How Reliable Are Streams? End-to-End Processing-Guarantee Validation and Performance Benchmarking of Stream Processing Systems

Jawad Tahir
jawad.tahir@tum.de
Technical University of
Munich
Germany

Ruben Mayer
ruben.mayer@uni-
bayreuth.de
University of Bayreuth
Germany

Christoph Doblander
christoph.doblander@tum.de
Technical University of
Munich
Germany

Hans-Arno Jacobsen
jacobsen@eecg.toronto.edu
University of Toronto
Toronto, Canada

ABSTRACT

Stream processing systems (SPSs) provide processing guarantees to ensure reliability under failure. However, no related work exists that empirically validates these guarantees. In this paper, we present PGVal, a tool that can end-to-end validate guarantees of SPSs. Additionally, we introduce new metrics for SPSs, such as reliability, reliable throughput, and failure cost, in addition to a refined definition of latency that results in improved measurements. We benchmark three popular SPSs, namely *Kafka Streams*, *Apache Storm*, and *Apache Flink*. Our results show that the reliability of SPSs depends on many characteristics, such as data rate, data partitions, processing topology, and parallelism factor. An SPS configuration may not continue to provide reliable outputs when any of these characteristics vary. PGVal can also inject faults into SPSs to observe their impact on reliability and performance. We provide a comprehensive failure model for fault-tolerance benchmarking of SPSs and report on the impact of faults on the reliability and performance of SPSs. Our experiments show that SPSs' reliability and performance drop varies by fault. Lastly, we provide suggestions to increase the reliability and performance of these systems.

1 INTRODUCTION

Stream processing systems (SPSs) have emerged as a pivotal technology in the era of real-time analytics, addressing the critical need for processing high-velocity and high-volume data streams generated by various sources such as social media, sensors, and financial transactions. Unlike batch processing systems, which handle high-volume data in discrete chunks, SPSs process and analyze data in

real-time to provide insights such as trending information, air quality index, and stock market trends [22, 41]. The state-of-the-art SPSs can horizontally scale to parallelize the processing of large data streams. As faults are a common occurrence in any computing infrastructure [48], SPSs offer fault tolerance and provide various processing guarantees (PG), claiming reliable processing of data. The reliability of SPS outputs is of paramount concern, as SPSs are increasingly being used to make consequential decisions, for example, fraud detection and healthcare monitoring [12, 16].

The correctness verification in SPSs is scarce. CSRBench and YABench are correctness benchmarks for RDF stream engines [19, 27]. However, these systems are not distributed systems, and the distributed nature of SPSs can affect their correctness. Correctness in SPSs is generally left to theoretical models and lacks empirical validation [36, 42]. Akidau et al. argued that configuring an SPS for latency, throughput, and correctness is a zero-sum game [2]. Furthermore, infrastructure faults are prone to cause data losses [3], yet no correctness evaluation of SPSs under faults exists to the best of our knowledge. Lastly, correct outputs do not ensure reliability, as we demonstrate the difference between correctness and reliability in Section 2.3.

Performance benchmarking of SPSs is an active research area, given their widespread adoption in the industry [1, 4, 8, 14, 26, 40, 46, 47]. Some of the related works provide fault-tolerance evaluations of SPS [32, 47, 49], but their failure model is limited to process failures, which, in reality, is just wishful thinking. Empirical studies have shown that network failures are a common occurrence in a compute infrastructure [23, 35, 48]. Additionally, network partitions are a given in any distributed system [9, 10]. Accurate performance measurement is another challenging task in the SPS context. Throughput can not be easily defined for SPSs as there exists a many-to-one mapping between inputs and outputs. Additionally, we show that the current definition of SPS throughput may give inaccurate measurements [26]. Also, latency measurement in a distributed system is a challenge due to the non-availability of a global clock.

In this paper, we present a tool called PGVal. It benchmarks SPSs for end-to-end processing-guarantee validation and performance under failures. It divides benchmarks into three phases: control, failure, and recovery. It establishes a baseline in the control phase and records SPS reliability and performance under the failure and recovery phase for comparative analysis. PGVal measures reliability by comparing the produced outputs with the correct outputs provided by an oracle. Furthermore, PGVal measures end-to-end

performance to provide a common ground for evaluations instead of relying on system-reported metrics. We propose a failure model for SPSs that can be used in further research to exhaustively assess SPSs' fault tolerance capabilities. We introduce new metrics for SPSs, such as reliability and failure cost, and refine the definitions of throughput and latency of SPSs. Lastly, we benchmark *Kafka Streams*, *Apache Storm*, and *Apache Flink* for reliability and performance during failures using PGVal. Our experiments show that, in practice, SPS reliability is sensitive to characteristics such as data rate, data partitions, processing topology, and parallelism factor. Furthermore, a fault's reliability and performance impact varies with SPSs, and SPSs handle process crashes better than network faults. The contributions of this paper are:

(1) *Failure model for SPSs.* The current failure model for SPSs is limited to process failures only. We provide a comprehensive failure model for SPS fault-tolerance benchmarking that covers process and network failures. Our experiments show that SPSs may behave differently during network failures compared to process failures, highlighting the need for network failures in the model.

(2) *Definition of reliability, reliable throughput, latency, and failure cost in the SPS context.* We introduce new metrics for SPSs to quantify reliability and failure cost. Reliability represents the percentage of data reliably processed, and failure cost helps practitioners configure time-outs for their systems. Our refined definitions of throughput and latency result in an accurate performance depiction of SPSs.

(3) *Reliability and performance benchmarking of SPSs under failures.* We comprehensively benchmark three widely-used SPSs for reliability and performance under failures. This work is the first of its kind to empirically validate PGs of SPSs. Our experiments helped us find a bug in Apache Storm[1]. Additionally, we provide suggestions for different SPSs to increase reliability. Lastly, we provide an extended version of the paper hosted on the artifact's GitHub repository for complete results [38].

(4) *PGVal.* We release our open-source processing-guarantee validation and performance benchmarking suite for SPSs [39]. Due to its modular architecture, it can be extended to further datasets and topologies by implementing an oracle to provide correct outputs. It can help practitioners measure the reliability and performance of their SPS configuration.

The rest of this paper is structured as follows. Section 2 provides a background on SPSs, fault tolerance, and reliability. Section 3 discusses PGVal system design. Section 4 summarizes our experimental results. Section 5 summarizes our insights, and Section 6 describes the related work.

2 BACKGROUND

2.1 Stream Processing Systems

SPSs abstract the data to a flow of events and can parallelize the processing topology to cater to high-volume and high-velocity data. The number of parallel topology instances is called the parallelism factor. A topology is a blueprint of processing logic and is defined

---

Table 1: Stream operator classification

State	No. of inputs	No. of outputs	Operator
Stateless	Single	Single	Transformation
Stateless	Single	Multiple	Group-by
Stateless	Multiple	Single	Union
Stateless	Multiple	Multiple	Custom
Stateful	Single	Single	Window
Stateful	Single	Multiple	Custom
Stateful	Multiple	Single	Join
Stateful	Multiple	Multiple	Custom

by a directed acyclic graph of stream processing operators. Operators define the processing logic for incoming events. Operators have three attributes: state, number of input streams, and number of output streams. Table 1 lists all possible combinations of state capability and the number of input and output streams, and it shows the corresponding operators. A transformation operator is a stateless single-input, single-output (SISO) operator. Map and filter operators are examples of transformation operators. Group-by is a stateless single-input, multi-output (SIMO) operator as it can create multiple streams based on different values of group-by attribute. Union is a stateless multi-input, single-output (MISO) operator as it redirects multiple input streams to one output stream. There isn't an individual operator for stateless multi-input, multi-output operations (MIMO), but a custom operator can be implemented by concatenating a union and group-by operators. A window operator is a stateful SISO operator that performs operations on events collected over a time window. Aggregators are a form of window operators. There isn't an individual operator for stateful SIMO operations, but a custom operator can be implemented by concatenating a window and group-by operators. A join operator joins multiple input streams into one over a time window and is a stateful MISO operator. There is no individual operator for stateful MIMO operations, but a custom operator can be implemented by concatenating a join with a group-by operator. Additionally, SPSs also provide special operators called source and sink to read and write data from external systems, respectively. Source and sink are stateful MISO and SIMO operators, respectively as they read from and write to multiple data partitions.

Stateful operators may have temporal constraints to keep the state size manageable. To specify temporal constraints, SPSs typically support two types of temporal semantics: *event time* and *processing time*. Event time is the notion of time at the event generation. On the other hand, processing time is the notion of time at the event processing. Event-time semantics allow the processing results to be deterministic and reproducible. SPSs keep track of event time using watermarks, where watermarks are the progression of event time during processing. The results of processing time semantics are neither deterministic nor reproducible. Hence, all topologies used in this work operate on event time semantics.

Events generally arrive at a data source from multiple data producers, and events generated at one global instant of time may arrive at the data source out-of-order. Furthermore, events coming from one data source may also arrive at an SPS out-of-order due to the distributed nature of SPS. In event time semantics, this situation may cause temporal operators (windows, joins) to produce

an incomplete output. To counter this issue, these operators can be configured to wait for a fixed delay before emitting an output. However, this configuration does not ensure correct results as the delay is set heuristically, and events may still arrive after the delay. Some SPSs provide an option to configure a side stream so that all late events can be collected and processed separately [37].

2.2 Fault tolerance of SPSs

It is essential to distinguish between faults and failure. A fault is an event that can induce an abnormal behavior, or a failure, in a system [24]; in other words, a fault is a *cause* and the failure is its *effect*. Many types of faults can occur in a computer system, ranging from network delays to node crashes. Consequently, SPSs employ failure recovery mechanisms to provide fault tolerance. Failure recovery depends on the type of operator. In the case of a stateless operator, such as transformation or filter, the operator is simply restarted, and it starts processing the new events. Stateful operators, such as window or join, require a sophisticated approach to recover from failures. SPSs store the operator state in an external system to properly handle failures in stateful operators. The operator is restarted when it encounters a failure, and it restores the last stored state to ensure correct results. Here, we explain the two stateful recovery mechanisms the benchmarked systems utilize.

