StreamOps: Cloud-Native Runtime Management for Streaming Services in ByteDance

Yancan Mao*
National University of Singapore
maoyancan@u.nus.edu

Zhanghao Chen
ByteDance Inc.
chenzhanghao.1997@bytedance.com

Yifan Zhang
ByteDance Inc.
zhangyifan.5@bytedance.com

Meng Wang
ByteDance Inc.
wangmeng.matt@bytedance.com

Yong Fang
ByteDance Inc.
fangyong.1218@bytedance.com

Guanghui Zhang
ByteDance Inc.
zhangguanghui@bytedance.com

Rui Shi
ByteDance Inc.
shirui@bytedance.com

Richard T. B. Ma
National University of Singapore
tbma@comp.nus.edu.sg

ABSTRACT

Stream processing is widely used for real-time data processing and decision-making, leading to tens of thousands of streaming jobs deployed in ByteDance cloud. Since those streaming jobs usually run for several days or longer and the input workloads vary over time, they usually face diverse runtime issues such as processing lag and varying failures. This requires runtime management to resolve such runtime issues automatically. However, designing a runtime management service on the ByteDance scale is challenging. In particular, the service has to concurrently manage cluster-wide streaming jobs in a scalable and extensible manner. Furthermore, it should also be able to manage diverse streaming jobs effectively.

To this end, we propose *StreamOps* to enable cloud-native runtime management for streaming jobs in ByteDance. *StreamOps* has three main designs to address the challenges. 1) To allow for scalability, *StreamOps* is running as a standalone lightweight control plane to manage cluster-wide streaming jobs. 2) To enable extensible runtime management, *StreamOps* abstracts control policies to identify and resolve runtime issues. New control policies can be implemented with a detect-diagnose-resolve programming paradigm. Each control policy is also configurable for different streaming jobs according to the performance requirements. 3) To mitigate processing lag and handling failures effectively, *StreamOps* features three control policies, i.e., auto-scaler, straggler detector, and job doctor, that are inspired by state-of-the-art research and production experiences at ByteDance. In this paper, we introduce the design decisions we made and the experiences we learned from building *StreamOps*. We evaluate *StreamOps* in our production environment, and the experiment results have further validated our system design.

PVLDB Reference Format:
Yancan Mao, Zhanghao Chen, Yifan Zhang, Meng Wang, Yong Fang, Guanghui Zhang, Rui Shi, and Richard T. B. Ma. StreamOps: Cloud-Native Runtime Management for Streaming Services in ByteDance. PVLDB, 16(12): 3501 - 3514, 2023.

doi:10.14778/3611540.3611543

*Work is done when the author is in ByteDance.

Proceedings of the VLDB Endowment, Vol. 16, No. 12 ISSN 2150-8097.
doi:10.14778/3611540.3611543

1 INTRODUCTION

ByteDance relies heavily on stream processing to handle massive amounts of real-time streaming data. We have adopted an open-sourced solution Apache Flink [8] as the Stream Processing Engine (SPE) to serve large-scale stream processing. According to the open-sourced statistics [13], popular applications such as TikTok, Douyin, and Toutiao in ByteDance serve 1.9 billion users across 150 countries. Those applications generate up to 9 billion online streaming records per second during the peak hour. Distributed stream processing plays an important role in those applications to handle such high-volume data in real-time. Typical stream processing scenarios include high-quality content recommendation, real-time streaming data warehouse, and security and risk control. Specifically, tens of thousands of stateless and stateful streaming jobs are deployed on large-scale in ByteDance cloud. They are allocated with millions of CPU cores and exabytes of memory to process massive amounts of data with high throughput and low latency. Streaming jobs under different scenarios have shown high diversity in terms of input workload characteristics, processing logic, and performance requirements.

An automated runtime management service is necessary to ensure that streaming jobs can keep up with the input stream and meet the performance requirements. As a fact, streaming jobs usually experience changes in workloads and resources during their lifetime. Such dynamic changes cause varying runtime issues for streaming jobs such as processing lag and runtime failures. ByteDance cloud statistics show that hundreds of streaming jobs may experience these issues on a daily basis. Resolving such runtime issues requires recursively analyzing the runtime status of streaming jobs and adjusting the job configurations such as processing logic, resources allocation, and workloads distribution scheme. As a result, manual runtime management of tens of thousands of streaming jobs on the ByteDance scale is not a feasible option due to its time-consuming and resource-intensive nature.

We propose *StreamOps*, a cloud-native runtime management service for cluster-wide streaming jobs. In general, *StreamOps* is inspired by Trisk [31] to run as a control plane. It applies runtime management in three main steps: 1) Monitoring runtime metrics for cluster-wide streaming jobs. 2) Applying control policies to identify runtime issues and their root causes. 3) Resolving runtime issues by dynamically reconfiguring affected streaming jobs. The design and implementation of *StreamOps* are following standard cloud-native principles [12], it addresses the runtime management problem of streaming jobs in ByteDance scenario with three main designs:

1) To scale *StreamOps* to manage cluster-wide streaming jobs, *StreamOps* is designed as a standalone service. In this way, it can be scaled to multiple parallel instances, allowing for concurrent management of varying numbers of streaming jobs. It further leverages global storage to store necessary state such as runtime metrics and control settings for streaming jobs.

2) To extend new control policies for future runtime management optimization opportunities, *StreamOps* separates the control policies from the execution of control mechanisms. Thus, new control policies can be implemented with a detect-diagnose-resolve procedure without knowing the details of control mechanisms. It further exposes control settings of different control policies for customization across different streaming jobs.

3) To mitigate processing lag and handle common failures effectively, *StreamOps* features three control policies: auto-scaler, straggler detector, and job doctor. We observe that the processing lag of streaming jobs is mainly caused by under-provisioned resources, stragglers, and/or data skewness. These three control policies are designed to resolve the corresponding scenarios. a) The auto-scaler identifies overloaded streaming jobs and resolves the issue by scaling out resources predictively. It can also scale in underloaded streaming jobs. b) The straggler detector identifies tasks that are slower than their peers due to resource issues and re-allocates resources for them. c) The job doctor consists of multiple rules to handle data skewness scenarios and common failure issues that require manual source code updates. Specifically, instead of applying dynamic reconfigurations, it fires alarms with insights, e.g., root causes and potential solutions of runtime issues, to help users handle them appropriately.

To summarize, we made the following contributions in this paper:

- We demonstrate the design decisions we made to build *StreamOps*, a cloud-native control plane that can manage large-scale streaming jobs in ByteDance cloud.
- *StreamOps* encapsulates a three-step programming paradigm for control policies implementation.
- *StreamOps* features three control policies, auto-scaler, straggler detector, and job doctor, to effectively mitigate processing lag and common failures for streaming services in ByteDance scenario.
- We present the experiment results in our production environment to demonstrate the effectiveness of *StreamOps*.

2 BACKGROUND AND MOTIVATION

In this section, we first introduce the concepts of stream processing. Then, we describe the features of streaming services in ByteDance's scenario. Finally, we discuss our runtime management requirements, which motivate our build of *StreamOps*.

2.1 Background

2.1.1 Distributed Stream Processing. A streaming job is instantiated from an *execution topology*. The *execution topology* describes the processing logic of a streaming job and can be represented as a directed acyclic graph, where vertices in the graph represent *operators* and edges represent the intermediate *streams* between operators. To deploy a streaming job, each *operator* can be further instantiated as parallel *tasks*, and the input *streams* of the operator are partitioned among those tasks for parallel execution. A *configuration* for a streaming job mainly describes its execution plan. It mainly covers the resources allocated for the Flink cluster, the processing logic of operators, and the distribution of the workload among parallel tasks.

We adopt an open-source solution Apache Flink as the SPE for large-scale stream processing. In most streaming scenarios, Flink streaming jobs are deployed in per-job mode on Kubernetes [6], ensuring each job has its own independent Flink cluster for performance isolation. The Flink Runtime comprises two main components: *JobManager* and *TaskManager*. The *JobManager* is a centralized process that ensures consistency and fault tolerance while the *TaskManager*s are allocated with resources and deployed as worker nodes for a streaming job.

2.1.2 Streaming Services in ByteDance. The streaming services in ByteDance cloud mainly exhibit three features: *large-scale deployment*, *diverse streaming scenarios*, and *changes of workloads and resources*.

Large-scale deployment. As aforementioned, to enable real-time online services under high volume input workloads, i.e., up to 9 billion streaming records per second, distributed stream processing has been widely adopted. Tens of thousands of streaming jobs are deployed in ByteDance cloud. They are allocated with millions of CPU cores and exabytes of memory to process high-volume streaming workloads.

Diverse streaming scenarios. Streaming jobs in ByteDance have also shown high diversity in terms of input workload characteristics, processing logic, and performance requirements. Based on our statistics, ByteDance has thousands of streaming scenarios that are divided according to various projects. Typical types of streaming scenarios in ByteDance are: 1) Online feature extraction and online machine learning. Online recommendation services must extract massive online features in real-time from multiple sources, which demand low latency, high throughput, and high availability to achieve a high-quality content recommendation. 2) Real-time data warehouse for statistical analysis. Data transmission services (DTS) in the real-time data warehouse are implemented as streaming jobs that continuously clean and transform data among varying data sources e.g., Kafka [25] and MySql [39]. Those services are less sensitive to processing latency but require more resource efficiency.

