Storm-RTS: Stream Processing with Stable Performance for Multi-cloud and Cloud-edge

Hai Duc Nguyen
University of Chicago
ndhai@cs.uchicago.edu

Andrew A. Chien
University of Chicago and
Argonne National Laboratory
achien@cs.uchicago.edu

2023 IEEE 16th International Conference on Cloud Computing (CLOUD) | 979-8-3503-0481-7/23/$31.00 ©2023 IEEE | DOI: 10.1109/CLOUD60044.2023.00015

***Abstract*—Stream Processing Engines (SPEs) traditionally deploy applications on a set of shared workers (e.g., threads, processes, or containers) requiring complex performance management by SPEs and application developers. We explore a new approach that replaces workers with Rate-based Abstract Machines (RBAMs). This allows SPEs to translate stream operations into FaaS invocations, and exploit guaranteed invocation rates to manage performance. This approach enables SPE applications to achieve transparent and predictable performance.**

We realize the approach in the Storm-RTS system. Exploring 36 stream processing scenarios over 5 different hardware configurations, we demonstrate several key advantages. First, Storm-RTS provides stable application performance and can enable flexible reconfiguration across cloud resource configurations. Second, SPEs built on RBAM can be resource-efficient and scalable. Finally, Storm-RTS allows the stream-processing paradigm to be extended from the cloud to the edge, using its performance stability to hide edge heterogeneity and resource competition. An experiment with 4 cloud and edge sites over 300 cores shows how Storm-RTS can support flexible reconfiguration and simple high-level declarative policies that optimize resource cost or other criteria.

***Index Terms*—Stream Processing, Serverless, FaaS, Real-time, Cloud Computing, Edge Computing**

I. Introduction

Recent years have seen increasing use of distributed stream processing engines (SPE). These vary from generic engines (e.g., Storm [1], Flink [2]) to optimized systems designed for specific deployments (e.g., Samza [3] and Turbine [4]). SPEs employ a stream processing model treating data as a stream of *tuples* and formulate analysis as a *workflow* – a directed acyclic graph of *operators*. Eager, data-driven processing provides low latency while parallel operation execution enables high-throughput [5], [6]. The resulting capabilities make stream processing an important paradigm for data analysis at scales from smart homes [7] to large-scale industries [8].

Most modern SPEs [1]–[3], [9]–[15] use the *worker model* for workflow deployment. In this model, the SPE maps operators onto workers (e.g., threads, processes, containers, etc.) that serve as a common abstraction for underlying compute resources. These workers are exposed as the basic performance abstraction to workflow developers, who can configure each operator to have one or more workers depending on its computational intensity to meet the workflow demand. However, changes in the execution environment or the underlying worker scheduling can disturb worker performance. For instance, collocated with an aggressive application could interfere with worker processing, reducing its throughput. Consequently, workflow performance tuning is a process of trial-and-error, adjusting worker configuration until reaching desired performance [5], [16]. Perhaps worse, the tuning produces a single configuration with little insight into how to adapt it as the load evolves. This lack of *performance transparency and predictability* is a challenge for SPE application developers.

We propose a new approach that solves these problems: hosting the SPE on a new abstraction called the rate-based abstract machine (RBAM). The RBAM model augments FaaS functions with guaranteed invocation rates. We show how SPE systems can exploit this foundation to achieve stable performance by mapping workflow operators into FaaS functions, accruing the benefits of performance transparency and portability. The RBAM approach exploits the FaaS abstraction interface, similar to other innovative works [2], [17]–[21], but differs in the critical aspect of providing guaranteed performance.

The Storm-RTS stream-processing engine realizes SPE on RBAM. Storm-RTS maps operator executions to FaaS invocations allocated at a rate guaranteed by RBAM to deliver efficient, scalable, and flexible stream processing. We describe Storm-RTS' design, implementation and compare it to several modern SPEs. Storm-RTS matches the resource efficiency of state-of-the-art worker-based SPEs while enabling easy reconfiguration with clouds, or across the cloud and edge. We illustrate how the performance transparency and predictability of Storm-RTS enable myriad opportunities such as declarative resource management to improve cost, reliability, and more. Finally, Storm-RTS provides scalability, enabling a workflow to easily exploit additional resources when pressed with an increased load without any redesign or reconfiguration. Contributions of the paper include:

- Describe how to translate stream processing applications into FaaS invocations with rate guarantee (RBAM) and achieve stable performance.
- Design and implementation of Storm-RTS, an SPE that realizes these ideas, replacing the worker abstraction with FaaS/RBAM to provide performance stability (and transparency and predictability) that enable both configuration flexibility and high-level declarative performance and configuration management.
- Evaluation of Storm-RTS, compared to state-of-the-art

2159-6190/23/$31.00 ©2023 IEEE
DOI 10.1109/CLOUD60044.2023.00015

SPEs, demonstrating stable performance, as well as its resource efficiency (comparable to worker-based SPEs), and dynamic scalability.
- Demonstration of several simple declarative approaches that exploit the flexible cloud-edge application configuration enabled by Storm-RTS' stable performance. The performance modularity enabled by the SPE on RBAM creates opportunities to optimize for other high-level objectives (e.g., cost, carbon footprint, etc.) in a fashion that is orthogonal to traditional SPE throughput.

The rest of the paper is organized as follows. Section II provides background in Stream Processing and FaaS computing. In Section III, we describe the worker model's drawbacks and then propose a new approach using the RBAM abstraction to address these issues in Section IV. Section V proposes Storm-RTS, an implementation for SPE on RBAM. We evaluate Storm-RTS in Section VI. Finally, we survey related work in Section VII and summarize the paper in Section VIII.

II. Background

1) Stream Processing.: The stream processing model enables performing real-time analytical tasks efficiently and scalably. The model treats input streams as flows of separate *tuples* and organizes applications as Directed Acyclic Graphs (DAGs) called *workflow* consisting of *operators* placed on a set of distributed computing nodes. Immediately after creation, tuples are taken through the workflow, and processed by their operators in an on-the-fly fashion, delivering analytical results with low latency. Also, each operator can have multiple copies running concurrently to exploit the hardware parallelism capability, easing high-throughput computation.

As such, many Stream Processing Engines (SPEs) have been proposed to automate workflow description, deployments, and operation with efficiency. Many of them are pure, general SPEs and act as a building block for larger data analysis systems [16]. Meanwhile, others are customized for specific infrastructures [22], applications [23], or workloads [3], [24].

2) Worker-based SPEs: Modern SPEs deploy stream processing workflows by mapping operators onto *workers* – a computation abstraction provided by the underlying resource manager for efficient hardware exploitation. Popular choices of worker abstraction are threads, processes, and containers. With all computation handled by operators, operator-to-worker mapping is crucial to workflow performance. Figure 1 shows how SPEs typically have it done. To deal with varied operator complexity, SPE assigns to each operator a *parallelism configuration* which is essentially the number of the operator's copies that can execute concurrently. SPE allocates a corresponding number of workers, each to run an operator copy, and distributes them across its cloud resources. For example, in Figure 1, O_2 and O_3 are compute-intensive operators so have their parallelism set to 2, resulting in two copies and getting two workers while O_1 and O_3 only have 1. This configuration creates 6 operators which need an allocation of 6 workers distributed over 2 machines. One hosts O_1 and O_2, and another hosts O_3 and O_4.

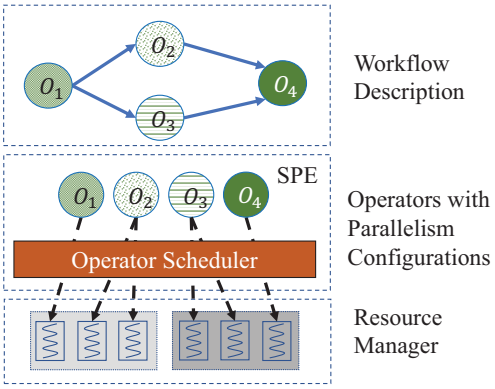


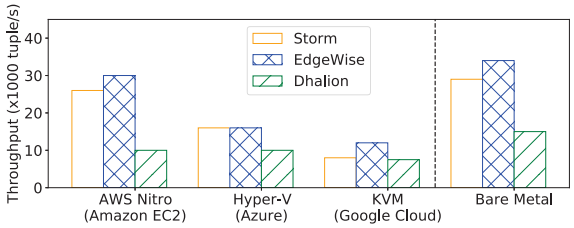
Fig. 1: Worker-based SPE Architecture. Operators are mapped onto workers across multiple machines. The parallelism configuration specifies high-cost operators mapped onto multiple workers for efficiency.