Replaying state updates: In this technique, the state updates are stored chronologically, forming a changelog. Any operation that updates the state will result in a new entry in the changelog. In case of a failure, all the state update operations are reapplied to reach the state just before the failure.

Snapshotting: In this technique, a state snapshot is taken at regular intervals (also called checkpoints) and the snapshot is stored in an external system. In case of a failure, the last persisted snapshot is reapplied, and the system's state is restored to the state before the failure [13, 30].

2.3 Correctness vs. Reliability

Correctness, in the SPS context, refers to the correctness of the outputs, and reliability refers to the correctness of processing. An SPS may produce a correct output without processing all input events. Take the trending topic topology in DSPBench as an example [8]. This topology computes trending topics from tweets. It consumes a stream of tweets, extracts topics from the tweets, counts the occurrences of topics in a time window, ranks these topics, and outputs the trending topics. Assume $e\langle id, ts, t\rangle$ represents a tweet with id about topic t created at timestamp ts, and $o\langle w, t, r, c\rangle$ represents the output that ranks topic t at rank r with c occurrences in a time window of w. Now, consider a scenario where an SPS running this topology experiences a failure such that it omits event $e\langle id_1, ts_1, t_1\rangle$ and processes event $e\langle id_2, ts_2, t_1\rangle$ twice to produce an output $o\langle w, t_1, r, c\rangle$, where ts_1 and $ts_2 \in w$. Let us also assume that we have an oracle that provides us with the correct output $s\langle w, t_1, r', c'\rangle$. Now, if we compare the produced output $o\langle w, t_1, r, c\rangle$ with its correct output $s\langle w, t_1, r', c'\rangle$, we can misconstrue that the SPS reliably processed all input events under the failure as $r = r'$ and $c = c'$, although the system did not process $e\langle id_1, ts_1, t_1\rangle$ and processed $e\langle id_2, ts_2, t_1\rangle$ twice. To ensure reliability, we need to observe which inputs were processed to compute an output. We can solve this problem by augmenting the output with a list of all input IDs l that were processed to generate that output, which can be represented as $o\langle w, t, r, c, l\rangle$ and $s\langle w, t, r', c', l'\rangle$. Now, if one compares l and l', the list of input IDs that *should* be processed to generate the correct output, we can accurately determine the reliability of the SPS.

SPSs offer PGs to ensure reliable processing under failures. *At most once* (M1) is the weakest PG as it tries to process every event but some events may not be processed. Applications that can tolerate a few missing events, such as calculating an air quality index from sensor measurements [41], may use M1. *At least once* (A1) is a more restrictive guarantee than M1. It implies that no event is unprocessed, but an event may be processed more than once. Applications that can tolerate a few duplicate events, such as a notification system [7], may use A1. *Exactly once* (E1) is the most restrictive PG. Exactly once implies that every event will affect the output exactly once, and there are no unprocessed events. Applications that cannot tolerate duplication and missed events, such as fraud detection [12], use E1.

It is important to note that these PGs do not guarantee correct outputs. Instead, these PGs just guarantee the correct processing of input events. To ensure end-to-end correct outputs, data sources and sinks must also provide corresponding delivery guarantees. For example, an SPS consuming events from a non-replayable source (taking measurements directly from a sensor) may not be able to provide A1 or E1 as the events during the failure-recovery phase may not be available to the SPS after the recovery.

3 SYSTEM DESIGN

In this section, we discuss the topologies designed for PGVal. After that, we explain our benchmarking methodology, the failure model, and our synthetic dataset. Finally, we explain important system metrics that we collect in our benchmarks.

3.1 Topology

There exist many topologies for SPS performance benchmarking, such as fraud detection, word count, and trending topics on social media [8, 26, 45]. However, these topologies are inadequate for PG validation as they don't preserve the IDs of all input events that were processed to generate the output (l). This raises the need for a new topology that generates outputs enriched with l so outputs can be mapped to inputs. We modified the trending topics topology to create a new resource demand monitoring topology to cater to a new use case (Figure 1). We call it a single-stream topology (SST) as this topology processes one event stream. However, SST lacks multi-stream processing to make it representative of complex queries. We modify SST further to create a multi-stream topology (MST) that includes multi-stream operators shown in Table 1. SST consumes
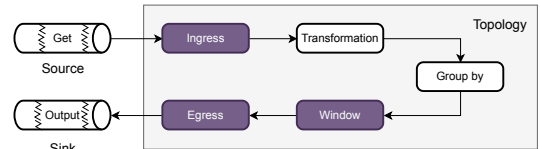

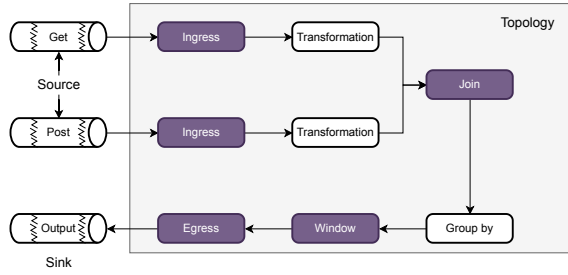**Figure 1: Single-stream topology (SST)**

Figure 2: Multi-stream topology (MST)

the event log of a web server that serves web resources (pictures, videos, web pages). The server receives GET requests for web resources, and SST computes the number of GET requests received in a time window for each resource to monitor the demand for web resources. We removed the ranking operator as it works as a filter and it produces only top-k elements. If we don't receive an input event in l, we cannot ascertain whether it is caused by filtering or PG violation. The topology consumes GET events through a source operator (ingress) from a data source. The ingress operator doubles as a union operator, merging streaming data from multiple data partitions into one stream. The transformation operator performs marshaling on input events. These events are then grouped by resources and are fed to a window operator that counts the number of requests of each resource over a specific time period. These counters are then forwarded to a sink operator (egress) that emits this information to a data sink. The stateful operators are darkly shaded in the figure.

The second topology, MST, employs a join operator and processes multiple streams to make our benchmarking broader (Figure 2). This topology lacks only a stateless MIMO operator. MST consumes and processes two event streams (GET and POST requests). Then, it joins these streams on a web resource in a time window. This joined stream is then grouped by web resources and sent to a window operator that counts the number of times a web resource is accessed and updated in a time window. These insights are then forwarded to a sink operator that emits this information to a data sink.

The topologies use a Kafka cluster as a data source and sink. Kafka is a performant, scalable, fault-tolerant, and distributed event streaming platform [21]. Events in Kafka are logically stored in topics, which can be partitioned to be distributed among Kafka instances for load balancing. These partitions can also be replicated to provide fault tolerance. Kafka tags events with ingress time, the time of insertion in Kafka. We create two topics, get and post, to store web requests. We create three partitions of the topics and

Figure 3: PGVal architecture

replicated them thrice, which is the minimum number of replications required to provide fault tolerance in Kafka. Kafka supports M1, A1, and E1 delivery guarantees, which are required by SPSs to provide corresponding PGs as explained in Section 2.3. Kafka provides data localization such that all inputs and outputs of one resource may be stored in one partition.

3.2 Benchmark Methodology

We separate SPS from PGVal to mitigate the effects of benchmark processing on stream processing and to modularize the PGVal architecture. This design decision allows for extending PGVal to other SPSs and topologies. The system architecture is shown in Figure 3. Components developed for PGVal are highlighted with a white background, while off-the-shelf components are marked with pink. The data generator persists events in a Kafka cluster, which the SPS consumes. The SPS processes these events and writes the results back to the same Kafka cluster. We configure the data generator, Kafka cluster, and our topology so that input events and their outputs for a given resource are sent to the same Kafka instance. The oracle consumes input events to compute expected outputs and share them with the PG Validation module. This module reads the produced outputs and compares them with their expected outputs to measure reliability and throughput. The latency calculator reads inputs and the produced outputs to measure latency. Lastly, the fault injector injects configured faults into the SPS.

3.2.1 Oracle. To verify reliable data processing, the produced outputs o must be compared with the correct outputs s. We cannot assume SPS outputs are correct for benchmarking. Therefore, we implemented a reference SPS called Oracle to provide correct outputs. Oracle lacks standard SPS functionalities, such as parallelization, fault tolerance, sliding windows, and late-event processing. Instead, it uses a single process with fixed time windows. A single process ensures no synchronization errors, assuming no faults exist in Oracle. It consumes input events and processes them according to the defined topology logic. Unlike SPSs with sliding windows, we use fixed-time windows for input events. Orcale shares correct output with the PG Validation module to measure reliability. Oracle's correctness is ensured by defining test cases. PGVal can be extended by implementing an oracle to calculate expected outputs s and modifying the topology to include input IDs l in the output. Data generators can be extended for further datasets. Note that implementing Oracle is challenging, and we spent considerable time developing test cases to ensure the correct implementation of topologies.

3.2.2 Dataset. Several datasets for web server logs exist [5, 15, 29, 33, 50]. However, these datasets are unsuitable for PG validation as the correct outputs of these events are not known a priori. Instead, we used a synthetic dataset designed to mimic real-world datasets. This synthetic dataset aids in developing test cases for Oracle. We curate a web server log dataset so that the outputs are deterministic and are compared with Oracle's output for correctness. Our dataset consists of two types of events: GET events and POST events. Each event contains a unique ID, an event timestamp, an ingress timestamp, and the resource's name. The event timestamp in GET events corresponds to web resource access times, while in POST events, it
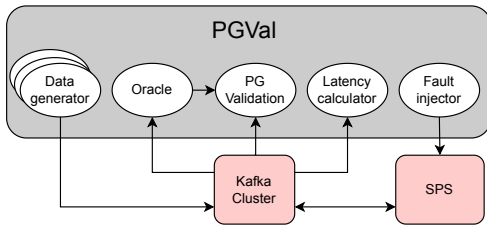
Table 2: Distribution of get and post requests in datasets

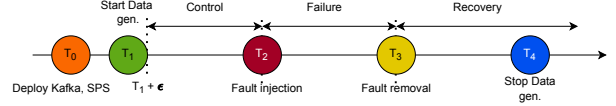| Dataset	Get requests	Post requests
Zaker et al. [50]	99%	1%
Lagopoulos et al. [29]	99.5%	0.5%
Maňásek et al. [33]	97%	2%
Chodak et al. [15]	95%	4%
Our dataset	99.5%	0.5%

corresponds to web resource update times. The ingress timestamp shows the timestamp at which the event is persisted on the data source.

