streamline: Accelerating Deployment and Assessment of Real-time Big Data Systems

Md. Monzurul Amin Ifath, *Student Member, IEEE,* Tommaso Melodia, *Fellow, IEEE,* and Israat Haque, *Senior Member, IEEE*

Abstract—Real-time stream processing applications (e.g., IoT data analytics and fraud detection) are becoming integral to everyday life. A robust and efficient big data system, especially a streaming pipeline composed of producers, brokers, and consumers, is at the heart of the successful deployment of these applications. However, their deployment and assessment can be complex and costly due to the intricate interactions between pipeline components and the reliance on expensive hardware or cloud environments. Thus, we propose *streamline*, an agile, efficient, and dependable framework as an alternative to assess streaming applications without requiring a hardware testbed or cloud setup. To simplify the deployment, prototyping, and benchmarking of end-to-end stream processing applications involving distributed platforms (e.g., Apache Kafka, Spark, Flink), the framework provides a lightweight environment with a developer-friendly, high-level API for dynamically selecting and configuring pipeline components. Moreover, the modular architecture of *streamline* enables developers to integrate any required platform into their systems. The performance and robustness of a deployed pipeline can be assessed with varying network conditions and injected faults. Furthermore, it facilitates benchmarking event streaming platforms like Apache Kafka and RabbitMQ. Extensive evaluations of various streaming applications confirm the effectiveness and dependability of *streamline*.

Index Terms—Stream Processing, Big Data Applications, Performance Evaluation, Reliability Testing, Apache Kafka, Apache Spark.

1 Introduction

THE high degree of big data generation and consumption transforms their usage in everyday life by the development of stream processing applications [1]. These applications have gained popularity due to their ability to perform real-time analytics for applications like financial transaction monitoring, IoT sensor data processing, sentiment analysis, or content recommendations. Usually, stream processing pipelines consist of data sources, streaming platforms, engines, storage, and visualization components. The majority of Fortune 100 companies (e.g., Cisco, Goldman Sacks, Uber) deployed stream processing platforms like Apache Kafka [2], Apache Spark [3], and Apache Flink [4] to manage complex, multi-component systems deployed across distributed servers [5]. However, the complexity and resource demand of distributed streaming systems may hinder innovations. Specifically, testing these systems relies on expensive hardware testbeds (typically a cluster of servers arranged in a predetermined, static configuration) or complex cloud setups, which limits accessibility, agility, and innovations. Moreover, setting appropriate communications (e.g., link capacity, routing scheme, faults) among components is pivotal for performance and robustness analysis, but is not easy across distributed components.

We may consider simulators (e.g., ns-3 [6] and OMNeT++ [7]) and emulators (e.g., CrystalNet [8], TurboNet [9]) to assess functional correctness or scale of stream processing applications; without inferring their performance in real deployments. Network testbeds such as PlanetLab [10] and Emulab [11] offer adequate computational resources but lack flexibility in network topology customization and require manual stream processing platform setup. Additionally, there are initiatives in developing stream processing assessment tools [12], [13], which focus on specific components of the pipeline or their integration. For instance, TRAK [12] focuses on assessing the reliability of the Apache Kafka streaming platform by strategically injecting faults at various levels within its architecture: broker, client, and network. While NAMB [13] supports rapid benchmarking and cross-platform evaluations of different platforms (e.g., Kafka, Flink, Storm), it lacks support for fine-grained network configurations, cluster-level testing, and system-level resource usage monitoring. Proprietary platforms like Confluent [14] and Databricks [15] offer component-specific connectors and specialized services, facilitating pipeline component integration. However, their cost and potential vendor lock-in can limit accessibility and flexibility. Unfortunately, there is no accessible, agile, efficient, and dependable stream processing framework to deploy and assess the end-to-end performance of streaming applications.