3) FaaS Computing.: Serverless or Function-as-a-Service (FaaS) is a resource abstraction that lets applications exploit the underlying resources through *invocations*. An invocation is a discrete execution unit limited in time and resource use (e.g., timeout, CPU, and memory). Applications associate invocations with their logic in the form of *stateless functions*. Each function is a specific task (e.g., resizing an image) with a unique identifier (usually an URL). A function is called (or invoked) by sending a request (e.g., HTTP Post), along with required arguments (e.g., a file content embedded inside the request body), to this identifier. The request is handled by a *FaaS platform* (e.g., AWS Lambda [25], Azure Function [26], Google Cloud Function [27], etc.) that allocates resources to launch an invocation that executes the function logic on the given arguments to complete the task. FaaS platforms can automatically scale up to thousands of invocations simultaneously in response to workload dynamics. This ability enables dynamic scalability with minimum efforts and costs opening great opportunities to implement cost-effective, scalable solutions [28], [29]. However, FaaS allocation is best-effort, executing a new invocation in response to a request may get delayed or even fails if the FaaS platform cannot find available resources resulting in performance instability.

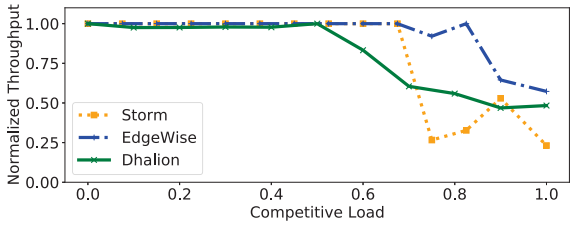
III. Problem: Limits of Worker-based SPEs

A. Performance Challenges

a) Performance Transparency Challenge.: Worker-based SPEs tie workflow performance to underlying worker resource configuration. Because these details are not part of the application abstraction, performance is not transparent. Figure 2a shows the maximum throughput of executing an ETL workflow on a 4-core machine deployed by three different worker-based SPEs: Storm [1], EdgeWise [30], and Dhalion [31] (see Section VI). We try four machine configurations (Section VI-A): one is bare metal while the others are VMs provisioned by different hypervisors while sticking with only one parallelism configuration (Figure 6a). The throughput is



(a) Throughput vs. Resource Configuration



(b) Throughput vs. Competitive Load

Fig. 2: Performance challenges in worker-based SPEs: (a) poor transparency as throughput greatly varies across different resource configurations, and (b) poor predictability as throughput is interfered with a competitive load.

extremely sensitive to resource configurations, with performance varying as much as 3-fold. For example, Storm running on KVM gets only 27% of bare metal throughput.

The results illustrate that workflow performance is not transparent yet strongly depends on the hardware resource configurations to which their workers have access. Hence, there would be no one-size-fits-all workflow configuration that can be used for every deployment. Instead, the SPE has to understand the underlying resource configurations and reconfigure workflow accordingly to maintain good performance.

b) Performance Predictability Challenge.: Most of the current distributed systems such as the cloud and edge are shared environments. Workers are typically collocated with other applications. Actual resources allocated to workers highly vary subject to these applications' behaviors. Tied to the transparency challenge, this further means that performance for SPE applications is almost not predictable. In Figure 2b, we plot the throughput of an ETL workflow normalized by its input rate when collocated with an aggressive competitive load on a single Azure VM. We slowly increase the computation demand of the competitive load, from no load until its computation demand is high enough to consume 100% CPU on the machine. We notice a significant drop in ETL's throughput after the competitive load exceeds 70%.

The results illustrate workflow performance is tied to its collocated applications. As a result, workflow performance is hard to predict. One deployment that works well may become ineffective when some surrounding applications change. Unfortunately, these changes are typically out of SPEs' control making performance predictability a challenge for them.

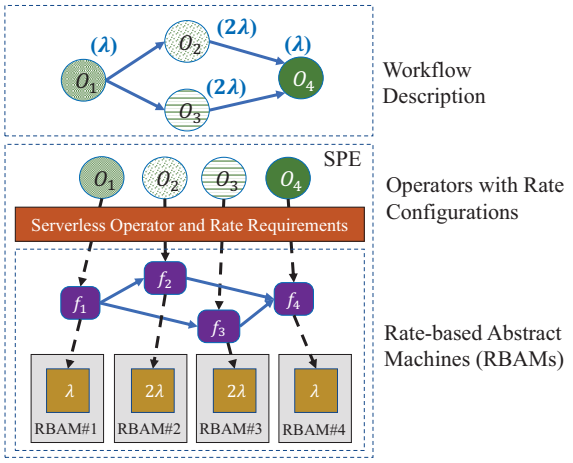


Fig. 3: The RBAM approach to stream-processing: Operators are wrapped by FaaS functions, providing invocation-level dynamic resource management. One RBAM for each FaaS function ensures its required tuple processing rate.

B. Implications for Applications

Application developers compensate for poor **performance transparency and predictability** by over-provisioning, wasting resources. A better approach is to repeatedly reconfigure workflow for any significant change to the environment or application until performance meets the desired level [5], [16], [31]. However, this only works if the SPE reacts properly to the change. Failing to select an appropriate configuration would result in multiple rounds of reconfiguration causing performance instability or over-provisioning.

Another implication of poor performance transparency and predictability is difficulty in changing workflow configuration (e.g., migrating workers from one machine to another). Such application reconfiguration can be desirable to manage cost, adjust to load dynamically, or move to other resources in response to outage, preemption, or perhaps power cost. For these reasons, most SPEs do not even support multi-site execution. For example, the design of a workflow deployment that spans two data centers or datacenter and the edge is a bespoke, manual activity [3], [24].

More directly, the above challenges make deploying a workflow over multiple data centers tricky; many manual efforts are required for each configuration. Worse, flexible reconfiguration across cloud and edge – a signature challenge for many applications – is difficult. In the edge's dramatically more complex environment of heterogeneous resources and networks, manual configuration and tuning may be impossible.

IV. Stream Processing on Rate-guaranteed FaaS

We resolve the performance challenges by replacing the worker model with FaaS invocations as shown in Figure 3. Operators are implemented as FaaS functions and the topology is encoded as FaaS chains, with tuples passed as function arguments. For example, operators O_1, O_2, O_3, O_4 becomes

	Worker Model	RBAM
Service Model	Continuous	Discrete (invocation)
Allocation	Static	Dynamic
Guarantee	None	Rate (invocations/sec)

TABLE I: Worker-model and RBAM Comparison

separate FaaS functions f_1, f_2, f_3, and f_4, respectively. The SPE handles each arriving tuple by first invoking f_1. After completion, f_1 triggers f_2 and f_3 with the output embedded inside their invocation requests. These invocations extract f_1's output from the requests, process it, and then pass their results downstream until reaching the sink operator.

Regular FaaS performance is best-effort. Invocation allocation may fail or get delayed, degrading workflow performance. To workaround, we let application developers configure per-operator rate requirements specifying the expected processing rate of these operators after deployment. Once a workflow is submitted, rate requirements are tied to their corresponding FaaS functions, each provisioned in a Rate-based Abstraction Machine (RBAM). As in Figure 3, f_1's rate requirement is λ, equal to the input rate of its operator, O_1. Rate configurations are equivalent to worker-based parallelism configurations yet are easier to determine by measuring input rate and can be guaranteed through RBAM enabling a simpler, straightforward way to specify, configure, and evaluate workflow performance.

A. Rate-based Abstract Machine

Each Rate-based Abstract Machine allocation has an invocation rate guarantee λ. This is a form of service/resource guarantee framed in the FaaS model. Once allocated, the RBAM gives a FaaS function a guarantee to execute *at least* λ *invocations* per second, where each invocation is access to specific resources within a limited time (e.g., 1 CPU for 10 seconds). For example, if an operator is mapped to an RBAM of 10 invocations/sec, then for every 1-second interval, the workflow is *guaranteed* to execute the operator logic at least 10 times. The RBAM abstraction departs from workers in many important ways (Table I):

- *Invocation-level Resource Management*: In contrast to worker abstraction's *continuous* resource access, RBAM lets applications access resources through invocations, a *discrete* notion in time and resources. This model naturally matches the stream processing workload which is also determined by discrete tuple arrivals.
- *Dynamic Allocation*: RBAM scales invocation allocation dynamically to tuple arrival and automatically releases them after finishing processing. This scheme supports both dynamic scaling and low resource waste. In comparison, worker allocation is rather static as one worker often represents a fixed set of resources.
- *Guaranteed-Rate*: while worker allocation offers no guarantee, RBAM allocation supports a guaranteed invocation rate that enables robust, simple QoS reasoning.

Further, RBAM performance is independent of underlying resource configurations due to FaaS invocation recycling and the time limit. For example, to support $\lambda = 1$ invocation/sec where one invocation gets 1 CPU for a maximum of 10 sec, we only need 10 CPU for invocations in the first 10 seconds. After that, allocated invocations terminate due to the time limit and their CPUs can be used for the next 10 seconds, and so on. Therefore, deploying this RBAM is as simple as reserving 10 CPUs. With isolation support (e.g., container), decoupling RBAM performance from underlying systems is straightforward (more in Section V-C).

B. Resolving Performance Challenges

By setting each RBAM rate guarantee to match the operator processing rate, the SPE guarantees the availability of resources to process tuples at the arrival rate, maintaining desired performance. This rate configuration is independent of any underlying resource configuration, so the SPE application has full performance transparency.

