metrics is stable. Therefore, throughput optimization is a prerequisite for discussing two latency optimization problems, and $\boldsymbol{\pi}^t$ is the lower bound of the solution configuration for P2 and P3. The waiting time in input/output network buffers on each instance affects data's processing-time latency, so we need to take this uncertain factor into account to solve **PLO** and obtain the parallelism configuration $\boldsymbol{\pi}^p$ when $l_p > L_p$. After solving **PLO**, we obtain a definite l_p and **ELO** starts to be solved. The logical relation of three optimization problems is shown in Fig. 3.

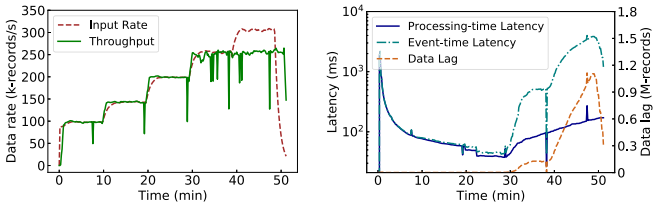
AuTraScale+ follows the above logic to achieve resource scaling in stream processing systems while meeting different QoS requirements. It tends to propose more accurate performance models against interference and more efficient search strategies for every optimization problem than existing solutions (see Section IV for details).

III. CASE STUDIES AND OPPORTUNITIES

In this section, we identify the challenges of solving the above resource scaling problem in streaming systems via case studies, and introduce the opportunities the Bayesian optimization algorithm brings to address these challenges.

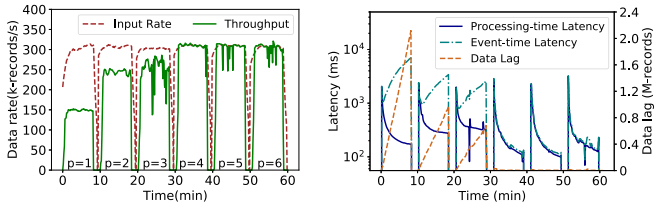
A. Case Studies and Challenges

Following the system model in Section II-A, we built a testbed in which the Kafka message system was deployed on the D-worker to cache data, and the Flink streaming system was deployed on multiple T-workers to process data. We run a simple WordCount Streaming job containing four operators (Source, FlatMap, Count, and Sink) on this testbed and monitor all key metrics (throughput, processing-time latency, event-time latency, data lag on the D-Worker) to help us obtain valuable observations.



(a) Throughput (b) Latency and data lag

Fig. 4. *The running results of the WordCount Streaming job with the fixed parallelism and increasing data rate:* Parallelism under-provisioning leads to non-trivial data lag and end-to-end latency.



(a) Throughput (b) Latency and data lag

Fig. 5. *The running results of the WordCount Streaming job with the fixed data rate and increasing parallelism. p* represents the operator parallelism. 1) There is a nonlinear relationship between the parallelism and QoS metrics. 2) The appropriate parallelism brings QoS benefit, but a higher parallelism may not gain lower latency or higher throughput.

CASE 1: Fixed parallelism and increased data rate to reveal the consequences of suboptimal configurations.

In this case, the parallelism of each operator is set to 2 and remains the same. The input data rate starts from 100k records/s and increases every 10 minutes by 50k records/s. Fig. 4(a) shows the changes in both the input data rate and throughput, and Fig. 4(b) shows the changes in the two types of latencies and the data lag during the experiment.

Observation 1: When the input data rate increases and exceeds the upper bound of the throughput with the fixed parallelism configuration, data are accumulated in Kafka, and both types of latency will continue to increase. Ultimately, this under-provisioning status results in serious QoS violations.

As shown in Fig. 4(a), when the input data rate is 100k records/s, 150k records/s, or 200k records/s, the throughput of the job can meet the input data rate and there is no data accumulation in the Kafka. In Fig. 4(b), two types of latency temporarily peak when the job starts and then gradually flatten out. When the input data rate reaches 250k records/s, the job throughput begins to lag behind for a short time. The data begins to accumulate in Kafka, and two types of latency begin to increase. Then, the input data rate increases to 300k records/s, which is significantly greater than the fixed parallelism configuration's processing capacity, while the job's throughput still maintains at 250k records/s. The growth rate of the accumulated data in Kafka and two types of latency in Flink increase until the test ends at the 50th minute.

This case shows that it is necessary to adjust the parallelism configuration and resource allocation when the data rate changes. The suboptimal configurations will result in under- or over-provisioning. Lack of resources will result in data processing lag and increased latencies, while redundant resources can be wasted if the data rate decreases. However, dynamically allocating resources for jobs is challenging, as shown by another set of our experiments as follows.

CASE 2: Fixed data rate and increased parallelism to identify the challenges of resource auto-scaling.

We execute six independent small tests on the above testbed, in which each test's input data rate maintains at about 300k records/s, and the operator parallelism is (1, 2, 3, 4, 5, 6) respectively. The results are shown in Fig. 5.

Observation 2.1: The relationship between operator parallelism and QoS metrics is not linear.

In Fig. 5(a), in the first three sets of tests, the parallelism is 1, 2, and 3, but the corresponding throughput is about 150k records/s, 250k records/s, and 275k records/s respectively. The multiplying growth of parallelism does not provide the proportional increase in throughput. Similarly, the relationship between processing-time latency and operator parallelism is also not linear, as shown in Fig. 5(b). The reasons for this phenomenon can be synchronization and resource competition between different operator instances.

Observation 2.2: The event-time latency highly correlates with data lag and processing-time latency.

In the first three tests of Fig. 5(b) (when the throughput cannot catch up with the data input rate), the event-time latency rises proportionally with the growth of accumulated data in Kafka. In the last three tests of Fig. 5(b) (when the throughput is equivalent to the data input rate), no data is accumulated, and the event-time latency is likely equal to the processing-time latency. A similar observation also appears in Fig. 4(b), which motivates us to optimize event-time latency by consuming accumulated data after processing-time latency optimization.

Observation 2.3: The appropriate parallelism brings throughput and latency benefits, and higher parallelism may not be better.

Comparing the latency between the first three tests and the last three tests in Fig. 5(b), we find that improving the parallelism is helpful in reducing two types of latency on the whole. However, the latter two tests (The latencies are 100 ms and 125 ms, respectively) also indicate that increased parallelism may increase communication cost [35] and thus increase the processing-time latency. In Fig. 5(a), the throughput is throttled by the input data rate in the last three tests, so redundant parallelism is wasted.

Key Challenges: Based on the above case studies, We state that the relationship between operators' parallelism (resource allocation) and different QoS metrics is greatly complicated by synchronization, resource contention, communication, and system scheduling policies. The main challenge to identifying the appropriate parallelism configuration when the workload changes is dealing with this unknown relationship.

B. Opportunities

To deal with the above challenges, we need a method that can capture the unknown relationship between resource allocation

and QoS metrics (e.g., throughput and latency), and recommend the most appropriate parallelism configuration for the application. Bayesian optimization (hereinafter referred to as "BO") is what we need. First, it uses the surrogate models, such as Gaussian process regression and random forests, to approximate the real relationship, and does not require the specific mathematical formula about the objective function and the input variables. Second, the goal of BO is to find the input variables that optimize the objective function with as few rounds of executions as possible. This is also consistent with the goal of minimizing reconfiguration overheads caused by scaling decisions in streaming systems. Therefore, this black-box method to solve optimizing objective functions is very suitable for auto-scaling scenario in the streaming system.

Mathematically, Bayesian optimization can be expressed by (1):

$$x^* = \arg \max_{x \in A \subset \mathbb{K}^d} f(x) \tag{1}$$

AuTraScale+ takes the parallelism of each operator as the input variable x, and takes a scoring function that quantifies the comprehensive benefits of service quality and resource usage with the given parallelism as the objective function f. The goal of AuTraScale+ is to find the most profitable parallelism configuration scheme to maximize the scoring function in the minimum number of iterations. In this paper, Bayesian optimization also involves the selection of the surrogate model, acquisition function and the initial training data, as elaborated in the next section.

IV. Design of AuTraScale+

We elaborate on the details of AuTraScale+ in this section. The frequently used notations throughout the paper are summarized in Table III.

A. Overview

Recall three scaling-related components on the master node depicted in Fig. 2, namely the Metric Aggregator, Scaling Manager, and Policy Controller. AuTraScale+ relies on these three components to implement resource auto-scaling workflow. More importantly, AuTraScale+ integrates three Bayesian-driven modules (TO Solver, PLO Solver, and ELO Solver) into the Policy Controller for solving three QoS optimization problems mentioned in Section II-B.

1) Auto-Scaling Workflow: AuTraScale+ achieves a control cycle of monitoring, analyzing, planning, and executing (MAPE) [36]. When the job is running, this control cycle runs periodically to guide the system to scale in or scale out resources appropriately for ensuring QoS requirements [37], [38], [39].

Specifically, metric monitors on different workers periodically gather throughput or latency information and report them into Metric Aggregator on the Master node (*Monitor*). Metric Aggregator selects and integrates the metric information monitored, such as calculating the total processing rate of all instances of each operator, and sends them to the Scaling Manager. The Scaling Manager judges whether the resource configuration needs to be adjusted and issues a scaling signal to Policy Controller (*Analyze*). Then Policy Controller runs scaling algorithms to make a new configuration decision and notifies Task Scheduler to execute (*Plan*). Task Scheduler stops and stores the current status information of the job, then restarts it using the new configuration. (*Execute*). This control cycle is driven by scaling algorithms in Section IV without user participation.

