Sponge: Fast Reactive Scaling for Stream Processing with Serverless Frameworks

Won Wook Song, *Seoul National University;* Taegeon Um, *Samsung Research;* Sameh Elnikety, *Microsoft Research;* Myeongjae Jeon, *UNIST;* Byung-Gon Chun, *Seoul National University and FriendliAI*

https://www.usenix.org/conference/atc23/presentation/song

This paper is included in the Proceedings of the 2023 USENIX Annual Technical Conference.

July 10–12, 2023 • Boston, MA, USA

978-1-939133-35-9

Open access to the Proceedings of the 2023 USENIX Annual Technical Conference is sponsored by

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

Sponge: Fast Reactive Scaling for Stream Processing with Serverless Frameworks

Won Wook Song
Seoul National University

Taegeon Um
Samsung Research

Sameh Elnikety
Microsoft Research

Myeongjae Jeon*
UNIST

Byung-Gon Chun*
Seoul National University and FriendliAI

Abstract

Streaming workloads deal with data that is generated in real-time. This data is often unpredictable and changes rapidly in volume. To deal with these fluctuations, current systems aim to dynamically scale in and out, redistribute, and migrate computing tasks across a cluster of machines. While many prior works have focused on reducing the overhead of system reconfiguration and state migration on pre-allocated cluster resources, these approaches still face significant challenges in meeting latency SLOs at low operational costs, especially upon facing unpredictable bursty loads.

In this paper, we propose Sponge, a new stream processing system that enables fast reactive scaling of long-running stream queries by leveraging serverless framework (SF) instances. Sponge absorbs sudden, unpredictable increases in input loads from existing VMs with low latency and cost by taking advantage of the fact that SF instances can be initiated quickly, in just a few hundred milliseconds. Sponge efficiently tracks a small number of metrics to quickly detect bursty loads and make fast scaling decisions based on these metrics. Moreover, by incorporating optimization logic at compile-time and triggering fast data redirection and partial-state merging mechanisms at runtime, Sponge avoids optimization and state migration overheads during runtime while efficiently offloading bursty loads from existing VMs to new SF instances. Our evaluation on AWS EC2 and Lambda using the NEXMark benchmark shows that Sponge promptly reacts to bursty input loads, reducing 99^{th}-percentile tail latencies by 88% on average compared to other stream query scaling methods on VMs. Sponge also reduces cost by 83% compared to methods that over-provision VMs to handle unpredictable bursty loads.

1 Introduction

Stream queries continuously process real-time data to extract insights and make business-critical decisions, such as analyzing real-time logs to extract statistics, detect anomalies, and provide notifications [2, 6, 29, 48, 52]. Latency is an essential service level objective (SLO) in these streaming workloads, as faster up-to-date results mean more value. Stream systems are expected to run 24/7 while meeting their SLOs [53].

Meanwhile, stream systems regularly face significant challenges due to sudden, unpredictable bursts of input loads caused by random events, e.g., influencer tweets, breaking news, and natural disasters [46, 47]. These bursts can abruptly increase the input load by more than 10× in just a few seconds [11, 17, 26, 37, 56]. If stream processing systems do not quickly acquire additional computing resources that can handle the bursty loads and do not promptly redistribute the load to the newly allocated computing resources, events will soon pile up on the existing resources, leading to cascading impacts on query latencies that can have fatal consequences such as reduced user satisfaction and revenues [48].

One approach to quickly acquiring additional computing resources is to over-provision resources. Existing work such as Rhino [18], Megaphone [25], and Chronostream [55] builds efficient stream load redistribution mechanisms by harnessing over-provisioned resources to minimize latency spikes on load bursts. For instance, Megaphone [25] smoothly migrates stream query loads to extra resources during stable load in preparation for load spikes. However, over-provisioning solutions can be *costly* and inefficient, as a significant amount of resources will remain idle for most of the time.

Cloud services can reduce operational costs by offering on-demand resource allocations. Existing scaling approaches for on-demand resources dynamically migrate stream operator instances, in units of parallel *tasks*, to the allocated on-demand virtual machines (VMs). They redistribute the tasks and their states, which are key-value pairs of aggregated intermediate results [7, 15, 16, 19, 36, 49]. However, migrating tasks and their states incurs extra overheads (e.g., (de)serialization), which increase proportionally to the state size (e.g., a large number of key-value pairs), and can violate low latency SLOs. Moreover, using VMs, which are popular on-demand cloud resources, can further exacerbate latency spikes due to the considerable launch delay of VM instances which can take

* Corresponding authors.

dozens of seconds (i.e., 25-30 secs) with conventional cloud providers [21, 31, 44].

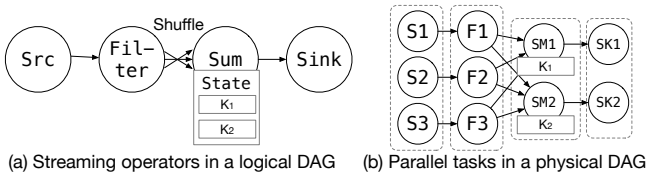
In this paper, we design Sponge, a new stream processing system that requires low operational costs and keeps low latency upon sudden bursty loads. Sponge is designed with the following three design principles:

Combining two heterogeneous cloud resources to have the best of both worlds: Sponge harnesses two heterogeneous cloud resources: VMs and serverless function (SF) instances. Serverless solutions provided by conventional cloud providers [11, 17, 26, 37, 56] only take hundreds of milliseconds (i.e., 300-750 ms) to launch and prepare and are designed to achieve high scalability, while the operational costs are much higher than those of VMs. Therefore, to achieve low latency and low operational costs, Sponge uses VMs for processing stable streaming loads for longer periods of time, while quickly invoking SF instances and using them for short periods of time to handle bursty loads. If the bursty input loads persist, we may consider launching new VM instances to permanently offload the tasks with existing state migration techniques [16, 18, 19, 23, 25, 28, 36, 45, 49, 55]. In such cases, on-demand SF instances can be used to accomplish system SLOs by hiding the launch overhead during the preparation of the new VM instances.

Keeping tasks with high migration overheads on VMs, while quickly redirecting data to SFs: When VMs process streaming data with stable loads over long periods of time, the states of stream tasks are materialized, and the state size may increase on the existing VMs. To avoid the state migration overheads from VMs to SFs, Sponge incorporates the *redirect-and-merge* mechanism: Sponge immediately redirects the increased load to SFs, which are imminent to offload, so that each SF instance can build small partial states and periodically send them back to the VMs to merge with the original states. This approach allows Sponge to promptly exploit fast-launching SF instances and bypass the prohibition of direct network communication between SF instances. For quick data redirection, Sponge exploits SF properties to prevent cold start latencies and pre-initiates copies of VM tasks on SFs to keep its runtime, process, and pre-initiated tasks readily available on time.

***Fast reactive scaling*:** On top of the fast resource scaling mechanisms on SF instances, Sponge identifies bottleneck tasks *reactively* and makes precise decisions on which part of the query to offload and how much of the compute resources to request. At runtime, Sponge continuously monitors the CPU usage, the major resource constraint of task execution, to quickly react to the changing input loads. Our offloading policy determines the fraction of input loads to offload based on excess events accumulated in the input queue and accounts for the optimal time to recover from load increases to meet the SLOs for a given query.

Sponge is built atop Apache Nemo [51, 57] with about 10K lines of code. We evaluate Sponge on EC2 instances (5× r5.xlarge) and AWS Lambda instances (up to 200 Lambda instances of 1,769MB memory with one full CPU core) with NEXMark [42], a popular benchmark for stream processing. The effectiveness of our optimizations varies according to the characteristics of queries (e.g., dataflow pattern, # of tasks, and state size). Our evaluations show that Sponge exhibits similar performance to costly over-provisioned approaches, and reduces input event 99^{th}-percentile tail latencies by 88% on average compared to scaling queries on VMs and by 70% compared to scaling on SFs without our techniques.



(a) Streaming operators in a logical DAG (b) Parallel tasks in a physical DAG

Figure 1: (a) Logical DAG of four operators including a stateful `Sum` operator with two key groups. (b) The corresponding physical DAG with parallel tasks.

2 Background

In this section, we describe the resource demand characteristics of stream processing and different on-demand resource provisioning methods provided by current cloud services.

2.1 Stream Processing

Execution model. A stream processing query processes an unbounded number of timestamped events to derive specific results (e.g., top K, statistics) on every temporal window. The execution of the query is generally expressed as a directed acyclic graph (DAG) of operators and data dependencies. As shown in Fig. 1, a vertex represents a stream operator that transforms input events and emits output events, and an edge represents how data flows between its adjacent operators. Popular stream engines like Flink [15], Spark Streaming [7], and Beam [12] aid users with high-level languages (e.g., declarative language) to facilitate query expressions. To provision compute resources over stream operators in response to the input data rate, the stream engine generates an optimized physical DAG (Fig. 1(b)) after translating a user query into a logical DAG (Fig. 1(a)). In a physical DAG, each logical operator is expanded into n parallel tasks, $p_0, ..., p_{n-1}$, where each task processes a disjoint data partition.