RBAMs also support performance predictability: RBAM allocations ensure their operators perform well against any load whose input rate is smaller or equal to the rate guarantee. Consequently, a workflow constructed from these operators also has a performance guaranteed up to a specific input rate. This performance predictability enables simple tuple rate comparisons and negotiation with the underlying RBAM systems to determine if a new configuration is feasible. This framework enables distributed SPE configuration management with stable performance possible (see Section V-D).

V. Storm-RTS: SPE for Distributed Stream Processing

A. Design Requirements

Storm-RTS – a new distributed SPE to translate workflow description into RBAM allocations – achieves performance transparency and predictability. This enables it to flexibly spread stream-processing applications over multiple machines across multi-datacenter from Cloud to Edge. We describe the design of Storm-RTS to demonstrate this new capability.

Storm-RTS is derived from Storm and reuses its workflow models to offer essential features of a modern SPE. However, mapping Storm's workflow model to RBAM abstraction is not straightforward. First, many essential FaaS configurations, such as time limit, cannot be inferred directly from Storm workflow configurations. Second, FaaS functions are highly modular and stateless while Storm, like other worker-based SPEs, collocate workers for efficiency and maintain operators' state for various functionalities, such as consistency and fault recovery. Naively replacing Storm's workflow executor with FaaS invocations would reduce efficiency and leave some features infeasible (e.g., stateful operators). We workaround these issues by meeting the following requirements.

- *Workflow performance stability*: achieve desired throughput and latency across distributed configurations, reconfiguration (migration), and varied competitive loads.
- *Predictable Resource Requirements*: operators and workflows characterized for their resource requirements.
- *Modular resource management*: can partition workflow across multiple sites/data centers. Individual site resource

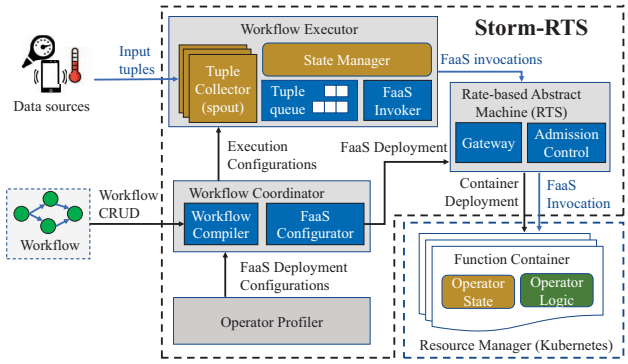


Fig. 4: Storm-RTS Architecture: Operator Profiler, Workflow Coordinator, Executor, and Rate-based Abstract Machine

managers can independently decide if a workflow can be placed and meet its performance requirements.
- *Compatibility*: support Storm workflows and features with similar efficiency and modest change.

B. Storm-RTS Architecture

The key elements of Storm-RTS are shown in Figure 4. At the high level, Storm-RTS has four main components, each is responsible for one of the design requirements listed above.

Workflow Coordinator: responsible for enforcing performance stability. It translates workflow operators received from developers into FaaS functions and associated them with appropriate configurations allowing the workflow to sustain the desired load. It is also responsible for protecting workflow performance from disruptions such as competitive loads, workflow reconfiguration, migration, etc.

Operator Profiler: responsible for resource requirement predictability. The component runs workflow operators offline to profile their computing and memory requirements. This information is used to configure FaaS functions' resource requirements, ensuring their invocations always have sufficient resources to execute their associate operators.

Rate-based Abstract Machine (RBAM): responsible for enabling modular resource management. FaaS functions created by the `workflow coordinator` are deployed separately inside RBAM allocations. Once established, each RBAM allocation ensures new invocations are executed at the configured rate independent of each other, underlying resource configuration, and other competitive loads.

Workflow Executor: responsible for executing workflows and compatibility supports. It collects tuples from data sources and then triggers corresponding FaaS invocations to start workflow execution. The `workflow executor` also reuses Storm's monitor and orchestration modules to offer similar data processing support as Storm.

C. Storm-RTS Implementation

1) Workflow Deployment: Workflow developers submit workflow descriptions directly to the `workflow compiler`. The description includes workflow topology and rate configuration. Rate configuration consists of a *desired rate* λ that developers expect the workflow to handle and per-*operator rate scales* μ_i representing the ratio of each operator's expected input rate and the desired rate. The `workflow compiler` extracts operators' logic from workflow topology and then encapsulates them inside FaaS functions.

Each FaaS function f_i has the `Operator Profile` determine its (i) per-invocation resource requirement s_i (mainly CPU and memory), (ii) time limit $tmax_i$ (i.e., timeout), and (iii) batch size b_i (i.e., number of tuples processed per invocation). This is done by running operators offline with tuples sampled from historical input stream data. The running environment is configured to be identical to the environment targeted to execute workflow operators.

- *Per-invocation resource (CPU and memory) requirement (s_i) and time limit ($tmax_i$)*: the profiler executes operators starting with excess resource allocation and gradually reduces the allocation until observing a 20% execution time increase. This last allocation configuration and the corresponding execution time are used to configure the FaaS wrapping the operator.
- *Batch size* (b_i): Since invocation overhead is typically much higher than tuple processing latency (a couple of milliseconds vs. $<$1ms), Storm-RTS batches multiple tuples in one invocation to amortize the overhead. However, this prolongs per-tuple processing latency. To mitigate this effect, the profiler compares naive operator execution versus FaaS varying the batch size and considers the batch size leading to an efficiency of 70% is acceptable and used to determine the batch size for this operator.

With the information, the `workflow coordinator` configures per-function rate guarantee A_i to the number of invocations expected to invoke per second if tuples are generated at the desired rate λ:

$$A_i = \frac{\lambda \cdot \mu_i}{b_i} \tag{1}$$

This A_i guarantees at least one invocation available for the operator wrapped by the FaaS function to process all incoming tuples sent at any rate less than or equal to λ, thereby satisfying the performance stability requirement. After determining the above information for all FaaS functions, the `FaaS Configurator` sends these functions and their configurations to RBAM to check whether the underlying resource manager can support their guarantee and wrap FaaS functions inside RBAMs with corresponding rate guarantees. If the process completes successfully, the desired rate is guaranteed so the `workflow executor` is triggered to begin execution, no further reconfiguration/profiling is needed.

2) RBAM Implementation: Storm-RTS leverages Real-time Serverless (RTS) [32] to deploy every FaaS functions deployment requested by the `workflow coordinator`. Each FaaS function f_i is initialized with s_i resources – just enough to handle one invocation request, if any. After $1/A_i$ seconds, we allocate s_i more resources for one more invocation. we then wait for $1/A_i$ more seconds, then allocate another s_i

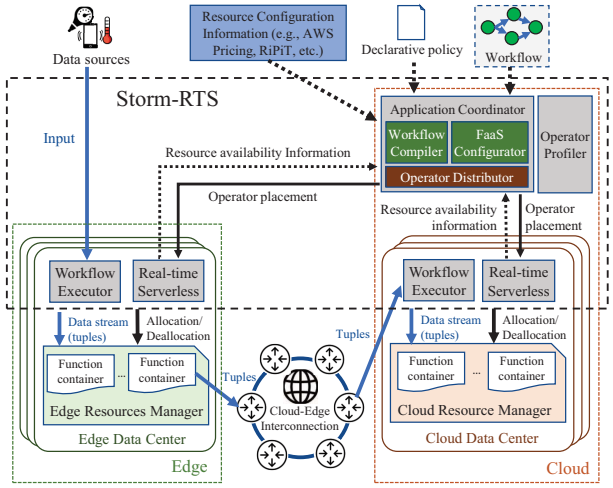


Fig. 5: Storm-RTS for Multi-site Deployment: the application coordinator manages multi-site deployment. As before, each site has admission control and performance monitoring that implements the local RBAM guarantees.

resources, and so on. In this way, for any $1/A_i$ interval, f_i always has resources for a new invocation, meeting the rate guarantee requirement. We repeat this process until $tmax_i$ seconds (i.e., the invocation execution time limit) have passed. By this time, the first s_i resources allocated at the beginning must be available in at most $1/A_i$ seconds and can be recycled for a new invocation. Similarly, resources allocated at the second $1/A_i$ interval can be used in the next $1/A_i$ interval, and so on. In the end, only

$$W_i = A_i \cdot tmax_i \cdot s_i \tag{2}$$

resources are needed to realize the rate guarantee. The RTS system reserves W_i resources for function containers via Kubernetes. If Kubernetes can successfully create these containers then RTS returns the deployment information to the `workflow coordinator`, and the workflow begins processing. Otherwise, RTS considers the deployment failed and cleans up. Note that if the request rate exceeds A_i, RTS provides best-effort service.