Changes of workloads and resources. Streaming jobs usually experience changes in their workloads and resources during their lifetime, resulting in varying runtime issues such as processing lag and failures. Our analysis reveals that on a daily basis, hundreds

of streaming jobs experience these runtime issues. Specifically, streaming jobs in ByteDance generate 9 billion online streaming records per second during peak hours, while during trough hours, the volume decreases by approximately 4-5 times. As a result, streaming jobs configured with under-provisioned resources may experience high processing lag when the input rate of streaming data is significantly increased during peak hours. Conversely, we can configure over-provisioned resources for high-priority streaming jobs based on their workload during peak hours, which avoids high processing lag but results in resource wastage. Besides the changes in data volume, the workload distribution among partitions of input streams may also change over time, resulting in data skewness among parallel tasks and accumulating processing lag. In addition, streaming jobs can be slowed down when parallel tasks are deployed on physical nodes experiencing resource issues, such as resource contention, slow disk I/O, and older machines with limited computational resources. Furthermore, runtime failures can occur due to data processing exceptions or server crashes.

2.2 Motivation

As more and more streaming jobs are deployed, applying runtime management manually for tens of thousands of streaming jobs on the ByteDance scale becomes extremely time-consuming and requires significant manpower resources. This process involves users reporting issues, experts analyzing runtime metrics and logs from Flink, Message Queues, and Kubernetes, and making manual adjustments to the job configurations until the issue is resolved. Thus, an automated runtime management service that can serve a control plane is necessary. By summarizing the runtime management scenarios, we conclude the following three main requirements for the control plane:

1) **Scalability.** The control plane has to be scalable enough to manage cluster-wide streaming jobs. In particular, the control plane needs to apply control policies on tens of thousands of streaming jobs concurrently. The runtime management on each streaming job should be completed in a short time, e.g., within five minutes. Additionally, as a cloud-native service, it should have an availability of at least 99.9% to provide stable runtime management for streaming jobs in ByteDance cloud.

2) **Effectiveness.** The control plane should incorporate effective control policies to address common runtime issues such as processing lags and failures Runtime issues in stream processing can be caused by various factors. For instance, processing lag can be the result of under-provisioned resources of the streaming job or load imbalance across parallel tasks. Therefore, to ensure effective runtime management for streaming jobs, the control plane must perform two key functions: a) identify the root causes of runtime issues by analyzing metrics from different sources, and b) resolve these issues using carefully designed models.

3) **Extensibility.** While the control plane currently handles a subset of observed runtime issues, it should be designed with extensibility to allow for the implementation of control policies for new scenarios that may arise in the future. For example, we plan to investigate the feasibility of using machine learning-based control policies to address a broader range of resource configuration issues. Furthermore, the control plane should be able to customize control policies to suit the diverse workload characteristics and performance requirements of different streaming jobs. Intuitively, runtime issues of high-priority streaming jobs need to be resolved promptly to prevent revenue loss.

We observe that novel research prototypes [16, 23] and in-production solutions [30, 33] have been proposed to apply runtime management for streaming jobs in the past decade. However, they fall short of satisfying all runtime management requirements of streaming services in ByteDance scenario.

Representative research prototypes [16, 23] mainly focus on designing effective control policies to detect and resolve the runtime issues for a single streaming job. Specifically, DS2 [23] mainly focus on the auto-scaling problem, it proposes a novel rate-based model to detect bottleneck among the entire pipeline and predict optimal parallelism for each operator. Dhalion [16] proposes a hybrid control policy to identify multiple resource issues such as resource over/under-provision, data skewness, and stragglers in a streaming job. However, both Dhalion and DS2 failed to provide an end-to-end solution to achieve large-scale runtime management for cluster-wide streaming jobs. Specifically, they failed to address 1) the lifecycle management of the control plane on a large scale and 2) how to customize control policies for streaming jobs with varying performance requirements and workload features.

In-production solutions [30, 33] are carefully designed on the self-maintained streaming engine and resource management platform. Specifically, to naturally leverage the scalability of their own SPE Flare on Orleans [7], Chi [30] embeds the control plane as a special operator into the streaming pipeline, which achieves efficient single job management but ignored cluster-level deployment details. Turbine [33] proposes a modularized system architecture to manage Jobs, Tasks, and Resources separately at the cluster level. But it is tightly coupled with their own infrastructure [10, 27] in Meta. In contrast, ByteDance leverages open-source solutions, such as Apache Flink as the SPE and Kubernetes as the cloud platform, for deploying its cloud-native streaming services. This requires customizations in those platforms to fill the gap for efficient runtime management of streaming services over ByteDance cloud. Although there have been efforts [14, 15] in the Flink community to explore resource management for streaming jobs, we advocate for a holistic solution that can effectively resolve runtime issues related to resources, workloads, and the processing logic for streaming jobs.

The unique requirements of runtime management in ByteDance's scenario necessitate a distinct design for the control plane that fulfills three key requirements. We propose *StreamOps*, a control plane for large-scale cloud-native stream processing runtime management. In the following chapters, we first describe our design decisions and the overall system architecture in Section 3. Next, in Section 4, we discuss how *StreamOps* enables extensible runtime management for varying streaming jobs. Then, in Section 5, we introduce three effective control policies to resolve common runtime issues in ByteDance.

3 *STREAMOPS* OVERVIEW

In this section, we discuss the design decisions and introduce an overview of the *StreamOps* architecture and workflow.

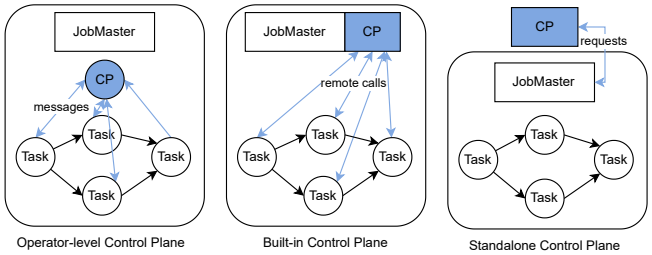


Figure 1: The design choices for where to deploy the control plane (CP).

3.1 Design Decisions

The main design goal of the *StreamOps* is to manage tens of thousands of Flink streaming jobs in ByteDance cloud concurrently with low response time. To achieve this goal, we mainly need to answer three questions: 1) Where to deploy the control plane? 2) How does the control plane apply cluster-wide runtime management? 3) Where to store the runtime management state?

3.1.1 Where to deploy the control plane? By summarizing the state-of-the-art work [16, 17, 23, 30, 31, 33], we mainly observed three design choices of the control plane deployment as shown in Figure 1:

1) **Operator-level control plane** [30]. To manage the runtime of each streaming job, the control plane can be implemented as a special control operator in a streaming job. The control operator interacts with other operators via the fine-grained control messages to retrieve metrics and apply reconfigurations efficiently. Such a design achieves low latency interaction between the control plane and parallel tasks in SPE runtime. In addition, the control policies of the control plane can be implemented flexibly inside the operator. However, it requires significant modifications to the SPE runtime to achieve the operator-level interaction via control messages. Additionally, the failure of the control plane affects the availability of the SPE runtime.

2) **Built-in control plane** [17, 31]. The control plane can be implemented as a centralized service inside the SPE runtime. For example, the control plane can be running as a process alongside JobManager in Flink Runtime. Subsequently, the control plane interacts with SPE runtime with a carefully designed control protocol to achieve efficient runtime management. The protocol can be implemented by leveraging the network stack in the SPE runtime for low-latency communication. In addition, the failure of the control plane does not affect the SPE runtime. However, since the control plane is implemented in SPE runtime, any update on the control plane involves modifications to the SPE source code. Moreover, in cases where control policies require metrics from sources outside of the SPE runtime, such as message queues and Kubernetes, the SPE runtime has to interact with third-party systems.

3) **Standalone control plane** [16, 23, 33]. The control plane can be implemented as a standalone centralized service running outside of the SPE runtime. Subsequently, the control plane can be deployed with a set of parallel instances on the cloud to manage streaming jobs concurrently. In particular, the lifecycle management of the control plane is independent of the SPE runtime. The control plane can also adapt resources on demand based on the number of streaming jobs it manages. Since the control plane is standalone, the development of the control plane does not affect the SPE runtime. But at the cost of remote communication cost.

Our decision. We choose to design *StreamOps* as a standalone service in ByteDance cloud to manage cluster-wide streaming jobs for the following reasons: 1) The development of the *StreamOps* can be independent of the SPE runtime development. Hence, we can add features to *StreamOps* easily in the future. 2) *StreamOps* is able to manage streaming jobs with runtime metrics from different sources without introducing additional dependencies on SPE runtime. 3) The lifecycle management of the *StreamOps* is decoupled from the SPE runtime, which provides greater flexibility and higher availability.