Streaming operators and resource demands. A stream operator is either stateless or stateful. Stateless operators, such as `map` and `filter`, are typically used to compute individual events or drop unnecessary events or fields by applying predicates. Due to their simplicity, stateless operators can be pipelined together within a single node to leverage data locality and reduce network overheads. On the other hand, stateful operators, such as `groupByKey` and `join`, perform data

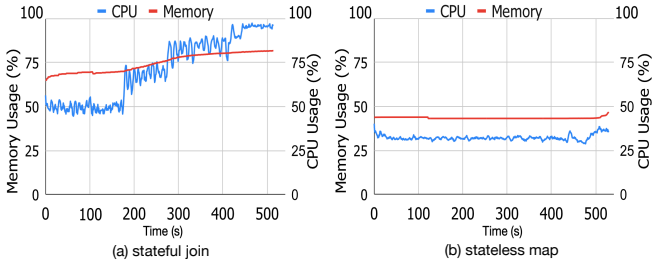


Figure 2: CPU and memory usage patterns for (a) stateful windowed `join` and (b) stateless `map` operators upon processing a fixed input rate of 80K events/s on identical 4 vCore nodes. The CPU and memory usage of the stateful operator increase until the window is full.

grouping within a window boundary to organize unbounded streaming events into disjoint groups based on timestamps and aggregation keys, requiring computationally extensive key lookups. Thus, most streaming engines apply parallelism specifically to stateful operators such that a single stateful task p_i processes events that only belong to a non-overlapping key partition group K_i out of the entire key space $K = \cup_{i=0}^{n-1} K_i$.

Stateful operators are often the major source of system bottlenecks [38, 50]. In particular, since each parallel stateful task is assigned to a key partition group, it incurs shuffle communication for the events in its key group that are collected from the preceding (upstream) operators. Shuffle communication often requires the data to travel across different nodes, requiring data serialization and deserialization on top of the computation performed for the key lookups. As a result, as shown in Fig. 2, it is prevalent to provision more CPUs to execute stateful operators rather than stateless operators [28, 54].

2.2 On-Demand Resource Provisioning

Several real-world stream analytics systems report high temporal variability in the event count of data streams, even across one-minute time windows [34, 36, 43, 48]. This means that stream processing may need to frequently adjust resource provisioning and query execution plans in response to changes in input loads. Upon facing increased input loads, the system needs to allocate more resources to avoid operators being congested and maintain stable query latency.

Cloud providers offer primarily two options for on-demand resource allocation: virtual machines (VM) and serverless functions (SF). We compare three representative characteristics between these two options in more detail.

Resource size. VMs are machine-isolated by bare-metal hypervisors, whereas SFs are process-isolated by OSes. Therefore, SFs are much more flexible in allocating resources. Cloud providers typically provide VMs in chunks of a predefined, fixed amount of resources (e.g., r5.xlarge with 4 vCores and 32GB memory). In contrast, SFs are allocated based on a specified memory size. For the memory size, cloud providers assign a certain number of CPU power (e.g., vCores)

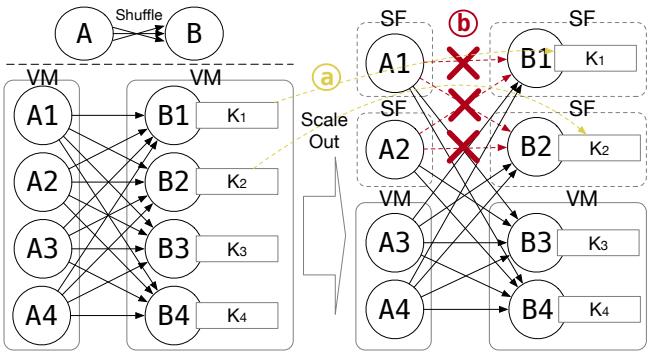


Figure 3: While scaling out on SF instances, the system must be aware that ⓐ task state migration overheads lead to latency spikes, and ⓑ direct data communication among adjacent tasks is prohibited between SF instances.

guided by their pricing model [8]. We observe that network bandwidth per SF instance is about 100Mbps and concurrently using multiple SFs can increase the bandwidth up to GBs of effective bandwidth, which VMs already support, providing enough capacities to handle most streaming workloads.

Start-up time. VM instances take a significant amount of time to launch and to prepare the runtime stack for query workload as they virtualize resources using bare-metal hypervisors. We observe that provisioning a new VM instance in major cloud service providers, like AWS, Azure, and GCP, mostly has a latency of over 25 seconds. On the contrary, SF instances provided by these cloud vendors take only 300-750 ms to launch and be ready to run because SF instances share runtimes and resources at the OS level.

Usage cost. SF instances are much more expensive to use than VM instances, e.g., 4× more expensive when running a 1GB SF instance with AWS Lambda (with $<$ 1 vCPU) compared to a t2.micro EC2 instance, which is equipped with 1 vCPU and 1GB RAM. However, temporarily using SF instances primarily for frequent short-lived bursty loads that constitute only a small fraction of time throughout the day [26] does not significantly increase the operational cost (§ 6.5).

3 Challenges

Based on these observations, we propose to use a combination of VMs and SFs to have the best of both worlds. To achieve low latency and cost, we use cheap and stable VMs for handling continuous loads for long periods of time, and costly and reactive SFs for bursty loads during short periods of time. In this section, we describe several challenges in scaling streaming loads from VMs to on-demand SF instances.

C1. Migration with large operator states. For stream scaling in the cloud, existing approaches trigger resource adaptation primarily by re-scaling operators (i.e., increasing or decreasing parallelism) and migrating the bottlenecked tasks to the instances with available resources (i.e., load redistribution) [7, 15, 16, 19, 28, 36, 49]. Thus, even if we can set

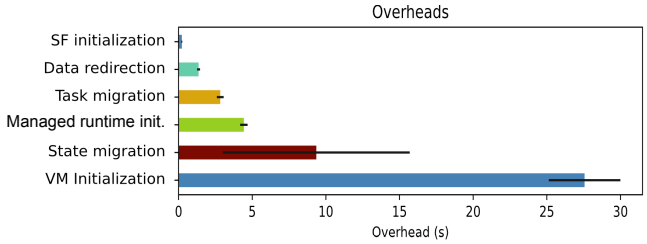


Figure 4: A comparison of the overheads of different steps of workload scaling on stream processing systems in the cloud. The VM, SF, and managed runtime initialization overheads are averaged across all instances, and the data redirection and task/state migration overheads are averaged across all single-scaling operations for the 5× input load experiment in § 6. Error bars indicate the 95% confidence interval.

up SF instances quickly, the task state migration overhead is inevitable with existing systems, as shown by Fig. 3ⓐ, and paradoxically often inflicts damage to system performance.

Fig. 4 illustrates the various overheads that occur during a single workload scaling for the queries evaluated in § 6. As shown in this figure, the task migration and reconfiguration require a few extra seconds (3-4 seconds) to resume the work after the migration. Also, the state migration takes several seconds (e.g., from 4 to 17 seconds) depending on the state size because of the (de)serialization overheads of states. These task and state migration overheads lead to increased query latency due to the delay in receiving events from upstream tasks. The system that aims to meet low-latency SLOs must correctly and rapidly carry out task offloading to SFs. In particular, some use cases are expected to generate outputs even in order of seconds or less, without query accuracy loss [36, 48].

C2. Indirect data communication between SF instances. As SFs are designed to be provisional and temporary, cloud vendors usually prohibit running a server process that can accept inbound network connections on an SF instance. Hence, direct data communication across SF instances is prohibited. This prevents neighboring stream operators (parent and child operators) from being offloaded to SFs simultaneously, as these operators require direct shuffle data transfers to group data by its key partitions, as shown in Fig. 3ⓑ.

Therefore, we can choose to migrate only certain tasks to SFs (e.g., either operator A or B in Fig. 3), but this eventually leads the bursty input load to end up on VMs on the adjacent operators and fails to alleviate latencies. Alternatively, we can offload all the tasks involved in the shuffle communication on a large SF instance. However, this forces parallel tasks to be located on a single SF instance, which can lead to network pressure while leaving VMs idle. Consequently, it is essential to design the system to be able to offload adjacent operators together to SFs while bypassing the prohibited direct communication across SF instances.

C3. Quick decision making and scaling. With frequent unpredictable changes in input events, offloading decisions must be made quickly at runtime. Stream systems often detect symptoms of bottlenecks from system metrics and decide on whether and how much to scale. However, existing approaches can be too slow, as they require multiple iterations of optimization that scale bottleneck operators one after another [19]. Other work prevents such iterations by providing a global optimum after collecting all metrics from all executors to redistribute tasks [28]. While these approaches effectively find the target throughput and may be suitable for throughput-oriented workloads, they only work in intervals of multiple 10s of seconds and may not be suitable for latency-oriented workloads. For a stream system operating with diverse intervals and window sizes, it is important to have a uniformly fast and effective optimization level to prevent window outputs from being delivered too late.

4 Sponge Design

In this section, we describe the key pillars of our system design and explain the details by illustrating our graph rewriting algorithm, dynamic offloading policy, and the mechanisms that prevent cold start latencies and enable system correctness.

4.1 Design Overview

Latency spikes occur when the input rate r_i exceeds the maximum throughput m_i on a particular task p_i. When this happens, data starts to accumulate on the event queue, along with the operator state in memory, leading to high CPU usage and memory pressure. In a cloud environment, the maximum throughput m_i often depends on the CPU capacity allocated to the task, regardless of the operator type. This is because most cloud providers are equipped with GBs of network bandwidths, and memory pressure starts to increase when the CPU becomes saturated, and the input data builds up in the event queue with $r_i > m_i$. Therefore, our goal is to primarily focus on relieving CPU pressure. To achieve this, we design a system that accurately estimates the amount of additional resources needed and provides fast mechanisms for offloading CPU computation from VMs to SFs through two design principles: *redirect-and-merge* and *fast reactive scaling*.