This dataset is disseminated by a configurable data generator. An SPS may consume a stream faster than its slowest operator processes it, causing event queues that grow over time, increasing latency. To counter this, we throttle data generator rate $Rate_{DG}$ to the optimum ingestion rate (OIR), the maximum rate at which latency remains stable [26]. Multiple generator instances can be spawned to mimic real-world scenarios, analogous to event logs of a load-balanced application. Events' timestamps generated by one generator are monotonically increasing. The generator can be configured to generate GET and POST requests for a configured number of resources, with all resources equally represented. However, the distribution of GET and POST events is skewed, as observed in real-world datasets (Table 2). We configure our dataset such that POST events make up 99.5% of the dataset, comparable to other real-world datasets. The datasets produced by Zaker et al. [50] and Chodak et al. [15] are server logs of e-commerce stores, whereas those by Lagopoulos et al. [29] and Maňásek et al. [33] are from the academic domain.

3.2.3 Selected Frameworks. Various SPSs cater to different use cases. We shortlisted three widely-used, open-source SPSs for benchmarking: *Kafka Streams* (KStreams), *Apache Storm* (Storm), and *Apache Flink* (Flink). Their differences are tabulated in Table 3. KStreams, a client library built on Apache Kafka, operates on a peer-to-peer architecture where each instance works independently. It does not process out-of-order events and discards late events. KStreams uses idempotent writes and the two-phase-commit protocol to provide end-to-end A1 and E1, respectively. In case of failure, KStreams recovers by reapplying state updates. Storm operates on a controller-worker architecture, offering out-of-order events redirection for both join and window operators. It provides A1 using acknowledgments [43] but does not offer end-to-end A1. Although Storm can provide end-to-end E1 through the Trident API, we did not use it due to its lack of event-time semantics, which is crucial for deterministic and reproducible outputs. Flink, a distributed processing engine for streams and batched events, also uses a controller-worker architecture. Flink's window operator can redirect out-of-order events to a side stream, but the join operator discards them. Flink ensures fault tolerance through the Chandy-Lamport snapshotting algorithm [13]. In case of failure,

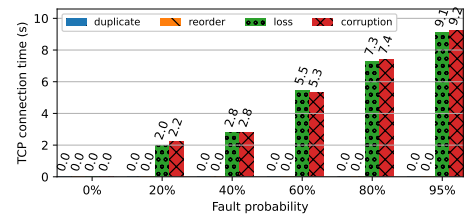**Figure 4: Benchmark workflow**

Flink redeploys the topology and resets each operator to the last stored snapshot (checkpoints). Flink provides end-to-end A1 and E1.

3.2.4 Benchmark Workflow. We perform multiple steps in each experiment, visualized in Figure 4. We deploy a Kafka cluster and a topology on a selected SPS at T_0, and start our data generator at T_1. A fault is injected into the SPS at T_2 and removed at T_3. The data generator is shut down at T_4, and the SPS runs until there are no more output events. The experiment is divided into three phases: The *control* phase is the period before fault injection (T_2). An SPS needs some time (ϵ) to stabilize. Metrics are collected ϵ seconds after the data generator starts, with ϵ observed to be around 5 seconds for both topologies. The *failure* phase is when the fault is active in the SPS. The *recovery* phase is from T_3, when the fault is removed until the last output is generated. We configured our workflow such that $T_2 - (T_1 + \epsilon) = T_3 - T_2 = T_4 - T_3 = 90$ seconds.

3.3 Failure Model

Coulouris et al. identified processes and networks as fundamental entities of a distributed system [17]. However, current related works limit their fault-tolerance evaluations of SPSs to process failures, ignoring network failures. Our failure model includes both network and process failures. Network failures can cause packet duplication, reorder, loss, corruption, and delay [6]. We do not need to evaluate SPSs against all network failures since they communicate over TCP, a reliable network transmission protocol [20, 34]. TCP's congestion control and error correction mechanisms ensure that applications receive correct data, albeit with increased delay, unless a network partition occurs. Thus, network failures in a reliable network reduce packet delays.

To demonstrate network faults' effects on TCP, we conducted chaos experiments on TCP connection time, as shown in Figure 5. Chaos engineering is a systematic study of a system's performance during failures [28]. We used `tc`, a Linux utility, to emulate network faults [25]. `tc` also allows configuring the probability of fault occurrence at a given time. Under normal conditions, a TCP connection takes a fraction of a millisecond to establish. As the probability of network faults increases, so does the connection time for packet

Table 3: Selected SPSs

Framework	Version	Architecture	out-of-order events processing	Fault recovery mechanism	PGs
Kafka Streams	3.6.1	Peer-to-peer	-	State updates	A1, E1
Apache Storm	2.6.0	Control-worker	Window, Join	Acknowledgments	A1
Apache Flink	1.18.0	Control-worker	Join	Snapshotting	A1, E1

Figure 5: TCP connection times under various probabilities of network fault occurrence (timeout = 10s)

loss and corruption, since TCP must run a checksum algorithm and resend missing or corrupted packets. When the probability approaches 100%, the connection times out, configured to ten seconds in our experiments. Packet duplication and reordering do not significantly affect connection times, as TCP discards duplicate messages and reorders packets appropriately within a batch without retries.

3.4 Collected Metrics

3.4.1 Reliability. Reliability in the stream processing context is offered through PGs. We benchmark reliability through end-to-end PG validation, which not only ensures both correct outputs and processing. This validation is crucial since SPSs consume data from external systems. We compare the produced output o with its expected output s to measure reliability, as illustrated in Algorithm 1. The algorithm takes three parameters: (1) a produced output by an SPS o, (2) the expected output computed by the oracle s, and (3) a list of previously processed IDs *processed*, initially empty. If an output is produced more than once, this list will contain the IDs from previous outputs. The algorithm iterates over the IDs in *produced.list* and searches for each ID in the *expected.list*. If an ID is not found, it indicates that the input event belongs to another output, marking the output as incorrect. If found, the ID is checked in the previously processed list; if it is a duplicate, it is noted; otherwise, it is added to the list. This algorithm is invoked for each output.

Algorithm 1 Process an output

---

procedure PROCESSOUTPUT (*produced, expected, processed*)
    for *id* **in** *produced.list* **do**   ▷ From SPS (o)
        if *id* **in** *expected.list* **then**   ▷ From Oracle (s)
            if *id* **in** *processed* **then**
                $duplicate \leftarrow duplicate + 1$
            else
                $processed.add(id)$
        else
            $incorrect \leftarrow incorrect + 1$

Out-of-order events may not be processed, and their IDs will not appear in the produced output. If the system routes these events to a side stream, we consider it reliable as no data is lost. These late events can be processed separately using Algorithm 2, which takes two parameters: (1) the late event, (2) previously processed IDs of input events contributing to the corresponding output. The algorithm searches the ID of each late event in the previously processed IDs list. If found, it is a duplicate; otherwise, the ID is added to the list.

Algorithm 2 Process a late event

---

procedure PROCESSLATE (*late, processed*)
    if *late.id* **in** *processed* **then**
        $duplicate \leftarrow duplicate + 1$
    else
        $processed.add(id)$

Algorithms 1 and 2 identify all input events processed by the system. To find unprocessed events, we execute Algorithm 3 with

two parameters: (1) the expected output computed by Oracle s, (2) a list of previously processed input event IDs. This algorithm iterates over the expected output list and checks each ID against the previously processed list. If an ID is not found, the event is not processed. After running these algorithms, we validate end-to-end PGs. If ($unprocCount = 0 \wedge incorrect = 0$), the system provides A1, indicating no event was left unprocessed and no incorrect outputs were found. If ($unprocCount = 0 \wedge incorrect = 0 \wedge duplicate = 0$), the system provides E1, indicating no event was left unprocessed, no incorrect outputs were found, and no event was processed more than once.