3.1.2 How does the control plane apply cluster-wide runtime management? The control plane can be scaled to multiple instances. To enable cluster-wide runtime management, control plane instances may interact with streaming jobs in the following two approaches:

1) **Proactive cluster-wide scanning.** The control plane may initiate multiple instances and allocate each instance with a subset of streaming jobs for periodic runtime status scans. In this way, runtime management can be applied in a top-down architecture, and streaming jobs are not aware of the upper layer control plane. However, it is hard to customize the runtime management check for different streaming jobs.

2) **Passive per-job triggering.** Instead of scanning streaming jobs proactively, the control plane may listen to runtime management requests from streaming jobs passively. Each streaming job can be associated with a runtime management trigger to send requests to the control plane. Subsequently, a control plane instance will be assigned to process runtime management requests. Such an approach enables customizable runtime management for different streaming jobs. However, the lifecycle management of the triggers introduces additional engineering efforts.

Our decision. We choose to adopt the passive per-job triggering as the default approach for the following reasons: 1) Enabling customizable runtime management matches the design goal of *StreamOps* in ByteDance. 2) Decoupling the runtime management triggers from the control plane makes the control plane architecture more lightweight with lower resource consumption. 3) Although this approach makes local runtime management decisions for each streaming job, it can also make global decisions to achieve multi-tenant runtime management, e.g., dynamically scaling cluster-wide streaming jobs, based on a global lock such as Zookeeper [21]. 4) The runtime management trigger can either be implemented within the SPE runtime to leverage SPE runtime for simplified lifecycle management or managed independently in a unified manner. We choose to implement them within the SPE runtime, leaving the unified management approach for future work.

3.1.3 Where to store the runtime management state? Control policies in the control plane can be stateful. This requires the control plane to maintain the necessary state for the associated control policies. There are mainly two types of state. 1) Control policies may need to read the historical metrics and configurations of each steaming job for more accurate and stable dynamic reconfiguration decisions. 2) Since varying streaming jobs may customize their

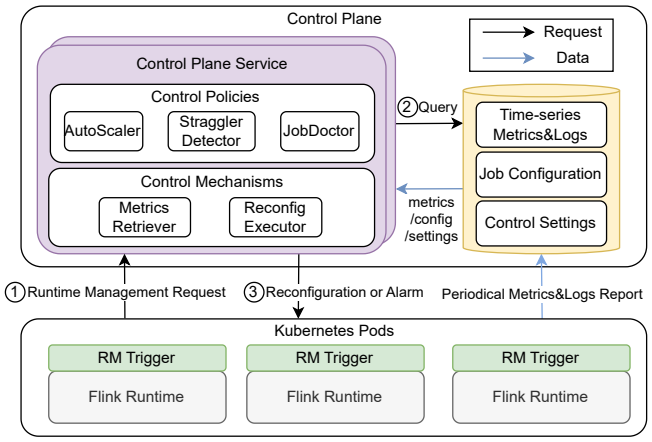


Figure 2: *StreamOps* architecture and a workflow overview.

control policies, the control plane needs to maintain control settings for the associated streaming job. We mainly considered two types of state storage approaches:

1) **Local state storage.** Streaming jobs can be pinned to a control plane instance. In this way, the associated state of a streaming job can be stored in the control plane instance locally. The main benefit of such an approach is to access the state efficiently without remote state access. However, additional state management overhead has been introduced to the control plane to maintain the state of the allocated streaming jobs. In addition, because the streaming jobs may have varying runtime management frequencies and dynamic reconfiguration overhead, the load balance among control plane instances is challenging.

2) **Global state storage.** The control plane may also maintain the state in global storage, such that all control plane instances access state remotely. In this way, the control plane is stateless and each instance can manage an arbitrary streaming job. The main benefit of this approach is to make the control plane more lightweight and scalable. It is also easier to achieve load balancing. Moreover, the maintenance of the global state storage introduces additional engineering overhead.

Our decision. We choose to adopt the global state storage approach for the following reasons: 1) Global state storage enables more desired benefits for ByteDance scenarios such as lightweight and scalability. 2) We can easily leverage existing in-production level global storage services in ByteDance to manage the state of the control plane correspondingly.

3.2 Overall Architecture

By summarizing our design decisions, we have developed *StreamOps*. An overview of the *StreamOps* architecture and an example workflow is depicted in Figure 2.

3.2.1 StreamOps Runtime. *StreamOps* mainly has three modules that can natively interact with Kubernetes and run in containers. 1) A *control plane service* to manage cluster-wide streaming jobs. 2) A *global storage* to manage the associated state for *control plane service*. 3) Each streaming job is attached with a runtime management trigger, i.e., *RM Trigger* to send runtime management requests to the control plane.

Control Plane Service. The control plane service is stateless and can be scaled to a set of parallel control plane instances. There is a load balancer running alongside parallel control plane instances. It receives runtime management requests and forwards them to parallel control plane instances. Specifically, the load balancer distributes the requests among control plane instances with the awareness of the workloads among control plane instances.

Global Storage. The global storage decouples the periodic metrics/logs report and the metrics/logs retrieval. During normal stream processing, the raw metrics and logs from different sources, e.g., Flink Runtime, Message Queues, and Kubernetes, are periodically reported to the global storage. During runtime management, i.e., a runtime management request is triggered, the *control plane service* queries necessary runtime metrics and logs to apply runtime management accordingly.

Runtime Management Trigger. The runtime management trigger can send runtime management requests on demand. In particular, a runtime management request can be triggered in three ways. 1) Scheduled trigger: streaming jobs may trigger runtime management requests periodically. This is useful to detect unpredictable runtime issues in time. For example, the frequency of runtime management can be customized for each streaming job. 2) Conditional trigger: streaming jobs can trigger runtime management requests on specific conditions. For instance, a streaming job can be configured to automatically trigger a request when processing lag exceeds a certain threshold or when the pipeline experiences backpressure. Additionally, some streaming jobs may exhibit periodic peaks at certain times and can be configured with a conditional trigger to adjust resources accordingly. 3) Manual trigger: the owners of streaming jobs may trigger runtime management requests manually, which is important for resolving emergent runtime issues.

3.2.2 StreamOps Workflow. *StreamOps* interacts with *Flink Runtime* in mainly three steps.

1) The *RM trigger* of a streaming job sends a runtime management request to the *control plane service* at a configurable frequency. The *RM trigger* sends a new runtime management request only when the last one is completed. Hence, there will only be one control plane instance to manage the streaming job at any time.

2) All runtime management requests will be handled by an instance of *control plane service*. Subsequently, the *control plane service* instance queries the runtime management state for the streaming job from *global storage*. Such as a) a subset of runtime metrics, b) the associated job configurations, and c) control settings Then, the *control plane service* makes control decisions according to the retrieved state.

3) The *control plane service* can either apply a dynamic reconfiguration or fire an alarm according to the control decisions. The dynamic reconfiguration is executed by instructing *Flink Runtime* to update its job configurations. An alarm will be fired when a system-level reconfiguration is not applicable. In the alarm, we reveal the root causes and possible solutions to help users resolve runtime issues for streaming jobs manually.

3.2.3 Fault Tolerance. *StreamOps* ensures stable and fault-tolerant runtime management services by considering various failure

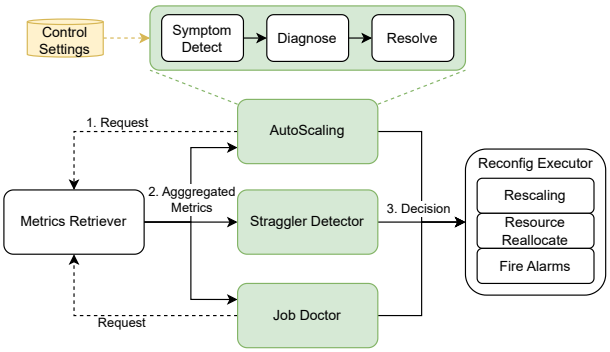


Figure 3: An execution workflow overview of control policies.

scenarios: 1) Failure of runtime modules. Since *control plane service* is stateless, its failures can be naturally handled by reallocating resources and restarting new instances from cloud platforms. *Global storage* is managed by an in-production storage service in ByteDance with fault tolerance. *RM Trigger* extends the fault tolerance semantics from Flink runtime. 2) Failure of metrics retrieval. Metrics reported from different sources in an ad-hoc manner may be missing or out-of-order, leading to unexpected control decisions. To mitigate this, *StreamOps* incorporates a metrics completeness check step before executing runtime management. 3) Failure of making control decisions. Control decisions can be ineffective or incorrect in resolving runtime issues. Similar to prior work [16], *StreamOps* employs a rollback mechanism when the runtime issue persists. It also limits or temporarily pauses frequent control decisions to prevent excessive resource updates that could impact the stability of cloud platforms. 4) Failure of reconfigurations. The execution of reconfigurations may fail due to stream processing errors or the unavailability of resources on cloud platforms. *StreamOps* handles such failures by gracefully skipping the current reconfiguration and allowing the streaming job to request the next round of runtime management in the future.