Redirect-and-merge. Sponge is designed to rapidly forward increased input load to available resources in SF instances. To ensure speed, we bypass expensive query optimizations during runtime by performing DAG optimizations during compile time, i.e., when the application is launched (Fig. 5ⓐ). During compile time, there are no concerns yet about runtime synchronization and progress, so it only takes about 200ms upon workload initialization to perform the DAG optimizations. After the optimization, Sponge scheduler places tasks on appropriate executors (Fig. 5ⓑ). This allows Sponge to focus on which operators and how much of their data volume

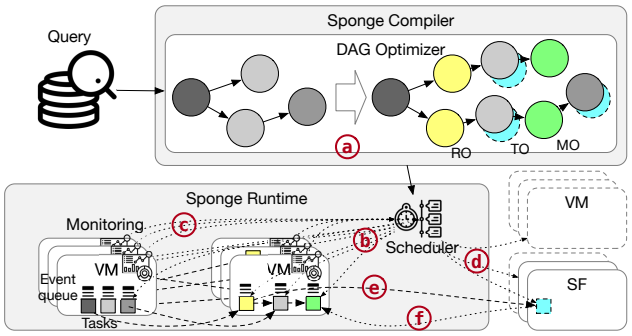


Figure 5: Sponge architecture.

to offload to SFs based on monitoring CPU usage (Fig. 5ⓒ) without having to relaunch queries at runtime.

While stateful operators are our primary focus as initial bottlenecks, any operator can become a subsequent bottleneck. Thus, we enable offloading for any operator, regardless of its type and statefulness. We design transient operators (TOs) so that operator logic can be prepared on SF instances to receive events immediately after detecting an increase in the input load and CPU usage on VMs (Fig. 5ⓓ). We also enable offloading to be activated at any time with high efficiency and responsiveness. To meet these requirements, we introduce a set of new proxy operators: router operators (ROs) and merge operators (MOs). ROs supervise the data communication to downstream VM and SF instances, in order to enable the system to rapidly and elastically *forward* data from any existing operators to the designated instances (Fig. 5ⓔ). To minimize state migration overhead, which is a major bottleneck in task migration [18, 23, 25, 55], the states, exclusively for the offloaded input load, are maintained separately on SFs. MOs enable the system to later *merge* the corresponding states of offloaded workload created on SFs with the states on the original VMs for any stateful operators (Fig. 5ⓕ). This way, the offloading overhead for both stateful and stateless operators is substantially reduced, as we only have to offload the computational logic, and not the states. The proxy operators are inactive during non-scaling periods to avoid extra costs and are only activated when needed.

Fast reactive scaling. With the principle above, we provide a fast reactive approach that prevents inaccurate predictions on resource provisioning by monitoring local performance metrics within the executors. Bottlenecks often occur individually on VMs, so it is sufficient to mitigate them *locally* within each VM. As briefly mentioned, relieving CPU pressure when the input rate r_i is greater than the operator throughput m_i is key to reducing CPU and memory strain in stream processing systems. Sponge has low monitoring overhead, with less than *10ms per observation*. Based on input rate and CPU usage observations, Sponge estimates the amount of CPU resources needed to increase operator throughput and meet our SLOs under increased input loads.

Algorithm 1: DAG rewriting for operator insertion.

```
1  Function OperatorInsertion(dag)
2      for vertex, inedges in dag.topological_sort() do
3          t_op = TransientOp.for(vertex)
4          for inedge in inedges do
5              if inedge.comm != 1to1 then
6                  r_op = RouterOp.create()
7                  dag.remove_edge(inedge)
8                  e1 = {inedge.src→r_op, inedge.comm}
9                  e2 = {r_op→vertex, 1to1}
                   // connect transient operators
10                 e3 = {inedge.src.t_op→r_op, inedge.comm}
11                 e4 = {r_op→vertex.t_op, 1to1}
12                 dag.add_edges([e1, e2, e3, e4])
13             else
14                 e = {inedge.src.t_op→t_op, 1to1}
15                 dag.add_edges([e])
16             if inedge.src.is_stateful() then
17                 m_op = MergeOp.create()
18                 dag.remove_edge(inedge)
19                 e1 = {inedge.src→m_op, inedge.comm}
20                 e2 = {m_op→vertex, 1to1}
21                 e3 =
                    {inedge.src.t_op→m_op,inedge.comm}
22                 e4 = {m_op→vertex, 1to1}
23                 dag.add_edges([e1, e2, e3, e4])
24     return dag
```

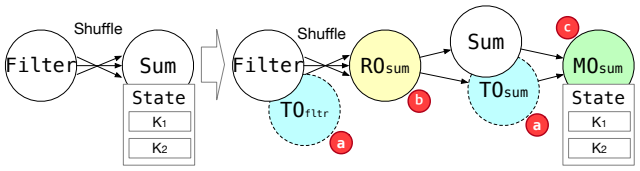


Figure 6: DAG transformation after graph rewriting.

4.2 Compile-time Graph Rewriting Algorithm

At the start of the application, our compiler applies the graph rewriting algorithm (Algorithm 1) to the application DAG, which produces a new DAG based on a set of conditional rules, as shown in Fig. 6. In our algorithm, TOs, ROs, and MOs are inserted as follows. *TOs* are cloned stream operators with additional features to run on SF instances, such as maintaining partial states for stateful operators. Since all original operators are potential candidates for offloading, we first create TOs for all operators (line 3, Fig. 6ⓐ). This way, all operators causing CPU bottlenecks can scale on SFs with TOs. *ROs* enable data communications between VM and SF instances when the communication pattern involves a shuffle or a broadcast (as one-to-one communications typically occur locally between pipelined operators) (§ 2). ROs run on existing VMs to redirect the input data to the downstream tasks running on either VMs or SFs without performing additional

computations (line 5-12, Fig. 6ⓑ). If the communication pattern involves the same number of partitions and tasks between two operators, we pipeline the corresponding TOs with a one-to-one edge (line 13-15). ROs incur almost no costs as they simply redirect events to the tasks on target instances (e.g., conventional or TO tasks). Lastly, we insert an *MO* after each stateful operator for every edge, so that the partially aggregated states on the TOs can be merged back into the original states on VMs (line 16-23, Fig. 6ⓒ), where the details of the merging mechanisms are provided in § 4.5. Stateless operators do not need to merge states, so they simply pass on their output to the following operators (e.g., filter operator in Fig. 6). During non-scaling periods, ROs are not activated and TOs and MOs do not receive any data, adding no computational costs to the runtime. The operators are only activated upon offloading actions.

4.3 Dynamic Offloading Policy

In this section, we describe when Sponge triggers offloading, how many SF instances it uses, and how many events it offloads. Our goal is to constantly maintain low query latencies while keeping CPU utilization stable across all active cloud instances. To achieve this, Sponge quickly calculates the total number of CPU cores needed to meet this goal and the Sponge scheduler redistributes the workload accordingly among the tasks placed on VMs and SFs.

Overall workflow. The Sponge runtime, shown in Fig. 5, is a main system component that performs monitoring of the resources and operator states to take immediate scaling actions as needed. Each executor continuously monitors CPU resources and input rates, typically every second, and observes if the CPU load falls outside a stable range for consecutive periods. If so, the Sponge runtime initiates the scaling phase by first calculating the target system throughput, based on the over-subscription period of CPUs and the current input rates (that jointly decide the number of events in the queue), and the recovery deadline (the time remaining to clear the events and return the system to a stable state). Subsequently, the Sponge runtime adds new SF instances as needed to meet the recovery deadline by sending requests to the Sponge scheduler. The number of new SF instances is chosen to be minimum to neither over-subscribe nor under-subscribe the active cloud instances, minimizing operational costs. After a scaling action is taken, the Sponge runtime returns to the monitoring phase. It is possible that the Sponge runtime may go through multiple monitoring-scaling phases before the system becomes stable.

4.3.1 Detailed Offloading Process

CPU utilization goals. Along with system metrics, such as the input rate and operator latency, Sponge measures the CPU load of the executor in order to maintain adequate CPU loads on individual nodes. Through extensive experiments, we have observed that the input rate r_i exceeds operator throughput m_i and event queues start to build up (i.e., $r_i > m_i$) when the CPU is occupied at around 75-80% of its capacity. We have also seen symptoms of over-provisioned system resources when the CPU load falls under 50-60%. Due to such reasons, we aim to maintain the CPU utilization range between 60-80%.

Events in the queue. Assuming $r_i(t) > m_i(t)$ between times t_p and t_{p+1} ($t_p < t_{p+1}$), the number of excess events accumulated in the queue can be formulated as $\int_{t_p}^{t_{p+1}} (r_i(t) - m_i(t))\,dt$. Obviously, the accumulated events in the queue will be smaller if the duration $d = t_{p+1} - t_p$ is smaller. This is the main reason for using SF instances over VMs – to reduce the duration of $r_i(t) > m_i(t)$.

Recovery deadline. Recovering from this event backpressure is achieved by providing the system with additional resources to achieve higher throughput, m_{i_o}. If additional resources are available from time t_o, we should set a deadline t_{o+1} until which we aim to empty the queue to return to a stable state for our streaming system. We base the deadline on the window interval of the query (e.g., 10 seconds) so that we can deliver the query results within the query's next output boundary. Then, the number of additional events that can be processed from the queue can be expressed as $\int_{t_o}^{t_{o+1}} (m_{i_o} - r_i(t))\,dt$, where the increased throughput is larger than the input rate ($m_{i_o} > r_i(t)$). As a result, we should adjust our throughput m_{i_o} with sufficient additional resources to meet our recovery deadline t_{o+1} (e.g., $t_{o+1} - t_o \leq 10$) such that the following equation holds:

$$\int_{t_p}^{t_{p+1}} (r_i(t) - m_i(t))\,dt \leq \int_{t_o}^{t_{o+1}} (m_{i_o} - r_i(t))\,dt \qquad (1)$$

The approximation of the integrals is based on Simpson's rule provided by [5], which turns complex calculations into simple arithmetic that incurs trivial overheads.