Algorithm 3 Count unprocessed events

---

procedure COUNTUNPROCESSED (*expected, processed*)
    for *id* **in** *expected.list* **do**   ▷ From Oracle (s)
        if *id* **not in** *processed* **then**
            $unprocCount \leftarrow unprocCount + 1$

3.4.2 Latency. Latency is generally calculated by subtracting the input record's ingress timestamp from its corresponding output record's ingress timestamp. However, accurate latency calculation in a distributed system is a challenge due to the non-availability of a global clock and clock synchronization issues. Modern computer systems use NTP to synchronize their clock [30], but this does not eliminate the clock skew problem. The clock skew may be negligible if computers are colocated, but as the deployment of an SPS can span over continents, the clock skew can reach the order of tens of milliseconds [11]. In an SPS, an event can be ingested by one machine, and its output can be generated by a different machine. If two computers cannot agree on the same time, their timestamps cannot be correlated to measure accurate latency. We side-stepped this issue by exploiting the data localization feature of Apache Kafka. The input and output events of one web resource are sent to the same broker in the Kafka cluster. We can subtract the input event's ingress timestamp from the output event's ingress timestamp to calculate the latency, as these timestamps are taken at the same machine.
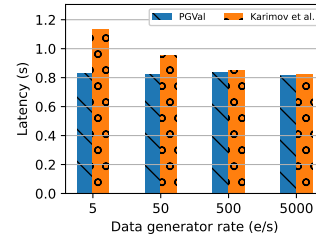
Figure 6: Latency measurements with varying date rate

Another question we have to answer is which input event's timestamp should we subtract from the output's timestamp for latency calculation? Our topology has aggregation operators (windows), so there is a many-to-one mapping between input and output events. If we subtract the timestamp of the first input event of a window from its output's timestamp, i.e., $\min_{ts}(input.ts \geq output.window.start)$, the latency would contain the time it took to fill the window, which can vary by $Rate_{DG}$. Karimov et al. [26] suggested that the last input event's timestamp should be subtracted from the output's timestamp for accurate latency measurement, i.e., $\max_{ts}(input.ts \leq$

$output.window.end$). However, our experiments on KStreams show that this method may still contain window filling times (Figure 6). Window operators do not start processing events when they consume the last event of the window. Instead, they start processing the events when they consume the first element of the next output, i.e., $\min_{ts}(input.ts > output.window.end)$. For each output event, PGVal gets the timestamp of the first input event whose timestamp is greater than the output's window end. This insertion time is then subtracted from the output's insertion time to calculate the latency. PGVal helps to eliminate the effects of $Rate_{DG}$ on the latency measurements, as shown in Figure 6. Karimov's algorithm is affected by $Rate_{DG}$ and may reduce the accuracy of measurements at a lower data generation rate.

3.4.3 Reliable throughput. Throughput is generally defined as the number of items a system processes per unit of time. However, aggregate operations make throughput calculation non-trivial. Simply tallying the outputs produced doesn't give a full picture because it overlooks the number of input events that were processed to produce those outputs. Since it takes multiple input events to generate a single output, relying solely on output count can lead to a misleading understanding of the system's performance. Karimov et al. measured throughput as the events *consumed* by an SPS [26]. However, if we count events consumed, an SPS that consumes input events at a high rate but does not produce any output would report a high but incorrect throughput as it did not produce anything at all. We define reliable throughput as the unique input events *processed* per unit of time, i.e., $\sum^{o \in outputs}\{o.list\}/t$. For all output events per unit of time, we count the unique IDs in their ID lists. We don't include duplicates in our throughput measurement as an SPS that repeats only one output would have a high throughput, even though it has only processed a few input events.

3.4.4 Failure cost and others. A system may have to perform certain tasks during the failure recovery that may hamper its processing, resulting in an increased latency. We define failure cost as the maximum increase in latency by a failure. This metric may help practitioners define an upper bound on time-outs for their systems. Furthermore, we collect all SPS and node metrics, such as CPU and memory consumption.

4 RESULTS

In this section, we present the results obtained from our experiments. For each SPS, we first run control experiments where the metrics are observed with no faults. All SPSs are comparably configured, such as state and heap size, and every instance of an SPS is run on a dedicated node. Every node contains 8 cores of Intel Xeon E5-2630, 20 GBs of RAM, and 35 GBs of SSD storage. These nodes are connected with a 10 Gbps network link. We run the SPS with varying degrees of parallelism factors and the number of data producers. We run the single-stream topology (SST) and multi-stream topology (MST) with the available PGs for all SPSs. We configure the data generator to operate at the OIR of the SPS. We then run chaos experiments where a specified fault is injected into one instance of the SPS through Pumba, which is a fault injection tool for containers [31]. We evaluate SPSs for fault tolerance with process crashes and packet delays. Furthermore, packet delays (PD) can

be either higher than the configured time-outs (TO), i.e., PD > TO, or they can be smaller than the configured time-outs (PD < TO). Every experiment is performed multiple (at least three) times to mitigate the effect of instantaneous aberrations, and the average is reported. For each experiment, we report the reliability percentage (the percentage of input events reliably processed), reliable throughput (which we will refer to as throughput in the following sections), and latency. Lastly, we provide an extended version of the paper hosted on GitHub which contains the results of additional experiments (Section 4.4) [38]. In this paper, we present only the most interesting results.

4.1 Kafka Streams

4.1.1 Control experiments. The results of control experiments are shown in Figure 7.

Reliability. Our experiments revealed that KStreams provides reliability for SST only and fails to ensure reliability for MST under any parallelism. This unreliability stems from its inability to process late events. Events may arrive out-of-order from two streams, and KStreams discards late events, which produces outputs with missing data. Increasing parallelism improves KStreams' reliability, as shown in Figures 7a to 7c. This improvement occurs because data is distributed across different instances, reducing the probability of out-of-order events. However, increasing parallelism from three to four did not enhance reliability further (Figures 7c and 7d) due to the number of data partitions. KStreams scales with the number of data partitions, leaving any extra worker idle. We also benchmarked topologies with single and multiple producers to assess their impact on performance and reliability. Experiments with parallelism of
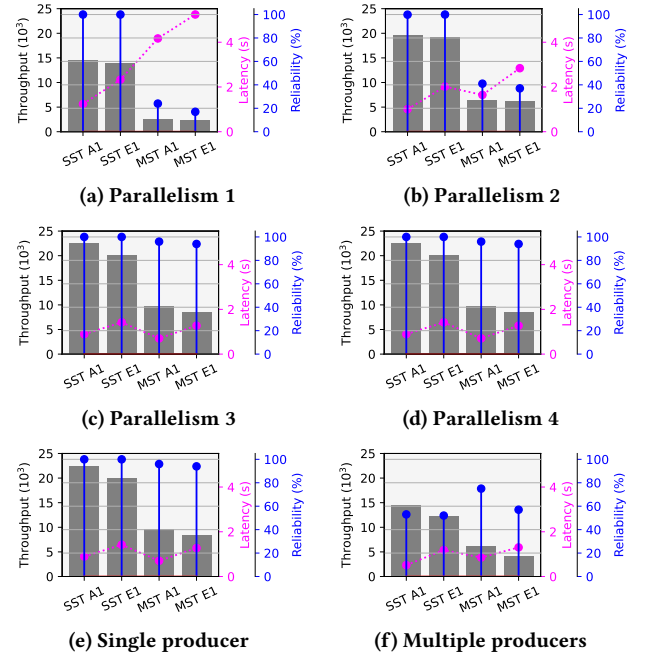

(a) Parallelism 1   (b) Parallelism 2

(c) Parallelism 3   (d) Parallelism 4

(e) Single producer   (f) Multiple producers

Figure 7: KStreams performance and reliability across various configurations in control experiments

Table 4: Failure cost of SPSs under various faults

Topology	PG	Fault	Failure cost (s)		
			KStreams	Storm	Flink
SST	A1	PD < TO	28	164	54
SST	A1	PD > TO	3	168	71
SST	A1	Process crash	2	427	78
SST	E1	PD < TO	27	-	54
SST	E1	PD > TO	3	-	66
SST	E1	Process crash	4	-	65
MST	A1	PD < TO	29	168	67
MST	A1	PD > TO	2	178	71
MST	A1	Process crash	3	440	81
MST	E1	PD < TO	27	-	58
MST	E1	PD > TO	2	-	69
MST	E1	Process crash	3	-	75

three, offering the best performance, showed KStreams fails to provide correct results with multiple producers, even in SST. This drop in reliability is again due to out-of-order events, with reliability decreasing proportionally to the probability of out-of-order events. Doubling the number of producers in Figure 7f resulted in a 50% reliability drop.

Throughput. KStreams increases system throughput by increasing parallelism. Our benchmarks showed that KStreams increased throughput by 55% (14.5K vs. 22.5K) and 284% (2.5K vs. 9.6K) for SST and MST, respectively, by increasing parallelism from one to three. The significant throughput increase in MST is also attributed to increased reliability. In contrast, SST reliability remained constant (100%), resulting in a less steep increase. The throughput difference between SST and MST was at least 57% (22.5K vs. 9.6K in Figure 7c). E1 may cause throughput to drop by up to 10% (22.5K vs. 20K in Figure 7c) compared to A1 due to the synchronous two-phase-commit protocol. Lastly, multiple producers decrease KStreams' throughput due to the drop in reliability.

Latency. Increasing parallelism reduced latency by up to 25% (1.25 vs. 0.9 s) and 83% (4.16 vs. 0.7 s) for SST and MST, respectively, as shown in Figures 7a to 7d. This reduction highlights the computational expense of join operators. Both topologies exhibit similar latency at the highest parallelism, but MST has significantly higher latency at lower parallelism levels. Additionally, E1 increase latency by 50% (2.35 vs. 1.24 s in Figure 7a) due to the synchronous two-phase-commit protocol. Lastly, the number of producers did not significantly affect latency.

4.1.2 Chaos experiments. We benchmarked KStreams for reliability and performance under the defined faults with parallelism three,

(a) PD < TO

(b) PD > TO

(c) Process crash
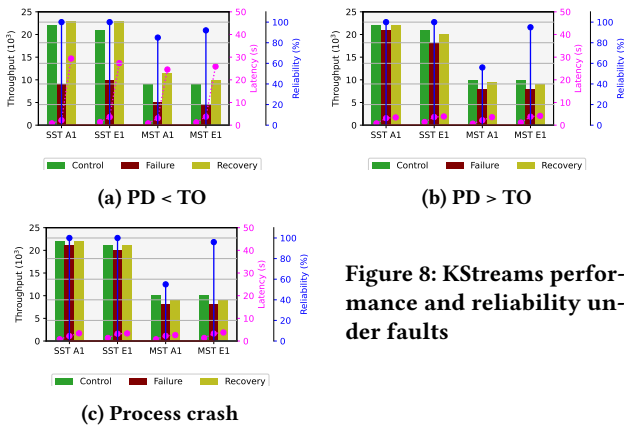
Figure 8: KStreams performance and reliability under faults

(a) PD < TO

(b) PD > TO

(c) Process crash
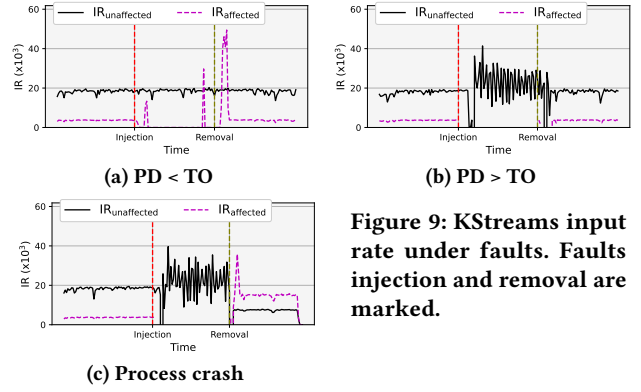
Figure 9: KStreams input rate under faults. Faults injection and removal are marked.

as it offers the best performance. The results of our benchmarking can be seen in Figure 8 and Table 4. The following observations can be drawn from our results.