To bridge this gap, we introduce a real-time big-data stream processing application assessment framework named *streamline*. Developers can implement and deploy it on a standalone computer and configure the required components of the processing pipeline using a developer-friendly high-level API. It abstracts the complexities of the underlying systems setup to developers and enables the rapid deployment and configuration of various stream processing pipelines. The modularized architecture allows one

- *MMA. Ifath and I. Haque are with the Faculty of Computer Science, Dalhousie University, Halifax, NS B3H 4R2, Canada. E-mail: monzurul.amin@dal.ca; israat@dal.ca*
- *T. Melodia is with the Department of Electrical and Computer Engineering, Northeastern University, Boston, MA 02115, United States. E-mail: melodia@northeastern.edu*

to choose different platforms (e.g. Apache Kafka, RabbitMQ, Apache Spark, and Apache Flink) based on the application demands. Once deployed and configured, developers can dynamically vary network conditions (link bandwidth, latency, faults) to assess the performance and robustness of streaming applications over single-node or cluster-based deployments. Typical use cases of *streamline* include measuring throughput or latency with varying network conditions (e.g., link bandwidth and latency), evaluating system behavior under network faults (e.g., network-level disruptions) and failures within streaming platforms, and benchmarking streaming platforms. Note that proprietary infrastructures that require native hardware execution are out of the scope of *streamline*.

We implement a prototype of *streamline* and evaluate it across eight streaming applications. These include sentiment analysis and network traffic monitoring, among others. To enable communication among pipeline components, we use Mininet [16], a lightweight yet flexible network emulator. The results show that *streamline* incurs less than 10% deviation in latency compared to hardware testbeds and achieves 15–33% lower latency compared to an OpenStack-based cloud setup, with only 7% additional CPU usage at scale. Finally, we assess its benchmarking capability by comparing Apache Kafka and RabbitMQ under high-latency network conditions, revealing that Kafka's latency escalates significantly relative to RabbitMQ. The contributions of *streamline* are summarized below.

- We present *streamline*, a lightweight and extensible *framework* for prototyping, deploying, and benchmarking real-time stream processing pipelines. Its modular architecture supports failure injection, network emulation, and clustered deployments under controlled conditions.
- *streamline* exposes a configurable *interface* that integrates diverse stream processing engines and event streaming platforms, supporting reproducible experiments through scenario-specific configurations.
- We demonstrate *streamline*'s *dependability* through extensive evaluations across varied environments (emulated, cloud-based, and hardware testbeds) showing less than 10% deviation in latency and comparable performance to state-of-the-art solutions like NAMB.
- We show *streamline*'s *capability* to assess reliability and fault tolerance of distributed stream applications by systematically injecting network and component faults and measuring their impact on performance (e.g., message loss, recovery time).
- To support *reproducibility* and extension, we release *streamline* as open-source software under the Apache license[1], including configuration templates, integration guides, and per-application workflows.

A preliminary version of this work appeared in [18], which is significantly extended in this work. In contrast to the prior version which focused on rapid prototyping of streaming application, *streamline* is a generalized and extensible framework designed to support a wide spectrum of performance and reliability evaluations. evaluations. The earlier tool can be considered a specialized instance of the more generalized *streamline* framework, which introduces significantly enhanced capabilities. To be specific, firstly, the revised architecture is completely modularized, allowing plug-in pipeline components through the extended API. For instance, developers can now choose combinations of various popular platforms and engines like Apache Kafka, RabbitMQ, Apache Spark, and Apache Flink. Secondly, we introduce new networking and reliability testing features, e.g., bandwidth conditions and impacts of component failures, which can be tested in a cluster deployment. Finally, we conduct extensive evaluations of *streamline* and compare results against hardware testbeds, virtualized cloud environment, and start-of-the-art solution, including benchmarking streaming platforms to confirm its usability, efficiency, and dependability. These enhancements collectively bolster *streamline*'s capabilities, establishing it as an accessible, efficient, and dependable tool for innovation in stream processing applications.

1. To be released in GitHub [17] upon paper acceptance.

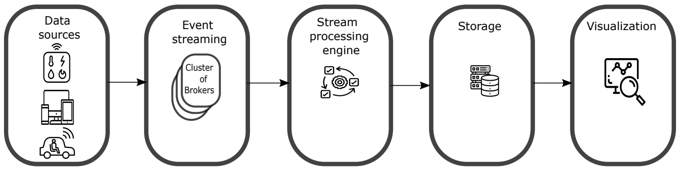


Fig. 1: Typical stream processing application pipeline.