Stable throughput per VM CPU core. To calculate the target number of SF instances required to achieve our target throughput, we maintain records of the CPU usage rate of the VM node during stable loads. Assuming that a task p_i runs on a single VM core with an average usage rate of $\overline{u_{cpu_i^{VM}}}(\%)$ and an average task input rate of $\overline{r_i}[events/sec]$ based on the records, we scale and approximate the input rate and throughput for 100% utilization of the VM core $rpc_i^{VM}[events/(sec \cdot core)]$ for task p_i, as follows:

$$rpc_i^{VM} = \frac{\overline{r_i}}{\frac{\overline{u_{cpu_i^{VM}}}}{100}} \qquad [events/(sec \cdot core)] \qquad (2)$$

Required number of SF instances and data redistribution. Based on the approximation of how much input throughput a VM core can handle, we can calculate the number of required SF instances to achieve our target throughput m_{i_o} with a simple division. We offload tasks from VMs to SFs to keep the CPU utilization of VM clusters between $60-80\%$ in order to prevent resource over-provisioning. Hence, we target the VM CPU core usages at 70%, for our approximation to solidly fall

into our target with a ±10% buffer even when our profiling measurements exhibit minor errors on time-varying variables like $r_i(t)$.

Assuming the CPU capacity of each SF instance core is different from that of a VM core, we can derive a relation between them with profiling: $capa^{SF}_{core} = \rho * capa^{VM}_{core}$. Correspondingly, $rpc^{SF}_i = \rho * rpc^{VM}_i$ because the throughput is proportional to the CPU capacity. Altogether, the number of total SF cores c that we need to prepare to meet our latency goal for task p_i can be derived with Equation 2 as follows:

$$c = \lceil \frac{m_{i_o}}{0.7 \cdot rpc^{SF}_i} \rceil = \lceil \frac{m_{i_o} \cdot \overline{u_{cpu_s}}}{0.7 \cdot 100 \cdot \rho \cdot \bar{r}_i} \rceil \quad (3)$$

where the number of required SF instance cores increases as ρ decreases. Finally, the number of SF instances can be calculated with $\frac{c}{k}$ where k is the number of cores per SF instance ($k = 1$ in our evaluation).

When offloading stateless or stateful tasks, Sponge evenly redirects data or redistributes keys to the $\frac{c}{k}$ SF instances, while processing remaining events on VMs to keep 70% CPU usage in average. If the event distribution is skewed across the key space, the solution can be extended to use key histograms for more accurate key partitioning, as in existing approaches [13, 30]. Both during scaling up and down, the target CPU usage is set at 70% within our target range.

4.4 Reducing Cold Start Latency

In order to timely gain access to SFs, Sponge provides two options: (1) warm-up SF workers in advance by sporadically processing short events [21, 39, 58] to minimize the managed runtime initialization overheads [35], and (2) use solutions like AWS SnapStart [32], which bring shorter initialization times of SFs by taking a snapshot of the initialized SF instance environment and caching it for low-latency access [1]. As SFs are charged based on the memory usage time and the number of requests [9, 10, 22], prices for pre-warming SFs are trivial (nearly zero). By default, Sponge prepares and keeps enough SFs warm to handle up to 5× the stable load during the workload. For bursty loads that exceed 5× the stable load, Sponge timely prepares new instances with SnapStart [32] on AWS. SFs on AWS SnapStart [1, 32] show a slightly worse start-up time compared to the instances that are kept warm in advance, but the overhead is reduced by more than 80% compared to unoptimized JVM initializations, resulting in a sub-second total start-up time for SFs (§ 6.4). As a result, by enabling both methods for initializing the managed runtime, Sponge can timely gain access to SFs upon facing unpredictable bursty input loads.

4.5 Correctness

As stream systems are designed to be long-running, progress is tracked by the positions of the *watermarks* that flow along

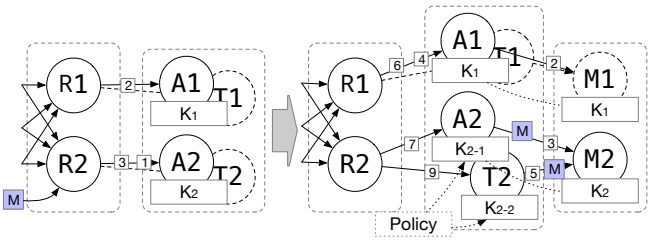


Figure 7: Once an operator (A2) tries to scale, an offload message (M) is generated at the RO (R2) to activate its TO (T2) and MO (M2). The offload message acts as a boundary among input events (1-9) for operator scaling and state merging.

with stream events [12, 15, 51, 57]. Based on the intuition, Sponge maintains correctness by (1) introducing a watermark in the event stream as a control message upon (de)activating an operator and (2) ensuring that all events between two watermarks are processed in the original system (i.e., without offloading) or on the offloaded operators [23, 36].

Concretely, upon detecting a possible bottleneck in an operator task p_i on an executor, Sponge fires a watermark message `M` into the data channel (Fig. 7). Upon receiving watermark messages, operators checkpoint their states to later recover from the checkpointed states, guaranteeing exactly-once processing. Sponge scales after temporarily pausing operators upon control messages and delivering messages to downstream tasks. Once all on-the-fly events in the data plane are consumed, downstream tasks send acks back to the upstream tasks to guarantee no event and state loss.

Thus, the events that arrive after `M` are immediately redirected to the tasks of the TOs on SFs, where partial states are aggregated if the operator is stateful. For stateful operators, TOs send the aggregated states to the following MOs placed on VMs, which know where to start merging the partial states with the original ones. Both incremental and appended aggregation can be mergeable with partial states, similar to how Flink [15] manages shared states, which causes moderate overhead on VMs. Even if events arrive out-of-order in the merge operators, they wait for the same watermark to arrive from the task in VM and its transient tasks so that the states can be synchronized. This ensures that all input data before and after `M` are processed according to the proposed optimizations.

5 Implementation

We have implemented Sponge with about 10K lines of Java with support for AWS Lambda, as follows:

Programming interface: To express a stream query as an application DAG, we use Apache Beam [12] application semantics, which is a widely used dataflow programming interface for various systems (e.g., Spark [7], Flink [15], Cloud-Dataflow [3]). In addition, as Beam provides APIs for developers to build associative and commutative operations (e.g.,

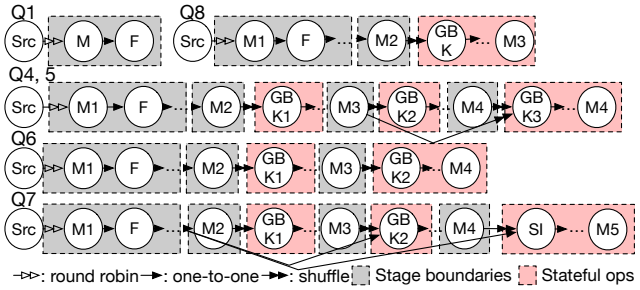


Figure 8: A simplified application DAG of stream queries used in our evaluation. `M` and `F` are map and filter operators, `GbK` is a stateful group-by-key operator for incremental aggregation on windows, and `SI` is a non-mergeable stateful operator for the join operation.

combiners), Sponge can extract this information to build the merge operators.

Compiler: Apache Nemo [51, 57] provides the intermediate representation and optimization pass abstractions, with which we can flexibly optimize application DAGs. We split our operator insertion into three separate optimization passes for inserting ROs, TOs, and MOs on Nemo to reshape the application DAG, defined by Apache Beam semantics.

Runtime: We modify the Nemo runtime [51, 57] to support the migration of tasks and the redirection of the data from VMs to SFs. Sponge executes worker processes on VM and SF instances, which each manages a thread pool that contains a fixed number of threads and assigns tasks to the threads. VM workers set up Netty [41] network channels and communicate with other VMs and SF workers, while there are no network channels set up between SF instances due to the communication constraint. For launching new VM and SF instances, as well as for deploying the worker code on AWS Lambda, we use boto3, the AWS SDK API for controlling AWS instances [14].

6 Evaluation

In our evaluation, we observe Sponge performance compared to other scaling mechanisms (§ 6.2), distinguish the factors that contribute to the Sponge performance (§ 6.3), compare the cold start latency reduction mechanisms (§ 6.4), and observe the latency-cost trade-off between SFs and VMs (§ 6.5).

6.1 Methodology

Environment. We use AWS EC2 r5.xlarge instances (32*GB* of memory and 4 vCores) as VM workers, and AWS Lambda instances as SF workers. As AWS Lambda offers one vCPU per 1,769*MB* and provides constant network bandwidth (i.e., $\sim$ 100Mbps) regardless of the instance size, we use single-core SF instances of 1,769*MB* to provision each instance with enough network bandwidth to achieve the throughput of the CPU core. VMs generally provide 10Gbps networks, which effortlessly cover the traffic generated by the CPU

Query	Stateful	State Size	# of Tasks (per Op.)	Stable input rate
Q1	X	-	120	550 K/s
Q4	O	~90 MB	60	190 K/s
Q5	O	~2.4 GB	70	19 K/s
Q6	O	~73 MB	70	230 K/s
Q7	O	~1.5 GB	90	15 K/s
Q8	O	~7 GB	60	60 K/s

Table 1: Characteristics of different NEXMark stream queries.