3) Workflow Execution and State Management: For each successful workflow deployment, the `workflow executor` creates a set of `tuple collectors` realizing the workflow source operators ("spout" in Storm terminologies) to continuously collect new tuples from data sources. New tuples are put into `tuple queues` by destination until their number is sufficient to form a batch, a `FaaS invoker` retrieves a batch from the queue and requests a new FaaS invocation for the appropriate workflow operator, passing tuples as an argument. Each invocation processes one batch. After completion, to pass on output tuples, the invocation calls the wrapper functions for the operators downstream, passing output tuples in batch as an argument. This allows the downstream function, in response, to extract the tuples, perform the operator computation, and call its downstream operator wrappers as needed, and so on. This mechanism forms tuple processing as FaaS function chains which are self-synchronized and do not need any dedicated messaging systems as in worker-based SPEs (e.g., Storm [1] relies on Netty [33] for inter-node messaging).

Storm-RTS provides equivalent state management and functionalities to Storm, including stateful supports, exactly-once processing, out-of-order events, etc. Since serverless invocations are stateless, we have to modify RTS to embed an in-memory store called `operator state` inside each FaaS container to maintain operator state (e.g., join keys), consistency, progress tracking, monitoring, and recovery. This information is updated every time a tuple completes processing and periodically synchronized with a centric `state manager`. We reuse Storm modules to implement both `operator states` and the `state manager`, ensuring the state information is handled properly and tuples are sent to FaaS containers in correct order and meet users' desired semantics.

D. Multi-site Deployment with Storm-RTS

Distributing workflow execution across multiple sites (e.g., cloud-cloud and cloud-edge) is challenging because distributed resources are both heterogeneous and can vary in availability. The Storm-RTS design brings new capabilities to address both issues. First, Storm-RTS accesses underlying resources through FaaS abstraction, so as long as FaaS are supported by underlying systems, Storm-RTS can mask heterogeneity via FaaS and assures performance via operator profiling and RTS guarantee enforcement. Second, resources with varying availability may require workflow reconfiguration. By leveraging the RBAM, Storm-RTS ensures that such reconfiguration will not affect workflow performance, enabling applications to optimize their deployments for cost, carbon, or other criteria. To illustrate this capability, we extend Storm-RTS architecture as shown in Figure 5. Each cloud or edge data center runs Storm-RTS as in Figure 4, but now the `workflow coordinator` is promoted to `application coordinator` to orchestrate FaaS deployments across the data centers. Apart from the original components, the `application coordinator` adds an `operator distributor` that places the FaaS-encapsulated operators across data centers, implementing the desired application policy.

Common policies include keeping operators close to data sources (often at the edge). If multiple data centers can host an operator, the coordinator implements the application's `deployment policy`, which picks application configurations from among the candidates. For example, if edge resources are zero-cost, when available, a policy that simply minimizes total deployment cost would push operators to the edge when it is idle, and pull them back to the cloud when it is not. If sustainability is the objective, then the `application coordinator` might push operators to the edge when solar panels create plentiful green power, but back to the cloud data center, when the solar panels stop generating sufficient green power. Storm-RTS implements policies by collecting and assessing two sources of information:

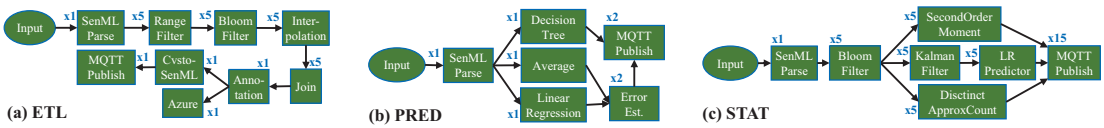


Fig. 6: RIoTBench workflows. Operators shown as green boxes with numbers representing parallelism configurations.

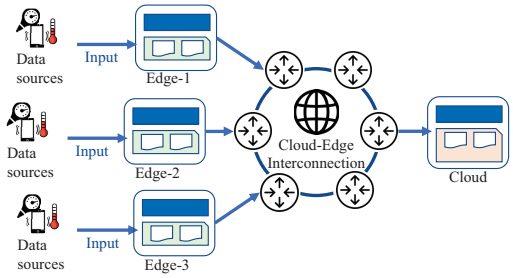


Fig. 7: A Cloud-edge resource configuration

- *Resource configuration* (e.g., resource pricing, Carbon intensity information [34], etc.) to give insights into resource properties for efficient exploitation.
- *Resource availability*: collected from resource managers in data centers. The `application coordinator` also communicates with RTS systems to determine if an operator placement is feasible at any particular site.

VI. Evaluation

A. Methodology

1) Workloads: We use the *RIoTBench* benchmark suite [35], designed specifically for evaluating SPE implementations. We select 3 workflows (Figure 6) capturing common stream processing activities over a real-world smart cities dataset [36]: PRED (make predictions on streamed data), ETL (perform data extraction, transformation, and load), and STATS (apply statistical summarization). Their parallelism configurations are selected based on the number of tuples each operator has to process per one input tuple.

2) Stream-processing Engines (SPEs): We compare five SPEs which are representative implementations of workflow deployment approaches discussed in Section III and IV.

- *Storm* [1]: Evaluation baseline. Workers are implemented as threads in a Java Virtual Machine. Worker allocation and mapping are static.
- *EdgeWise* [30]: a Storm variation that replaces static worker mapping with a dynamic one prioritizing operators experiencing long input queues for higher efficiency.
- *Dhalion* [31] a worker-based SPE with heuristic dynamic scaling. The SPE allocates more resources if workflow throughput fails to match the input rate and frees unused resources if the workflow is over-provisioned.
- *Storm-Serverless* implements the Storm API on FaaS. Its implementation is identical to Storm-RTS, except the RBAM is replaced with OpenFaaS [37], thereby operators have no rate-guarantee.
- *Storm-RTS* implements the Storm API with rate-based abstract machine as described in Section V.

In the following experiments, unless stated otherwise, worker-based SPEs use parallelism configurations shown in Figure 6. Storm-RTS also sets operator scale factors μ_i identical to these parallelism configurations and desired rate λ equal to the workflow input rate.

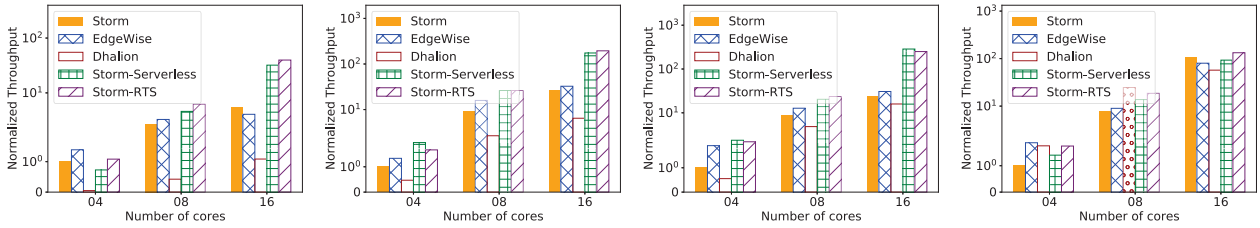
3) Hardware/Resource Configurations: Experiments are conducted over three configurations

- *Cloud VM*: workflows are hosted by virtual machines in public clouds, including Amazon EC2 (m5zn instances), Microsoft Azure (Dasv4 instances), and Google Cloud (e2-standard instances) to evaluate SPE performance over realistic settings where they typically run over a virtual, oversubscribed environment inside data centers.
- *Bare Metal*: for raw performance measurement (no sharing). The machine has 1 Intel Xeon Gold 6138 (80 cores), 512GB RAM and uses *cgroup* for resource control.
- *Cloud-Edge* We create four clusters (Figure 7) where the *cloud* emulates the cloud side with an unlimited number of machines, each has 92 cores and 192GB of memory. *edge1*, *edge2*, and *edge3* represent edge data centers. Each has 4 VMs (12 cores and 48GB memory). We configure the network based on Amazon Cloud Infrastructure's network performance [38]. All connections have 100Gbps bandwidth. Intercloud connections have 5.5ms latency while Cloud-Edge latency is randomized with Gaussian distribution with 5.5ms mean and 2ms standard deviation.

4) Metrics: We evaluate SPEs based on *throughput* (measured at sink operators), end-to-end processing *latency*, CPU utilization (100% per core), and *cost*, measured as CPU utilization * cost-factor. The cost-factor is a dimensionless relative measure of resource cost, reflecting resource location.