4 CONTROL PLANE SERVICE

The design goal of the *control plane service* is to achieve extensible control policies for streaming jobs. To achieve this, by following the principle of the separation of policy and mechanism [26], we separate control policies from the detailed control mechanisms that are physically interacting with Flink Runtime. Specifically, control policies can be implemented by leveraging the exposed three-step programming paradigm. Control mechanisms, i.e., metrics retrieval and reconfiguration, are encapsulated and can be executed by interacting with Flink Runtime and other systems.

4.1 Control Policies

Control policies read runtime metrics and logs to detect runtime issues and make control decisions to resolve them if necessary. To enable extensible and customizable control policies, we separate control policies from the execution mechanisms. Thus, control policies can be implemented with a detect-diagnose-resolve procedure without knowing the details of control mechanisms. We discuss the design of the auto-scaler, straggler detector, and job doctor following this procedure in Section 5. We encapsulate a programming paradigm according to the detect-diagnose-resolve procedure. This allows us to extend new control policies easily in the future. In addition, each control policy is further exposed with control settings to customize the runtime management for each streaming job. This allows users to fine-tune control policies according to their performance requirements.

Algorithm 1: Code sketch of a control policy in *StreamOps*.

```
Data: Job_Name // The Streaming Job Name.
Data: Metrics_Name // The runtime metrics required by
      the control policy.
while True do
    // StreamOps listens runtime management requests.
    if receive a runtime management request then
        Run_Control_Policy(Job_Conf, Metrics_Name);
Function Run_Control_Policy(Job_Name, Metrics_Name):
    Conf_job, Conf_contl, Metrics =
      Metrics_Retriever.retrieve(Job_Name, Metrics_Name);
    // Get job configuration, control settings, and the
       associated metrics.
    Symptom = Detect(Metrics, Conf_contl);
    Root_Cause = Diagnose(Symptom, Conf_contl);
    Conf'_job = Resolve(Root_Cause, Conf_job, Conf_contl);
    Reconfig_Executor.execute(Conf'_job);
```

Control Policy Workflow. An overview architecture and execution workflow of control policies inside a control plane instance is depicted in Figure 3. When a runtime management request is triggered, the associated control policy will be applied. It mainly works in three steps. 1) The control policy relies on *Metrics Retriever* to retrieve the runtime management state such as runtime metrics of the streaming job. 2) The control policy makes control decisions based on a detect-diagnose-resolve procedure. It can be customized for each streaming job with pre-defined control settings. 3) The control decisions can be applied on the Flink Runtime based on the *Reconfig Executor*. Specifically, the *Reconfig Executor* may either apply reconfiguration on the Flink Runtime or fire alarms to the users. Multiple control policies can be executed during each round of runtime management on a control plane instance, but only one control decision can be applied at a time. This is decided by the priority assigned to each control policy.

Control Policy Programming Paradigm. Based on the overall execution workflow, we expose the detect-diagnose-resolve programming paradigm for control policies implementation. This is similar to the programming model provided by Dhalion [16] and DS2 [23]. We show an example of implementing a control policy based on the detect-diagnose-resolve programming paradigm in Algorithm 1. The control policy is long-running and listens to the associated runtime management requests. When the control policy receives a request, it applies a detect-diagnose-resolve procedure for the associated streaming job. 1) The control policy has a pre-defined subset of metrics and logs as the input, which is retrieved from the *Metrics Retriever*. Subsequently, it detects symptoms based on those metrics and logs. 2) If a symptom is detected, the

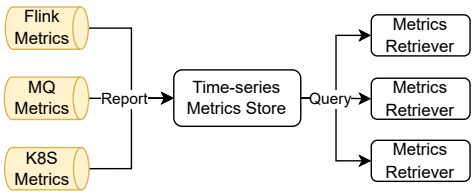


Figure 4: An overview of Time-series Metrics Store.

control policy diagnoses the root causes for the symptom. The diagnosis is executed by carefully analyzing related metrics. For example, the root cause of the processing lag can be either under-provisioned resources or load imbalance. To further confirm the root cause, the control policy has to identify the bottlenecked operator, and then, check the detailed execution status and resource utilization of parallel tasks. 3) If the root cause is located, the control policy needs to propose a new configuration to resolve the performance bottleneck if necessary. This is achieved by invoking reconfiguration APIs provided by *Reconfig Executor*.

Control Settings. To allow for more precise identification of runtime issues, the execution of control policies on a specific streaming job can be customized through user-defined *control settings*. For instance, users can adjust the lag size threshold for the auto-scaler to fine-tune its sensitivity in making dynamic scaling decisions. Control policies may have different *control settings*, and developers may expose these settings to users according to the detailed execution logic in the control policy.

4.2 Control Mechanisms

Control mechanisms interact with other systems such as global storage and Flink Runtime for metrics retrieval and dynamic reconfiguration. In the following, we discuss the detailed techniques and design decisions that have been made to achieve efficient execution of control mechanisms.

Metrics Retrieval. The runtime metrics required by control policies are retrieved by the *Metrics Retriever*. The runtime metrics can be complex in two dimensions: the types of metrics and the time range over which they are measured. While control policies usually require only a small subset of runtime metrics as input from those two dimensions. Thus, it is important that *Metrics Retriever* is able to query a subset of metrics for efficient runtime management.

Existing SPEs such as Apache Flink enable queryable metrics by simply reading runtime metrics of the streaming job on the fly. Such a metrics management approach is simple and works well in small-scale clusters, but does not meet the goal of *StreamOps* in three folds. 1) It does not support the range query of historical metrics. Instead, only the latest runtime metrics are available. 2) In a large-scale cluster, collecting metrics is time-consuming. It requires gathering the latest metrics of parallel tasks across different physical nodes. 3) Metrics from other platforms such as Kubernetes and Message Queues are also important for identifying runtime issues.

The main idea of metrics management in ByteDance is to cache historical metrics and decouple the metrics retrieval and report via an in-production time-series database. An overview of metrics management over ByteDance cloud is depicted in Figure 4. There are mainly three types of metrics sources, i.e., metrics from Flink, Kubernetes, and Message Queues, that periodically report

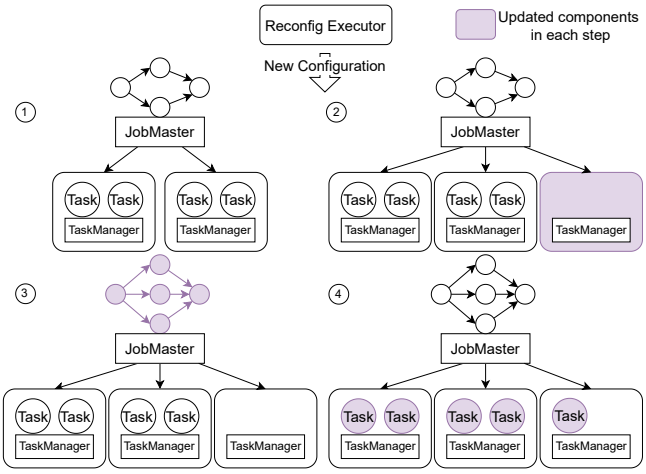


Figure 5: A workflow overview of scaling execution.

metrics to the database. Control policies can get the expected metrics by declaring the required metrics for the *Metrics Retriever*. Subsequently, the *Metrics Retriever* queries a subset of metrics and aggregates the raw metrics as the input for those control policies. The metrics retrieval is efficient with SLO guarantee based on the time-series database service.

Reconfiguration Execution. The reconfiguration of a streaming job is physically executed by the *Reconfig Executor*, which requires support from Flink Runtime. For example, when the auto-scaler makes a scaling out decision, the *Reconfig Executor* proposes a new configuration with respect to the estimated resources. Then, the Flink Runtime needs to rescale the deployed streaming job by acquiring more resources from Kubernetes and creating more task instances according to the new configuration.

The reconfiguration causes the unavailability of the original stream processing. Hence, the impact of reconfiguration needs to be as low as possible. Based on our profiling results of deploying streaming tasks, we observe that a majority of the time (up to 70%) is used for scheduling resources during restarting the entire streaming job in ByteDance cloud. To this end, inspired by the adaptive scheduling [15] in the latest Flink-1.16, we adopt a dynamic reconfiguration mechanism without releasing existing resources. In particular, the Flink Runtime negotiates with Kubernetes to add (or remove) additional Kubernetes pods. Subsequently, the streaming job will be updated without re-compiling or re-submitting.