2 Background and Motivation

This section presents the necessary background to understand our contribution along with the motivation for developing *streamline*.

Stream Processing Pipeline: In *batch processing*, data is accumulated and stored for a fixed time interval and then processed in bulk at the end of each interval. Batch processing systems require large amounts of data storage and computation power to handle the massive influx of data, which may incur latency between minutes and days. *Stream processing*, on the other hand, offers real-time processing of a continuous stream of data in personalized recommendations, financial fraud detection, log monitoring in smart grids, IoT telemetry, i.e., applications requiring high throughput and low latency [5]. A stream processing pipeline consists of data sources, event streaming platforms, processing engines, storage systems, logging mechanisms, and visualization stages (Figure 1). As each such component is a distributed system, we can refer to the pipeline as a *system of systems*. For example, Uber's data analytics infrastructure [19] integrates third-party platforms like Apache Kafka and Apache Flink for event streaming and stream processing, respectively, with their proprietary workflow management solutions. These systems typically operate on a distributed cluster of servers, relying on high-speed networking for coordination.

Event Streaming Platforms: These systems collect and distribute events (e.g., transactions, sensor readings, or database updates) across data sources and sinks. Event streaming platforms like Apache Kafka, Google Cloud

Pub/Sub, and Apache Pulsar operate on the publish-subscribe messaging pattern to facilitate flexible data flow and dynamic addition or removal of clients without requiring them to be constantly online. This asynchronous architecture supports decoupled components, enabling servers and clients to interact conveniently and significantly enhance scalability and adaptability. Apache Kafka [2] is the most widely used publish-subscribe system that combines log aggregation and messaging over a comprehensive log for reliable and ordered data retrieval.

Stream Processing Engine: Stream processing engines allow developers to conduct operations like querying, filtering, and aggregating on streaming data without knowing low-level deployment intricacies. These engines generally fall into two categories based on their programming models and execution modes. *Compositional* engines like Apache Storm, Apache Samza, and Apache Apex allow developers to construct directed acyclic graphs (DAGs) of operators for data processing. In contrast, *Declarative* engines, including Apache Spark Structured Streaming, Apache Flink, and Azure Stream Analytics, enable query formulation using declarative languages like SQL, abstracting the processing logic from implementation details.

Network Platforms: Communication among pipeline elements is pivotal for effective stream processing. Simulators like NS-3 and OMNeT++ are popular choices for assessing the functional correctness and scale of networking systems without considering real deployments. On the other hand, PlanetLab [10], Emulab [11], CloudLab [20], and Chameleon [21] are testbeds for networking systems with realistic computation and communication capabilities, i.e., offer a platform to assess the deployment feasibility of a system. This benefit comes at the cost of testbeds accessibility and their complex and time consuming setups. For instance, developers need to configure required topologies and the forwarding behaviors of elements to enable required communications among the components of stream processing pipelines. Additionally, they must instantiate data processing platforms (e.g., stream processing engines and messaging systems) from scratch.

Emulators are a middle ground between the above two alternatives. CrystalNet [8] and TurboNet [9] have their own strength and weaknesses. The former emulates large-scale network control plane functionalities without supporting dynamically configuring data plane attributes like link delay and bandwidth. TurboNet can emulate both planes but requires additional hardware support for host applications. The container-based emulator Mininet [22] emulates software-defined networking (both data and control plane) with portability and agility in testing network functions. Thus, we adopt Mininet in *streamline* to enable realistic communications among stream processing pipeline components.