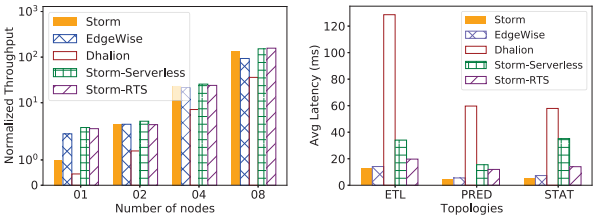
B. Resource Efficiency

1) Single Machine: We deploy RIoTBench workflows separately over a single machine with fixed CPUs (4, 8, and 16 cores). The workflows are fed tuples at a constant rate, and we gradually increase the rate until saturation (i.e., the tuple processing latency increases sharply and the throughput fails to match the tuple input rate). We report the throughput just before this point, calling it the maximum throughput. We plot the geometric mean of the normalized maximum throughputs of three RIoTbench workflows on 4 different machine configurations in Figure 8. The performance of Storm, Edgewise, and Dhalion scale poorly, falling slightly behind Storm-Serverless and Storm-RTS at 8 cores and badly behind at 16 cores. Both Storm-RTS and Storm-Serverless scale well



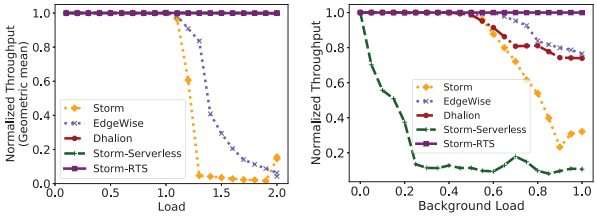
(a) AWS Nitro (Amazon EC2) (b) Hyper-V (Azure) (c) KVM (Google Cloud) (d) Bare Metal

Fig. 8: Maximum throughput of RIoTBench workflows on a single machine (varying from 4 to 16 cores). Geometric mean of workflows' throughput, each is normalized by Storm throughput on a 4-core machine.



(a) Throughput (b) Average Latency

Fig. 9: Storm-RTS achieves comparable throughput and latency versus worker-based SPEs.



(a) Dynamic Scaling (b) Performance Isolation

Fig. 10: Storm-RTS flexibly reconfigures for various workloads and protect workflow performance from collocated applications while other SPEs fail to do so. (Results from Azure VMs only, other configurations are omitted due to similarity).

with the system capacity, with workflow maximum throughput increasing almost linearly with the number of cores. These results are consistent across all of the cloud VMs and also the bare metal configuration confirming that FaaS-based SPEs can achieve equal or superior resource efficiency.

2) Multiple machines: We deploy workflow separately over multiple 4-core VMs and report the geometric mean normalized throughput for each SPE on Azure in Figure 9a. The other resource configurations are omitted because their results are the same as we have presented for Azure. All SPEs have comparable performance. Both Storm-Serverless and Storm-RTS scale well, increasing throughput with more machines. This result confirms their resource efficiency, compared to worker-based SPEs, in a distributed computing setting.

3) Processing Latency: Figure 9b shows the average per-tuple end-to-end latency of RIoTBench workflows at the steady state when the load is at around 70% of available capacity for all SPEs in Azure (we also omit other configurations due to similarity). Compared to Storm and EdgeWise, Storm-RTS and Storm-Serverless experience higher latency due to FaaS invocation overhead. However, by batching tuples into a single invocation request, the overhead is amortized. Storm-RTS keeps the latency below 20ms, just slightly above Storm and EdgeWise while significantly better than Dhalion. The results demonstrate that Storm-RTS is efficiently equivalent to other worker-based SPEs in terms of processing speed.

C. Performance Stability

1) Scalable Workflow Performance: We run each RIoTBench workflow separately in a system having ample resources, at varying input rates but keep their parallelism and rate configuration fixed. The results are presented in Figure 10a. The x-axis represents the input rate normalized by the saturation rate (maximum throughput) of Storm. The y-axis represents the geometric mean of workflow throughputs normalized by input rate. A perfect system would produce a flat line across the top – full performance with no saturation.

Our results show that all five SPE systems scale well up to Storm's saturation rate (normalized to 1.0). Beyond this point, among worker-based SPEs, only Dhalion with dynamic scaling support can handle the load. Storm and EdgeWise static worker allocations are both overwhelmed, causing their throughput to drop. At a saturation ratio of 1.5, both of their throughputs are below 20% of the input rate, and at 2.0, their throughput drops further approaching 0%. In contrast, Storm-Serverless and Storm-RTS perform dynamic allocation, using the underlying FaaS dynamic allocation to acquire more resources and support higher tuple processing rates. As a result, their performance is not limited by workflow configuration and continues to match the growing tuple for all workflows well beyond 1.0x and even 2.0x the Storm saturation rate.

The results above reveal the configuration inflexibility of the worker-based model. Any changes in workflow and input tuple rate require configuration adjustment, either manual or automatic, to achieve desired performance. On the other hand, FaaS-based SPEs do not require any parameter tuning to meet performance goals. This eases the deployment effort.

2) Performance Isolation: We consider the case of multiple workflows competing for shared resources. This is a common occurrence in production settings and can lead to performance

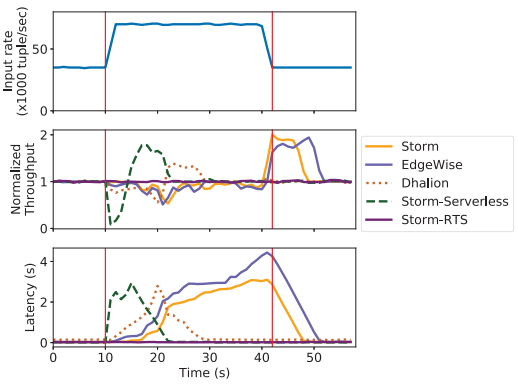


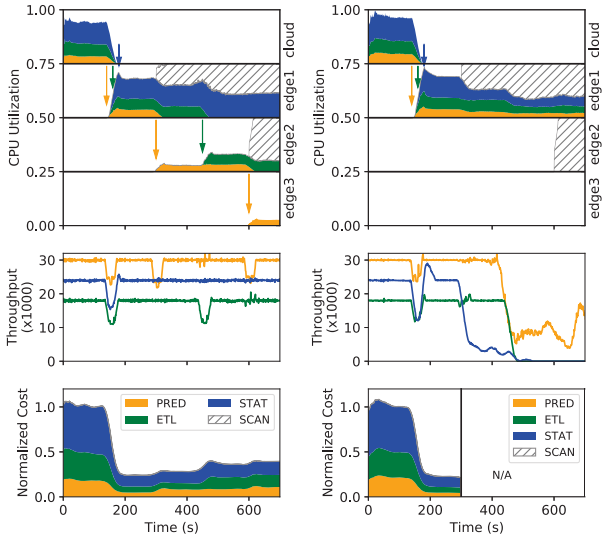
Fig. 11: Storm-RTS guarantees the performance of bursty workloads while other SPEs fail to do so.

interference. To evaluate how well SPEs protect workflow from interference, we run each of the RIoTBench workflows with SCAN. This is a single-bolt workflow performing expensive arithmetic operators on input tuples, so it competes for CPU cycles with the foreground RIotBench workflows.

In Figure 10b, we report the geometric mean of the throughputs for the RIoTBench workflows normalized by their saturation input rate. The x-axis values are normalized background load (SCAN), with 1.0 indicating the ability to consume 100% of the CPU capacity. All worker-based SPEs fail to provide performance isolation, showing a throughput decrease after the background load exceeds 50%. Due to relying on best-effort invocation allocation, Storm-Serverless sees its throughput drop from the introduction of very small levels of resource competition. The decrease is severed, and nearly 100% loss of throughput with about 30% competitive load. In contrast, the RBAM allocations enforce rate guarantees with strong resource isolation allowing Storm-RTS to provide good performance isolation all the way up to 100% competitive load. This demonstrates the ability to deliver predictable performance of RBAM SPEs as discussed in Section IV.

3) Supporting Bursty Workloads: We consider a common load pattern in practice: bursty workflows whose input rate varies over time. Workflow developers can configure Storm-RTS to handle bursty loads by setting the desired input rate equal to the peak input rate when the load bursts. We deploy a PRED workflow that operates at around 35 thousand tuples/sec on Azure VMs. However, after the 10-th second, the input rate is doubled and lasts for around 30 seconds (see the first graph of Figure 11). We execute this load with different SPEs. The workflow's throughput and latency are shown in the second and third graphs of Figure 11, respectively.

Storm and EdgeWise have their resource allocated statically. When the burst arrives, they are unable to process the excessive tuples in time causing significant high processing latency with a noticeable throughput drop. Dhalion and Storm-Serverless support dynamic allocation so they can scale up during the burst. However, it takes time for both to detect the burst, and scale resource allocation accordingly. Thus, both see



(a) Storm-RTS (b) Storm-Serverless

Fig. 12: Storm-RTS shifting workflows across edge datacenters, while maintaining stable performance. The flexibility enabled by Storm-RTS enables simple optimization of cost.

significant performance degradation for 10-20s (35 to 65% of the burst period). Storm-RTS, on the other hand, has the desired rate set to the burst peak (70 thousand tuples/sec) and it maintains the desired throughput and latency throughout the burst period. This demonstrates the robustness of performance stability provided by Storm-RTS.

D. Flexible Cloud-Edge Reconfiguration

Performance stability allows Storm-RTS to simplify application management for other objectives. Consider a simple declarative policy MinCost: minimize resource cost of stream processing workflows at any point in time. Storm-RTS (Figure 5) reduces the policy to placing operators in the data center with the lowest cost. If this data center is full, then operators will be placed in the data center with the next lowest cost, and so on. Consider a resource environment shown in Figure 7, where the cost of *edge1*, *edge2*, and *edge3* are equal to 25%, 50%, and 75% respectively relative to the *cloud*'s 100%. On this testbed, we conduct an experiment showing how Storm-RTS operate workflows stably at optimal cost.