2) Solver Modules: Three solver modules (TO Solver, PLO Solver, and ELO Solver) in the Policy Controller are designed to make scaling decisions for ensuring throughput, processing-time latency, and event-time latency requirements. Their design details are summarized in Fig. 6. When the Scaling Manager sends a scaling signal, AuTraScale+ first calls TO Solver to find minimum parallelisms for all operators to make the throughput catch up with the input data rate, ensuring accumulated data on the D-Worker is no longer increasing. Then, the PLO Solver is

TABLE III
Symbols Used in This Paper

Symbol	Meaning
$o_1, o_2, ..., o_N$	Operators in a streaming job (N is the total)
$\boldsymbol{\pi}^t, \boldsymbol{\pi}^p, \boldsymbol{\pi}^e$	The solution configs of TO,PLO,ELO
L_p, L_e	Thresholds of processing/event-time latency
λ	The input rate of a job
v_t	The system throughput at the current config
λ_i, v_i	The total input, processing rate of operator i
v_{ij}	The rate of the jth instance of operator i
$\bar{v}_i$	The average processing rate of operator i
$\boldsymbol{p}' = (p'_1, ..., p'_N)$	The output of throughput optimization
$\boldsymbol{p} = (p_1, ..., p_N)$	The current parallelisms of all operators
$\mathcal{M}$	Gaussian process model
$P_{\max}$	Maximal allowable parallelism of system
$S = \{(\boldsymbol{p}_j, s_j)\}_{j=1}^{M+N}$	An bootstrapping sample set
$S_p = \{\boldsymbol{p}_j\}_{j=1}^{M+N}$	An bootstrapping configuration set
w	The over-allocation ratio
s_l	The benefit score threshold of a job
T_d	The stabilization time after reconfiguration
l_p, l_e, l_w	The processing/event/waiting time latency
$l_p^{(s)}, l_e^{(s)}, l_w^{(s)}$	Three types of latency at the steady state
$R_{lag}^{(c)}, R_{lag}^{(s)}$	The cached data at current/steady state
$R_{lag}^{(t)}$	The consumed data to meet L_e
V_t	The target throughput to meet L_e
T_{limit}	The limited time to meet L_e
$D_c = \left\{\left(\boldsymbol{p}_j^c, s_j^c\right)\right\}_{j=1}^{T}$	An available sample set at $Rate_c$
$P_c = \left\{\boldsymbol{p}_j^c\right\}_{j=1}^{M+N}$	The bootstrapping parallelism sample set
Num	The maximum number of iterations

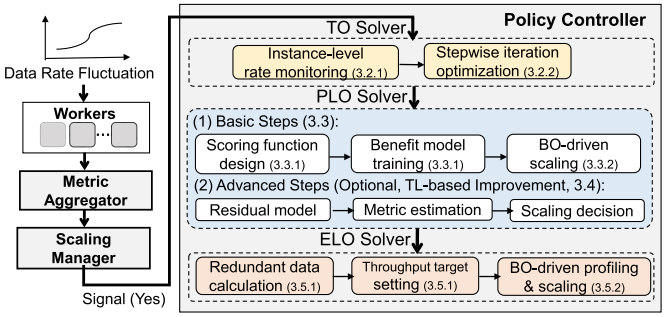


Fig. 6. The design overview of AuTraScale+.

called to optimize processing-time latency. Finally, in the steady-state system with known processing-time latency, ELO Solver targets to find optimal parallelisms that can consume redundant accumulated data within a specific time limit so that event-time latency is lower than its threshold.

Specifically, TO Solver relies on an instance-level rate model and an iteration solution with forward stepwise along data flow. First, the instance-level rate model is used to measure the actual processing rate of each operator instance, eliminating the impact of data pending time on the rate calculation, and more truly reflects the instance's processing ability. Second, TO Solver calculates the number of instances required for each operator to reach the target throughput, starting from the data source node of the DAG and following the data flow direction. This process is called stepwise iteration optimization in Section IV-B.

PLO Solver follows Bayesian Optimization (BO) framework to train a benefit model and recommend the better parallelism configuration using an expected improvement-based acquisition function iteratively until the processing-time latency is lower than the threshold (Basic steps). Specifically, the Gaussian process-based benefit model can cover the impact of interference and model the nonlinear relationship between latency and parallelism more accurately; the acquisition function can speed up the search process of the solution and minimize the number of iterations to find the optimal solution. Besides, the PLO solver can trigger transfer learning-based advanced steps to make more accurate scaling decisions when the benefit model is not fully trained in the initial phase of a new data rate.

ELO Solver temporarily increases parallelisms to consume redundant accumulated data on the Data Worker until the event-time latency is lower than its threshold after optimizing the processing-time latency. The key is to find the appropriate parallelism configuration ($\boldsymbol{\pi}^e$) to consume a specified amount of data within a given time limit, i.e., reach a specific throughput target while minimizing resource usage. To this end, in Section IV-E, ELO Solver first uses a data backlog model to determine this new throughput target and then calls BO-driven offline profiling to find the optimal solution.

B. TO Solver

Throughput optimization solver (TO Solver) finds the minimum parallelism configuration that allows the throughput to catch up with input data rate based on the instance-level rate model and a stepwise iteration solution.

1) Instance-Level Rate Model: In the stream processing system, the actual processing rate of an operator is usually defined as the ratio of the number of processed records over the processing time. However, the observed data processing time from the system's metric interface often contains a large amount of waiting time at the input/output buffer of the operator due to blocking [40] and hence cannot be used directly. Here, we use the concept of *Useful Time* in DS2 [25] to define the true processing rate of an operator instance, which is formulated as (2).

$$v_u = \frac{R}{T_u} \tag{2}$$

where R is the total number of records processed by an operator instance in a period of time T, and T_u represents the time used to process data (i.e., *Useful Time*) in T. T_u includes three parts: the deserialization time, processing time, and serialization time. Based on the above definition, we can compute the average processing rate of all instances of operator i is $\bar{v}_i = \frac{1}{p_i}\sum_{j=1}^{p_i} v_{ij}$. p_i is the parallelism of the operator i and v_{ij} is the true processing rate of the jth instance of operator i. We use $\bar{v}_i$ and λ_i (the date input rate of operator i) to identify optimal parallelism configuration.

2) Stepwise Iteration Optimization: We use a stepwise iteration solution along data flow [25] to determine appropriate parallelism for each operator and quickly make the throughput as close to the input data rate as possible. This method relies on the true processing rate measurements among all operators and their instances.

Suppose that operator $i-1$ and operator i are two connected operators in the DAG. The operator $i-1$ only contains one instance, and its total true processing rate v_{i-1} is equal to its external input data rate v_e and its total output rate o_{i-1} at time t. The operator i is the successor of the operator $i-1$, and its total input data rate is λ_i. There is a simple equivalence, $\lambda_i = o_{i-1} = v_{i-1}$. To make the total true processing rate of the operator i catch up with the input data rate, the number of instances of operator i can be set to $\left\lceil \frac{v_{i-1}}{\bar{v}_i} \right\rceil$ ($\bar{v}_i$ is the average true processing rate of all the instances of operator i).

However, the parallelism of an operator is greater than one, and the total true processing rate may not meet the external input data rate in practice. For this general case, AuTraScale+ uses (3) to calculate the optimal parallelism of each operator in each iteration step:

$$p_i' = \begin{cases} \left\lceil \frac{\lambda}{\bar{v}_1} \right\rceil & i = 1 \\ \left\lceil \frac{v_{i-1} \times \frac{p_{i-1}'}{p_{i-1}}}{\bar{v}_i} \right\rceil & i > 1 \end{cases} \tag{3}$$

where p_i' and p_{i-1}' are the optimal parallelisms of operator i and operator $i-1$, respectively, at the current iteration step.

The advantage of this approach is to quickly and accurately find the minimum resource configuration that enables the throughput to meet the requirements. Nevertheless, the throughput is often constrained by other factors like the key-value store in third-party systems and can not meet the input data rate in practice (see the results of the Yahoo Streaming job in Section V-B for details). The DS2 method does not address this issue and thus can fall into an infinite loop due to that the throughput does not reach the target value. Therefore, except the throughput is greater than or equal to the input data rate, AuTraScale+ adds a new termination condition to deal with this situation, i.e., two consecutive identical configurations occur. We denote the best parallelism configuration found at the end of the iteration as $\boldsymbol{\pi}^t$, which is the lower bound of solution configurations to the processing-time and event-time latency optimization.

C. PLO Solver

Processing-time latency optimization solver (PLO Solver) finds optimal parallelisms to meet the processing-time latency requirement and minimize resource usage using the Gaussian process model and Bayesian optimization framework.

1) GP-Based Benefit Model: The processing-time latency of a streaming job is affected by many uncertain factors like operators' input/output buffer status and performance interference between them besides the parallelism of operators. Traditional mathematical models like the queueing model can not model these uncertain factors. Their strong assumptions about input data rate distribution and jobs' logic structure are also not satisfied in a practical stream computing scenario. Therefore, we consider obtaining a posteriori distribution of the latency under a given parallelism configuration instead of a deterministic latency value for covering underlying uncertain factors. The Gaussian process (GP) model fits our above needs very well and is proper to model the relationship of a comprehensive benefit on latency and resource usage and the parallelism configuration.

Scoring function construction: The simple GP-based latency model cannot capture the resource usage status, which cannot help us find the parallelism that meets the latency requirement while minimizing resource usage. So AuTraScale+ designs a scoring function to quantify the comprehensive benefits of processing-time latency and resource usage. We regard this scoring function as the objective function of the Gaussian process model. Specifically, the scoring function needs to satisfy two basic rules: (a) the lower the latency, the higher the score; (b) the closer the parallelism is to the basic configuration (the parallelism for maximizing throughput), the higher the score. So AuTraScale+ defines the scoring function as (4):

$$F = \alpha \times \min\left(1.0, \frac{L_p}{l_p}\right) + (1-\alpha) \times \frac{1}{N} \times \sum_{i=1}^{N} \frac{\pi_i^t}{p_i} \quad (4)$$

where π_i^t represents the minimum parallelism of operator i that can achieve throughput target, i.e., the solution configuration of throughput optimization. p_i represents the current parallelism of operator i. l_p is the average processing-time latency of data with the current configuration. L_p is the target processing-time latency. The first half of the formula is used to judge whether the current latency meets the requirements, and the second half is used to prevent the over-provisioning of parallelism. α is an adjustable parameter indicating the relative importance of the two targets.