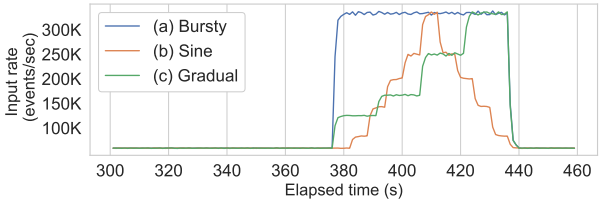


Figure 9: Examples of different bursty input patterns used in some experiments, where input rates increase at time $t = 380$. (a) shows a sudden increase from $60K$ to $300K$ ($5\times$) for 60 seconds, (b) shows a sine-curve increase and decrease, and (c) shows a gradual increase.

core throughput (i.e., $< 10\%$ bandwidth utilization when offloading 450K events/sec). We set up Amazon Virtual Private Cloud (VPC) for the data communication between the VM and SF instances for stable network connections.

Workloads. NEXMark [42] is a widely used streaming benchmark [28, 33] that analyzes auctions and bid data streams. NEXMark contains diverse stream queries with complex dataflow and stateful operations. Among the 8 (Q1-8) NEXMark queries, we choose 6 queries as shown in Table 1 because they represent distinctive data communication patterns and stateless and stateful operations. We omit Q2-3 because Q2 is a stateless query similar to Q1, and Q3 is a non-associative stateful query like Q7.

Fig. 8 illustrates the simplified original DAG of NEXMark queries, and Table 1 summaries the characteristics of the queries. The queries except for Q1 contain windowed operations. We configure the window size of queries as 60 seconds and the window interval as 1 second. While the system throughput declines with larger and more frequent windows, we evaluate under a frequent window interval to test Sponge under requirements for frequent, time-critical resource scaling. The throughput of the evaluated engine [51, 57] is similar to the performance of other stream processing engines [7, 15] when the same window size and interval are used. Nonetheless, in our evaluation, the bursty traffic is increased by up to $10\times$ events/sec and represents a wide range of realistic input rates in the field (§ 6.2).

Baseline. We compare Sponge with the following baselines:

- **NoScaling** executes stream queries on static VMs without scaling out stream queries.
- **VMBase** dynamically creates new VMs and migrates tasks

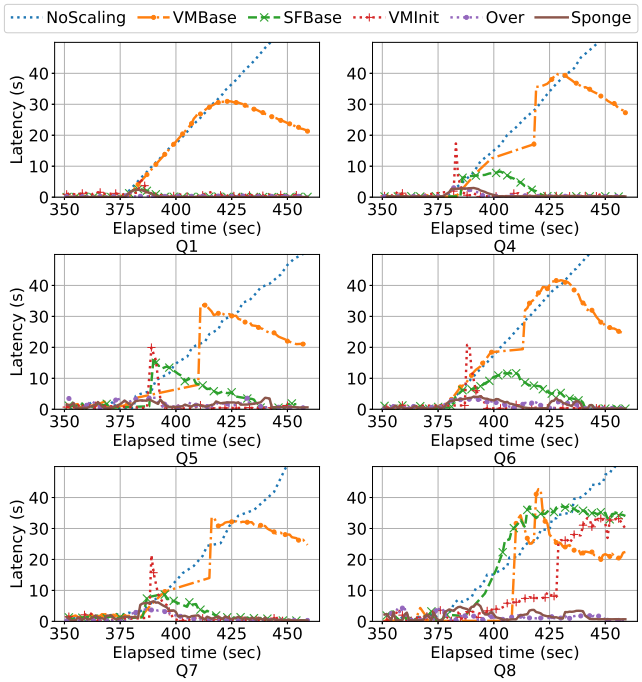


Figure 10: The tail latency graph, under a bursty load (Fig. 9(a)) at $t = 380s$ and scaling is triggered at $t = 381s$.

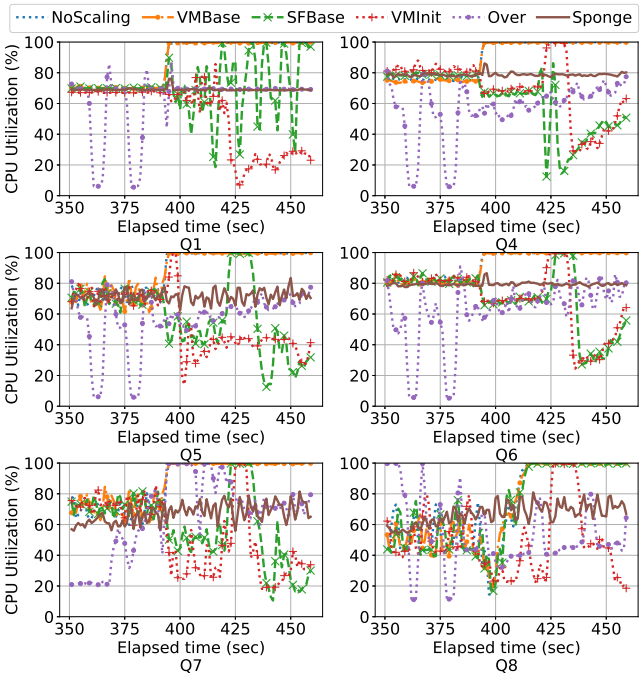


Figure 11: The CPU utilization graph, under a bursty load (Fig. 9(a)) at $t = 380s$ and scaling at $t = 381s$.

to the new VMs for scaling without dataflow reshaping.

• **SFBase** dynamically creates SF instances and migrates tasks to SFs for scaling without dataflow reshaping. For SFBase and Sponge, we prevent cold start latencies on SF workers as described in § 4.4.

• **VMInit** initializes new VMs in advance and migrates tasks to the new VMs for scaling without dataflow reshaping.

• **Over** over-provisions VMs and already has enough resources to cover input loads without dataflow reshaping.

Bursty traffic and resource allocation. We emulate bursty traffic by increasing the input rate over a short period of time, as shown in Fig. 9. In this traffic pattern, we first generate stable input streams where the input rate is stable and does not fluctuate. At a specific point (e.g., $t = 380s$ in our evaluation), we increase the input rate for a short period of time (e.g., 60 seconds) to emulate an increased load and then decrease the rate back to the stable input rate. In § 6.2, we observe the average performance of the different systems under up to $10\times$ burstiness ($\frac{\text{bursty input rate}}{\text{stable input rate}}$), and we provide detailed analysis on the effects of the burstiness rising from $3\times$ to $6\times$ in § 6.3. By default, the burstiness is set to $5\times$, as it distinctly shows the limitations of existing approaches comparatively. For example, as the stable CPU load is kept at $60-80\%$, most baselines already experience high latencies from $2\times$ burstiness, but the performance results are more clearly distinguishable between the baselines under the $5\times$ burstiness.

During the stable load, we run 5 VM workers. We generate events (per second) for the stable load such that all 5 VM workers undergo CPU usage between 60% and 80%, preventing the VM cluster from being under-loaded or overloaded. As queries have different computational complexity, the stable input rate is configured differently for each query as shown in the last column of Table 1. Once bursty loads occur, we dynamically allocate up to 200 single-core SF instances for *Sponge* and *SFBase*, and up to $50(10\times)$ new extra VM instances for *VMBase* depending on the query load.

6.2 Performance Analysis

Fig. 10 and Fig. 11 illustrate the 99^{th}-percentile tail latency and CPU utilization, respectively, of the different systems across different queries for the Burst traffic pattern in Fig. 9. Overall, *Sponge* and *Over* exhibit lower latencies compared to others during bursty periods and successfully keep the CPU utilization stable. The latency of *NoScaling* continuously increases with full CPU utilization as the existing VMs are overloaded and never perform offloading. Henceforth, we discuss Sponge and other baselines that perform scaling. For SF-based strategies that are restricted by the prohibited direct communication between SF instances, we profile their operator costs and manually configure them to make the best scaling decisions.

Sponge. Sponge reduces the tail latency on average by 88% compared to *VMBase* and 70% compared to *SFBase* and performs comparably to *Over*. Sponge also keeps the CPU utilization relatively stable across time, as shown in Fig. 11. Subsequently, we illustrate below why other scaling strategies cannot deliver the same benefits as Sponge in further detail.

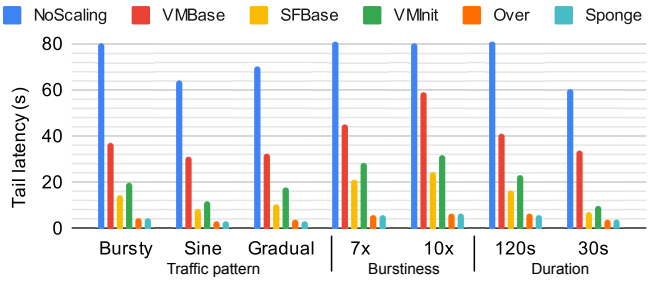


Figure 12: Summarized results of the experiments, with similar settings as in Fig. 10, displaying the average peak tail latency across the different NEXMark queries under diverse input patterns and burstiness.

VMBase. The latency of *VMBase* in Fig. 10 increases by at least 30s due to the slow start-up time of the VMs. Specifically, we observe that it takes around 25-30 seconds for the VMs to start, and around 4 extra seconds for managed runtime (i.e., JVM) worker processes to start on the newly started VMs. Moreover, as JVM processes are cold at the beginning and JIT compilation is not triggered, the processing throughput is low in the beginning, which causes extra latency of up to 44 seconds. After new VMs are instantiated, tasks are migrated to new VMs, and the latency of the VM decreases as the throughput eventually becomes larger than the input rate. Nevertheless, the CPU utilization of *VMBase* shown in Fig. 11 is continually kept high after the peak load, as it tries to climb down from the latency peak by heavily processing the data in the event queue.