The first graph of Figure 12a shows events that happen during the experiment and decisions made by Storm-RTS in response. At $t = 0$, Storm-RTS deploys three RIoTBench workflows in *cloud*. At $t = 150$, three edge data centers become available. The MinCost policy dictates a move to the cheapest data center, *edge1*, so the `operator distributor` shifts the operators for all three workflows to *edge1*. However, at $t = 300$, a SCAN workflow starts at *edge1*, consuming CPU resources. *edge1* becomes oversubscribed, the local RTS reports this situation to the `application coordinator`. The `application coordinator` has the `operator distributor` move PRED, the smallest

workflow, to maintain adequate performance. To minimize resource cost, *edge2* is selected. At $t = 450$, the SCAN load increases. *edge1*'s RTS system notifies the `application coordinator` again, leading to a move of ETL to *edge2*. And when SCAN expands to *edge2* at $t = 600$, its resource consumption there causes the RTS system on *edge2* to notify the `application coordinator` that it cannot maintain its guarantees. In response, Storm-RTS moves PRED to *edge3* ensuring resource sufficiency for all workflows. Through these many workflow reconfigurations, Storm-RTS maintains their performance, ensuring all three workflows stably achieve the desired throughput (the second graph of Figure 12a). And, as the `application coordinator` always moves workflows to the data centers with the lowest cost available, the total cost is minimized (the last graph in Figure 12a).

To understand the importance of Storm-RTS in implementing such declarative policy, consider the same scenario with Storm-Serverless (Figure 12b). Since Storm-Serverless allocates resources in best-effort manner, can neither detect a shortfall nor choose a suitable destination for a migration (has enough resources available). This results in poor workflow performance in these changing resource environments.

VII. Related Work

1) Solving Performance Challenges: Worker-based SPEs try to provide transparent performance by carefully considering workflow topology and the underlying system details for every scheduling decision. Many SPEs dynamically map operators to workers with heuristic scheduling strategies based on performance profiling [24], [39]–[42] and/or workflow characterization, including operator dependencies [41], [43], queue size [30], and query context [44], [45]. In distribution settings, SPEs distribute workers in traffic-aware [46]–[48] or topology-aware [46], [49], [50] fashion ensuring tuple transmission is supported by the underlying network. On low-end systems, e.g., Edge, resource heterogeneity and scarcity are common, great efforts on workload partitioning [51]–[55] and task placement [53], [56]–[62] are needed.

To resolve performance predictability challenges, worker-based SPEs leverage control mechanisms, which are typically full loops of two steps: performance degradation detection and recovery. SPEs typically detect performance degradation by monitoring stream traffic [40], [63] and throughput [64]. Some approaches try to predict potential degradation [64]–[66] and then proactively prevent it beforehand. Performance is recovered with heuristic algorithms, which either dynamically adjust resource sharing among competitive workflows [31], [64]–[67] or migrate them to other machines [40].

2) Stream Processing and FaaS: Many SPEs have leveraged FaaS for dynamic scalability [2], [17]–[21], [68]. However, these SPEs only outsource the processing logic to FaaS. Other parts of operators, such as transmission and synchronization, are implemented through the worker abstraction inheriting worker-based performance limitations. Storm-RTS wraps entire operators inside FaaS deployments. This completely removes the worker abstraction from SPE implementation, eliminating its performance limitation legacies. Additionally, SPEs relying solely on regular FaaS (e.g., [25]–[27], [37], [69], [70]) experience performance degradation when these systems fail to acquire needed resources (Section VI-C). Recent years witnessed many attempts on minimizing the chance of these failures, including optimizing invocation resource consumption [71]–[75], proactive pre-allocation, and invocation recycling [76]–[81]. There are also active studies on intelligent resource sharing [73], [82]–[85], function placement [84], [86], [87], and exploiting hardware heterogeneity [88] to improve resource efficiency and avoid interference.

3) Stream Processing across Multiple Sites: Most of the solutions for stream processing across multiple sites adopt the worker abstraction or use worker-based SPEs as a building block (e.g., [4], [89]–[91]). Worker abstraction limitations combined with new challenges that arise from distribution require additional efforts on reliability [92]–[96], communication latency and overhead [97], [98], and managing limited, heterogeneous resource pools [99], [100], balancing task placement and parallelism [101]–[103].

4) Summary: Current solutions to performance issues in stream processing and FaaS are heuristics. When facing uncovered situations, they may misbehave causing performance instability. In contrast, Storm-RTS provides robust performance stability, deployment optimization for latency (i.e., prioritize data centers with fast connections), reliability (i.e., automatic migration at power shortage), and more.

VIII. Summary and Future Work

RBAM abstraction realizes stream processing workflows as chains of rate-guarantee FaaS invocations to provide transparent and predictable performance. Storm-RTS exploits this capability to enable workflow deployment over heterogeneous and distributed resources, unlocking myriad application flexibilities and opportunities for optimized management, and simplifying distributed stream processing. Experimental results show the comparable performance of Storm-RTS versus state-of-the-art worker-based SPEs while offering excellent performance stability, great flexibility and robustness for multi-site deployment, and automatic reconfiguration capability.

With new capabilities, Storm-RTS open many research questions. For example, what new classes of distributed resource optimization for cost or reliability does this create? Can it be used to increase capability or efficiency? Also, pre-allocate resources to implement RBAMs can be insufficient in many cases, (e.g., bursty load) yet the solution is still remaining as questions waiting for the answer.

Acknowledgment

We thank the anonymous reviewers for their insightful reviews, including those who reviewed the earlier versions of this paper. This work is supported in part by NSF Grants CMMI-1832230, OAC-2019506, CNS-1901466, and the VMware University Research Fund. We also thank the Large-scale Sustainable Systems Group members for their support of this work!

REFERENCES

[1] "Apache Storm," https://storm.apache.org, May 2017.

[2] "Apache Flink," https://flink.apache.org, 2014.

[3] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R. H. Campbell, "Samza: stateful scalable stream processing at linkedin," *Proceedings of the VLDB Endowment*, vol. 10, no. 12, pp. 1634–1645, 2017.

[4] R. Tudoran, A. Costan, O. Nano, I. Santos, H. Soncu, and G. Antoniu, "Jetstream: Enabling high throughput live event streaming on multi-site clouds," *Future Generation Computer Systems*, vol. 54, pp. 274–291, 2016.

[5] M. Dias de Assunção, A. da Silva Veith, and R. Buyya, "Distributed data stream processing and edge computing: A survey on resource elasticity and future directions," *Journal of Network and Computer Applications*, vol. 103, pp. 1–17, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1084804517303971

[6] W. Wingerath, F. Gessert, S. Friedrich, and N. Ritter, "Real-time stream processing for big data," *it-Information Technology*, vol. 58, no. 4, pp. 186–194, 2016.

[7] A. AlHammadi, A. AlZaabi, B. AlMarzooqi, S. AlNeyadi, Z. Al-Hashmi, and M. Shatnawi, "Survey of iot-based smart home approaches," in *2019 Advances in Science and Engineering Technology International Conferences (ASET)*, 2019, pp. 1–6.

[8] PTC, "Howden Creates Mixed Reality Solutions to Enhance Customer Experience," https://www.ptc.com/en/case-studies/howden-mixed-reality, Feb 2019.

[9] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, "Twitter heron: Stream processing at scale," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 239–250. [Online]. Available: https://doi.org/10.1145/2723372.2742788

[10] "Kafka Streams," https://apex.apache.org/docs.html, 2022.

[11] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proceedings of the twenty-fourth ACM symposium on operating systems principles*, 2013, pp. 423–438.

[12] "Apache Apex," https://kafka.apache.org/documentation/streams/, 2017.

[13] "Apache Gearpump," http://gearpump.github.io/overview.html, 2022.

[14] "Apache Nifi," https://nifi.apache.org/, 2018.

[15] "Mantis," https://netflix.github.io/mantis/, Jun 2022.

[16] H. Röger and R. Mayer, "A comprehensive survey on parallelization and elasticity in stream processing," *ACM Comput. Surv.*, vol. 52, no. 2, Apr. 2019. [Online]. Available: https://doi.org/10.1145/3303849

[17] "Serverless Streaming Architectures and Best Practices," https://d1.awsstatic.com/whitepapers/Serverless_Streaming_Architecture_Best_Practices.pdf, Jun. 2018.

[18] S. Nastic, T. Rausch, O. Scekic, S. Dustdar, M. Gusev, B. Koteska, M. Kostoska, B. Jakimovski, S. Ristov, and R. Prodan, "A serverless real-time data analytics platform for edge computing," *IEEE Internet Computing*, vol. 21, no. 4, pp. 64–71, 2017.

[19] P. A. Bernstein, T. Porter, R. Potharaju, A. Z. Tomsic, S. Venkataraman, and W. Wu, "Serverless event-stream processing over virtual actors." in *CIDR*, 2019.