Benefit model building: AuTraScale+ models the relationship between the comprehensive benefit and the parallelism configuration based on the above scoring function and the Gaussian process model. We assume this benefit model as $\mathcal{M}$, which is subject to a Gaussian process $y \sim GP(\mu(x), k(x, x'))$, where x denotes the parallelism variable and y denotes the benefit score variable. We adopt the Matern covariance kernel. Given a available sample set $D = \{(x_j, y_j)\}_{j=1}^{t}$ ($D_y = \{y_1, \ldots, y_t\}$ and $D_x = \{x_1, .., x_t\}$), the posterior over y is still a GP distribution, denoted as $p(y \mid x, D) = \mathcal{N}(y \mid \mu_t(x), \sigma_t^2(x))$. $\mu_t(x)$ and $\sigma_t^2(x)$ are given by (5):

$$\begin{aligned} \mu_t(x) &= k_t(x)^T \left(K_t + \sigma^2 I\right)^{-1} D_y \\ \sigma_t^2(x) &= k(x,x) - k_t(x)^T \left(K_t + \sigma^2 I\right)^{-1} k_t(x) \end{aligned} \quad (5)$$

where $k_t(x) = [k(x_1, x), k(x_2, x), \ldots, k(x_t, x)]^T$ and $K_t = [k(x, x')]_{x,x' \in D_x}$.

Algorithm 1: BO-Driven Auto-Scaling for PLO.

Input: $\boldsymbol{\pi}^t$, L_p, w, α, S
Output: $\boldsymbol{\pi}^p$
1: Initialize parallelism: $\boldsymbol{p} = (p_1, \ldots, p_N)$.
2: Model pre-training using samples in S.
3: Get the BO's search space Ω.
4: $\boldsymbol{x} = (x_1, \ldots, x_N) \in \Omega, x_i \in [\pi_i', P_{max}]$
5: Get the score threshold: $s_l \leftarrow \alpha + (1-\alpha)/(1+w)$
6: **while** $true$ **do**
7: $l_p \leftarrow Run\_Job(\boldsymbol{p}, T_d)$
8: $score \leftarrow Score\_Function(\boldsymbol{p}, \boldsymbol{\pi}^t, l_p, L_p)$
9: Add $(\boldsymbol{k}, score)$ to the existing set
10: Update the surrogate model $\mathcal{M}$
11: **if** $l_p <= L_p$ **and** $score >= s_l$ **then**
12: $\boldsymbol{\pi}^p \leftarrow \boldsymbol{p}$
13: **break**
14: **else**
15: $\boldsymbol{p}_{new} \leftarrow \arg\max_{\boldsymbol{x} \in \Omega \subset \mathbb{R}^N} EI(\boldsymbol{x}, \mathcal{M})$
16: $\boldsymbol{p} \leftarrow \boldsymbol{p}_{new}$
17: **end if**
18: **end while**

Bootstrapping samples selection: To improve the fitting accuracy of the benefit model $\mathcal{M}$, we need to select the bootstrapping samples to train the initial model carefully. There are two types of samples in the initial training set of AuTraScale+. *(a) All operators in a sample have the same parallelism, and different samples have different parallelisms.* First, the parallelism of all operators is set to $\pi_{\max}^t$, which is the maximum parallelism of throughput optimization configuration $\boldsymbol{\pi}^t$. Then we divide the remaining parallelism (the difference between the current parallelism $\pi_{\max}^t$ and the maximum allowable parallelism $P_{\max}$ of the system) into $M-1$ parts, each of which is called an *interval*. The parallelism of all operators in the i-th sample is set to $\pi_{\max}^t + i \times interval$. Those M samples can help the model perceive the benefit results of different configurations and also help us to determine whether the current resources can meet the processing-time latency requirements. *(b) The parallelism of only one operator is set to $P_{\max}$, and the parallelism of other operators is kept in the basic configuration.* There are N such samples (where N is the number of operators in a DAG) that can make the model capture the different impact of each operator on latency as far as possible and have a more precise prediction.

2) BO-Driven Online Scaling: AuTraScale+ adopts the Bayesian optimization (BO) framework to make resource scaling decisions to meet the processing-time latency requirement and minimize resource usage (The details are shown in Algorithm 1). It uses a scoring function to evaluate the performance

benefits of different configurations and a surrogate model to fit the relationship between the parallelism of operators and the scoring function (as mentioned in Section IV-C1). When the current resource is over-provisioned or the QoS violation occurs, the acquisition function of AuTraScale+ will recommend new parallelism samples for the next job run. Then, the model is updated using the current metric information. If the termination condition is not met, AuTraScale+ will iteratively perform the above process (corresponding to the scaling workflow in Section IV-A1). We will introduce the essential parts involved in the above process as follows.

Acquisition function: The acquisition function aims to find the next sample closer to the optimal solution. It also balances the proportion of exploration and exploitation during the sampling period. In the stream computing scenario, a suitable acquisition function satisfies the following two conditions: (a) Try to find the global optimal value; (b) The evaluation cost should not be too high. AuTraScale+ wants to minimize the expected deviation from the true maximum to find the global optimal value. However, its expense is very high when we consider multiple steps ahead [41]. So AuTraScale+ chooses an alternative, which is to maximize the expected improvement with respect to the best value known. Besides, to adjust the proportion of global search and local optimization, AuTraScale+ introduces parameter ξ in the expectation of the improvement function [42]. Mathematically, the acquisition function is defined as (6):

$$EI(x) = \begin{cases} K\Phi(Z)+\sigma(x)\phi(Z) & \text{If } \sigma(x)>0 \\ 0 & \text{If } \sigma(x)=0 \end{cases} \tag{6}$$

$$K = \mu(x) - f\left(x^{+}\right) - \xi \tag{6a}$$

$$Z = \begin{cases} \frac{K}{\sigma(x)} & \text{If } \sigma(x) > 0 \\ 0 & \text{If } \sigma(x) = 0 \end{cases} \tag{6b}$$

where $\mu(x)$ and $\sigma(x)$ is the GP mean and standard deviation at the sample x. $\Phi(Z)$ and $\phi(Z)$ is the standard normal CDF and PDF of Z respectively. $f(x^{+})$ is the best value known.

Termination condition: The termination condition of AuTraScale+'s BO algorithm is that the latency requirement is met and the resource score is greater than the threshold. AuTraScale+ calculates the resource score threshold using the over-allocation ratio w of the resource specified by the user. w is defined as (7):

$$\frac{C_{now} - C_{opt}}{C_{opt}} < w \Longleftrightarrow \quad \frac{C_{opt}}{C_{now}} > \frac{1}{1+w} \tag{7}$$

where C_{now} is the current parallelism and C_{opt} is the optimal parallelism. $\frac{C_{opt}}{C_{now}}$ is the resource allocation ratio. For a job with N operators, AuTraScale+ defines the resource allocation ratio as the mean of the corresponding values of all operators, which can be expressed as $\frac{C_{opt}}{C_{now}} = \frac{1}{N} \times \sum_{i=1}^{N} \frac{k'_i}{k_i}$. Therefore, the termination condition of Bayesian optimization is that the latency target is met and the benefit score surpasses the predefined threshold as indicated by (8):

$$F \geqslant \alpha + (1-\alpha) \times \frac{1}{1+w} \tag{8}$$

D. Transfer Learning-Based Improvement for PLO

The performance model mentioned in the above basic steps of PLO Solver is binded to the specific input data rate λ, which leads to a lot of time and resources spent in training new models from scratch when the input data rate changes. At that time, it is difficult for AuTraScale+ to make accurate scaling decisions in time due to the model is not fully trained. To tackle this issue, AuTraScale+ refers to the idea of transfer learning [43], [44] and fully uses existing trained models on old data rates and limited samples on the new rate to recommend better parallelism configurations in the initial phase of the new data rate. The details are presented in Algorithm 2.

When the input data rate changes, AuTraScale+ first optimizes throughput to obtain the solution $\boldsymbol{\pi}_t$ and then calls the transfer learning method. Suppose that there are $c-1$ trained models $\{\mathcal{M}_i\}_{i=1}^{c-1}$, where the corresponding rate $Rate_{c-1}$ of the model $\mathcal{M}_{c-1}$ is the closest to the new rate $Rate_c$. $D_c = \{(\boldsymbol{p}_t, s_t)\}_{t=1}^{T}$ is an available sample set at the current rate. First, we use model $\mathcal{M}_{c-1}$ to calculate the score estimate $\mu_{c-1}(\boldsymbol{p}_t)$ of parallelism vector $\boldsymbol{p}_t$ in the set D_c, denoted as $\mathcal{M}_{c-1}(\boldsymbol{p}_t)$. Then we use the Gaussian Process Regression to train a residual model $\mathcal{M}'_c$ with the training sample set $D'_c = \{(\boldsymbol{p}_t, s_t - \mu_{c-1}(\boldsymbol{p}_t))\}_{t=1}^{T}$. Next, we use the residual model $\mathcal{M}'_c$ and the model $\mathcal{M}_{c-1}$ to calculate the Gaussian process mean and variance of point $\boldsymbol{x}$ in the configuration set $\Omega - P_c$ respectively. The results $\mu'_c, \sigma'_c(\boldsymbol{x})$ and $\mu_{c-1}, \sigma_{c-1}(\boldsymbol{x})$ are used to determine the objective function estimation of the current point $\boldsymbol{x}$. Finally, the acquisition function recommends an optimal configuration based on these estimations. The above procedure is iterated until the latency target or the maximum number of iterations Num is reached. The Num is a signal to switch AuTraScale+ from Algorithm 2 to Algorithm 1 to avoid accuracy loss caused by the above estimation after the new model is ready.

In the above process, AuTraScale+ directly predicts the mean and variance of all the parallelisms in the sample space and uses these values as the inputs of the acquisition function to recommend a new parallelism configuration. Compared with the traditional transfer learning method in our earlier version [45], AuTraScale+ trims the estimation process of the bootstrapping samples and avoids additional model pre-training costs. This lightweight method reduces the algorithm's running time and further speeds up the scaling decision-making (see comparison results in Section V-D).

E. ELO Solver

The event-time latency is affected by the amount of accumulated data except the parallelism configuration. The amount of accumulated data may vary from time to time, leading to different event-time latency even under the same parallelism. Therefore, it is impractical to model the event-time latency like the processing time field. AuTraScale+ targets to reduce the waiting latency of data on the premise of the processing-time latency is stable and known, which means to consume accumulated data within a limited time by increasing parallelism configuration for reaching the event-time latency target. Specifically, AuTraScale+ calculates the amount of accumulated data

Algorithm 2: The TL-Based Improvement for PLO.