Reliability. Reliability results are similar to the control experiments. KStreams provides reliability for SST only, and faults may cause the reliability of MST to drop by half (Figure 8c). Another counterintuitive observation is the reliability drop for PD > TO and process crash faults (Figures 8b and 8c) is higher than the reliability drop in PD < TO. This drop can be investigated by observing the input rate of KStreams (Figure 9). The input rate of the nodes where we inject failure, IR_{affected} (dashed line), and the input rate of the rest of the system, $IR_{\text{unaffected}}$ (solid line), are measured separately. When we inject PD < TO, the affected node does not timeout and continues processing events but fails to perform significant processing (Figure 9a). However, when we inject PD > TO and process crashes, KStreams detects a failure and transfers the failed nodes' tasks to unaffected nodes, which is observed by an increase in the $IR_{\text{unaffected}}$ (Figures 9b and 9c). In this scenario, KStreams has fewer workers than data partitions, and a worker may have to process two data partitions, which increases the probability of out-of-order events. Hence, lower reliability is observed when we inject PD > TO and process crashes.

Latency. Latency increases 2-3 folds in the failure phase for all faults. Latency increase in the recovery phase is conditioned on the fault induced. For PD < TO, the latency is observed to increase up to 11 folds. Comparatively, a negligible increase in latency is experienced for PD > TO and process crashes. This behavior can be understood by looking at KStreams input rates during their respective failures (Figure 9). For PD < TO, unaffected nodes continue to consume inputs and produce outputs at the optimal rate in the failure phase, but the affected node throttles consuming inputs and producing outputs. Hence, a slight increase in latency in the failure phase. The unaffected nodes continue to perform optimally in the recovery phase, and the affected nodes resume producing outputs in the recovery phase of events produced during the failure phase. It causes latency to increase proportionally to the duration of the failure phase. It also explains the comparatively higher failure cost for PD < TO in Table 4. As the failure phase is 30 seconds long, KStreams may produce outputs with latency as high as 30 seconds. For PD > TO and process crashes, the tasks of affected nodes are transferred to the unaffected nodes, causing a slight increase in latency. KStreams rebalances tasks after the fault is removed and

continues to perform optimally, resulting in a slight increase in latency.

Throughput. Throughput is also observed to follow the latency trend. For PD > TO and process crashes, a slight decrease (3% - 7%) in throughput is experienced in the failure phase due to the rebalancing of tasks and lower compute resources. In the recovery phase, KStreams rebalances the tasks and continues to perform optimally, causing the throughput to recover quickly. For PD < TO, the affected nodes neither process events nor release control of the tasks to allow rebalancing. Hence, the data of the affected nodes is simply not processed in the failure phase. It also causes an increase in throughput in the recovery phase, as the events in the failure phase are also processed in the recovery phase.

4.1.3 Takeaways. KStreams provides reliable data processing as long as the topology does not contain a join operator. KStreams scales with data partitions and offers the highest reliability and performance when the parallelism factor is equal to the number of data partitions. Furthermore, it can not appropriately handle out-of-order events. Fault-tolerance evaluations show that the instance experiencing network delay should be removed to achieve the best possible reliability and performance.

4.2 Apache Storm

4.2.1 Control experiments. Storm allows late events to be processed through a side channel. We experienced a system crash whenever Storm received an out-of-order event. We investigated this issue and found a bug in the Storm source code. We reported this bug to the Storm community. The bug has not been resolved, and we have not used this side channel in our topologies. All experiments use A1, the only PG provided by Storm.

Reliability. Our experiments to determine the OIR of Storm revealed that its reliability varies with $Rate_{DG}$ (Figures 10a to 10c). Storm demonstrated the highest reliability, 98%, at $Rate_{DG}$ of 16K. Reliability decreased as $Rate_{DG}$ increased, even though throughput

increased. At $Rate_{DG}$ of 66K, Storm achieved its highest throughput with 86% and 70% accuracy for SST and MST, respectively, which we adopted as the OIR for further experiments. Increases in $Rate_{DG}$ reduced reliability and throughput, with reliability dropping to 19% and 0.1% at 92K for SST and MST, respectively. This decrease in reliability by increasing data rate is caused by out-of-order events. The higher data rate causes the ingestion operator to read larger batches, which contain events from multiple windows. Once a time window of one resource crosses the watermark, all windows of the same time period of other resources are processed even though their events are still behind the watermark. This phenomenon increases the number of out-of-order events and reduces the reliability.

We also vary the parallelism levels of the topologies (Figures 10d to 10f). Reliability increases with parallelism due to data partition distribution among different machines, reducing event intermingling and out-of-order events. SST reliability increases from 84% to 93%, and MST reliability from 70% to 86%. We use parallelism of three for chaos experiments, as it offers the best reliability and performance. Unlike KStreams, Storm did not achieve 100% reliability with parallelism three. We investigated this issue and found that Storm and KStreams use different hashing algorithms. KStreams relies on the `murmur2` hashing algorithm, and Storm uses the `SHA256` hashing algorithm. This results in events from different partitions being processed at one machine, causing out-of-order events and reducing reliability. Storm did not provide 100% reliability in any controlled experiment due to multiple partitions. Having multiple partitions causes the events to arrive out of order at an instance. Storm only provided 100% reliability when we replaced multi-partitioned data with single-partition data (Figures 10g and 10h).

Throughput. Storm's throughput increases with $Rate_{DG}$ up to a point (Figures 10a and 10b). Beyond that, higher out-of-order events lead to discarded events, reducing throughput (Figure 10c). This experiment highlights the advantage of using reliable throughput. Using Karimov et al.'s definition [26], our measurements would report higher throughput, even though Storm produced minimal outputs and most input events consumed were discarded. Throughput is increased by increasing parallelism. This increase in throughput is attributed to increased reliability.

Latency. Storm demonstrated the lowest latency (180 ms) among all SPSs. The latency of SST does not change by varying $Rate_{DG}$ (Figures 10a to 10c), as expected by our definition of latency. However, the latency of MST is affected by varying data rates. We investigated this issue and found it is caused by two consecutive window operators in MST. For MST, Storm first joins two streams over a time period, and once it reaches its watermark, the join operator produces an output. However, this output does not cross the watermark of the second window operator. The window operator waits for the second output from the join operator that crosses its watermark, i.e., the outputs of Storm are two windows behind the input events. As our latency measurements subtract one window length, the latency contains the window-filling time. Hence, the latency of MST is varied by $Rate_{DG}$. No effect is observed on latency by varying parallelism. Latency increases by 35% (300 vs. 190 ms) when consuming data from a single partition as compared to multi-partitioned data.
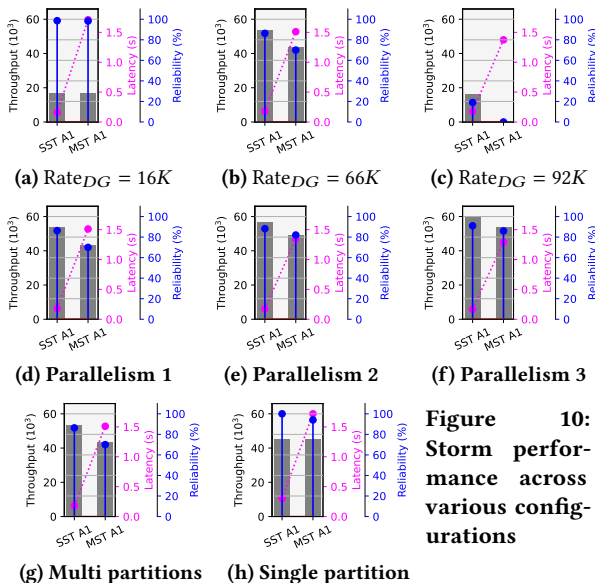

(a) $Rate_{DG} = 16K$ **(b)** $Rate_{DG} = 66K$ **(c)** $Rate_{DG} = 92K$

(d) Parallelism 1   **(e) Parallelism 2**   **(f) Parallelism 3**

(g) Multi partitions   **(h) Single partition**

Figure 10: Storm performance across various configurations

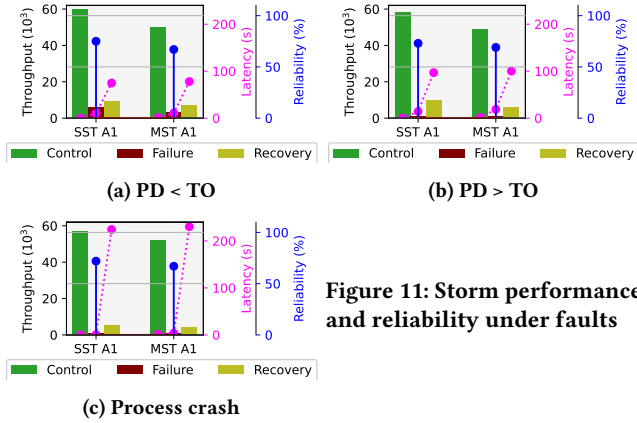**(a) PD < TO**

(b) PD > TO

(c) Process crash

Figure 11: Storm performance and reliability under faults

4.2.2 Chaos experiments. The results of our chaos testing of Storm are shown in Figure 11 and Table 4. Storm has the highest failure cost among all benchmarked SPSs.