[20] "Amazon Kinesis Data Streams," https://aws.amazon.com/kinesis/data-streams/, 2019.

[21] S. Poojara, C. K. Dehury, P. Jakovits, and S. N. Srirama, *Serverless Data Pipelines for IoT Data Analytics: A Cloud Vendors Perspective and Solutions*. Cham: Springer International Publishing, 2023, pp. 107–132.

[22] Z. Chen, J. Xu, J. Tang, K. A. Kwiat, C. A. Kamhoua, and C. Wang, "Gpu-accelerated high-throughput online stream data processing," *IEEE Transactions on Big Data*, vol. 4, no. 2, pp. 191–202, 2016.

[23] R. Tönjes, P. Barnaghi, M. Ali, A. Mileo, M. Hauswirth, F. Ganz, S. Ganea, B. Kjærgaard, D. Kuemper, S. Nechifor, A. Sheth, V. Tsiatsis, and L. Vestergaard, "Real time iot stream processing and large-scale data analytics for smart city applications," in *poster session, European Conference on Networks and Communications*. sn, 2014, p. 10.

[24] Y. Mei, L. Cheng, V. Talwar, M. Y. Levin, G. Jacques-Silva, N. Simha, A. Banerjee, B. Smith, T. Williamson, S. Yilmaz, W. Chen, and C. Jerry, "Turbine: Facebook's service management platform for stream processing," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 2020, pp. 1591–1602.

[25] "AWS Lambda," https://aws.amazon.com/lambda/, 2017.

[26] "Microsoft Azure Function," https://azure.microsoft.com/en-us/services/functions/, 2017.

[27] "Google Cloud Function," https://cloud.google.com/functions, 2017.

[28] S. Fouladi, R. S. Wahby, B. Shacklett, K. V. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein, "Encoding, fast and slow: Low-latency video processing using thousands of tiny threads," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, Mar. 2017, pp. 363–376. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/fouladi

[29] A. Ali, R. Pinciroli, F. Yan, and E. Smirni, "Batch: machine learning inference serving on serverless platforms with adaptive batching," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–15.

[30] X. Fu, T. Ghaffar, J. C. Davis, and D. Lee, "Edgewise: A better stream processing engine for the edge," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, Jul. 2019, pp. 929–946. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/fu

[31] A. Floratou, A. Agrawal, B. Graham, S. Rao, and K. Ramasamy, "Dhalion: Self-regulating stream processing in heron," *Proc. VLDB Endow.*, vol. 10, no. 12, p. 1825–1836, aug 2017. [Online]. Available: https://doi.org/10.14778/3137765.3137786

[32] H. D. Nguyen, C. Zhang, Z. Xiao, and A. A. Chien, "Real-time serverless: Enabling application performance guarantees," in *Proceedings of the 5th International Workshop on Serverless Computing*, ser. WOSC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1–6. [Online]. Available: https://doi.org/10.1145/3366623.3368133

[33] "netty," https://netty.io/, 2022.

[34] "Right Place, Right Time (RiPiT) Carbon Emissions Service," https://http://ripit.uchicago.edu//, May 2022.

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

[36] D. Canvas, "Sense your city: Data art challenge." http://datacanvas.org/sense-your-city/, Jun 2022.

[37] "Oppenfaas," https://www.openfaas.com/, 2017.

[38] Amazon, "Amazon Cloud Infrastructure," https://aws.amazon.com/about-aws/global-infrastructure/, Feb 2021.

[39] X. Liu and R. Buyya, "D-storm: Dynamic resource-efficient scheduling of stream processing applications," in *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 2017, pp. 485–492.

[40] 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 Transactions on Parallel and Distributed Systems*, vol. 28, no. 12, pp. 3553–3569, 2017.

[41] D. Cheng, Y. Chen, X. Zhou, D. Gmach, and D. Milojicic, "Adaptive scheduling of parallel jobs in spark streaming," in *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*. IEEE, 2017, pp. 1–9.

[42] V. Cardellini, V. Grassi, F. L. Presti, and M. Nardelli, "On qos-aware scheduling of data stream applications over fog computing infrastructures," in *2015 IEEE Symposium on Computers and Communication (ISCC)*. IEEE, 2015, pp. 271–276.

[43] H. Miao, H. Park, M. Jeon, G. Pekhimenko, K. S. McKinley, and F. X. Lin, "Streambox: Modern stream processing on a multicore machine," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017, pp. 617–629.

[44] L. Xu, S. Venkataraman, I. Gupta, L. Mai, and R. Potharaju, "Move fast and meet deadlines: Fine-grained real-time stream processing with cameo," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, 2021, pp. 389–405.

[45] S. Venkataraman, A. Panda, K. Ousterhout, M. Armbrust, A. Ghodsi, M. J. Franklin, B. Recht, and I. Stoica, "Drizzle: Fast and adaptable stream processing at scale," in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 374–389.

[46] A. Muhammad, M. Aleem, and M. A. Islam, "Top-storm: A topology-based resource-aware scheduler for stream processing engine," *Cluster Computing*, vol. 24, no. 1, pp. 417–431, 2021.

[47] J. Xu, Z. Chen, J. Tang, and S. Su, "T-storm: Traffic-aware online scheduling in storm," in *2014 IEEE 34th International Conference on Distributed Computing Systems*. IEEE, 2014, pp. 535–544.

[48] L. Eskandari, J. Mair, Z. Huang, and D. Eyers, "T3-scheduler: A topology and traffic aware two-level scheduler for stream processing systems in a heterogeneous cluster," *Future Generation Computer Systems*, vol. 89, pp. 617–632, 2018.

[49] X. Wei, X. Wei, and H. Li, "Topology-aware task allocation for online distributed stream processing applications with latency constraints," *Physica A: Statistical Mechanics and its Applications*, vol. 534, p. 122024, 2019.

[50] H. Moussa, I.-L. Yen, and F. Bastani, "Service management in the edge cloud for stream processing of iot data," in *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*. IEEE, 2020, pp. 91–98.

[51] P. Liu, D. Da Silva, and L. Hu, "Dart: A scalable and adaptive edge stream processing engine," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021, pp. 239–252.

[52] M. A. U. Nasir, G. D. F. Morales, N. Kourtellis, and M. Serafini, "When two choices are not enough: Balancing at scale in distributed stream processing," in *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. IEEE, 2016, pp. 589–600.

[53] N. R. Katsipoulakis, A. Labrinidis, and P. K. Chrysanthis, "A holistic view of stream partitioning costs," *Proceedings of the VLDB Endowment*, vol. 10, no. 11, pp. 1286–1297, 2017.

[54] X. Wang, Z. Zhou, P. Han, T. Meng, G. Sun, and J. Zhai, "Edge-stream: a stream processing approach for distributed applications on a hierarchical edge-computing system," in *2020 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, 2020, pp. 14–27.

[55] J. Fang, R. Zhang, T. Z. Fu, Z. Zhang, A. Zhou, and J. Zhu, "Parallel stream processing against workload skewness and variance," in *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, 2017, pp. 15–26.

[56] V. Cardellini, V. Grassi, F. Lo Presti, and M. Nardelli, "Optimal operator placement for distributed stream processing applications," in *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, 2016, pp. 69–80.

[57] J. Jiang, Z. Zhang, B. Cui, Y. Tong, and N. Xu, "Stromax: Partitioning-based scheduler for real-time stream processing system," in *International Conference on Database Systems for Advanced Applications*. Springer, 2017, pp. 269–288.

[58] M. D. de Assuncao, A. da Silva Veith, and R. Buyya, "Distributed data stream processing and edge computing: A survey on resource elasticity and future directions," *Journal of Network and Computer Applications*, vol. 103, pp. 1–17, 2018.

[59] M. Nardelli, V. Cardellini, V. Grassi, and F. L. Presti, "Efficient operator placement for distributed data stream processing applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 8, pp. 1753–1767, 2019.

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

[61] A. da Silva Veith, M. D. de Assuncao, and L. Lefevre, "Latency-aware placement of data stream analytics on edge computing," in *International conference on service-oriented computing*. Springer, 2018, pp. 215–229.

[62] G. Amarasinghe, M. D. De Assuncao, A. Harwood, and S. Karunasekera, "A data stream processing optimisation framework for edge computing applications," in *2018 IEEE 21st International Symposium on Real-Time Distributed Computing (ISORC)*. IEEE, 2018, pp. 91–98.

[63] J. Li, C. Pu, Y. Chen, D. Gmach, and D. Milojicic, "Enabling elastic stream processing in shared clusters," in *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*. IEEE, 2016, pp. 108–115.

[64] M. R. H. Farahabady, A. Y. Zomaya, and Z. Tari, "Qos-and contention-aware resource provisioning in a stream processing engine," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2017, pp. 137–146.

[65] M. R. HoseinyFarahabady, A. Jannesari, J. Taheri, W. Bao, A. Y. Zomaya, and Z. Tari, "Q-flink: A qos-aware controller for apache flink," in *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. IEEE, 2020, pp. 629–638.