Input: Ω, $\{\mathcal{M}_i\}_{i=1}^{c-1}$, D_c, P_c, Num
Output: $\boldsymbol{\pi}_p$ or $\boldsymbol{\pi}_e$
while $true$ **do**
  function TRAINRESIDUALMODEL($\mathcal{M}_{c-1}, D_c$)
    $(\boldsymbol{k}_t, s_t) \in D_c$
    $\mu_{c-1}(\boldsymbol{k}_t) \leftarrow \mathcal{M}_{c-1}.predict(\boldsymbol{k}_t)$
    $D'_c \leftarrow \{(\boldsymbol{k}_t, s_t - \mu_{c-1}(\boldsymbol{k}_t))\}_{t=1}^{T}$
    $\mathcal{M}'_c \leftarrow Gaussian\_Process\_Regress\ (D'_c)$
    return $\mathcal{M}'_c$
  end Function
  Initial the estimated sample set $D_e \leftarrow D_c$
  for each sample $\boldsymbol{x}$ in $(\Omega - P_c)$ **do**
    $\mu_{c-1}(\boldsymbol{x}), \sigma_{c-1}(\boldsymbol{x}) \leftarrow \mathcal{M}_{c-1}.predict(\boldsymbol{x})$
    $\mu'_c(\boldsymbol{x}), \sigma'_c(\boldsymbol{x}) \leftarrow \mathcal{M}'_c.predict(\boldsymbol{x})$
    $\mu_c(\boldsymbol{x}) = \mu_{c-1}(\boldsymbol{x}) + \mu'_c(\boldsymbol{x})$
    $\beta \leftarrow \alpha|D_c|/(\alpha|D_c| + |D_{c-1}|)$
    $\sigma_c(\boldsymbol{x}) = \sigma'_c(\boldsymbol{x})^{\beta}\sigma_{c-1}(\boldsymbol{x})^{1-\beta}$
    $D_e.\mathrm{add}(\boldsymbol{x}, \mu_c(\boldsymbol{x}), \sigma_c(\boldsymbol{x}))$
  end for
  $\boldsymbol{p}_{new} \leftarrow \arg\max_{\boldsymbol{x}\in\Omega\subset\mathbb{R}^N} EI(D_e)$
  Obtain $score$ at the point $\boldsymbol{p}_{new}$
  $D_c.\mathrm{add}(\boldsymbol{p}_{new}, score)$
  if The terminational condition is met **then**
    break
  end if
  $num++$
  if $num >= Num$ **then**
    Switch to $Algorithm\ 1$
  end if
end while

to be consumed (denote as $R_{lag}^{(t)}$) and the target throughput V_t that can consume data $R_{lag}^{(t)}$ within a limited time T_{limit} based on the data backlog model. Then, it performs throughput profiling offline and adopts the Bayesian optimization framework again to find the optimal parallelism configuration for meeting the target throughput V_t. The overall workflow is shown in Algorithm 3.

1) Data Backlog Model: We analyze the relationship between the amount of accumulated data and event-time latency by the following data backlog model and deduce the target throughput V_t that can consume data $R_{lag}^{(t)}$ within T_{limit} based on it. According to definitions in Section II-B, the event-time latency is equal to the sum of the processing-time latency and the waiting-time latency. In a steady state, i.e., the system throughput is equal to the input data rate, and the amount of accumulated data keep steady so that the waiting-time latency is also stable. Meanwhile, the processing-time and event-time latency are also stable. At this time, the three types of latencies and the amount of accumulated data satisfy the relationships shown in (9) and (10), respectively.

$$l_w^{(s)} = l_e^{(s)} - l_p^{(s)} \tag{9}$$

$$R_{lag}^{(s)} = \lambda \times l_w^{(s)} \tag{10}$$

where $l_e^{(s)}, l_p^{(s)}, l_w^{(s)}$ respectively represent the event-time latency, processing-time latency, and waiting-time latency. λ is the input data rate equal to the system throughput v at the steady state. $R_{lag}^{(s)}$ is the amount of accumulated data waiting in the Data Worker corresponding to the target event-time latency in a steady state. Equation (10) follows Little's Law [46], [47] that states the long-term average number of data in a system is equal to the long-term average arrival rate of data multiplied by the average time that the data spends in the system.

Next, AuTraScale+ computes the difference between $R_{lag}^{(c)}$ and $R_{lag}^{(s)}$ as the amount of data that currently needs to be consumed for meeting event-time latency using (11). $R_{lag}^{(c)}$ is the current amount of accumulated data in the Data Worker.

$$R_{lag}^{(t)} = R_{lag}^{(c)} - R_{lag}^{(s)} \tag{11}$$

Finally, the target throughput V_t that can consume the amount of data $R_{lag}^{(t)}$ with the limited time T_{limit} can be defined by (12).

$$V_t = \lambda + \frac{R_{lag}^{(t)}}{T_{limit}} \tag{12}$$

2) BO-Driven Offline Profiling: After determining the target throughput V_t, AuTraScale+ needs to find the appropriate parallelism to achieve it. Unlike the throughput optimization in Section IV-B, V_t is much larger than the input data rate λ. We can not measure the rate information online and compute the parallelism configuration by the stepwise iteration method. Therefore, AuTraScale+ conducts an offline profile to obtain the throughput information under different parallelism. However, traversing all parallelism configurations is expensive due to huge search space $P_{\max}^N$ where N is the number of operators and $P_{\max}$ is the maximum parallelism allowed by the system. In order to improve the efficiency of the offline profile, AuTraScale+ considers the Bayesian optimization method again to reduce the number of iterations and resource consumption. The key parts of the BO framework are similar to Section IV-C except for the scoring function. Here, the scoring function is only related to throughput, as shown in (13):

$$F_{tp} = \frac{\beta}{|v_t - V_t| + \beta} \tag{13}$$

where v_t represents the system throughput corresponding to the current parallelism. β is a constant used to control the magnitude of the scoring function variation and can be preferably set as the same order of magnitude with $|v_t - V_t|$. The range of the scoring function is (0,1). The closer it is to 1, the closer the current throughput is to the target, and the better the current parallelism configuration is. The absolute value ensures that neither too high nor too low throughput can obtain high scores, so the trained model can be used to find the optimal configuration that meets throughput requirement (and therefore the event-time latency requirement) while preventing resource over-provisioning.

Algorithm 3: BO-Driven Auto-Scaling for ELO.

Input: $\boldsymbol{\pi}^p, T_d, L_e, T_{limit}, S_p, \beta, \Omega, s_l$
Output: $\boldsymbol{\pi}^e$
$l_e, l_p^{(s)}, R_{lag}^{(c)} \leftarrow Run\_Job(\boldsymbol{\pi}^p, T_d)$
if $l_e <= L_e$ **then**
  $\boldsymbol{\pi}^e \leftarrow \boldsymbol{\pi}^p$
  exit
else
  $V_t \leftarrow \lambda + (R_{lag}^{(c)} - \lambda \times (L_e - l_p^{(s)}))/T_{limit}$
  $S \leftarrow Throughput\_Profile(S_p, V_t, \beta)$
  $\mathcal{M} \leftarrow Pretraining\_Model(S)$
  while *true* **do**
    $\boldsymbol{p}' \leftarrow \arg\max_{\boldsymbol{x} \in \Omega \subset \mathbb{R}^N} EI(\boldsymbol{x}, \mathcal{M})$
    $v_t \leftarrow Throughput\_Profile(\boldsymbol{p}', T_d)$
    $F_{tp} \leftarrow \frac{\beta}{|v_t - V_t| + \beta}$
    Add $(\boldsymbol{p}', F_{tp})$ to the existing set
    Update the surrogate model $\mathcal{M}$
    if $F_{tp} >= s_l$ **then**
      $\boldsymbol{\pi}^e \leftarrow \boldsymbol{p}'$
      break
    end if
  end while
end if
$Run\_Job(\boldsymbol{\pi}^e, T_{limit})$ ▷ Let $l_e < L_e$
$Restart\_Job(\boldsymbol{\pi}^p)$ ▷ Resume steady state

V. Evaluation

In this section, we elaborate on the experiment environments for evaluation and present the evaluation results and analyses of AuTraScale+ compared with other existing methods.

A. Experiment Setup

Machine configuration: We evaluate AuTraScale+ on a physical cluster composed of 3 Dell PowerEdge R730xd (20-core CPUs and 256 GB RAM) and 1 Dell PowerEdge R740xd (32-core CPUs and 256 GB RAM). AuTraScale+ is implemented in Flink 1.10.0 and Hadoop Yarn 2.7.5. We run Flink applications in the yarn-per-job mode, which launches Flink within YARN only for executing a single job. Hadoop and Flink are deployed on three R730xd machines, one as the Master node and two as T-Worker nodes. To simulate the real production environment, we also deploy Zookeeper and Kafka on the other R740xd machine (i.e., D-Worker node) in a pseudo-distributed mode to store data. The InfluxDB database on the Master node stores Flink and Kafka metric information.

Metric collection: Stream processing systems usually provide users with a metric interface to gather job-running information or expose metrics to external systems. The monitor component of AuTraScale+ periodically calls this interface to gather performance metrics like latency and throughput. These metrics are stored uniformly in a third-party database system, such as Prometheus and InfluxDB. Besides, we modify the source code of Flink system to gather the true processing rate of the operator mentioned in Section IV-B and expose them to the same metric interface avoiding additional overheads.

Baselines: For throughput optimization, we compare AuTraScale+ with the recently proposed method DS2 [25], which can reach the optimal configuration at most three steps. For processing-time latency optimization, we compare the efficiency of AuTraScale+ with the DRS [17] method. DRS targets to guarantee end-to-end processing-time latency and minimize resource usage based on the queuing theory and greedy algorithm during the auto-scaling process. However, its accuracy relies heavily on strong assumptions of queuing theory and is easily affected by uncertain factors like network buffer. Moreover, we also compared the performance of the lightweight transfer learning (TL) method proposed in this paper with the traditional TL method in the earlier version [45]. For event-time latency optimization, we compare AuTraScale+ with the following three straw-man solutions.

1) Max consumes the accumulated data using the maximal allowable parallelism of the system.
2) *Proportion* adjusts the parallelism based on the ratio of current throughput to target throughput.
3) *Exponent* increases the parallelism exponentially until the throughput target is reached.