Reliability. Storm did not provide 100% reliability for any control experiment. Unsurprisingly, similar behavior is observed for all chaos experiments, and all faults impact the reliability (Figure 11). Storm reported a drop of 20% (from 91% to 72%) for SST and 22% (from 86% to 68%) for MST.

Throughput. Storm's throughput dropped by 90% (60K vs. 6K) in the failure phase for PD < TO, and even more severely by 98% (60K vs. 1K) for PD > TO and process crashes. During the recovery phase, Storm showed a slight improvement, reaching 10K for PD < TO and PD > TO, while for process crashes, it hovers around 5K. This behavior can be explained by Storm's output rate (Figure 12). Storm's throughput drops significantly in the failure phase. A fault causes the reshuffling of tasks. After the fault is removed, Storm resumes processing, but events being processed just before the fault must first time out, leading to a long tail of outputs and extending the recovery phase, which reduces throughput. Storm doesn't immediately resume processing after fault resolution (Figure 12c), taking over six minutes to restart worker processes. This delay also contributes to a high failure cost for process crashes (Table 4).

Latency. Storm reported about a 90x increase (14 vs. 0.16 s) in latency for SST and about a 12x increase (18.70 vs. 1.45 s) for MST in the failure phase. However, it pales in front of the latency we experience in the recovery phase. The average latency observed in

(a) PD < TO
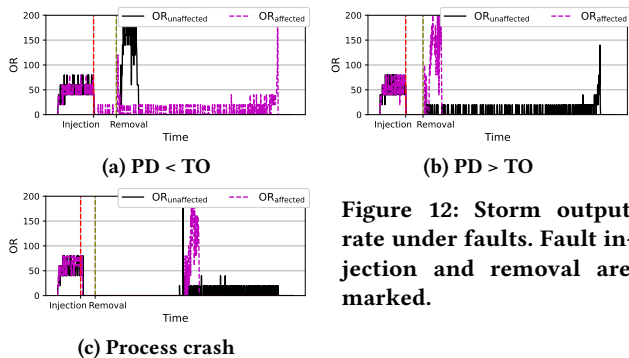
(b) PD > TO

(c) Process crash

Figure 12: Storm output rate under faults. Fault injection and removal are marked.

the recovery phase was around 230 seconds for process crashes, a whopping 1400X increase in latency. For PD < TO and PD > TO, the latency increase was comparatively better at 600X. These inflated latencies are attributed to a slow restart of the worker process and acknowledgment timeouts.

4.2.3 Takeaways. Storm offers the lowest latency, making it ideal for low-latency streaming applications. However, its reliability is among the lowest we tested, primarily due to the bug we discovered. If resolved, this could improve Storm's reliability. Additionally, Storm performs better when ingesting unpartitioned data. Fault-tolerance evaluations reveal a high failure cost for Storm.

4.3 Apache Flink

4.3.1 Control experiments. Control experiment results are shown in Figure 13.

Reliability. Varying parallelism in Flink has no effect on its reliability, in contrast to KStreams and Storm (Figure 13). Flink produces correct outputs for all topology-PG combinations with a single producer. We also benchmarked the topologies with multiple producers to see the effects of out-of-order events on performance and reliability. Out-of-order events do not cause a drop in SST reliability, but they reduce correctness. Out-of-order data may cause a window operator to process its events prematurely, causing incorrect outputs. However, Flink provides the functionality to reroute out-of-order events to a side channel. We collect these events and mark them against their input events. This allows Flink to process complete data even for out-of-order events. However, the reliability of MST with multiple producers is reduced. This behavior is expected

(a) Parallelism 1

(b) Parallelism 2

(c) Parallelism 3

(d) Parallelism 4

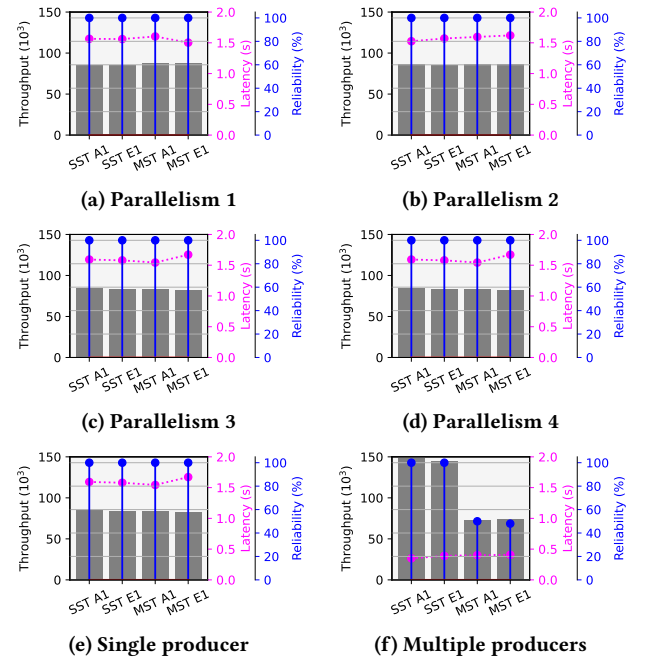**(e) Single producer**

(f) Multiple producers

Figure 13: Flink performance and reliability across various configurations in control experiments

as the additional join operator does not offer a side channel for late events.

Throughput. The parallelism configuration doesn't affect throughput, which remains constant at 92K events per second, even at higher parallelism levels. However, this doesn't mean Flink fails to scale with parallelism. This is due to the $Rate_{DG}$ limit, as a single data generator can produce only 92K events per second. Even with a parallelism factor of 1, Flink operates faster than the maximum $Rate_{DG}$. A faster data generator would result in higher throughput. Flink processes events concurrently, unlike KStreams, which processes events sequentially, resulting in higher throughput for Flink. SST's throughput increases with multiple producers, showing that Flink can handle more data than a single producer generates. Additionally, Flink's side channel reroutes out-of-order events, further increasing throughput. SST in Flink achieved 160K events per second, the highest among all benchmarks. MST saw a slight drop in throughput due to the absence of a side channel for out-of-order events in join operators, causing premature window firing and more late events.

Latency. The latency remains constant at around 1.5 seconds in all our experiments. Even changing the PG to E1 does not increase the latency, as Flink relies on an asynchronous snapshotting algorithm to provide E1. The latency is observed to drop for multiple producers. However, this drop is not due to high performance. In fact, this drop in latency is attributed to premature processing of window operators due to out-of-order events.

4.3.2 Chaos experiments. Flink achieved peak performance with multiple producers but did not provide full reliability. It demonstrated reliability only with a single producer. Hence, we conduct all chaos experiments with a single producer to study reliability. Insights for throughput and latency can be scaled for a high-performance producer. The results of our chaos testing are shown in Figure 14 and Table 4.

Reliability. Flink provides the highest reliability across all faults. However, in PD > TO and process crash (Figures 14b and 14c), reliability is not 100%. We observed that Flink restarts the current processing job for these faults. After the restart, we experienced duplicate events in the first few outputs, violating E1 and reducing reliability (Figures 15b and 15c). We configured PGVal to read
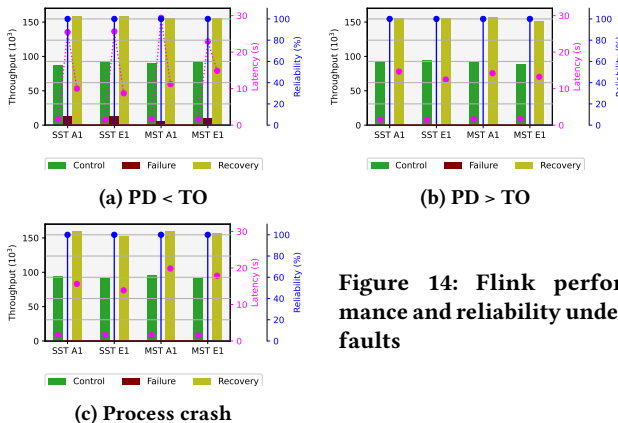

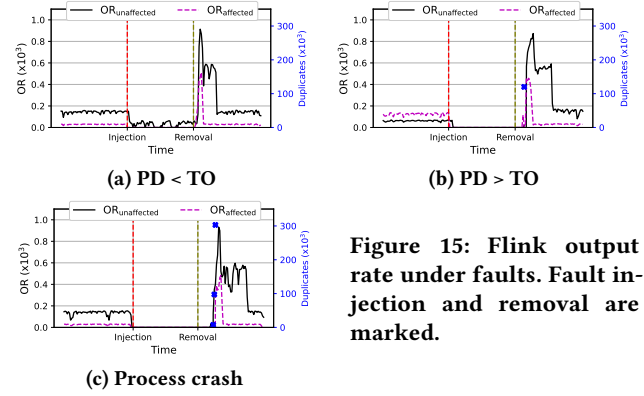
(a) PD < TO   (b) PD > TO

(c) Process crash

Figure 15: Flink output rate under faults. Fault injection and removal are marked.

only committed outputs, so any uncommitted output at fault injection is not read. This implies that Flink consumed, processed, and produced some events multiple times despite correct checkpoint restoration.

Throughput. Flink's highly synchronized architecture halts the entire system even when only part is affected (Figure 14). Throughput drops by 90% (90K vs. 9K) for PD < TO, while for PD > TO, Flink stops processing completely. Examining the input rate Figure 15 explains this behavior. For PD < TO, the affected worker halts, but others continue at a throttled rate, causing a 90% throughput drop (Figure 15a). For other faults, even unaffected workers stop, causing throughput to drop to zero (Figures 15b and 15c). Flink restarts the job during recovery and quickly restores optimal throughput. This restart time is visible in Table 4.