An overview workflow of reconfiguration is depicted in Figure 5. The Flink Runtime maintains the execution plan for the current streaming job. The reconfiguration is executed by updating the execution plan and redeploying parallel tasks accordingly. Once the *Reconfig Executor* receives a new job configuration that changes the parallelism of operators, it requests the Flink Runtime to scale out/in the streaming job in mainly three steps. 1) The JobMaster creates a new TaskManager for new task deployment without affecting the current stream processing. 2) Once the resource is allocated, JobMaster updates the execution plan of the current streaming job according to the new configuration. 3) Finally, JobMaster stops and redeploys the parallel tasks from the latest checkpoint to let the new configuration take effect. Overall, dynamic reconfiguration takes

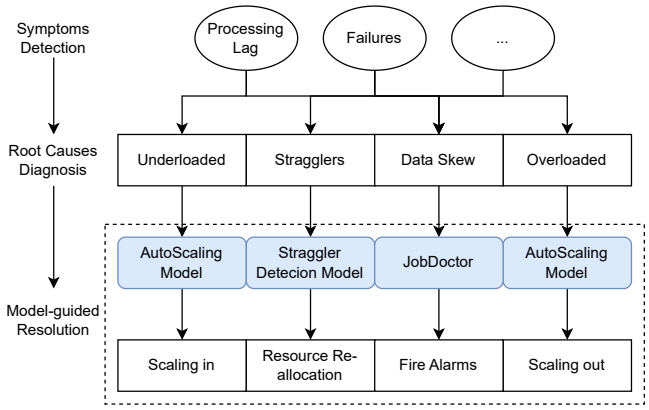


Figure 6: Model-guided control policies in *StreamOps*.

2-3 minutes to complete over large-scale streaming jobs deployed with thousands of parallel tasks, which is up to 2x faster than the default stop-and-restart mechanism in the original Flink Runtime.

5 CONTROL POLICIES INSTANTIATION

The main design goals for control policies in ByteDance are to 1) ensure streaming jobs keep up with the input stream and mitigate common runtime issues such as processing lag and failures. 2) make better use of cluster resources. *StreamOps* features an *auto-scaler*, a *straggler detector*, and a *job doctor* to achieve these goals. Processing lag is mainly caused by under-provisioned resources or load imbalance. The load imbalance can be further caused by stragglers or data skewness. This can be handled effectively with the three proposed control policies. The auto-scaler identifies overloaded and underloaded resources and rescales resources predictively. The straggler detector identifies stragglers and re-allocates resources. While the job doctor handles data skewness and other failure issues that require manual source code updates. It fires alarms with insights to help resolve the issue.

An overview of control policies architecture is shown in Figure 6. As aforementioned, all control policies can be implemented based on the detect-diagnose-resolve paradigm: 1) The frequent runtime issues (or symptoms) can be detected based on pre-defined detection mechanisms over the associated runtime metrics. 2) Control policies diagnose root causes based on carefully designed diagnosis rules. 3) Once a root cause is confirmed, control policies introduce model-guided resolvers in control policies to propose effective solutions.

5.1 Auto-Scaler

The main design goals of auto-scaler are to 1) mitigate high processing lag issues caused by overloaded stream processing with under-provisioned resources by scaling out; and 2) improve resource utilization efficiency when the streaming job is underloaded with over-provisioned resources by scaling in. To achieve these goals, the auto-scaler must answer two key questions: 1) "When to scale": To identify whether the streaming job is overloaded or underloaded and dynamic scaling is required. 2) "How much to scale": To predict the appropriate resources, i.e., what is an optimal resource configuration for current input workloads.

Algorithm 2: Auto-Scaler workflow.

```
1  Function Detect(Metrics, Conf_contl):
2      if Processing_Lag ≥ Overload_Threshold or
         Processing_Lag ≤ Underload_Threshold then
3          return try_rescaling;
4  Function Diagnose(Symptom, Conf_contl):
5      if Symptom is not try_rescaling then
6          return null;
7      if Load balanced then
8          if Processing_Lag ≥ Overload_Threshold and
             Processing_Lag is increasing and
             ∑Overloaded_tasks ≥ n1 then
9              return need_scaling_out;
10         else if Processing_Lag ≤ Underload_Threshold and
             ∑Underloaded_tasks ≥ n2 then
11             return need_scaling_in;
12 Function Resolve(Root_Cause, Conf_job, Conf_contl):
13     if Root_Cause is need_scaling_in or need_scaling_out
         then
14         Try Scaling In/Out with AutoScaling Model;
```

Diagnosis rule. The main challenge of deciding "When to scale" is to accurately identify whether the streaming job is overloaded/underloaded and rescale it accordingly. In particular, simply relying on the backpressure related metrics is not general enough to cover all overloaded cases in ByteDance. For example, the data transmission services in ByteDance are running with a single operator and do not incur backpressure when overloaded. Therefore, the auto-scaler is designed with a *diagnosis rule* over the processing lag related metrics.

An overview of auto-scaler diagnosis workflow is depicted in Algorithm 2. The prerequisite of scaling decisions is the load balance among parallel tasks. The parallel tasks are considered as load balanced with two conditions: 1) Most parallel tasks have similar *useful time ratio*. In particular, the *useful time ratio* of a task follows the definition in DS2 [23], which is measured by the actual working time of a parallel task over a time period. 2) The *CPU resource utilization* of allocated containers for TaskManagers is in a similar range. Subsequently, the streaming job is considered overloaded if: a) It has high processing lag, i.e., the lag size exceeds a predefined threshold and is still increasing. b) Most parallel tasks are overloaded, i.e., having a high useful time ratio. Note that the streaming job may not need to be scaled out when the processing lag is decreasing. In contrast, the streaming job is considered underloaded and can be scaled in when the streaming job has very low processing lag and most parallel tasks have a low useful time ratio. Since the SLO and stability of the streaming job is more important than resource utilization, we set the scaling in threshold more conservatively compared to scaling out. Besides, users can also tune the configurations of the diagnosis rule to adapt to their own streaming scenarios.

Resolution. A streaming job usually requires to be rescaled multiple times to converge to a stable configuration for current workloads. The main challenge of "How much to scale" is: to predict

accurate resources in order to reduce the convergence time for a stable resource configuration. The auto-scaler adopts a predictive *autoscaling model* to answer this question. The autoscaling model predicts the parallelism for each operator inspired by prior work DS2 [23]. This is achieved by quantifying the processing capability and input workloads of each operator. The model requires four inputs. 1) *Streaming DAG* of the job. 2) *Input Rate* of the job. 3) *Useful Time* per operator. 4) *In/Out Ratio* per operator. Intuitively, the *Streaming DAG* can be retrieved directly from Flink Runtime. While the other three inputs have to be derived from runtime metrics. In the following, we discuss how we derive all inputs by querying the associated runtime metrics within a time period T, i.e., from $now - T$ to now.

The *Input Rate* of the streaming job is not always equivalent to the output rate of source operators. In particular, when source operators are backpressured and throttled, the *Input Rate* will be greater than the output rate of source operators, and the processing lags are increasingly accumulated. As a result, we need to get the estimated *Input Rate* with the consideration of the processing lag increase rate, which can be derived as follows:

$$LagIncreaseRate = (Lag_{now} - Lag_{now-T})/T$$

Based on the processing lag increase rate, the estimated *Input Rate* of a source operator can be derived by:

$$InputRate = OutputRate + LagIncreaseRate$$

For the *Useful Time* over a time period T, we observe that *Busy Time Per Second* metrics provided by Flink Runtime records the useful time of a task per second. Subsequently, the *Useful Time* per operator over a time period T can be derived by:

$$UsefulTime = \sum_{now-T}^{now} BusyTimePerSecond$$

The *In/Out Ratio* per operator can be simply derived by summarizing the number of records in/out over the time period T.

When the new parallelism has been calculated based on the DS2 model, the physical resources can be linearly estimated according to the new parallelism, which is similar to the prior work Turbine [33].

5.2 Straggler Detector

The main design goal of the straggler detector is to identify stragglers and migrate them to other nodes in order to mitigate high processing lag issues. Stragglers are parallel tasks that typically run slower than their peers due to various resource issues on the physical nodes. Although cloud platforms such as Kubernetes may alleviate some resource issues among physical nodes, such as resource contention, they may not be able to respond to application-level stragglers in a timely manner due to the overhead involved in scanning all physical nodes. The stragglers cause load imbalance among parallel tasks and further introduces high processing lag. As a result, the main requirement of the straggler detector is to identify stragglers among parallel tasks accurately when the processing lag threshold is exceeded. Subsequently, the straggler detector mainly handles the resource problems of the parallel tasks by reallocating resources for stragglers.

Diagnosis rule. The straggler detector is a rule-based model summarized from the runtime management experience in ByteDance. An overview of the execution workflow of the straggler detector can be depicted in Algorithm 3. we identify stragglers by leveraging the load, i.e., useful time ratio and the processing rate of parallel tasks. In particular, the stragglers that have resource issues usually have higher loads and lower or similar processing rates than their peers. And they are usually located in the same subset of machines. Based on this feature, we diagnose the root cause resulting from stragglers with mainly two steps: 1) We find out a subset of tasks that have a higher load but a lower processing rate than their peers. This is achieved based on Z-score [37]. 2) We consider the subset of tasks as stragglers if the ratio of those tasks is small compared to the total number of parallel tasks. Similar to Dhalion [16], the data skewness scenario is filtered out by comparing the processing rate of overloaded tasks with their peers. We will discuss how we identify data skewness issues in Section 5.3.