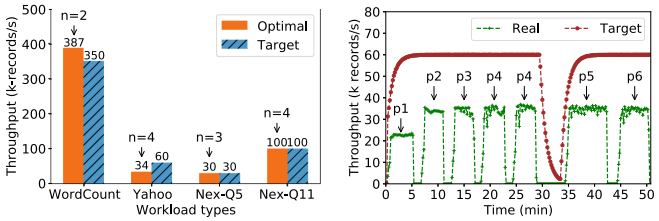
They all aim to find the parallelism configuration to achieve the target throughput and consume the accumulated data to meet the event-time latency requirement. To simplify the time complexity analysis, we assume that the number of operators in the workload is N, and the algorithm iterates M times in configuration adjustment. The time complexity of Max, *Proportion*, and *Exponent* solution is $O(N), O(NM), O(NM)$ respectively. In contrast, Bayesian optimization used by AuTraScale+ has a higher time complexity $O(N^2 M)$ [48] but brings good resource benefits (see Section V-E for details). Moreover, as shown in Table V of Section V-F, the overhead of the BO algorithm is completely acceptable in our experiment.

Benchmarks: We use three representative workloads to verify the performance of AuTraScale+. WordCount Streaming Job has a simple linear DAG structure containing four operators: Source, FlatMap, Count, and Sink. Yahoo Streaming Benchmarks [49] is an advertisement event processing case, and we use an extended version [50] that contains five operators: Source, FlatMap (Parse data), Filter, Project, and FlatMap (Join data). Nexmark [51] is a benchmarking suite of Apache Beam that contains multiple continuous data stream queries. We use Query5 (sliding window) and Query11 (session window) to evaluate the special operators.

B. Throughput Optimization

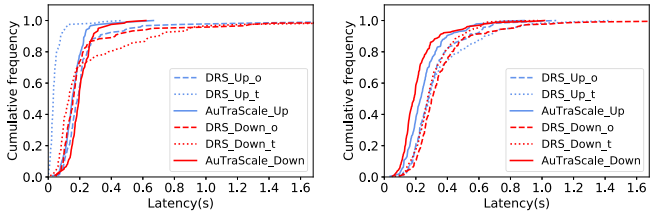
To verify the effect of our throughput optimization method in Section IV-B, we run four workloads: WordCount, Yahoo, Nexmark-Query5, and Nexmark-Query11. The initial parallelisms of all operators for four workloads are set to 1, and the stabilization time after reconfiguration is 5 minutes. The input data rates of the four jobs are set to 350k records/s, 60k records/s, 30k records/s, and 100k records/s, respectively.

After the job starts with initial parallelism, if the throughput is lower than the input data rate and running time is greater than 5



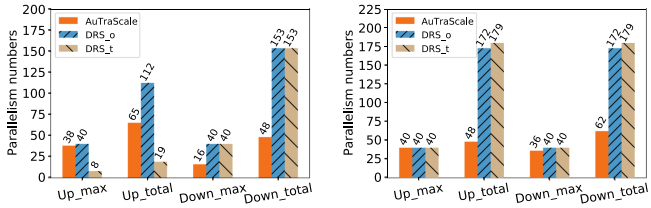
(a) (b)

Fig. 7. (a) Throughput optimization results for different workloads. *n* represents the number of iterations. Because of the data backlog in Kafka, WordCount's throughput is temporarily higher than the targe. The throughput of Yahoo jobs is limited by Redis's read/write rate and cannot reach the target rate. (b) Throughput optimization process for Yahoo Streaming job. In the fourth iteration, the parallelism is p4 (40, 1, 1, 1, 40) again, and the algorithm is terminated. When the parallelism is larger, p5(40, 40, 40, 40, 40) and p6(50, 50, 50, 50, 50), the throughput does not continue to increase.



(a) WordCount Streaming job (b) Yahoo Streaming job

Fig. 8. Processing-time latency comparison of optimal configuration for different methods in elasticity tests. *DRS_Up_o* represents the final optimized configuration found by running the DRS method with the observed rate in the scale-up test. *DRS_Down_t* represents the final optimized configuration found by running the DRS method with the true rate in the scale down test, and so on.



(a) WordCount Streaming job (b) Yahoo Streaming job

Fig. 9. Parallelism comparison of optimal configuration for different methods in elasticity tests. *Up_max* represents the max parallelism of optimal configuration in the scale-up test. *Down_total* represents the sum of all operators' parallelism in optimal configuration in the scale-down test.

minutes, the Scaling Manager will inform the Policy Controller to call the throughput optimization method and return new parallelism. The Task Scheduler receives the new configuration and restarts the job. After running 5 minutes, if the Scaling Manager detects that the current throughput is greater than the input data rate or the current parallelism is the same as the last iteration, the throughput optimization will terminate.

When the throughput optimization algorithm is terminated, the operator parallelism of the four workloads are (3, 4, 12, 10), (40, 1, 1, 1, 40), (1, 18), (1, 11). Fig. 7(a) shows the final throughput of the job and the number of iterations. For WordCount job, the data accumulated in Kafka due to not being processed in time are also processed with the newly arrived data under the current optimized configuration, so its throughput will be higher than the input data rate. Due to the limitation of the read/write rate of Redis, the optimal throughput of the Yahoo streaming job cannot reach the input data rate. In this situation, the parallelism obtained at the end of the iteration is not always optimal. Therefore, AuTraScale+ reviews the iterative process and selects the solution with maximum throughput and less resource utilization as the final result. In Fig. 7(b), the parallelism p2 (4, 2, 1, 1, 34) is selected as the final optimal configuration. The experiment after the 35th minute in Fig. 7(b) is to verify that higher parallelism does not lead to throughput optimization.

From the results, AuTraScale+ can achieve the optimal throughput with four iterations at most. Although the convergence is slower than DS2 (three iterations at most), AuTraScale+ can terminate the iteration when external factors limit the throughput to prevent the resource waste caused by the infinite loop of the algorithm.

C. Processing-Time Latency Optimization

We use WordCount and Yahoo streaming jobs to evaluate the performance of Bayesian optimization-driven scaling method in this section. AuTraScale+ trains the initial model using the bootstrapping samples and iteratively updates the model as the job runs. When a QoS violation occurs or the benefit score is below the threshold, AuTraScale+ starts scaling up or scaling down by adjusting the parallelism configuration. We do the experiments with two jobs: WordCount job (target throughput is 350k records/s and target processing-time latency is 180 ms) and Yahoo job (target throughput is 34k records/s and target processing-time latency is 300 ms). The parallelisms are set differently to test scaling up and down, respectively. The initial training set of two jobs contains 10 and 40 samples, respectively. AuTraScale+ terminates when the processing-time latency, throughput, and benefit scores meet the requirements concurrently. The benefit score is 0.9, and the running time of each parallelism configuration is 10 minutes. The DRS method with either the true or the observed processing rate is used for comparison. It runs until the processing-time latency meets the requirements or the total number of parallelisms in the new configuration exceeds the upper limit of resources.

We can get the following information from the results. First, Table IV shows that AuTraScale+ can find a parallelism configuration that meets QoS requirements in fewer steps as long as the resources are sufficient. The more training samples, the fewer iterations. Second, Fig. 9 shows that the optimal parallelism that meets the QoS requirements of AuTraScale+ consumes fewer resources than the DRS method for most tasks. AuTraScale+ can reduce 66.6% and 36.7% resource consumption respectively in the scale-down and scale-up scenarios. Although the true rate version of DRS can find solutions that use fewer resources than AuTraScale+ in the WordCount scale-up experiment, it can not meet the throughput requirements. Third, Fig. 8 shows that less parallelism may bring better latency benefits. It is worth noting that sometimes DRS does not meet the QoS requirement.

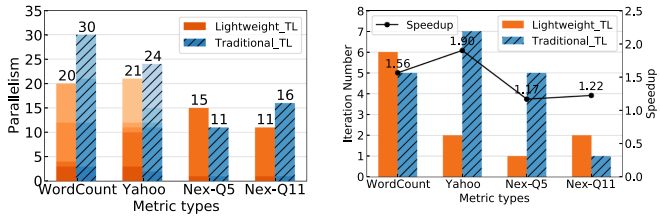
TABLE IV
COMPARISON OF OPTIMAL CONFIGURATION IN THE SCALING TEST FOR WORDCOUNT AND YAHOO STREAMING JOBS

Job	Type	Initial value*	Method	Rate type	Optimal value	Throughput (records/s)	Latency (ms)	Score	Iteration	meet QoS
WC	Up	1,3,12,2	DRS	observed	23,33,40,16	381k	246	–	9	no
				true	2,1,8,8	338k	146	–	2	no
			AuTraScale	true	7,8,12,38	405k	172	0.9096	3	yes
	Down	36,32,24,20	DRS	observed	40,40,40,33	405k	120	–	1	yes
				true	40,33,40,40	406k	242	–	1	no
			AuTraScale	true	10,16,12,10	355k	160	0.9275	4	yes
Yahoo	Up	4,2,1,1,4	DRS	observed	40,33,33,33,33	35.3k	329	–	4	no
				true	40,33,33,33,40	35.4k	242	–	2	yes
			AuTraScale	true	4,2,1,1,40	34.7k	223.7	0.994	1	yes
	Down	40,40,40,40,40	DRS	observed	40,33,33,33,33	35.1k	276	–	2	yes
				true	40,33,33,33,40	34.5k	247	–	2	yes
			AuTraScale	true	22,2,1,1,36	36.2k	246	0.965	1	yes

* The initial value of parallelism configuration here is the best configuration to meet QoS requirements at the last data rate.

TABLE V
THE EXPERIMENT SETUP OF THE TRANSFER LEARNING ALGORITHM COMPARISON

Workloads	Old rate	New rate	Latency
WordCount	150k	200k	250ms
Yahoo	20k	30k	130ms
Nex-Q5	8k	10k	100ms
Nex-Q11	60k	80k	100ms



(a) Parallelism. (b) Iteration number and speedup.

Fig. 10. The comparison of the transfer learning algorithm between traditional one and lightweight one. *Speedup* is the ratio of the average running time of the traditional algorithm over that of the lightweight one.