[66] M. R. HoseinyFarahabady, J. Taheri, A. Y. Zomaya, and Z. Tari, "Qspark: Distributed execution of batch & streaming analytics in spark platform," in *2021 IEEE 20th International Symposium on Network Computing and Applications (NCA)*. IEEE, 2021, pp. 1–8.

[67] Y. Morisawa, M. Suzuki, and T. Kitahara, "Resource efficient stream processing platform with {Latency-Aware} scheduling algorithms," in *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*, 2020.

[68] Y. Cheng and Z. Zhou, "Autonomous resource scheduling for real-time and stream processing," in *2018 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*. IEEE, 2018, pp. 1181–1184.

[69] "Openwhisk," https://openwhisk.apache.org/, 2016.

[70] "Knative," https://knative.dev/, 2021.

[71] S. Eismann, L. Bui, J. Grohmann, C. Abad, N. Herbst, and S. Kounev, "Sizeless: Predicting the optimal size of serverless functions," in *Proceedings of the 22nd International Middleware Conference*, 2021, pp. 248–259.

[72] D. Mvondo, M. Bacou, K. Nguetchouang, L. Ngale, S. Pouget, J. Kouam, R. Lachaize, J. Hwang, T. Wood, D. Hagimont *et al.*, "Ofc: an opportunistic caching system for faas platforms," in *Proceedings of the Sixteenth European Conference on Computer Systems*, 2021, pp. 228–244.

[73] A. Suresh, G. Somashekar, A. Varadarajan, V. R. Kakarla, H. Upadhyay, and A. Gandhi, "Ensure: Efficient scheduling and autonomous resource management in serverless environments," in *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. IEEE, 2020, pp. 1–10.

[74] A. Mampage, S. Karunasekera, and R. Buyya, "Deadline-aware dynamic resource management in serverless computing environments," in *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 2021, pp. 483–492.

[75] V. M. Bhasi, J. R. Gunasekaran, A. Sharma, M. T. Kandemir, and C. Das, "Cypress: Input size-sensitive container provisioning and request scheduling for serverless platforms," in *Proceedings of the 13th Symposium on Cloud Computing*, ser. SoCC '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 257–272. [Online]. Available: https://doi.org/10.1145/3542929.3563464

[76] M. Shahrad, R. Fonseca, I. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Jul. 2020, pp. 205–218. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/shahrad

[77] A. Fuerst and P. Sharma, "Faascache: keeping serverless computing alive with greedy-dual caching," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 386–400.

[78] A. U. Gias and G. Casale, "Cocoa: Cold start aware capacity planning for function-as-a-service platforms," in *2020 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 2020, pp. 1–8.

[79] C. Denninnart and M. A. Salehi, "Harnessing the potential of function-reuse in multimedia cloud systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 3, pp. 617–629, 2021.

[80] K. Kaffes, N. J. Yadwadkar, and C. Kozyrakis, "Hermod: Principled and practical scheduling for serverless functions," in *Proceedings of the 13th Symposium on Cloud Computing*, ser. SoCC '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 289–305. [Online]. Available: https://doi.org/10.1145/3542929.3563468

[81] A. Mahgoub, E. B. Yi, K. Shankar, S. Elnikety, S. Chaterji, and S. Bagchi, "ORION and the three rights: Sizing, bundling, and prewarming for serverless DAGs," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. Carlsbad, CA: USENIX Association, Jul. 2022, pp. 303–320. [Online]. Available: https://www.usenix.org/conference/osdi22/presentation/mahgoub

[82] Z. Li, L. Guo, Q. Chen, J. Cheng, C. Xu, D. Zeng, Z. Song, T. Ma, Y. Yang, C. Li, and M. Guo, "Help rather than recycle: Alleviating

cold startup in serverless computing through Inter-Function container sharing," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. Carlsbad, CA: USENIX Association, Jul. 2022, pp. 69–84. [Online]. Available: https://www.usenix.org/conference/atc22/presentation/li-zijun-help

[83] Y. Fu, L. Liu, H. Wang, Y. Cheng, and S. Chen, "Sfs: Smart os scheduling for serverless functions," in *2022 SC22: International Conference for High Performance Computing, Networking, Storage and Analysis (SC) (SC)*. Los Alamitos, CA, USA: IEEE Computer Society, nov 2022, pp. 584–599. [Online]. Available: https://doi.ieeecomputersociety.org/

[84] Y. Zhang, Í. Goiri, G. I. Chaudhry, R. Fonseca, S. Elnikety, C. Delimitrou, and R. Bianchini, "Faster and cheaper serverless computing on harvested resources," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021, pp. 724–739.

[85] V. Dukic, R. Bruno, A. Singla, and G. Alonso, "Photons: Lambdas on a diet," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, 2020, pp. 45–59.

[86] R. B. Roy, T. Patel, and D. Tiwari, "Icebreaker: Warming serverless functions better with heterogeneity," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 753–767. [Online]. Available: https://doi.org/10.1145/3503222.3507750

[87] Z. Li, Y. Liu, L. Guo, Q. Chen, J. Cheng, W. Zheng, and M. Guo, "Faasflow: Enable efficient workflow execution for function-as-a-service," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 782–796. [Online]. Available: https://doi.org/10.1145/3503222.3507717

[88] D. Du, Q. Liu, X. Jiang, Y. Xia, B. Zang, and H. Chen, "Serverless computing on heterogeneous computers," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 797–813. [Online]. Available: https://doi.org/10.1145/3503222.3507732

[89] M. Branson, F. Douglis, B. Fawcett, Z. Liu, A. Riabov, and F. Ye, "Clasp: Collaborating, autonomous stream processing systems," in *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer, 2007, pp. 348–367.

[90] V. Cardellini, V. Grassi, F. Lo Presti, and M. Nardelli, "Distributed qos-aware scheduling in storm," in *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, 2015, pp. 344–347.

[91] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle, "Millwheel: Fault-tolerant stream processing at internet scale," *Proceedings of the VLDB Endowment*, vol. 6, no. 11, pp. 1033–1044, 2013.

[92] Z. Zhang, Y. Gu, F. Ye, H. Yang, M. Kim, H. Lei, and Z. Liu, "A hybrid approach to high availability in stream processing systems," in *2010 IEEE 30th International Conference on Distributed Computing Systems*. IEEE, 2010, pp. 138–148.

[93] M. Gorawski and P. Marks, "Towards reliability and fault-tolerance of distributed stream processing system," in *2nd International Conference on Dependability of Computer Systems (DepCoS-RELCOMEX'07)*. IEEE, 2007, pp. 246–253.

[94] J.-H. Hwang, U. Cetintemel, and S. Zdonik, "Fast and reliable stream processing over wide area networks," in *2007 IEEE 23rd International Conference on Data Engineering Workshop*. IEEE, 2007, pp. 604–613.

[95] X. Wei, Y. Zhuang, H. Li, and Z. Liu, "Reliable stream data processing for elastic distributed stream processing systems," *Cluster Computing*, vol. 23, no. 2, pp. 555–574, 2020.

[96] Y. Zhuang, X. Wei, H. Li, M. Hou, and Y. Wang, "Reducing fault-tolerant overhead for distributed stream processing with approximate backup," in *2020 29th International Conference on Computer Communications and Networks (ICCCN)*. IEEE, 2020, pp. 1–9.

[97] A. Jonathan, A. Chandra, and J. Weissman, "Wasp: wide-area adaptive stream processing," in *Proceedings of the 21st International Middleware Conference*, 2020, pp. 221–235.

[98] F. Yin, X. Li, X. Li, and Y. Li, "Task scheduling for streaming applications in a cloud-edge system," in *Security, Privacy, and Anonymity in Computation, Communication, and Storage*, G. Wang, J. Feng, M. Z. A. Bhuiyan, and R. Lu, Eds. Cham: Springer International Publishing, 2019, pp. 105–114.

[99] F. R. de Souza, M. D. de Assunçao, E. Caron, and A. da Silva Veith, "An optimal model for optimizing the placement and parallelism of data stream processing applications on cloud-edge computing," in *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, 2020, pp. 59–66.

[100] E. Kalyvianaki, M. Fiscato, T. Salonidis, and P. Pietzuch, "Themis: Fairness in federated stream processing under overload," in *Proceedings of the 2016 International Conference on Management of Data*, 2016, pp. 541–553.

[101] F. R. de Souza, A. D. S. Veith, M. D. de Assunçao, and E. Caron, "Scalable joint optimization of placement and parallelism of data stream processing applications on cloud-edge infrastructure," in *International Conference on Service-Oriented Computing*. Springer, 2020, pp. 149–164.

[102] A. Dasilvaveith, M. D. de Assuncao, and L. Lefevre, "Latency-aware strategies for deploying data stream processing applications on large cloud-edge infrastructure," *IEEE Transactions on Cloud Computing*, 2021.

[103] S. K. Sharma and X. Wang, "Live data analytics with collaborative edge and cloud processing in wireless iot networks," *IEEE Access*, vol. 5, pp. 4621–4635, 2017.