SFBase. The slow start-up time of VMs can be mitigated by using SFs as shown in *SFBase*. Upon scaling out Q1 (a simple stateless query), *SFBase* significantly reduces the latency and CPU compared to *VMBase*, as the start-up time of an SF instance only takes a few hundred milliseconds in our evaluation. This result suggests that only by using SFs instead of VMs, we can significantly improve the latency for scaling out a simple stateless query, similar to MArk which handles bursty loads of stateless inference jobs [58].

However, for scaling out other complex queries with N-to-N shuffle data communications and stateful operations, the performance gain of *SFBase* compared to *VMBase* declines. It indicates that naïvely scaling queries on SFs without any operator insertion has limitations due to the challenges explained in § 3. In Q4 and Q6, latency increases by up to 12 seconds because the operators with shuffle edges cannot be redistributed to SFs and VMs become the bottleneck. In Q5, Q7, and Q8, latency and CPU spikes are caused by task and state migration overheads.

VMInit. Like *SFBase*, *VMInit* reduces the slow start-up time of VMs by starting them in advance. For *VMInit*, queries with N-to-N shuffle data communications can be offloaded, but we can see that it still incurs task and state migration overheads resulting in short steep peaks of tail latencies and CPU usage, which is highlighted in Q8.

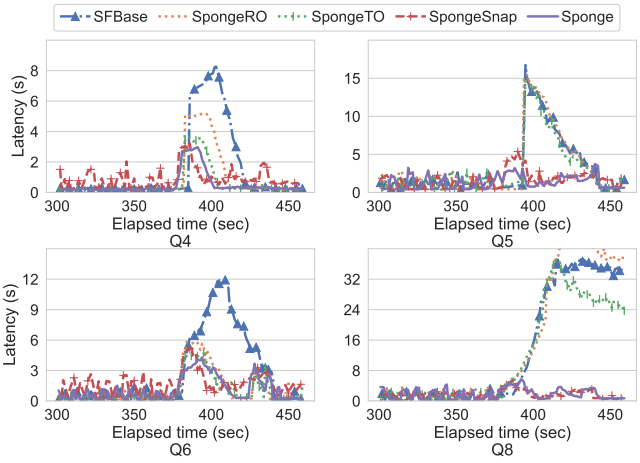


Figure 13: The latency graphs for SF, *SpongeRO*, *SpongeTO*, SpongeSnap, and Sponge to analyze and break down the performance improvements of Sponge.

Over. The over-provisioned case is the most expensive solution, providing enough resources for the peak loads without considering an upper bound for runtime costs. In Fig. 10, we can see a slight increase in latency as the input load increases, but it soon stabilizes back. The CPU usage in Fig. 11 displays an under-utilization before the peak load, but shows an adequate utilization rate afterward, as it is allocated with an adequate amount of resources for the peak load.

Input patterns. In Fig. 12, we can see the average tail latency among the different queries along the different input patterns. We can see that *Sponge* and *Over* show good performance among all settings, and *NoScaling* continuous increases in most cases. The sine and gradual bursts show a relatively mild effect compared to others, as their bursts are more gentle. We can see that while 120s and 30s bursts show somewhat similar results, 7× and 10× bursts show higher tail latencies due to the increased load.

6.3 Graph Rewriting Effect

To validate our design, we analyze the performance gain on Sponge with the following additional baselines:

- **SpongeRO** scales queries on SFs while allowing direct communication between SF instances with ROs only.
- **SpongeTO** scales queries on SFs by adding event redirection atop *SpongeRO* with ROs *and* TOs.
- **SpongeSnap** shows performance for Sponge, with ROs, TOs, and MOs, on SnapStart, without pre-warming instances.

Fig. 13 illustrates the tail latencies of *SFBase*, *SpongeRO*, *SpongeTO*, *SpongeSnap*, and *Sponge* in more detail. Q1 and Q7 are omitted in the figure, as Q1 is a simple stateless query, and Q7 is represented by Q5 and Q8.

Router operator effect. Comparing *SpongeRO* with *SFBase* shows the effect of router operators. In Q4 and Q6, SFs exhibit higher latencies as VMs are bottlenecked while processing

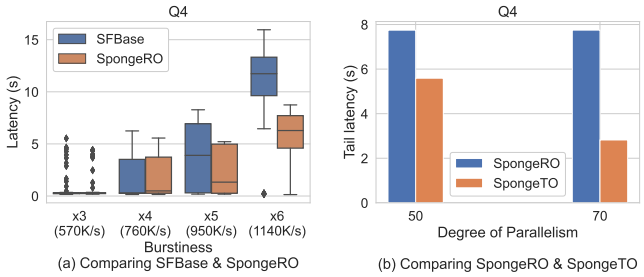


Figure 14: Comparison on Q4 for (a) *SFBase* and *SpongeRO* on diverse burstiness, and (b) *SpongeRO* and *SpongeTO* on different degrees of parallelism (# of parallel tasks).

events for `M` operators (Fig. 8) on VMs (only 3% of input events are filtered before `M2`). As naive SFs can only offload one of `M` and `GbK`, we choose to offload `GbK`, as the amount of computation on `M` is smaller than that of `GbK` due to the additional aggregation. However, the input rate of `M` on VMs becomes higher than the maximum throughput on the VMs with 5× bursts in Q4 and Q6, and events pile up in `M` operators, incurring latency increases in SFs. In Q5 and Q8, the latency of *SFBase* is similar to *SpongeRO* as VMs sufficiently handle the load on `M` operators. The main bottlenecks in Q5 and Q8 are `GbK` operators, which incur large aggregate computations. This result indicates that the RO is effective when the input rate and the overhead caused by the operators running on VMs are high.

We also evaluate when VMs become bottlenecks on `M` operators, by varying the burstiness ($\frac{\text{bursty input rate}}{\text{stable input rate}}$ from 3 to 6 in Q4 (Fig. 14(a)). In the figure, the bottom and top of the box are the 25^{th} and 75^{th} percentiles, the line indicates the median, error bars are the 95% confidence interval, and outliers are dotted as rhombi. VMs sufficiently handle 3× and 4× burstiness, and the latency of *SFBase* does not increase and is similar to *SpongeRO*. However, when the burstiness increases to 6×, VMs become the bottleneck in processing the input events of `M`. Unlike *SFBase*, *SpongeRO* adds an RO between `M` and `GbK`, and migrates both `M` and `GbK` to SFs while keeping the RO on VMs. As an RO does not (de)serialize events and does not perform computation, the amount of computation of RO is always smaller than that of `M`, and reduces latencies by up to 70%.

Transient operator effect. Transient operators enable Sponge to redirect data without stopping the workload for rescheduling. The effectiveness of transient operators increases as the number of tasks to be migrated (or redirected) increases. Q4 requires a large number of tasks to be migrated or redirected. For Q4, we had to migrate or redirect 85% of total tasks to SFs to mitigate the bottleneck in the VMs in *SpongeRO* and *SpongeTO*. *SpongeRO* takes around 2.8 seconds for migrating its tasks. In contrast, *SpongeTO* takes around 1.4 seconds for redirecting its tasks. Due to the fast redirection mechanism, *SpongeTO* additionally reduces the latency by up to 28% compared to *SpongeRO*.

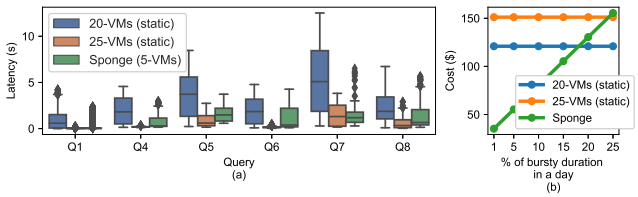


Figure 15: (a) The latency during a bursty period, and (b) a rough calculation of the cost according to the % of the bursty duration throughout the day.

When the degree of parallelism increases, the number of tasks to be migrated or redirected also increases. Fig. 14(b) illustrates the tail latency under different degrees of parallelism in Q4. With 50 parallel tasks for each operator, the task migration/redirection overhead is small, but the latency increases after the migration and redirection in both *SpongeRO* and *SpongeTO*, as a smaller degree of parallelism makes the system more prone to unevenly skewed tasks. With 70 parallel tasks, the overall latency decreases but the task migration overhead increases in *SpongeRO*. As a result, the peak latency increases by up to 8 seconds. In contrast, due to the lightweight redirection optimization, the peak latency of *SpongeTO* is kept at around 3.5 seconds, which is 56% smaller than *SpongeRO*.

Merge operator effect. Even with ROs and TOs, *SpongeTO* still suffers from high latencies in Q5 and Q8 due to the state encoding/decoding overheads. The state migration overhead is trivial in Q4 and Q6 (< 100MB), but the overhead increases with the state size. The time to encode/decode the states of Q5 and Q8 takes around 13s (for ∼2.4GB state) and 35s (for ∼ 7 GB state), respectively. As a result, the latency of *SpongeTO* increases by up to 15 and 38 seconds in Q5 and Q8. In contrast, Sponge significantly reduces the latencies to 4 seconds in Q5, and to 6 seconds in Q8, preventing state migration overheads with MOs.

6.4 Cold Start Latency Reduction Methods

In § 4.4, we describe two methods for reducing the cold start latency: by keeping warm SF instances and by using snapshots of SFs through tools like SnapStart [32]. In Fig. 13, we can see that the performance of *SpongeSnap*, which solely bases its initialization method on SnapStart [32], is slightly worse, but comparable with *Sponge*, which uses a hybrid of both methods in optimizing the managed runtime (e.g., JVM) initialization overhead. Since the overhead is reduced by more than 80% with SnapStart [32], and > 90% with warm SF instances compared to the original managed runtime initialization methods on SF instances, both methods succeed to timely supply SFs within a sub-second total start-up time.