It shows that the error of the queueing model becomes more significant in complex resource mapping schemes. In contrast, the Gaussian process model used by AuTraScale+ has better performance.

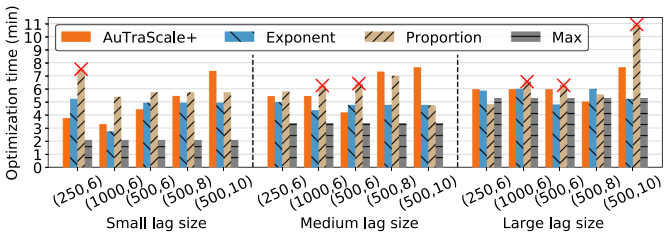
D. Transfer Learning-Based Improvement

We compare the performance of the lightweight transfer learning acceleration with the traditional transfer learning method in our earlier study [45] by performing processing-time latency optimization when the input data rate changes. For each of the four workloads mentioned in Section V-A, we run two transfer learning algorithms independently to explore the optimal parallelism that meets the processing-time latency requirement at a changed input data rate. The experimental setup is summarized in Table V. We first train models at an old input data rate and get a set of actual samples at the new rate as inputs of the transfer learning algorithm. Then we adopt two different transfer learning algorithms to explore the new parallelism configuration to reach the latency target at the new input data rate. We record the throughput, processing-time latency, the number of iterations, the running time of the algorithm, and other related information during the execution of workloads. When the latency requirement is met, the algorithm terminates and outputs the optimal parallelism.

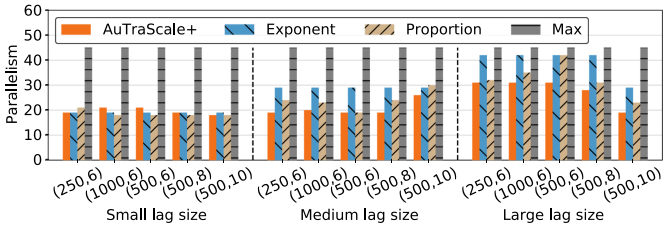
We compare the final results (parallelisms and the numbers of iterations) of the two transfer learning algorithms in Fig. 10. From the results, we find that for WordCount and Nex-Q11 workload, the lightweight algorithm saves about 30% parallelism resource with only one more iteration. For the Yahoo Streaming workload, the lightweight algorithm achieves better parallelism with less than one-third of iterations. For Nex-Q5, the final parallelism of the lightweight algorithm is just a little bit worse than the traditional one, but it saves 80% of iterations. In addition, the execution efficiency of the lightweight algorithm is higher than the traditional one, as shown in Fig. 10(b). Specifically, the lightweight algorithm achieves 1.17x-1.9x speedup on average per iteration and saves 20%-84% CPU resources in all iterations except Nex-Q11. On the whole, the lightweight algorithm outperforms the traditional one.

E. Event-Time Latency Optimization

We use Yahoo Streaming Job to evaluate the event-time latency optimization algorithm. First, AuTraScale+ constructs an initial set of the parallelisms referring to Section IV-C1 and conducts an offline profile to obtain the throughput corresponding to different parallelism. After that, we set the input data rate to 40% of the maximum system throughput, set the stabilization time after reconfiguration to 3 minutes, and start the Yahoo Job with the initial parallelism of every operator set to 1. Initially, the system runs with under-provisioned resources, and AuTraScale+ will sequentially execute throughput and processing-time optimization. Then AuTraScale+ calls Algorithm 3 to stabilize the event-time latency below the target value. The running process for the other three comparison methods (Max, *Proportion* and *Exponent*) are the same in general, and they adjust the parallelisms to optimize the event-time latency according to the strategies in Section V-A respectively. The control variable method is used for analysis under different conditions. We mainly consider three types of variables: the initial size of accumulated data (6/12/16 million records for small/medium/large size), event-time latency



(a) Optimization time.



(b) Parallelism.

Fig. 11. The comparison of different methods for event-time latency optimization. The tuple *(x,y)* on the x-axis represents event-time latency target (*x* ms) and time limit (*y* minutes). The red cross indicates a method did not achieve the event-time latency target in time. Note that there is no positive or negative correlation between the optimization time and the total parallelism of all operators because potential redundant parallelism may not increase throughput or even decrease throughput due to resource contention.

target (250 ms, 500 ms, and 1000 ms), and the time limit (6 min, 8 min, 10 min).

The experimental results are shown in Fig. 11. The optimization time in Fig. 11(a) represents the time consumed by optimizing event-time latency using the optimal parallelism configuration found by the algorithm. The parallelism in Fig. 11(b) represents the total number of all operators' parallelism corresponding to the optimal configuration found by the algorithm. We can draw the following conclusions. First, AuTraScale+ can complete event-time latency optimization within the same time limit as Max and *Exponent*. *Proportion*, which adjusts parallelisms according to throughput ratio and ignores the impact of resource interference, is conservative and results in some timeouts during the optimization as shown in Fig. 11(a). Second, AuTraScale+ uses fewer resource to meet the event-time latency target. Specifically, AuTraScale+ saves 21.9%, 9.3%, and 49.5% parallelism resources on average compared with *Exponent*, *Proportion*, and Max, respectively. From Fig. 11(b), we also find that AuTraScale+ shows a greater advantage when there is more accumulated data, and it is comparable to other methods even in the worst case. *Exponent* is radical and tends to allocate more resources to jobs compared with AuTraScale+ and *Proportion*.

Although the time limit varies from 6 to 10 minutes in the above experiments, it can be extended to other appropriate values according to the system state and user demand. When the time limit is too tight, the parallelism and resource utilization keep at a high level until the event-time latency meets the requirement, which may affect the performance of other jobs in the cluster. Conversely, when the time limit is too loose, the tedious optimization may impair the user experience. The tradeoff between resource utilization and QoS mainly depends on the service provider and is out of the scope of this paper.

TABLE VI
COMPUTATION OVERHEADS IN SECOND AT THE DIFFERENT NUMBER OF OPERATORS

Method	Algorithm	The number of operators				
		2	4	6	8	10
BO	Algorithm 1	0.042	0.053	0.065	0.076	0.088
	Algorithm 3	0.027	0.035	0.043	0.048	0.056
TL	Traditional [45]	0.070	0.081	0.094	0.104	0.116
	Algorithm 2	0.036	0.044	0.050	0.060	0.073

F. Overheads

Our solution directly calls the native *metric group* interface of Flink to expose the true processing rate information to the outside, so it does not bring additional overhead to the system. Users can get the rate information by the path *taskmanager_job_task_trueProcessingRate* like accessing other native metrics in Flink.

To evaluate the running overhead, we run all algorithms with a different number of operators to obtain the consumed CPU time. We repeat the experiment 100 times to eliminate the influence of random factors. The results are listed in Table VI, where *BO* represents the Bayesian optimization method and *TL* represents the Transfer learning method. On the whole, the running overhead of all algorithms increases approximately linearly with the number of operators, attributable to the expanded solution search space. The CPU time consumption is less than 90 ms when the number of operators is less than 10, and can stay within 1 s within reasonable predictions. This magnitude does not significantly impact the quality of service (QoS) of the tested job. Besides, the overhead associated with the lightweight transfer learning method (Algorithm 2) consistently remains lower than that of the traditional method with an equivalent number of operators, thus highlighting the superiority of the lightweight transfer learning method once more.

VI. RELATED WORK

A. Resource Scaling in Traditional Cloud Systems

Cloud systems provide the service for users in the form of virtual machines (VMs) or more lightweight containers. When the workload varies over time, the manager can add/remove physical resources to/from these virtual resource units (Vertical Scaling) or directly add/remove these virtual resource instances (Horizontal Scaling) to avoid the QoS degradation [52]. There are two types of auto-scaling techniques, reactive and proactive, to guide the manager when and how to execute vertical/horizontal scaling.

The reactive techniques trigger resource scaling when the collected metric information does not reach the target. These metrics include CPU utilization [53], response time [54], message queue status [55], and their thresholds can be dynamic [56], [57], or hierarchical [58] for better tuning scaling decisions. The simple method to make scaling decisions is table look-up, but it has low scalability and fails to use in practical [59]. There are

many more effective methods have been proposed, including learning-based [23], [24], [60], [61], control theory [62], fuzzy control [63], [64], and model-based [65], [66].

The proactive techniques predict QoS metrics or workloads near future and make scaling decisions according to these predictions. Queuing theory is often used to build performance models for prediction relying on the system simulation and the observation that request arrivals follow the Poisson distribution. The scaling methods usually adopt single-tiered [67], or multi-tiered [66], [68] queuing models according to their workload characteristics. Petri nets [69], [70] is also used to compute a compromise plan between several adaptation plans. Time series analysis techniques like autoregressive integrated moving average (ARIMA) [71] and diversified machine learning methods [72], [73], [74] are used to estimate future workload fluctuation and help resource managers proactively plan resource allocation for the next time interval.

The above resource scaling solutions in traditional cloud systems mainly focus on throughput optimization based on their workload characteristics and requirements. Applying these solutions to resource scaling problems in streaming systems will lead to severe and inevitable latency violations due to inaccurate performance modeling without latency consideration. Latency violations significantly decrease user satisfaction and cause substantial financial losses, which can be fatal for streaming applications. Therefore, it is essential to design latency-aware resource scaling schemes for streaming systems, as AuTraScale+ does.

B. Resource Scaling in Stream Processing Systems

The workloads in stream processing systems change more frequently and are more sensitive to latency violation than in traditional cloud computing. Therefore, the threshold-based scaling methods [4], [5], [6] focus on not only traditional metrics like CPU utilization [12][26], memory utilization [26] and throughput [10], but also other latency-related metrics like network congestion status [10], [27] and backpressure [10]. Some other methods define new metrics, such as the Effective Throughput Percentage(ETP) in Stela [13].

The model-based methods mainly use queuing theory, such as Kingman's formula and Jackson open queueing networks, to model the end-to-end latency of the system as prior work [16], [17], [21], [22]. These methods predict the latencies obtained by different schemes before the scaling action and find the best one to execute. However, uncertain factors like network buffer status and interference between multiple tasks can easily affect their accuracy.