Pipeline Assessment: Like traditional software testing, developers must address the stream processing testing challenge across various levels of granularity, encompassing unit, integration, and system level testing [23]. While unit tests are prevalent and many stream processing systems offer their own unit testing modules (e.g., Kafka test-utils [24], Flink Spector [25]), integration and system testing receive less attention and often necessitate developers to devise custom ad-hoc solutions. A frequently employed method for integration testing involves mocking, where frameworks like Mockito [26] simulate interactions among different components within a data processing pipeline. Proprietary platforms such as Confluent [14] and Databricks [15] provide diverse services tailored to specific pipeline components. Still, they can be expensive and may lead to vendor lock-in when transitioning to alternative technologies. However, none of these solutions adequately address system-level testing needs or offer control over underlying infrastructure variables (e.g., networking delays, failure occurrences) while supporting various stream processing platforms with consistent performance, reliability, and scalability. To address these shortcomings, we developed *streamline* as a modular, configurable, efficient, and reliable platform to deploy and assess stream processing applications.

3 Related Work

This section presents works related to *streamline* to position its contribution.

There are a few attempts to develop tools to deploy and assess the entire stream processing pipeline, including communications among components. Kallas *et al.* present DiffStream [27], a library designed for differential testing of stream processing systems. This library facilitates the comparison of output streams from two distinct stream processing programs given identical input streams, even allowing for reordering within the output. DiffStream also enables developers to define dependencies among stream items and, through integration with Apache Flink, provides bug detection, program parallelization, and monitoring with minimal overhead. However, DiffStream's utility is curtailed by its inability to support stateful or windowed operations, external interactions, or side effects, nor does it aid in crafting equivalent program versions or determining dependence relations. On another front, TRAK [12] serves as an instrument for assessing the reliability of event streaming platforms like Apache Kafka, employing fault injection via docker containers and Pumba to gauge message loss and duplication rates in unreliable networks. Nonetheless, TRAK's focus is narrowly confined to a pair of reliability metrics and does not extend to the broader spectrum of stream processing application concerns. NAMB [13], another relevant tool in this domain, is a high-level prototype generator that supports rapid benchmarking of stream processing platforms like Flink and Storm using YAML-based topology descriptions. It allows synthetic or Kafka-based data injection and simulates workload via busy-wait loops. However, NAMB focuses on application-level prototyping without supporting realistic network emulation, fault injection, or runtime monitoring.

In the realm of benchmarking[2], a couple of tools have been developed for stream processing engines and pipelines. Gadget [28] supports stream processing engines Apache Flink and RocksDB to analyze streaming state access workloads. The developers require configuring and deploying a stream processing system integrated with an appropriate key-value store to execute queries reflective

2. In benchmarking, system performance is measured and compared against established standards or metrics.

TABLE 1: *streamline* vs. existing approaches. ESP = Event Streaming Platform, SPE = Stream Processing Engine, DS = Data Store.

Approach	Quality attribute	Network condition	Fault injection	Stateful operation	Platform support	Open source
DiffStream	Performance Scalability	No	No	No	SPE	Yes
TRAK	Reliability	No	Yes	No	ESP	No
NAMB	Performance	No	No	Partial (simulated)	ESP, SPE	Yes
Gadget	Performance Scalability	No	No	Yes	SPE, DS	Yes
Karimov	Performance Scalability	No	No	Yes	SPE	No
Chintapalli	Performance	Yes	No	Yes	ESP, SPE, DS	Yes
ShuffleBench	Performance Scalability	No	No	Yes	SPE	Yes
streamline	Performance Reliability Scalability	Yes	Yes	Yes	ESP, SPE, DS	Yes

of the performance of streaming state stores. However, the tool does not support crucial state measures like fault tolerance, consistency, and recovery, which are essential in applications prone to network partitions, such as `Military coordination` (see Section 6).

Karimov *et al.* [29] develop a benchmarking suite with novel metrics and methodologies for quantifying the performance of stream processing systems, assessing metrics such as latency, throughput, and windowed events timing. However, the assessment considers synthetic data for certain workloads. Also, it does not support different topologies with varying resources and fault tolerance. Chintapalli *et al.* [30] propose a complete pipeline benchmarking tool with Apache Kafka and Redis to emulate practical production environments. Yet, it falls short in considering the impacts of network conditions, data representation, and manipulation on streaming engines' performance. Recent efforts such as ShuffleBench [31] presents a cloud-native, open-source benchmark tailored to large-scale data shuffling scenarios in observability platforms. As ShuffleBench primarily focuses on isolated performance benchmarking of Flink, Kafka Streams, Spark, and Hazelcast, it lacks support for realistic application contexts, modular extensibility, or network emulation under multi-platform configurations, limiting its utility for prototyping end-to-end pipeline deployments involving diverse event streaming platforms and stream processing engines.