Latency. Flink exhibited high latency during the failure phase for PD < TO. Despite only a 500 ms delay, latency increased 15x (from 1.6 to 25.6 seconds), driven by a drop in throughput. While Flink achieves optimal performance in the recovery phase, events generated during the failure phase are processed later, raising the average latency. For PD > TO and process crashes, latency is undefined in the failure phase since no output is produced.

Flink supports over-provisioning and configuring standby workers for fault tolerance. We provisioned one extra worker beyond the parallelism factor to determine if standby workers could take over tasks from failed ones. As shown in Figure 16, the results for PD < TO remained unchanged. However, for PD > TO, Flink reassigned the failed worker's task to the standby worker, limiting the throughput drop to 4% (92K vs. 88K) and ensuring uninterrupted processing during failures. Process crashes yielded similar results to PD > TO.

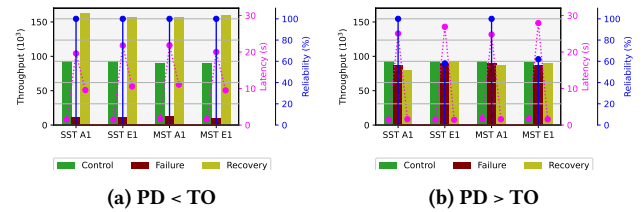
(a) PD < TO   (b) PD > TO

Figure 16: Flink performance and reliability under faults with standby workers

(a) PD < TO   (b) PD > TO

(c) Process crash

Figure 14: Flink performance and reliability under faults

4.3.3 Takeaways. Flink delivers the highest throughput and reliability among all evaluated SPSs. However, Flink lacks reliability for multi-stream topologies. Fault-tolerance evaluations show that Flink halts processing entirely even if part of the infrastructure fails. To ensure uninterrupted processing during failures, Flink should be provisioned with more workers than the defined parallelism factor, allowing standby workers to replace faulty ones. Alternatively, deploying Flink with a resource provider like Kubernetes ensures continuous processing, as Kubernetes automatically replaces failed pods [47].

5 DISCUSSION

Our experiments show that SPS reliability depends on many factors such as data partitions, data rate, parallelism, and topology. An SPS that delivers reliable outputs may fail to do so if any of these factors change, as it may cause events to be out-of-order. Our experiments highlight the need to process late events as there are a plethora of reasons why events may arrive out-of-order. Therefore, all SPSs should implement side channel functionality to prevent data loss. Additionally, join operators in a topology often result in unreliable outputs. Our results also indicate that each system has pros and cons, and the functional and non-functional requirements—such as correctness, throughput, latency, parallelism, data partitions, data rate, and fault profile—should be considered when selecting an SPS. Lastly, all SPSs achieve the highest reliability when the parallelism matches the number of partitions.

Our work also raises the need to consider network faults in the failure model instead of only relying on process crashes. We observed that SPSs respond to process and network failures differently. Additionally, our work raises the need for end-to-end PGs. SPSs do not exist independently and always integrate with other external systems. A system is as reliable as it outputs. Results produced by a system that generates incomplete or incorrect outputs cannot be relied on for data-sensitive applications. Furthermore, our experiments highlight the benefits of our refined metric definitions. The reliable throughput correctly pointed out a throughput drop at a higher data rate for Storm. We would have observed a high throughput if we had used the previous definitions. Similarly, our latency metric definition helped us get fine-grained latency measurements at all data rates. Lastly, had we relied on just correctness, we would have reported that Flink produced correct results. Our reliability metric helped us capture PG violations.

To conclude, we cross-compare selected frameworks to highlight their unique strengths for specific use cases. KStreams demonstrates the lowest failure cost, making it well-suited for fault-prone infrastructures. Storm excels in delivering minimal latency, which is ideal for low-latency applications, though it sacrifices reliability. Flink offers the highest throughput and reliability, positioning it as the optimal choice for data-intensive applications where both performance and fault tolerance are critical.

Future direction. Despite our earnest efforts to make our evaluation as representative as possible, the results are inherently limited to the specific topologies and the dataset used in this study. To this end, PGVal can be extended to benchmark additional topologies by implementing an oracle and modifying the existing topologies to track input events.

6 RELATED WORK

The work related to this paper falls in two categories: verification and performance benchmarking. The verification of SPSs' outputs is scarce in the literature. Most of the work relies on theoretical models to verify correctness [36, 42]. CSRBench and YABench empirically measure the correctness of SPSs [19, 27]. However, they benchmarked previous-generation SPSs, which are not able to scale horizontally. No work has been done to verify the correctness of distributed SPSs. Furthermore, failures can also cause data loss and corruption, and yet no work has been done to verify the correctness of SPSs under failure. Lastly, empirical PG validation has never been tried before.

Performance benchmarking of SPSs is an active research area. Lopez et al. [32] benchmarked SPSs for throughput. They also injected machine crashes into SPSs to test their fault-tolerance capabilities. Karimov et al. [26] benchmarked SPSs for throughput and latency. They highlighted the importance of SPS separation from the benchmarking utilities. They also defined several terms, such as latency and OIR. However, we demonstrated in this paper that their definitions may result in coarse-grained measurements at lower data rates. Bordin et al. [8] developed DSPBench, consisting of 15 topologies to benchmark SPSs. The topologies designed for PGVerifier are inspired by their trending topic topology. Van Dongen et al. [46] developed OSPBench that can measure throughput and latency. They highlighted the problem of latency measurement in a distributed environment. They proposed the concept of end-to-end latency to counter this issue, in which they used one machine for input and output records. However, they did not exploit data locality. Hence, their benchmark would produce coarse-grained measurements in the case of a multi-broker Kafka cluster. In their subsequent work [44], they also benchmarked various DSPSs under process crashes. However, their failure model was limited to process crashes. Tahir et al. [40] developed a benchmark for fault-tolerance evaluation of streaming solutions submitted for the ACM DEBS Grand Challenge [18, 22, 41]. Their work is limited to performance measurement of SPS under faults. Agnihotri et al. [1] developed PSDP-Bench to benchmark Flink for operator parallelization. Most recently, Vogel et al. [47] benchmarked SPSs for failure recovery and measured the impacts of failure recovery on performance in a cloud-native environment. However, their failure model was still limited to process crashes, and they measured only performance.

7 CONCLUSIONS

In this work, we developed an open-source benchmarking suit, PG-Val, that can validate end-to-end PGs and measure the performance of SPSs. PGVal can also inject faults into SPSs to observe their reliability and performance under failures. We benchmarked three open-sourced SPSs (Kafka Streams, Apache Storm, and Apache Flink) for performance, reliability, and failure cost. Our experiments show that the reliability of an SPS relies on many factors, such as data rate, data partitions, processing logic, and parallelism factor. Furthermore, every SPS has its own strengths and weaknesses under different failures. Lastly, we provided various suggestions to improve reliability and decrease failure costs based on our experimental evaluation.

REFERENCES

[1] Pratyush Agnihotri, Boris Koldehofe, Roman Heinrich, Carsten Binnig, and Manisha Luthra. 2024. PDSP-Bench: A Benchmarking System for Parallel and Distributed Stream Processing. (2024), 22 pages. Accepted at TPCTC.

[2] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, et al. 2015. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-Of-Order Data Processing. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1792–1803.

[3] Ahmed Alquraan, Hatem Takruri, Mohammed Alfatafta, and Samer Al-Kiswany. 2018. An analysis of {Network-Partitioning} failures in cloud systems. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, 51–68.

[4] Arvind Arasu, Mitch Cherniack, Eduardo Galvez, David Maier, Anurag S. Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. 2004. Linear road: a stream data management benchmark. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30* (Toronto, Canada) *(VLDB '04)*. VLDB Endowment, 480–491.

[5] Martin F. Arlitt and Carey L. Williamson. 1996. Web Server Workload Characterization: The Search for Invariants. *SIGMETRICS Perform. Eval. Rev.* 24, 1 (may 1996), 126–137. https://doi.org/10.1145/233008.233034

[6] Peter Bailis and Kyle Kingsbury. 2014. The Network Is Reliable: An Informal Survey of Real-World Communications Failures. *Queue* 12, 7 (jul 2014), 20–32. https://doi.org/10.1145/2639988.2655736

[7] Snehasis Banerjee and Debnath Mukherjee. 2013. Towards a Universal Notification System. In *2013 IEEE/WIC/ACM International Joint Conferences on Web Intelligence (WI) and Intelligent Agent Technologies (IAT)*, Vol. 3. IEEE, 286–287.

[8] Maycon Viana Bordin, Dalvan Griebler, Gabriele Mencagli, Claudio F. R. Geyer, and Luiz Gustavo L. Fernandes. 2020. DSPBench: A Suite of Benchmark Applications for Distributed Data Stream Processing Systems. *IEEE Access* 8 (2020), 222900–222917. https://doi.org/10.1109/ACCESS.2020.3043948

[9] Eric Brewer. 2012. Cap Twelve Years Later: How the "Rules" Have Changed. *Computer* 45, 2 (2012), 23–29. https://doi.org/10.1109/MC.2012.37

[10] Eric A. Brewer. 2000. Towards Robust Distributed Systems (Abstract). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing* (Portland, Oregon, USA) *(PODC '00)*. Association for Computing Machinery, New York, NY, USA, 7. https://doi.org/10.1145/343477.343502

[11] M Caporaloni and R Ambrosini. 2002. How Closely Can a Personal Computer Clock Track the UTC Timescale via the Internet? *European Journal of Physics* 23, 4 (jun 2002), L17–L21. https://doi.org/10.1088/0143-0807/23/4/103

[12] Fabrizio Carcillo, Andrea Dal Pozzolo, Yann-Aël Le Borgne, Olivier Caelen, Yannis Mazzer, and Gianluca Bontempi. 2018. Scarff: A Scalable Framework for Streaming Credit Card Fraud Detection with Spark. *Information fusion* 41 (2018), 182–194.