Besides, some rule-based scaling methods [7], [8], [9], [14] are used in practical stream processing scenarios. Dhalion [9] detects system bottlenecks by analyzing metrics such as the backpressure, then generates a diagnosis and selects a scaling plan according to the self-defined rule. Google Dataflow [14] heuristically adjusts the number of workers in a cloud environment based on several signals, such as CPU utilization, the amount of work remaining, and throughput. Based on the data flow model, DS2 [25] determines the appropriate parallelism for Apache Flink and Timely Dataflow according to the input data rate and the true processing rate of the operator. A topology-based scaling policy [28] for Apache Storm makes up for some drawbacks of rebalance command and improves scaling performance.

Compared to existing work, the superior performance of AuTraScale+ comes from more accurate performance modeling and a more efficient scaling decision process. AuTraScale+ models the relationship between performance and resource allocation as a Gaussian process covering performance interference caused by resource contention. This modeling method is more suitable for complex application environments than those that rely on fixed-structure mathematical formulas like queuing models. Meanwhile, AuTraScale+ capitalizes on the effectiveness of the Bayesian optimization for exploring better configurations with fewer iterations within an expansive parameter space, thereby expediting the decision-making process for resource scaling. Besides, AuTraScale+ is the first effort to optimize event-time latency, significantly improving the user satisfaction of real-time applications.

VII. Conclusion

This paper studies the resource auto-scaling problem for dynamic workloads in the stream processing system. The goal is to find an appropriate parallelism configuration for a streaming job with the new data rate to meet three QoS metric targets (throughput, the processing/event-time latency) while minimizing resource usage. To achieve it, our versatile solution, called AuTraScale+, uses the Gaussian process to build performance models and follows the Bayesian optimization framework to make scaling decisions. Experiment results on representative workloads show that AuTraScale+ outperforms existing scaling methods on resource-saving and accuracy. For future work, we will investigate efficient methods to predict the fluctuation of input data rates for guiding scaling decisions to minimize reconfiguration probability over the near future.

References

[1] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *Bull. IEEE Comput. Soc. Tech. Committee Data Eng.*, vol. 36, no. 4, pp. 28–38, 2015.

[2] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proc. 24th ACM Symp. Operating Syst. Princ.*, 2013, pp. 423–438.

[3] "Storm." Accessed: May 17, 2024. [Online]. Available: https://storm.apache.org/

[4] Y. Mao et al., "StreamOps: Cloud-native runtime management for streaming services in bytedance," in *Proc. VLDB Endowment*, vol. 16, no. 12, pp. 3501–3514, 2023.

[5] J. Liu, Q. Wang, S. Zhang, L. Hu, and D. Da Silva, "Sora: A latency sensitive approach for microservice soft resource adaptation," in *Proc. 24th Int. Middleware Conf.*, 2023, pp. 43–56.

[6] M. R. Hossen, M. A. Islam, and K. Ahmed, "Practical efficient microservice autoscaling with QoS assurance," in *Proc. 31st Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2022, pp. 240–252.

[7] W. W. Song, T. Um, S. Elnikety, M. Jeon, and B.-G. Chun, "Sponge: Fast reactive scaling for stream processing with serverless frameworks," in *Proc. USENIX Annu. Tech. Conf.*, 2023, pp. 301–314.

[8] A. F. Baarzi and G. Kesidis, "Showar: Right-sizing and efficient scheduling of microservices," in *Proc. ACM Symp. Cloud Comput.*, 2021, pp. 427–441.

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

[10] B. Gedik, S. Schneider, M. Hirzel, and K.-L. Wu, "Elastic scaling for data stream processing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 6, pp. 1447–1463, Jun. 2014.

[11] D. J. Abadi et al., "The design of the borealis stream processing engine," in *Proc. Conf. Innov. Data Syst. Res.*, 2005, 2005, pp. 277–289.

[12] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente, and P. Valduriez, "Streamcloud: An elastic and scalable data streaming system," *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 12, pp. 2351–2365, Dec. 2012.

[13] L. Xu, B. Peng, and I. Gupta, "Stela: Enabling stream processing systems to scale-in and scale-out on-demand," in *Proc. IEEE Int. Conf. Cloud Eng.*, 2016, pp. 22–31.

[14] M. Dvorský and E. Anderson, "Comparing cloud dataflow autoscaling to spark and hadoop," 2016. [Online]. Available: https://cloud.google.com/blog/products/gcp/comparing-cloud-dataflow-autoscaling-to-spark-and-hadoop

[15] D. Sun, H. He, H. Yan, S. Gao, X. Liu, and X. Zheng, "LR-stream: Using latency and resource aware scheduling to improve latency and throughput for streaming applications," *Future Gener. Comput. Syst.*, vol. 114, pp. 243–258, 2020.

[16] B. Lohrmann, P. Janacik, and O. Kao, "Elastic stream processing with latency guarantees," in *Proc. IEEE 35th Int. Conf. Distrib. Comput. Syst.*, 2015, pp. 399–410.

[17] T. Z. Fu, J. Ding, R. T. Ma, M. Winslett, Y. Yang, and Z. Zhang, "DRS: Auto-scaling for real-time stream analytics," *IEEE/ACM Trans. Netw.*, vol. 25, no. 6, pp. 3338–3352, 2017.

[18] T. De Matteis and G. Mencagli, "Keep calm and react with foresight: Strategies for low-latency and energy-efficient elastic data stream processing," *ACM SIGPLAN Notices*, vol. 51, no. 8, pp. 1–12, 2016.

[19] V. Cardellini, V. Grassi, F. Lo Presti, and M. Nardelli, "Optimal operator replication and placement for distributed stream processing systems," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 44, no. 4, pp. 11–22, 2017.

[20] V. Cardellini, M. Nardelli, and D. Luzi, "Elastic stateful stream processing in storm," in *Proc. Int. Conf. High Perform. Comput. Simul.*, 2016, pp. 583–590.

[21] Z. She, Y. Mao, H. Xiang, X. Wang, and R. T. Ma, "Streamswitch: Fulfilling latency service-layer agreement for stateful streaming," in *Proc. IEEE Conf. Comput. Commun.*, 2023, pp. 1–10.

[22] K. Razavi, M. Luthra, B. Koldehofe, M. Mühlhäuser, and L. Wang, "FA2: Fast, accurate autoscaling for serving deep learning inference with SLA guarantees," in *Proc. IEEE 28th Real-Time Embedded Technol. Appl. Symp.*, 2022, pp. 146–159.

[23] Z. Wang et al., "Deepscaling: Microservices autoscaling for stable CPU utilization in large scale cloud systems," in *Proc. 13th Symp. Cloud Comput.*, 2022, pp. 16–30.

[24] H. Qiu et al., "{AWARE}: Automate workload autoscaling with reinforcement learning in production cloud systems," in *Proc. USENIX Annu. Tech. Conf.*, 2023, pp. 387–402.

[25] 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 *Proc. 13th {USENIX} Symp. Operating Syst. Des. Implementation*, 2018, pp. 783–798.

[26] T. Heinze, L. Roediger, A. Meister, Y. Ji, Z. Jerzak, and C. Fetzer, "Online parameter optimization for elastic data stream processing," in *Proc. Sixth ACM Symp. Cloud Comput.*, 2015, pp. 276–287.

[27] T. Heinze, Z. Jerzak, G. Hackenbroich, and C. Fetzer, "Latency-aware elastic scaling for distributed data stream processing systems," in *Proc. 8th ACM Int. Conf. Distrib. Event-Based Syst.*, 2014, pp. 13–22.

[28] C.-K. Shieh, S.-W. Huang, L.-D. Sun, M.-F. Tsai, and N. Chilamkurti, "A topology-based scaling mechanism for apache storm," *Int. J. Netw. Manage.*, vol. 27, no. 3, 2017, Art. no. e1933.

[29] T. Buddhika, R. Stern, K. Lindburg, K. Ericson, and S. Pallickara, "Online scheduling and interference alleviation for low-latency, high-throughput processing of data streams," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 12, pp. 3553–3569, Dec. 2017.

[30] T. Li, J. Tang, and J. Xu, "Performance modeling and predictive scheduling for distributed stream data processing," *IEEE Trans. Big Data*, vol. 2, no. 4, pp. 353–364, Dec. 2016.

[31] S. Kulkarni et al., "Twitter heron: Stream processing at scale," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2015, pp. 239–250.

[32] B. Zhao, N. Q. V. Hung, and M. Weidlich, "Load shedding for complex event processing: Input-based and state-based techniques," in *Proc. IEEE 36th Int. Conf. Data Eng.*, 2020, pp. 1093–1104.

[33] A. Slo, S. Bhowmik, and K. Rothermel, "eSPICE: Probabilistic load shedding from input event streams in complex event processing," in *Proc. 20th Int. Middleware Conf.*, 2019, pp. 215–227.

[34] K. Chapnik, I. Kolchinsky, and A. Schuster, "Darling: Data-aware load shedding in complex event processing systems," in *Proc. VLDB Endowment*, vol. 15, no. 3, pp. 541–554, 2021.

[35] J. Xu, Z. Chen, J. Tang, and S. Su, "T-storm: Traffic-aware online scheduling in storm," in *Proc. IEEE 34th Int. Conf. Distrib. Comput. Syst.*, 2014, pp. 535–544.

[36] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[37] V. Cardellini, F. L. Presti, M. Nardelli, and G. R. Russo, "Decentralized self-adaptation for elastic data stream processing," *Future Gener. Comput. Syst.*, vol. 87, pp. 171–185, 2018.

[38] H. Arkian, G. Pierre, J. Tordsson, and E. Elmroth, "Model-based stream processing auto-scaling in geo-distributed environments," in *Proc. Int. Conf. Comput. Commun. Netw.*, 2021, pp. 1–10.

[39] X. Liu, A. V. Dastjerdi, R. N. Calheiros, C. Qu, and R. Buyya, "A stepwise auto-profiling method for performance optimization of streaming applications," *ACM Trans. Auton. Adaptive Syst.*, vol. 12, no. 4, pp. 1–33, 2017.

[40] "A deep-dive into flink's network stack." Accessed: May 17, 2024. [Online]. Available: https://flink.apache.org/2019/06/05/flink-network-stack.html

[41] E. Brochu, V. M. Cora, and N. De Freitas, "A tutorial on Bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning," 2010, *arXiv:1012.2599*.