Algorithm 3: Straggler Detector workflow.

```
1  Function Detect(Metrics, Conf_contl):
2      if Processing_Lag ≥ Threshold then
3          return try_straggler_detection;
4  Function Diagnose(Symptom, Conf_contl):
5      if Symptom is not try_straggler_detection then
6          return null;
7      if Load imbalanced then
8          Straggler_tasks ← Identify Stragglers with higher load
             from Tasks;
9          if rate of Straggler_tasks ≤ rate of Normal_tasks and
             ∑Straggler_tasks ≤ n then
10             return stragglers_diagnosed;
11 Function Resolve(Root_Cause, Conf_job, Conf_contl):
12     if Root_Cause is stragglers_diagnosed then
13         Re-allocate resources with Straggler Detection Model;
```

Resolution. To resolve the straggler issue, instead of simply migrating a straggler task from one TaskManager to another inside the Flink cluster, we need to migrate TaskManagers containing stragglers to other physical nodes. Thus, migrating stragglers affects other parallel tasks and introduces non-negligible scheduling overhead. In addition, there are also false positive cases among stragglers that are not necessarily needed to be migrated. To this end, we design a simple but effective model, i.e., *Straggler Detection Model* to check and decide whether a TaskManager needs to be migrated. The *Straggler Detection Model* works as follows: 1) For the identified subset of stragglers, we further find out the TaskManagers and their allocated physical nodes. 2) If the majority of TaskManagers containing stragglers are in the same physical node (the number of nodes is configurable), we blacklist the physical node for the streaming job and no further resources will be allocated to the node. 3) We migrate TaskManagers in the blacklisted nodes to other nodes.

5.3 Job Doctor

We design the job doctor to detect and identify the root causes of the runtime issues that cannot be resolved by reconfigurations

Table 1: Job Doctor diagnosis rules for varying runtime issues.

Category	Runtime Issues
Resource Usage Analysis	JM/TM CPU usage
	JM/TM Memory usage
Failover Analysis	All Failovers e.g., OOM
Flink Configuration Analysis	JVM Frequent Full GC
	Skewed State Distribution
	Inappropriate State Backend Configuration
Data Processing Analysis	Processing Lag Size Exceeded
	Backpressured

on Flink Runtime. It provides a web dashboard for streaming jobs to request diagnosis of runtime issues and receive insights into their root causes. Instead of applying reconfiguration to resolve the runtime issues dynamically, the job doctor fires alarms to users with the diagnosis results for the runtime issues. The runtime issues are to be resolved from the users' side manually. Job doctor is especially useful when the associated reconfiguration mechanisms for resolution are not supported. For example, resolving data skewness mainly involves updating the workload distribution among parallel tasks. The modification of the workload distribution further requires updating the source code of a streaming job.

Diagnosis rules. Job doctor is generally a combination of diagnosis rules. Those diagnosis rules are summarized from expert knowledge and experiences. Table 1 summarizes representative four categories of runtime diagnosis rules for associated issues, including Resource Usage Analysis, Failover Analysis, Memory Analysis, and Data Processing Analysis. 1) *Resource Usage Analysis* suggests a better CPU/Memory configuration for JobManagers and TaskManagers of the Flink cluster according to the average and peak usage of resources. 2) *Failover Analysis* handles exceptions by suggesting pre-defined exception handling rules such as increasing memory size to resolve OOM exceptions. 3) *Flink Configuration Analysis* monitors Flink configuration related issues. For example, the job doctor can suggest state backend configurations in Flink according to the state size for better state access performance. 4) *Data Processing Analysis* detects and diagnoses data processing issues such as processing lags and backpressure, which may be caused by data skewness. Unlike the straggler detector, data skewness results in tasks with a higher load having a higher processing rate, and based on this condition, the job doctor can differentiate between the diagnosis of stragglers and data skewness.

Resolution. To help resolve the runtime issues, the job doctor notifies users with the insights (i.e., the diagnosis results) of the runtime issues and recommends resolution guidance that may potentially resolve the problem. For example, the job doctor notifies users with the current workload distribution and the resource utilization among parallel tasks. Subsequently, users can reconfigure the key partitioning strategy of their streaming job to resolve the data skewness.

6 EXPERIMENTAL EVALUATION

This section reports the performance evaluation results of *StreamOps* from the in-production environment in ByteDance. First, we evaluate the overall scalability of *StreamOps* for managing cluster-wide streaming jobs. Second, we evaluate the effectiveness of each control policy for resolving the runtime issues under the associated scenarios.

6.1 Experiment Setup

We measure the overview performance of *StreamOps* on one of our in-production clusters. The cluster runs thousands of streaming jobs for varying online services. We measure the overall CPU and Memory consumption of the cluster over 24 hours before integrating with *StreamOps*, which is shown in Figure 7. Figure 7a reveals that nearly 75% of streaming jobs have relatively low CPU consumption, i.e., under 50%. Figure 7b reveals that nearly 80% of streaming jobs have a relatively healthy memory consumption i.e. the memory usage is between 50% $\sim$ 80%. In summary, the detailed resource usage indicates that there is a large resource optimization space for the in-production cluster in ByteDance. In this experiment, we show that *StreamOps* scales well and is able to increase the overall resource utilization and reduce the processing lag runtime issues effectively.

6.2 Performance Overview

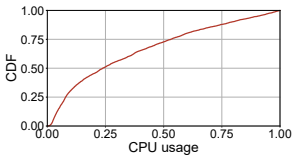
We first evaluate the overall scalability of *StreamOps* over our in-production environment. We deployed *StreamOps* with 50 instances over the cluster. Each instance is configured with 16 CPU cores and 32GB memory. To guarantee stability, most streaming jobs have been configured with an in-frequent runtime management trigger, e.g., 4 times per day by default. While streaming jobs may customize their own trigger for better runtime management.

We measure the runtime management requests received by *StreamOps* in 1 day and summarized the statistics of the number of requests received and the time spent to respond. Figure 8 shows the overall scalability of the *StreamOps*. Specifically, Figure 8a shows the CDF of the requests per second received by *StreamOps*. Figure 8b shows the CDF of 95% response latency per second for all requests handled by *StreamOps*. Based on the in-production performance results, we have the following observations: In general, *StreamOps* receives up to 33k requests per second over the entire cluster, and it can apply runtime management within 60 seconds for 99% of runtime management requests. This confirms the good scalability of *StreamOps* because of the lightweight and stateless system architecture.

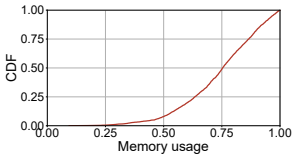
6.3 Auto-Scaler in Action

In this section, we demonstrate the effectiveness of the auto-scaler. We select a large-scale in-production streaming job to evaluate the auto-scaler. The streaming job contains a single operator to collect and analyze the runtime statistics of online web services. Figure 9a shows the input rate of the streaming job over 2 days. Specifically, the input rate of the streaming job periodically increases to nearly 3 million events per second and decreases to nearly 800k events per second. The input stream has 3000 partitions. In addition, the initial parallelism for the streaming job has been set to 750 by default, which is 25% of the number of partitions.

Figure 9a shows changes in parallelism and the input rate over time. Based on the results, we have the following observations. 1) Although the initial parallelism of the streaming job is able to avoid processing lags, it wastes a large number of resources if auto-scaling is not enabled. Specifically, the detailed CPU consumption metrics indicate that the overall CPU utilization is 26%. 2) Auto-scaler makes in-time scaling decisions according to the changes in

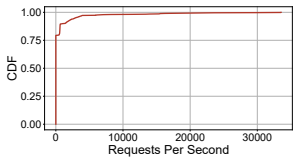


(a)

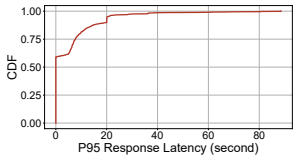


(b)

Figure 7: CPU and Memory usage of streaming jobs in selected ByteDance cluster.

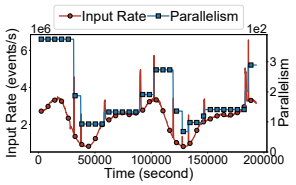


(a) CDF for Requests per Second.

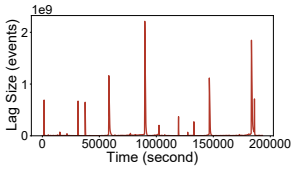


(b) CDF for P95 Response Latency.

Figure 8: *StreamOps* scalability overview.

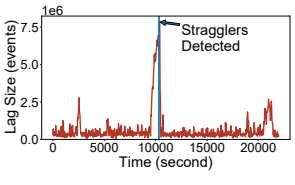


(a) Input Rate and Parallelism.

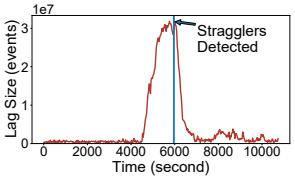


(b) Processing Lag.

Figure 9: Auto-scaler for real scenario in ByteDance.



(a) DTS_Job_1.