6.5 Latency-Cost Trade-Off

The cost of using SF instances may be higher than overprovisioning VMs when the bursty input persists. In such cases, it makes more sense to launch new VMs while Sponge

handles the bursty traffic and offload our tasks to the VM. To investigate the latency-cost trade-off and figure out when it is more beneficial to launch new VMs, we compare the following two VM over-provisioning approaches with Sponge in terms of latency and cost on the workload shown in Fig. 9(a). One is *20-VMs (static)*, where 20 VMs are statically allocated without dynamic scaling, and the other is *25-VMs (static)*, where 25 VMs are statically allocated. As the default number of VMs used in *Sponge* is 5, *20-VMs* and *25-VMs* allocate 4× and 5× more VMs compared to *Sponge*, respectively.

Fig. 15(a) illustrates the latency of *20-VMs*, *25-VMs*, and *Sponge* during the bursty period. The latency of *Sponge* is in between *20-VMs* and *25-VMs*. Compared to *25-VMs*, which has enough resources to handle 5× bursty loads, Sponge has inherent scaling overheads due to the redirection and migration protocols. This is why the latency of *Sponge* is slightly higher than *25-VMs*.

In terms of cost, Fig. 15(b) shows a rough calculation of cost according to the bursty duration in a day. For instance, 1% of bursty duration represents that bursty loads happen for $24hr * 0.01$ during a day. Basically, the cost of *Sponge* is smaller than others when bursty loads occur in short durations. When the duration of the bursty load is less than 15%, *Sponge* has lower latency and cost compared to *20-VMs*. When the bursty load persists (at more than 25% in Fig. 15(b)), the cost of *Sponge* exceeds *25-VMs* due to the high cost of the SF instances. In this case, it is more beneficial to statically over-provision VM resources in terms of latency and cost. Nevertheless, as presented in existing works [36, 40], bursty loads are mostly short-lived, and persistent peaks are comparatively much rare, resulting in their duration generally falling much below 15% of the total time. Sponge provides mechanisms to initially provide prompt scaling with fast-starting SFs regardless of the peak duration and later expands the cluster to additional slow-starting VMs if the peaks persist, making the solution effective with any bursty traffic in terms of both cost and latency.

7 Related Work

To the best of our knowledge, Sponge is the first work that addresses all technical challenges described in § 3. There are some existing works that partially address the challenges, and we compare them with Sponge.

Data communication across SF instances. Researchers have exploited fast-starting SF instances for various workloads such as interactive data analytics [27, 44], video analytics [4, 21], and daily applications [20]. These applications are also represented as DAGs, and shuffle operations are required between SF instances. Their solutions to enable data communication across SF instances enable using additional VM relay servers [21], using HDFS in VMs [27], building an ephemeral storage service [31], and using a NAT-traversal technique [20]. Sponge router operators enable data communication across SF instances preserving event-based stream processing with low latency, without requiring any of the additional VM resources or NAT-traversal technique.

Optimizing state migration. Rhino [18] and ChronoStream [55] replicate states of stream queries across extra (over-provisioned) machines to minimize state migration overheads. Replicating and holding states requires costly over-provisioning of long-running resources like VMs. Holding states on SFs will cause additional state recovery and cost when SFs are reclaimed by cloud vendors. Megaphone [25] proposes fluid migration that smoothly migrates states from source to destination resources for a long period to reconfigure system configurations. However, when bursty loads happen, the reconfiguration must be executed in a short period of time. As a result, a large amount of state migration is inevitable to quickly migrate the load on VMs. In contrast, Sponge avoids state migration from VMs to SFs by just forwarding data to SFs and merging partial states in SFs into the existing VMs.

Scaling policy. Regarding scaling policies, SEEP [16], StreamCloud [24], and Dhalion [19] use metrics like CPU utilization for their decisions. Systems such as DS2 [28] aim to measure the processing and output rates of individual dataflow operators through system instrumentation. Many of these scaling policies are designed to be agnostic to the underlying scaling mechanisms and resource acquisition schemes. In contrast, the Sponge scaling policy also explicitly considers the characteristics of SF instances and offloads the right amount of computations to keep the CPU utilization high.

8 Conclusion

Sponge harnesses SF instances for offloading bursty loads from existing VMs in streaming workloads. Sponge minimizes task migration overheads and addresses data communication constraints on SF instances by inserting new stream operators in the application DAG: router, transient, and merge operators. Sponge also provides an offloading policy that determines when and how to offload the increased input loads. Our evaluation on AWS EC2 and Lambda shows that the Sponge operators are effective in significantly reducing tail latencies in stream processing upon unpredictable bursty loads, compared to existing scaling mechanisms on VMs and SFs.

Acknowledgments

We thank our shepherd Maria Carpen-Amarie and the anonymous reviewers for their feedback. This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No.2015-0-00221, Development of a Unified High-Performance Stack for Diverse Big Data Analytics), the 2023 Research Fund (1.230019) of UNIST, and Electronics and Telecommunications Research Institute(ETRI) grant funded by the Korean government [23ZS1300].

References

[1] AGACHE, A., BROOKER, M., FLORESCU, A., IORDACHE, A., LIGUORI, A., NEUGEBAUER, R., PIWONKA, P., AND POPA, D.-M. Firecracker: Lightweight Virtualization for Serverless Applications. In *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation* (USA, 2020), NSDI'20, USENIX Association, p. 419–434.

[2] AKIDAU, T., BALIKOV, A., BEKIROĞLU, K., CHERNYAK, S., HABERMAN, J., LAX, R., MCVEETY, S., MILLS, D., NORDSTROM, P., AND WHITTLE, S. MillWheel: fault-tolerant stream processing at internet scale. *VLDB Journal 6*, 11 (2013), 1033–1044.

[3] AKIDAU, T., BRADSHAW, R., CHAMBERS, C., CHERNYAK, S., FERNÁNDEZ-MOCTEZUMA, R. J., LAX, R., MCVEETY, S., MILLS, D., PERRY, F., SCHMIDT, E., AND WHITTLE, S. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-scale, Unbounded, Out-of-order Data Processing. *VLDB Journal 8*, 12 (2015), 1792–1803.

[4] AO, L., IZHIKEVICH, L., VOELKER, G. M., AND PORTER, G. Sprocket: A Serverless Video Processing Framework. In *Proceedings of the ACM Symposium on Cloud Computing* (New York, NY, USA, 2018), SoCC '18, Association for Computing Machinery, p. 263–274.

[5] Apache Commons Math. https://commons.apache.org/proper/commons-math/, 2023.

[6] ARASU, A., CHERNIACK, M., GALVEZ, E., MAIER, D., MASKEY, A. S., RYVKINA, E., STONEBRAKER, M., AND TIBBETTS, R. Linear road: a stream data management benchmark. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30* (2004), pp. 480–491.

[7] ARMBRUST, M., DAS, T., TORRES, J., YAVUZ, B., ZHU, S., XIN, R., GHODSI, A., STOICA, I., AND ZAHARIA, M. Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark. In *Proceedings of the 2018 International Conference on Management of Data* (New York, NY, USA, 2018), SIGMOD '18, Association for Computing Machinery, p. 601–613.

[8] AWS. Configuring Lambda function options, 2023. https://docs.aws.amazon.com/lambda/latest/dg/configuration-function-common.html.

[9] AWS Lambda. https://aws.amazon.com/lambda, 2023.

[10] Azure Function. https://docs.microsoft.com/en-us/azure/azure-functions/, 2023.

[11] Various Traffics in the Cloud. https://intercept.cloud/en/news/checklist-part-1-choose-your-strategy-before-you-migrate-to-azure/, 2023.

[12] Apache beam. https://beam.apache.org/.

[13] BINDSCHAEDLER, L., MALICEVIC, J., SCHIPER, N., GOEL, A., AND ZWAENEPOEL, W. Rock You like a Hurricane: Taming Skew in Large Scale Analytics. In *Proceedings of the Thirteenth EuroSys Conference* (New York, NY, USA, 2018), EuroSys '18, Association for Computing Machinery.

[14] AWS SDK for Python (Boto3). https://boto3.amazonaws.com/v1/documentation/api/latest/index.html, 2023.

[15] CARBONE, P., KATSIFODIMOS, A., EWEN, S., MARKL, V., HARIDI, S., AND TZOUMAS, K. Apache Flink™: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull. 38*, 4 (2015), 28–38.

[16] CASTRO FERNANDEZ, R., MIGLIAVACCA, M., KALYVIANAKI, E., AND PIETZUCH, P. 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* (New York, NY, USA, 2013), SIGMOD '13, Association for Computing Machinery, p. 725–736.

[17] Stream Processing with IoT Data: Challenges, Best Practices, and Techniques. https://www.confluent.io/blog/stream-processing-iot-data-best-practices-and-techniques/, 2023.

[18] DEL MONTE, B., ZEUCH, S., RABL, T., AND MARKL, V. 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* (New York, NY, USA, 2020), SIGMOD '20, Association for Computing Machinery, p. 2471–2486.

[19] FLORATOU, A., AGRAWAL, A., GRAHAM, B., RAO, S., AND RAMASAMY, K. Dhalion: self-regulating stream processing in heron. *Proceedings of the VLDB Endowment 10*, 12 (2017), 1825–1836.

[20] FOULADI, S., ROMERO, F., ITER, D., LI, Q., CHATTERJEE, S., KOZYRAKIS, C., ZAHARIA, M., AND WINSTEIN, K. From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)* (Renton, WA, July 2019), USENIX Association, pp. 475–488.

[21] FOULADI, S., WAHBY, R. S., SHACKLETT, B., BALASUBRAMANIAM, K. V., ZENG, W., BHALERAO, R., SIVARAMAN, A., PORTER, G., AND WINSTEIN, K. 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, Mar. 2017), USENIX Association, pp. 363–376.