[42] D. J. Lizotte, "Practical bayesian optimization," Univ. Alberta, Edmonton, Canada, 2008.

[43] S. J. Pan and Q. Yang, "A survey on transfer learning," *IEEE Trans. Knowl. data Eng.*, vol. 22, no. 10, pp. 1345–1359, Oct. 2010.

[44] F. Zhuang et al., "A comprehensive survey on transfer learning," in *Proc. IEEE*, vol. 109, no. 1, pp. 43–76, Jan. 2021.

[45] L. Zhang, W. Zheng, C. Li, Y. Shen, and M. Guo, "Autrascale: An automated and transfer learning solution for streaming system auto-scaling," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2021, pp. 912–921.

[46] J. D. Little, "A proof for the queuing formula: L= λ w," *Operations Res.*, vol. 9, no. 3, pp. 383–387, 1961.

[47] J. D. Little, "Or forum–little's law as viewed on its 50th anniversary," *Operations Res.*, vol. 59, no. 3, pp. 536–549, 2011.

[48] J. Wenger, G. Pleiss, P. Hennig, J. Cunningham, and J. Gardner, "Preconditioning for scalable gaussian process hyperparameter optimization," in *Proc. Int. Conf. Mach. Learn.*, 2022, pp. 23751–23780.

[49] S. Chintapalli et al., "Benchmarking streaming computation engines: Storm, flink and spark streaming," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops*, 2016, pp. 1789–1792.

[50] "Extended Yahoo Streaming job." Accessed: May 17, 2024. [Online]. Available: https://github.com/dataArtisans/yahoo-streaming-benchmark

[51] "Apache beam nexmark benchmark suite." Accessed: May 17, 2024. [Online]. Available: https://beam.apache.org/documentation/sdks/java/testing/\\nexmark/

[52] F. Rossi, M. Nardelli, and V. Cardellini, "Horizontal and vertical scaling of container-based applications using reinforcement learning," in *Proc. IEEE 12th Int. Conf. Cloud Comput.*, 2019, pp. 329–338.

[53] E. Casalicchio, "A study on performance measures for auto-scaling cpu-intensive containerized applications," *Cluster Comput.*, vol. 22, no. 3, pp. 995–1006, 2019.

[54] W. Iqbal, M. Dailey, and D. Carrera, "SLA-driven adaptive resource management for web applications on a heterogeneous compute cloud," in *Proc. IEEE Int. Conf. Cloud Comput.*, 2009, pp. 243–253.

[55] M. Gotin, F. Lösch, R. Heinrich, and R. Reussner, "Investigating performance metrics for scaling microservices in cloudiot-environments," in *Proc. ACM/SPEC Int. Conf. Perform. Eng.*, 2018, pp. 157–167.

[56] F. Rossi, V. Cardellini, and F. L. Presti, "Self-adaptive threshold-based policy for microservices elasticity," in *Proc. 28th Int. Symp. Model. Anal. Simul. Comput. Telecommun. Syst.*, 2020, pp. 1–8.

[57] S. Horovitz and Y. Arian, "Efficient cloud auto-scaling with sla objective using q-learning," in *Proc. IEEE 6th Int. Conf. Future Internet Things Cloud*, 2018, pp. 85–92.

[58] M. Z. Hasan, E. Magana, A. Clemm, L. Tucker, and S. L. D. Gudreddi, "Integrated and autonomic cloud resource scaling," in *Proc. IEEE Netw. Operations Manage. Symp.*, 2012, pp. 1327–1334.

[59] M. Amiri and L. Mohammad-Khanli, "Survey on prediction models of applications for resources provisioning in cloud," *J. Netw. Comput. Appl.*, vol. 82, pp. 93–113, 2017.
[60] L. Toka, G. Dobreff, B. Fodor, and B. Sonkoly, "Machine learning-based scaling management for kubernetes edge clusters," *IEEE Trans. Netw. Service Manag.*, vol. 18, no. 1, pp. 958–972, Mar. 2021.
[61] M. Imdoukh, I. Ahmad, and M. G. Alfailakawi, "Machine learning-based auto-scaling for containerized applications," *Neural Comput. Appl.*, vol. 32, no. 13, pp. 9745–9760, 2020.
[62] L. Yazdanov, "Towards auto-scaling in the cloud: Online resource allocation techniques," Ph.D. dissertation, Technische Universität Dresden, Dresden, Germany, 2016.
[63] P. Lama and X. Zhou, "Autonomic provisioning with self-adaptive neural fuzzy control for percentile-based delay guarantee," *ACM Trans. Auton. Adaptive Syst.*, vol. 8, no. 2, pp. 1–31, 2013.
[64] H. Arabnejad, C. Pahl, P. Jamshidi, and G. Estrada, "A comparison of reinforcement learning techniques for fuzzy cloud auto-scaling," in *Proc. 17th IEEE/ACM Int. Symp. Cluster Cloud Grid Comput*, 2017, pp. 64–73.
[65] J.-C. Lin, J. Mauro, T. B. Røst, and I. C. Yu, "A model-based scalability optimization methodology for cloud applications," in *Proc. IEEE 7th Int. Symp. Cloud Service Comput.*, 2017, pp. 163–170.
[66] A. U. Gias, G. Casale, and M. Woodside, "Atom: Model-driven autoscaling for microservices," in *Proc. IEEE 39th Int. Conf. Distrib. Comput. Syst.*, 2019, pp. 1994–2004.
[67] J. H. Novak, S. K. Kasera, and R. Stutsman, "Cloud functions for fast and robust resource auto-scaling," in *Proc. 11th Int. Conf. Commun. Syst. Netw.*, 2019, pp. 133–140.
[68] F. Rossi, V. Cardellini, and F. L. Presti, "Hierarchical scaling of microservices in kubernetes," in *Proc. IEEE Int. Conf. Autonomic Comput. Self-Organizing Syst.*, 2020, pp. 28–37.
[69] S. Merkouche and C. Bouanaka, "A proactive formal approach for microservice-based applications auto-scaling," in *Proc. 11th Seminary Comput. Sci. Res. Feminine LIRE Lab. Constantine*, vol. 2, pp. 15–28, 2022.
[70] A. Shahidinejad, M. Ghobaei-Arani, and L. Esmaeili, "An elastic controller using colored petri nets in cloud computing environment," *Cluster Comput.*, vol. 23, no. 2, pp. 1045–1071, 2020.
[71] T.-Y. Hsu and A. D. Kshemkalyani, "A proactive, cost-aware, optimized data replication strategy in geo-distributed cloud datastores," in *Proc. 12th IEEE/ACM Int. Conf. Utility Cloud Comput.*, 2019, pp. 143–153.
[72] D. Saxena and A. K. Singh, "A high availability management model based on VM significance ranking and resource estimation for cloud applications," *IEEE Trans. Services Comput.*, vol. 16, no. 3, pp. 1604–1615, May/Jun. 2022.
[73] D. Saxena, J. Kumar, A. K. Singh, and S. Schmid, "Performance analysis of machine learning centered workload prediction models for cloud," *IEEE Trans. Parallel Distrib. Syst.*, vol. 34, no. 4, pp. 1313–1330, Apr. 2023.
[74] A. K. Singh, D. Saxena, J. Kumar, and V. Gupta, "A quantum approach towards the adaptive prediction of cloud workloads," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 12, pp. 2893–2905, Dec. 2021.

Liang Zhang (Student Member, IEEE) received the BE degree in computer science and technology from Northeastern University in China, in 2018. She is currently working toward the PhD degree with the Department of Computer Science and Engineering, Shanghai Jiao Tong University. Her research interest includes stream processing and resource scheduling in the cloud or edge computing environment. For more information, please visit https://zl-cs.github.io/.

Wenli Zheng (Member, IEEE) received the PhD degree in electrical and computer engineering from The Ohio State University, in 2016. He is currently a tenure-track assistant professor with Shanghai Jiao Tong University. His research interests include computer architecture, data center power management, and data-driven computing.

Kuangyu Zheng (Member, IEEE) received the PhD degree in computer engineering from the Ohio State University. He is an associate professor with the School of Electronic and Information Engineering, Beihang University. His research interests include energy-efficient communication, computer, and network systems. He has received the Distinguished Paper Award and Best Student Paper Award of SAGC 2021; He has also received the Distinguished Service Award from IEEE GreenCom 2023. He is the committee member of IMT-2020 (5 G) and IMT-2030 (6 G) research promotion group of MIIT, China, and the executive committee member of CCF TCARCH and CCF TCCOMM.

Hongzi Zhu (Senior Member, IEEE) received the PhD degree in computer science from Shanghai Jiao Tong University, in 2009. He was a posdoctoral fellow with the Department of Computer Science and Engineering, Hong Kong University of Science and Technology, and the Department of Electrical and Computer Engineering, University of Waterloo, in 2009 and 2010, respectively. He is now a professor with the Department of Computer Science and Engineering, Shanghai Jiao Tong University. His research interests include mobile sensing, mobile computing, and Internet of Things. He received the Best Paper Award from IEEE Globecom 2016. He is an associate editor for *IEEE Transactions on Vehicular Technology* and *IEEE Internet of Things Journal* For more information, please visit http://lion.sjtu.edu.cn.

Chao Li (Senior Member, IEEE) received the PhD degree from the University of Florida, in 2014. He is a full professor with the Department of Computer Science and Engineering, Shanghai Jiao Tong University. His primary research area is system architecture design with an emphasis on energy-efficient, high-performance computers of large scale. His broader research interests also include emerging technologies and evolving applications that could ultimately shape the next-generation computing paradigms.

Minyi Guo (Fellow, IEEE) received the PhD degree in computer science from the University of Tsukuba, Japan. He is a Zhiyuan chair professor with the Department of Computer Science and Engineering, Shanghai Jiao Tong University, China. He is currently Zhiyuan chair professor. His present research interests include parallel/distributed computing, compiler optimizations, embedded systems, pervasive computing, Big Data and cloud computing. He is now on the editorial board for *IEEE Transactions on Parallel and Distributed Systems*, *IEEE Transactions on Cloud Computing* and *Journal of Parallel and Distributed Computing*.