[13] K. Mani Chandy and Leslie Lamport. 1985. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems* 3, 1 (Feb 1985), 63–75. https://doi.org/10.1145/214451.214456

[14] Sanket Chintapalli, Derek Dagit, Bobby Evans, Reza Farivar, Thomas Graves, Mark Holderbaugh, Zhuo Liu, Kyle Nusbaum, Kishorkumar Patil, Boyang Jerry Peng, and Paul Poulosky. 2016. Benchmarking Streaming Computation Engines: Storm, Flink and Spark Streaming. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, Chicago, IL, USA, 1789–1792. https://doi.org/10.1109/IPDPSW.2016.138

[15] Grzegorz Chodak, Grażyna Suchacka, and Yash Chawla. 2020. HTTP-Level E-commerce Data Based on Server Access Logs for an Online Store. *Computer Networks* 183 (Dec. 2020), 107589. https://doi.org/10.1016/j.comnet.2020.107589

[16] Rudyar Cortés, Xavier Bonnaire, Olivier Marin, and Pierre Sens. 2015. Stream Processing of Healthcare Sensor Data: Studying User Traces to Identify Challenges from a Big Data Perspective. *Procedia Computer Science* 52 (2015), 1004–1009.

[17] George F Coulouris, Jean Dollimore, and Tim Kindberg. 2005. Distributed Systems: Concepts and Design. *DISTRIBUTED SYSTEMS* (2005), 67–70.

[18] Luca De Martini, Jawad Tahir, Christoph Doblander, Sebastian Frischbier, and Alessandro Margara. 2024. The DEBS 2024 Grand Challenge: Telemetry Data for Hard Drive Failure Prediction. In *Proceedings of the 18th ACM International Conference on Distributed and Event-based Systems*. ACM, Villeurbanne France, 223–228. https://doi.org/10.1145/3629104.3672538

[19] Daniele Dell'Aglio, Jean-Paul Calbimonte, Marco Balduini, Oscar Corcho, and Emanuele Della Valle. 2013. On Correctness in RDF Stream Processor Benchmarking. In *The Semantic Web–ISWC 2013: 12th International Semantic Web Conference, Sydney, NSW, Australia, October 21-25, 2013, Proceedings, Part II 12*. Springer, 326–342.

[20] W. Eddy. 2022. *RFC 9293: Transmission Control Protocol (TCP)*. RFC Editor, USA.

[21] Apache Software Foundation. 2024. Apache Kafka. https://kafka.apache.org/

[22] Sebastian Frischbier, Jawad Tahir, Christoph Doblander, Arne Hormann, Ruben Mayer, and Hans-Arno Jacobsen. 2022. Detecting Trading Trends in Financial Tick Data: The Debs 2022 Grand Challenge. In *Proceedings of the 16th ACM International Conference on Distributed and Event-Based Systems*. ACM, Copenhagen Denmark, 132–138. https://doi.org/10.1145/3524860.3539645

[23] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. 2011. Understanding network failures in data centers: measurement, analysis, and implications. (2011), 350–361. https://doi.org/10.1145/2018436.2018477

[24] Vassos Hadzilacos and Sam Toueg. 1994. *A Modular Approach to Fault-Tolerant Broadcasts and Related Problems*. Technical Report. Cornell University. 86 pages.

[25] Bert Hubert. 2001. tc(8) - Linux manual page. Retrieved December 22, 2024 from https://man7.org/linux/man-pages/man8/tc.8.html

[26] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, and Volker Markl. 2018. Benchmarking Distributed Stream Data Processing Systems. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, Paris, 1507–1518. https://doi.org/10.1109/ICDE.2018.00169

[27] Maxim Kolchin, Peter Wetz, Elmar Kiesling, and A Min Tjoa. 2016. *YABench: A Comprehensive Framework for RDF Stream Processor Correctness and Performance Assessment*. Lecture Notes in Computer Science, Vol. 9671. Springer International Publishing, Cham, 280–298. https://doi.org/10.1007/978-3-319-38791-8_16

[28] Mathias Lafeldt and GU Yu. 2016. Principles of Chaos Engineering. https://principlesofchaos.org

[29] Athanasios Lagopoulos and Grigorios Tsoumakas. 2019. Web Robot Detection - Server Logs. https://doi.org/10.5281/ZENODO.3477932

[30] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (Jul 1978), 558–565. https://doi.org/10.1145/359545.359563

[31] Alexei Ledenev. 2024. alexei-led/pumba. Retrieved December 22, 2024 from https://github.com/alexei-led/pumba

[32] Martin Andreoni Lopez, Antonio Gonzalez Pastana Lobato, and Otto Carlos M. B. Duarte. 2016. A Performance Comparison of Open-Source Stream Processing Platforms. In *2016 IEEE Global Communications Conference (GLOBECOM)*. IEEE, Washington, DC, USA, 1–6. https://doi.org/10.1109/GLOCOM.2016.7841533

[33] Martin Maňásek and Petr Tůma. 2019. Charles University SIS Access Log Dataset. https://doi.org/10.5281/ZENODO.3241445

[34] J. Nagle. 1984. *Congestion Control in IP/TCP Internetworks*. Number RFC0896. RFC0896 pages. https://doi.org/10.17487/rfc0896

[35] Rahul Potharaju and Navendu Jain. 2013. When the Network Crumbles: An Empirical Study of Cloud Network Failures and Their Impact on Services. In *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, Santa Clara California, 1–17. https://doi.org/10.1145/2523616.2523638

[36] Caleb Stanford, Konstantinos Kallas, and Rajeev Alur. 2022. Correctness in Stream Processing: Challenges and Opportunities. In *12th Conference on Innovative Data Systems Research, CIDR 2022, Chaminade, CA, USA, January 9-12, 2022*. www.cidrdb.org. https://www.cidrdb.org/cidr2022/papers/p103-stanford.pdf

[37] Donghan Sun and Soochan Hwang. 2018. DSSP: Stream Split Processing Model for High Correctness of Out-of-Order Data Processing. In *2018 IEEE First International Conference on Artificial Intelligence and Knowledge Engineering (AIKE)*. IEEE, Laguna Hills, CA, 193–197. https://doi.org/10.1109/AIKE.2018.00044

[38] Jawad Tahir. 2024. PGVal. https://github.com/jawadtahir/DSPF-BM/blob/main/Docs/DSPS-BM-Extended.pdf.

[39] Jawad Tahir. 2024. PGVal. https://github.com/jawadtahir/DSPF-BM.

[40] Jawad Tahir, Chiheb Baili, Matej Svaral, Johannes Friedlein, Christoph Doblander, and Hans-Arno Jacobsen. 2024. Challenger 2.0: A Step Towards Automated Deployments and Resilient Solutions for the DEBS Grand Challenge. In *Proceedings of the 18th ACM International Conference on Distributed and Event-Based Systems* (Villeurbanne, France) *(DEBS '24)*. Association for Computing Machinery, New York, NY, USA, 6–17. https://doi.org/10.1145/3629104.3666027

[41] Jawad Tahir, Christoph Doblander, Ruben Mayer, Sebastian Frischbier, and Hans-Arno Jacobsen. 2021. The Debs 2021 Grand Challenge: Analyzing Environmental Impact of Worldwide Lockdowns. In *Proceedings of the 15th ACM International Conference on Distributed and Event-based Systems*. ACM, Virtual Event Italy, 136–141. https://doi.org/10.1145/3465480.3467836

[42] Nesime Tatbul, Stan Zdonik, John Meehan, Cansu Aslantas, Michael Stonebraker, Kristin Tufte, Chris Giossi, and Hong Quach. 2015. Handling Shared, Mutable State in Stream Processing with Correctness Guarantees. *IEEE Data Eng. Bull.* 38, 4 (2015), 94–104.

[43] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. 2014. Storm@twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. ACM, Snowbird Utah USA, 147–156. https://doi.org/10.1145/2588555.2595641

[44] Giselle van Dongen and Dirk Van Den Poel. 2021. A Performance Analysis of Fault Recovery in Stream Processing Frameworks. *IEEE Access* 9 (2021), 93745–93763. https://doi.org/10.1109/ACCESS.2021.3093208

[45] Giselle van Dongen, Bram Steurtewagen, and Dirk Van den Poel. 2018. Latency Measurement of Fine-Grained Operations in Benchmarking Distributed Stream Processing Frameworks. In *2018 IEEE International Congress on Big Data (BigData Congress)*. 247–250. https://doi.org/10.1109/BigDataCongress.2018.00043

[46] Giselle van Dongen and Dirk Van den Poel. 2020. Evaluation of Stream Processing Frameworks. *IEEE Transactions on Parallel and Distributed Systems* 31, 8 (Aug

2020), 1845–1858. https://doi.org/10.1109/TPDS.2020.2978480

[47] Adriano Vogel, Sören Henning, Esteban Perez-Wohlfeil, Otmar Ertl, and Rick Rabiser. 2024. A Comprehensive Benchmarking Analysis of Fault Recovery in Stream Processing Frameworks. , 12 pages. https://doi.org/10.1145/3629104.3666040

[48] Guosai Wang, Lifei Zhang, and Wei Xu. 2017. What Can We Learn from Four Years of Data Center Hardware Failures?. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, Denver, CO, USA,

25–36. https://doi.org/10.1109/DSN.2017.26

[49] Xiaotong Wang, Chunxi Zhang, Junhua Fang, Rong Zhang, Weining Qian, and Aoying Zhou. 2022. A Comprehensive Study on Fault Tolerance in Stream Processing Systems. *Frontiers of Computer Science* 16, 2 (April 2022), 162603. https://doi.org/10.1007/s11704-020-0248-x

[50] Farzin Zaker. 2019. Online Shopping Store - Web Server Logs. https://doi.org/10.7910/DVN/3QBYB5