(b) DTS_Job_2.

Figure 10: Straggler Detector for real scenario in ByteDance.

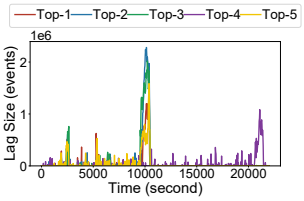
the input workloads. It can scale in the parallelism to 200 during trough hours while scaling the parallelism out to 600 during peak hours. The allocated resources can be adjusted proportionally. On average, auto-scaling is able to save up to 60% resources during peak hours and 87% during trough hours. 3) The scaling execution may cause an input rate spike due to blocked input data during scaling. When the streaming job is updated, the blocked input data is consumed in a short time, causing short input rate spikes. This highlights the need for careful tuning of the scaling model to balance dynamic scaling costs with sensitivity to workload changes.

Figure 9b shows the detailed changes in lag size over time. Based on the results, we have the following observations. 1) The overall lag size remains consistently low, with almost 0 lag in 99.4% of the entire time period. This confirms that auto-scaler is able to make better use of resources while mitigating the processing lag for large-scale streaming jobs. 2) However, multiple processing lag spikes are also observed. As aforementioned, this is mainly caused by the inevitable scaling execution overhead. We leave the exploration of more efficient scaling execution mechanisms in cloud-native environments as our future work.

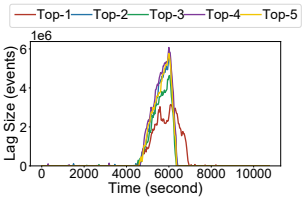
6.4 Straggler Detector in Action

In this section, we demonstrate the effectiveness of the straggler detector. According to our in-production statistics, the straggler detector is able to detect nearly 300 streaming jobs with stragglers per day. It can resolve the processing lag issue for them effectively within 10 minutes. This confirms that the straggler detector can reduce runtime management costs and handle straggler issues automatically. To understand the detailed execution behavior of the straggler detector, we have sampled two representative straggler cases from our in-production environment.

We selected two streaming jobs named *DTS_Job_1* and *DTS_Job_2*. Both streaming jobs are data transmission services, i.e., to read the high volume input stream from Kafka and apply data cleaning correspondingly. Initially, they are deployed with 2000 parallel tasks and 400 TaskManagers. Each of them has an input stream with 2000 partitions. Each parallel task is allocated with a partition of the input stream for parallel stream processing.



(a) DTS_Job_1.



(b) DTS_Job_2.

Figure 11: The processing lag size breakdown.

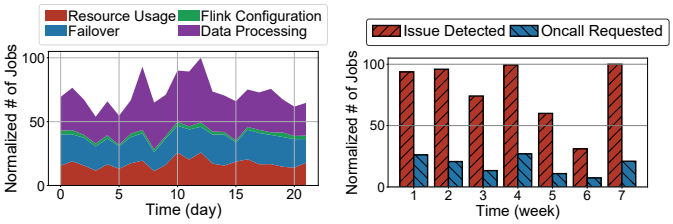
In both scenarios, the stragglers are mainly caused by lacking CPU resources on the overloaded physical nodes.

Figure 10 reports the changes in the processing lag for each streaming job over a period of time. Based on the results, we have the following observations. 1) Both streaming jobs have low processing lag, i.e., under 200k which can be processed within a few minutes, during normal data processing. This indicates that they are allocated plenty of physical resources for data processing. 2) The processing lag of the *DTS_Job_1* started to increase from time 9500 seconds, while the processing lag of the *DTS_Job_1* started from time 4000 seconds. 3) The processing lag decreased after the straggler issue was detected and resolved.

To further confirm the effectiveness of the straggler detection, we show the detailed processing lag size breakdown in Figure 11. In particular, we sampled the top 5 partitions with the largest lag size over the period. The detailed processing lag results have shown that over 80% of the increased processing lag is accumulated in those partitions. This indicates that parallel tasks assigned with those partitions are stragglers. They cannot keep up with the input rate of those partitions. By further analyzing the allocated physical nodes for those stragglers, we observe that those nodes are overloaded because of resource sharing among multiple jobs. The straggler detector is able to identify stragglers and resolve the issue effectively by migrating the associated TaskManagers out to other nodes.

6.5 Job Doctor in Action

In this section, we demonstrate the effectiveness of job doctor.



(a) Diagnosis statistics. (b) In-production user experiences.

Figure 12: Job Doctor for real scenarios.

The job doctor is widely utilized, with thousands of runtime management requests triggered via the dashboard per day. Figure 12a shows the statistics over 7 days, summarizing the normalized number of streaming jobs with successfully diagnosed runtime issues based on four categories of rules. Based on the detailed results, we have the following observations: 1) The Flink configuration issues are the least frequent. This indicates that based on the development and management of Flink streaming jobs in ByteDance over the years, we have accumulated massive experience in setting appropriate Flink configurations for varying streaming jobs. 2) However, data processing, failover, and resource usage issues still commonly exist in ByteDance cloud. As a result, *StreamOps* is important in ByteDance's scenario to help resolve those runtime issues effectively.

Previously, when runtime issues were detected, users had to request oncalls and rely on Flink experts to manually diagnose root causes. With the introduction of the job doctor, users can now diagnose root causes for runtime issues, reducing the need for oncalls. Figure 12b shows statistics from a 7-week period, summarizing the normalized number of streaming jobs with detected runtime issues and the normalized number of jobs with oncalls requested after using the job doctor. Based on the number of oncall per week, we confirm that the job doctor can help users resolve runtime issues based on the four categories of diagnosis rules. In particular, the actual number of oncalls is up to 4.5x lower than the number of runtime issues incurred daily.

7 RELATED WORK

In the past decade, a lot of stream processing engines have emerged in both academia and industry [1–5, 18, 22, 28, 32, 34–36, 40–43]. The runtime management of streaming services over SPEs has been widely discussed in both academia and industry. Specifically, the runtime management topics can be generally divided into three categories: 1) *Control plane*, 2) *Control policies*, and 3) *Dynamic reconfiguration optimization*.

Control Plane. Existing research prototypes [30, 31, 33] propose control planes to manage the runtime of streaming jobs with different design goals. Chi [30] and Trisk [31] focus on extensibility and efficiency. They both expose programming models for designing varying control policies, and enable efficient reconfiguration based on fine-grained atomic task-level operations in the SPE runtime. Turbine [33] provides cluster-wide runtime management for streaming jobs in Meta and decouples the runtime management into job management, task management, and resource management for effectiveness. The design of *StreamOps* draws significant inspiration from the prior work Trisk [31]. *StreamOps* adopts the concept of a standalone control plane with modularized functionalities, as seen in Trisk. However, *StreamOps* has different design decisions and goals compared to existing works. Specifically, *StreamOps* incorporates multiple designs to achieve effective runtime management for cluster-wide streaming jobs in ByteDance's scenario.

Control Policies. Control policies play a crucial role in optimizing the runtime performance of streaming jobs. Prior research prototypes [9, 16, 17, 23, 24, 29] have proposed different control policies to dynamically reconfigure streaming jobs with various performance objectives. DS2 [23], DRS [17], Nephele [29], and Dhalion [16] focus on making dynamic scaling to maximize throughput or guarantee latency. Henge [24] aims to achieve SLO/SLAs while maximizing the overall system utilization through automata-based cluster-wide resource management. In contrast, *StreamOps* focuses on the general runtime management of streaming jobs over ByteDance cloud. *StreamOps* is able to integrate novel control policies for effective runtime management.

Reconfiguration Optimization. The reconfiguration of streaming jobs introduces inevitable downtime. Existing work [9, 11, 19, 20, 38] propose optimized mechanisms for efficient dynamic reconfiguration. Seep [9] proposes a partial-pause-and-resume reconfiguration execution mechanism to partially update the streaming pipeline without affecting the rest of the parallel tasks. Rhino [11] and Clonos [38] mainly focus on optimizing the state management framework by proactive state replication to enable fast state migration and recovery. Megaphone [20] and Meces [19] propose fine-grained state migration scheduling strategies to reduce the latency spike incurred by transferring state remotely. Due to the complicated streaming scenario in ByteDance, enabling fine-grained reconfiguration optimization requires large-scale refactorization on the codebase and potentially affects the stability of services. We leave the investigation of the efficient reconfiguration in ByteDance cloud as a future work.

8 CONCLUSION

In conclusion, we present *StreamOps*, a standalone lightweight control plane to manage cluster-wide streaming jobs in ByteDance cloud. We show that *StreamOps* is scalable to handle tens of thousands of runtime management requests from streaming jobs within one minute. Streaming jobs can also customize their runtime management by tuning the control settings of different control policies. Our carefully designed auto-scaler, straggler detector, and job doctor are also able to handle processing lag and common runtime failure issues effectively.

9 ACKNOWLEDGEMENT

We would like to thank all anonymous reviewers for reviewing the paper and insightful comments. We also would like to thank all those who contributed to *StreamOps* including Yuming Liang, Pengwei Zhao, Zhiwen Li, Ying Zhao, Clarence Castillo, Yan Le, Yung Che Li, Yuan Huang, Weihua Hu, and Chong Liu. This research was supported in part by the TikTok Research Grant A-8000409-00-00.