The existing tools, while insightful, fall short of providing a holistic solution that tackles the complexities of end-to-end stream processing pipeline testing, particularly in ensuring performance, reliability, and scalability under real-world conditions. An ideal tool should not only support diverse stream processing platforms and engines but also enable the emulation of realistic network conditions, facilitate fault injection for reliability testing, and provide comprehensive monitoring and debugging capabilities. This gap in the testing and benchmarking landscape underscores the need for a more versatile and comprehensive tool like *streamline*. Table 1 offers an overall comparative analysis between *streamline* and the aforementioned testing methodologies.

4 *streamline* OVERVIEW

This section presents the design goals of *streamline*, its architecture, workflow, and configuring API. We conclude the section with an illustrative example for *streamline*.

Properties of *streamline*: *streamline* offers the following features for efficient deployment and assessment of stream processing pipelines. It is *easy to configure, monitor, and manage*, enabling developers to test a prototype within their computing systems without accessing, configuring, and using a complex distributed testing platform (e.g., Chameleon), i.e., a *cost-effective* tool. Specifically, they can use a container-based deployment of *streamline* without having in-depth knowledge of low-level distributed system details (e.g., network addressing, routing, platform interoperability).

The modular architecture with developer-friendly API makes *streamline* an *agile stream processing applications testing playground*, where they can choose a combination of streaming platforms, stream processing engines, and storage systems within *streamline*. Furthermore, developers can integrate additional components of the processing pipeline as per the demand of applications. *streamline* simplifies the evaluation of stream processing applications with *diverse network topologies and operational conditions* (e.g., route delays, link bandwidths, disconnections). It continuously monitors the entire pipeline, providing developers with real-time insights to quickly identify and debug issues during testing and deployment. *streamline* accurately mimics the operational behavior and performance characteristics of real-world stream processing applications, ensuring that the application code executes identically in comparison with the testbed and cloud environment.

Challenges: *streamline* must resolve a number of design challenges to offer the desired properties. First, it must offer an *efficient and extensible API* to provide the necessary system configuration parameters and performance measurement metrics without knowing the underlying infrastructure details. Second, it must efficiently utilize *computing resources* of the host machine to implement and deploy every component of the stream processing pipeline. Specifically, *streamline* must carefully allocate resources across network elements, stream processors, and data loggers while meeting the application performance requirements. Furthermore, ensuring seamless *compatibility and integration* among various stream processing components can be complex, often requiring the development of proxy or wrapper code. Additionally, integrating *streamline* with existing stream processing platforms and libraries may necessitate addressing compatibility and interoperability issues. Finally, *streamline* must offer a *dependable* (similar to real deployment) performance with system performance *monitoring and debugging* facilities.

4.1 Architecture and Workflow

Figure 2 shows *streamline*'s architecture. The framework takes a high-level description of the configuration and assessment tasks as input over the developed API. The input contains: i) a set of stream processors (e.g., Spark/Flink programs) specified by the application developer and accompanied by sample input data; ii) necessary configuration parameters (e.g., number of message brokers, event topics, and stream processing engine workers) for setting up the underlying stream processing platform, including its data stores, data producers, and message brokers in case these are present; and iii) a desired network topology to host the entire stream processing system. This design separates the application logic from its testing setup, which enables the