[22] Google Cloud Function. https://cloud.google.com/functions/docs/, 2023.

[23] GU, R., YIN, H., ZHONG, W., YUAN, C., AND HUANG, Y. Meces: Latency-efficient Rescaling via Prioritized State Migration for Stateful Distributed Stream Processing Systems. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)* (Carlsbad, CA, July 2022), USENIX Association, pp. 539–556.

[24] GULISANO, V., JIMENEZ-PERIS, R., PATINO-MARTINEZ, M., SORIENTE, C., AND VALDURIEZ, P. Streamcloud: An elastic and scalable data streaming system. *IEEE Transactions on Parallel and Distributed Systems 23*, 12 (2012).

[25] HOFFMANN, M., LATTUADA, A., AND MCSHERRY, F. Megaphone: latency-conscious state migration for distributed streaming dataflows. *Proceedings of the VLDB Endowment 12*, 9 (2019), 1002–1015.

[26] ISLAM, S., VENUGOPAL, S., AND LIU, A. Evaluating the Impact of Fine-Scale Burstiness on Cloud Elasticity. In *Proceedings of the Sixth ACM Symposium on Cloud Computing* (New York, NY, USA, 2015), SoCC '15, Association for Computing Machinery, p. 250–261.

[27] JAIN, A., BAARZI, A. F., KESIDIS, G., URGAONKAR, B., ALFARES, N., AND KANDEMIR, M. SplitServe: Efficiently Splitting Apache Spark Jobs Across FaaS and IaaS. In *Proceedings of the 21st International Middleware Conference* (New York, NY, USA, 2020), Middleware '20, Association for Computing Machinery, p. 236–250.

[28] KALAVRI, V., LIAGOURIS, J., HOFFMANN, M., DIMITROVA, D., FORSHAW, M., AND ROSCOE, T. 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)* (Carlsbad, CA, Oct. 2018), USENIX Association, pp. 783–798.

[29] KALIM, F., XU, L., BATHEY, S., MEHERWAL, R., AND GUPTA, I. Henge: Intent-driven multi-tenant stream processing. In *Proceedings of the ACM Symposium on Cloud Computing* (2018), pp. 249–262.

[30] KE, Q., ISARD, M., AND YU, Y. Optimus: A Dynamic Rewriting Framework for Data-Parallel Execution Plans. In *Eurosys 2013* (April 2013), ACM.

[31] KLIMOVIC, A., WANG, Y., STUEDI, P., TRIVEDI, A., PFEFFERLE, J., AND KOZYRAKIS, C. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)* (Carlsbad, CA, Oct. 2018), USENIX Association, pp. 427–444.

[32] Lambda SnapStart. https://docs.aws.amazon.com/lambda/latest/dg/snapstart.html, 2023.

[33] LI, S., GERVER, P., MACMILLAN, J., DEBRUNNER, D., MARSHALL, W., AND WU, K.-L. Challenges and Experiences in Building an Efficient Apache Beam Runner for IBM Streams. *Proc. VLDB Endow. 11*, 12 (aug 2018), 1742–1754.

[34] LIN, M., WIERMAN, A., ANDREW, L. L. H., AND THERESKA, E. Dynamic right-sizing for power-proportional data centers. In *2011 Proceedings IEEE INFOCOM* (2011), pp. 1098–1106.

[35] LION, D., CHIU, A., SUN, H., ZHUANG, X., GRCEVSKI, N., AND YUAN, D. Don't Get Caught in the Cold, Warm-up Your JVM: Understand and Eliminate JVM Warm-up Overhead in Data-Parallel Systems. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (USA, 2016), OSDI'16, USENIX Association, p. 383–400.

[36] MAI, L., ZENG, K., POTHARAJU, R., XU, L., SUH, S., VENKATARAMAN, S., COSTA, P., KIM, T., MUTHUKRISHNAN, S., KUPPA, V., DHULIPALLA, S., AND RAO, S. Chi: A Scalable and Programmable Control Plane for Distributed Stream Processing Systems. *Proceedings of the VLDB Endowment* (2018), 1303–1316.

[37] MI, N., CASALE, G., CHERKASOVA, L., AND SMIRNI, E. Injecting Realistic Burstiness to a Traditional Client-Server Benchmark. In *Proceedings of the 6th International Conference on Autonomic Computing* (New York, NY, USA, 2009), ICAC '09, Association for Computing Machinery, p. 149–158.

[38] MIAO, H., JEON, M., PEKHIMENKO, G., MCKINLEY, K. S., AND LIN, F. X. StreamBox-HBM: Stream Analytics on High Bandwidth Hybrid Memory. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2019), ASPLOS '19, Association for Computing Machinery, p. 167–181.

[39] MÜLLER, I., MARROQUÍN, R., AND ALONSO, G. Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2020), SIGMOD '20, Association for Computing Machinery, p. 115–130.

[40] NARAYANAN, D., DONNELLY, A., THERESKA, E., ELNIKETY, S., AND ROWSTRON, A. Everest: Scaling Down Peak Loads Through I/O Off-Loading. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08)* (San Diego, CA, Dec. 2008), USENIX Association.

[41] Netty. http://netty.io/, 2023.

[42] Nexmark benchmark suite. https://beam.apache.org/documentation/sdks/java/testing/nexmark/, 2023.

[43] POTHARAJU, R., KIM, T., WU, W., ACHARYA, V., SUH, S., FOGARTY, A., DAVE, A., RAMANUJAM, S., TALIUS, T., NOVIK, L., AND RAMAKRISHNAN, R. Helios: Hyperscale Indexing for the Cloud & Edge. *Proc. VLDB Endow. 13*, 12 (Aug 2020), 3231–3244.

[44] PU, Q., VENKATARAMAN, S., AND STOICA, I. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)* (Boston, MA, Feb. 2019), USENIX Association, pp. 193–206.

[45] RAJADURAI, S., BOSBOOM, J., WONG, W.-F., AND AMARASINGHE, S. Gloss: Seamless Live Reconfiguration and Reoptimization of Stream Programs. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2018), ASPLOS '18, Association for Computing Machinery, p. 98–112.

[46] RASTEGAR, S. H., ABBASFAR, A., AND SHAH-MANSOURI, V. Rule Caching in SDN-Enabled Base Stations Supporting Massive IoT Devices With Bursty Traffic. *IEEE Internet of Things Journal 7*, 9 (2020), 8917–8931.

[47] ROBINSON, B., POWER, R., AND CAMERON, M. A Sensitive Twitter Earthquake Detector. In *Proceedings of the 22nd International Conference on World Wide Web* (New York, NY, USA, 2013), WWW '13 Companion, Association for Computing Machinery, p. 999–1002.

[48] SANDUR, A., PARK, C., VOLOS, S., AGHA, G., AND JEON, M. Jarvis: Large-scale Server Monitoring with Adaptive Near-data Processing. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)* (2022), IEEE, pp. 1408–1422.

[49] SHAH, M., HELLERSTEIN, J., CHANDRASEKARAN, S., AND FRANKLIN, M. Flux: an adaptive partitioning operator for continuous query systems. In *Proceedings 19th International Conference on Data Engineering (Cat. No.03CH37405)* (2003), pp. 25–36.

[50] SONG, W. W., JEON, M., AND CHUN, B.-G. SWAN: WAN-Aware Stream Processing on Geographically-Distributed Clusters. In *Proceedings of the 13th ACM SIGOPS Asia-Pacific Workshop on Systems* (New York, NY, USA, 2022), APSys '22, Association for Computing Machinery, p. 78–84.

[51] SONG, W. W., YANG, Y., EO, J., SEO, J., KIM, J. Y., LEE, S., LEE, G., UM, T., CHO, H., AND CHUN, B.-G. Apache Nemo: A Framework for Optimizing Distributed Data Processing. *ACM Transactions on Computer Systems (TOCS) 38*, 3-4 (2021), 1–31.

[52] UM, T., LEE, G., AND CHUN, B.-G. Pluto: High-performance iot-aware stream processing. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)* (2021), pp. 79–91.

[53] VENKATARAMAN, S., PANDA, A., OUSTERHOUT, K., ARMBRUST, M., GHODSI, A., FRANKLIN, M. J., RECHT, B., AND STOICA, I. Drizzle: Fast and Adaptable Stream Processing at Scale. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, Association for Computing Machinery, p. 374–389.

[54] WANG, L., FU, T. Z. J., MA, R. T. B., WINSLETT, M., AND ZHANG, Z. Elasticutor: Rapid Elasticity for Realtime Stateful Stream Processing. In *the ACM International Conference on Management of Data conference (SIGMOD)* (2019), ACM.

[55] WU, Y., AND TAN, K.-L. ChronoStream: Elastic stateful stream computation in the cloud. In *2015 IEEE 31st International Conference on Data Engineering* (2015), pp. 723–734.

[56] XU, D., LIU, X., AND VASILAKOS, A. V. Traffic-aware resource provisioning for distributed clouds. *IEEE Cloud Computing 2*, 1 (2015), 30–39.

[57] YANG, Y., EO, J., KIM, G.-W., KIM, J. Y., LEE, S., SEO, J., SONG, W. W., AND CHUN, B.-G. Apache Nemo: A Framework for Building Distributed Dataflow Optimization Policies. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)* (Renton, WA, July 2019), USENIX Association, pp. 177–190.

[58] ZHANG, C., YU, M., WANG, W., AND YAN, F. MArk: Exploiting Cloud Services for Cost-Effective, SLO-Aware Machine Learning Inference Serving. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)* (Renton, WA, July 2019), USENIX Association, pp. 1049–1062.