REFERENCES

[1] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, et al. 2005. The design of the Borealis stream processing engine.. In *CIDR*. 277–289.

[2] Daniel J Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. 2003. Aurora: a new model and architecture for data stream management. *The VLDB Journal* 12, 2 (2003), 120–139.

[3] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. 2013. MillWheel: fault-tolerant stream processing at Internet scale. *Proceedings of the VLDB Endowment* 6, 11 (2013), 1033–1044.

[4] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, et al. 2015. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1792–1803.

[5] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, Felix Naumann, Mathias Peters, Astrid Rheinländer, Matthias J. Sax, Sebastian Schelter, Mareike Höger, Kostas Tzoumas, and Daniel Warneke. 2014. The Stratosphere Platform for Big Data Analytics. *The VLDB Journal* 23, 6 (2014), 939–964.

[6] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. 2016. Borg, omega, and kubernetes. *Commun. ACM* 59, 5 (2016), 50–57.

[7] Sergey Bykov, Alan Geller, Gabriel Kliot, James R Larus, Ravi Pandya, and Jorgen Thelin. 2011. Orleans: cloud computing for everyone. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*. 1–14.

[8] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36, 4 (2015). http://flink.apache.org/

[9] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. 2013. Integrating scale out and fault tolerance in stream processing using operator state management. In *Proceedings of the 2013 ACM SIGMOD international conference on Management of data*. 725–736.

[10] Guoqiang Jerry Chen, Janet L Wiener, Shridhar Iyer, Anshul Jaiswal, Ran Lei, Nikhil Simha, Wei Wang, Kevin Wilfong, Tim Williamson, and Serhat Yilmaz. 2016. Realtime data processing at facebook. In *Proceedings of the 2016 International Conference on Management of Data*. 1087–1098.

[11] Bonaventura Del Monte, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2020. Rhino: Efficient management of very large distributed state for stream processing engines. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2471–2486.

[12] Christoph Fehling, Frank Leymann, Ralph Retter, Walter Schupeck, and Peter Arbitter. 2014. *Cloud computing patterns: fundamentals to design, build, and manage cloud applications*. Vol. 545. Springer.

[13] Apache Flink. (2021). *Flink Forward Asia 2021*. https://flink-forward.org.cn/

[14] Apache Flink. (2022). *AutoScaling model proposal in Apache Flink*. https://cwiki.apache.org/confluence/display/FLINK/FLIP-271%3A+Autoscaling

[15] Apache Flink. (2022). *Elastic scaling execution in Apache Flink*. https://nightlies.apache.org/flink/flink-docs-master/docs/deployment/elastic_scaling/

[16] Avrilia Floratou, Ashvin Agrawal, Bill Graham, Sriram Rao, and Karthik Ramasamy. 2017. Dhalion: self-regulating stream processing in heron. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1825–1836.

[17] Tom ZJ Fu, Jianbing Ding, Richard TB Ma, Marianne Winslett, Yin Yang, and Zhenjie Zhang. 2015. DRS: dynamic resource scheduling for real-time analytics over fast streams. In *2015 IEEE 35th International Conference on Distributed Computing Systems*. IEEE, 411–420.

[18] Jon Gjengset, Malte Schwarzkopf, Jonathan Behrens, Lara Timbó Araújo, Martin Ek, Eddie Kohler, M. Frans Kaashoek, and Robert Morris. 2018. Noria: dynamic, partially-stateful data-flow for high-performance web applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 213–231.

[19] Rong Gu, Han Yin, Weichang Zhong, Chunfeng Yuan, and Yihua Huang. 2022. Meces: Latency-efficient Rescaling via Prioritized State Migration for Stateful Distributed Stream Processing Systems. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 539–556.

[20] Moritz Hoffmann, Andrea Lattuada, Frank McSherry, Vasiliki Kalavri, John Liagouris, and Timothy Roscoe. 2019. Megaphone: Latency-conscious state migration for distributed streaming dataflows. *Proceedings of the VLDB Endowment* 12, 9 (2019), 1002–1015.

[21] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. 2010. ZooKeeper: wait-free coordination for internet-scale systems.. In *USENIX annual technical conference*, Vol. 8.

[22] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS operating systems review*. ACM, 59–72.

[23] Vasiliki Kalavri, John Liagouris, Moritz Hoffmann, Desislava Dimitrova, Matthew Forshaw, and Timothy Roscoe. 2018. Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 783–798.

[24] Faria Kalim, Le Xu, Sharanya Bathey, Richa Meherwal, and Indranil Gupta. 2018. Henge: Intent-driven multi-tenant stream processing. In *Proceedings of the ACM Symposium on Cloud Computing*. 249–262.

[25] Jay Kreps, Neha Narkhede, Jun Rao, et al. 2011. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, Vol. 11. Athens, Greece, 1–7.

[26] Butler W Lampson and Howard E Sturgis. 1976. Reflections on an operating system design. *Commun. ACM* 19, 5 (1976), 251–265.

[27] Sangmin Lee, Zhenhua Guo, Omer Sunercan, Jun Ying, Thawan Kooburat, Suryadeep Biswal, Jun Chen, Kun Huang, Yatpang Cheung, Yiding Zhou, et al. 2021. Shard manager: A generic shard management framework for geo-distributed applications. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 553–569.

[28] Wei Lin, Zhengping Qian, Junwei Xu, Sen Yang, Jingren Zhou, and Lidong Zhou. 2016. Streamscope: continuous reliable distributed processing of big data streams. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. 439–453.

[29] B. Lohrmann, P. Janacik, and O. Kao. 2015. Elastic Stream Processing with Latency Guarantees. In *IEEE 35th International Conference on Distributed Computing Systems*. 399–410. https://doi.org/10.1109/ICDCS.2015.48

[30] Luo Mai, Kai Zeng, Rahul Potharaju, Le Xu, Steve Suh, Shivaram Venkataraman, Paolo Costa, Terry Kim, Saravanan Muthukrishnan, Vamsi Kuppa, et al. 2018. Chi: A scalable and programmable control plane for distributed stream processing systems. *Proceedings of the VLDB Endowment* 11, 10 (2018), 1303–1316.

[31] Yancan Mao, Yuan Huang, Runxin Tian, Xin Wang, and Richard TB Ma. 2021. Trisk: Task-Centric Data Stream Reconfiguration. In *Proceedings of the ACM Symposium on Cloud Computing*. 214–228.

[32] Yancan Mao, Jianjun Zhao, Shuhao Zhang, Haikun Liu, and Volker Markl. 2022. MorphStream: Adaptive Scheduling for Scalable Transactional Stream Processing on Multicores. (2022).

[33] Yuan Mei, Luwei Cheng, Vanish Talwar, Michael Y Levin, Gabriela Jacques-Silva, Nikhil Simha, Anirban Banerjee, Brian Smith, Tim Williamson, Serhat Yilmaz, et al. 2020. Turbine: Facebook's service management platform for stream processing. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1591–1602.

[34] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 439–455.

[35] Derek G Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. 2011. CIEL: a universal execution engine for distributed data-flow computing. In *Proc. 8th ACM/USENIX Symposium on Networked Systems Design and Implementation*. 113–126.

[36] Zhengping Qian, Yong He, Chunzhi Su, Zhuojie Wu, Hongyu Zhu, Taizhi Zhang, Lidong Zhou, Yuan Yu, and Zheng Zhang. 2013. Timestream: Reliable stream computation in the cloud. In *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM, 1–14.

[37] Peter J Rousseeuw and Mia Hubert. 2011. Robust statistics for outlier detection. *Wiley interdisciplinary reviews: Data mining and knowledge discovery* 1, 1 (2011), 73–79.

[38] Pedro F Silvestre, Marios Fragkoulis, Diomidis Spinellis, and Asterios Katsifodimos. 2021. Clonos: Consistent causal recovery for highly-available streaming dataflows. In *Proceedings of the 2021 International Conference on Management of Data*. 1637–1650.

[39] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. 2009. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment* 2, 2 (2009), 1626–1629.

[40] Guozhang Wang, Lei Chen, Ayusman Dikshit, Jason Gustafson, Boyang Chen, Matthias J Sax, John Roesler, Sophie Blee-Goldman, Bruno Cadonna, Apurva Mehta, et al. 2021. Consistency and completeness: Rethinking distributed stream processing in apache kafka. In *Proceedings of the 2021 international conference on management of data*. 2602–2613.

[41] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2–2.

[42] Shuhao Zhang, Jiong He, Amelie Chi Zhou, and Bingsheng He. 2019. Briskstream: Scaling data stream processing on shared-memory multicore architectures. In *Proceedings of the 2019 International Conference on Management of Data*. 705–722.

[43] Yunhao Zhang, Rong Chen, and Haibo Chen. 2017. Sub-millisecond stateful stream querying over fast-evolving linked data. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles*. 614–630.