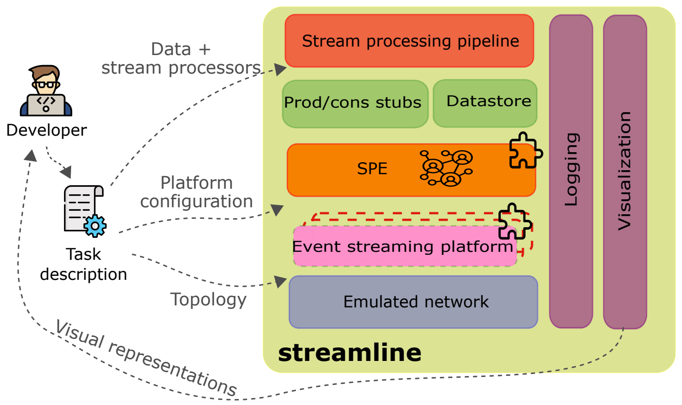


Fig. 2: *streamline* architecture and workflow. SPE = Stream Processing Engine.

re-use of testing scenarios by modularly plugging different stream processors.

streamline instantiates the specified topology using a network emulator, Mininet [16]. Even though our framework focuses on single computers, it can operate with minimal modifications on distributed clusters (e.g., using [32] or [33]) if an extreme scale is needed. Once the network is set, *streamline* starts a modular event streaming platform (ESP) for communication among different application components, which is common in current data processing pipelines (see Section 2). *streamline* then initializes the various components that the specified application encompasses, which may include stream processing engines (SPEs), data stores, producer and consumer stubs, among others.

streamline provides a repository containing standard producer/consumer stubs that developers can use to quickly ingest data into or extract data from stream processing pipelines according to desired patterns. Also, each application component runs as an independent process, which enables them to be balanced and prioritized among multiple cores in the underlying server to mitigate load imbalances due to diverse resource demands. *streamline* provides several parameters that can be tuned to support various operational conditions from production environments, including routing algorithms and failure profiles. It also facilitates fine-grained parameter tuning to optimize resource usage while maintaining accuracy compared to real-world deployments. *streamline* aims to address integration challenges by compatibility testing with standard stream processing platforms and libraries with integration guidelines.

Finally, it offers a series of monitoring tasks that log relevant network and application information (e.g., bandwidth measurements, timestamped events) and employs a robust error handling mechanism that gracefully captures and logs exceptions arising from any component within the pipeline. Moreover, a visualization module presents a rich set of statistics to the developer, which includes per-port throughput, message latency, and event ordering. *streamline* incorporates modular ESPs and SPEs for seamless integration and interchangeability of different ESP and/or SPE components. Also, *streamline* supports clustered setups to test larger-scale and complex deployments.

TABLE 2: *streamline* attributes.

Graph attributes	Description
topicCfg	Topic configuration for the event streaming system
faultCfg	Fault configuration (e.g., link down) for reliability tests
Node attributes	**Description**
prodType	Data source type (used for data ingestion)
prodCfg	Data source configuration
consType	Data sink type (used for data consumption)
consCfg	Data sink configuration
streamProcType	Stream processing engine type (e.g., Spark, Flink, KStream)
streamProcCfg	Stream processing engine configuration
storeType	Data store type (e.g., MySQL, MongoDB, RocksDB)
storeCfg	Data store configuration
brokerType	Message broker type (e.g., Kafka, RabbitMQ)
brokerCfg	Message broker configuration
cpuPercentage	Cap on overall system CPU usage
Link attributes	**Description**
lat	Link latency (in milliseconds)
bw	Link bandwidth (in Mbps)
loss	Link loss (%)
st	Source port
dt	Destination port

4.2 Configuration API

The interface builds on GraphML [34] and YAML [35] due to their agility and offers a unified, platform-agnostic interface. Table 2 lists the attributes *streamline* supports, which either points to a configuration file or developer-specified values. The integrated event streaming platform moves data among network nodes based on a publish/subscribe model after setting a network topology.

Graph Attributes: Developers can establish a collection of topics, enabling application components to efficiently publish or subscribe to specific data streams. In this context, *streamline* leverages a broker-based messaging (or event streaming) system for inter-component communication. For each topic, developers can designate a primary broker and specify the desired number of replicas. Additionally, *streamline* provides a simplified API for simulating various failure scenarios, such as link failures.

Node Attributes: Nodes can host a variety of application components, e.g., data stores, producers, consumers, message brokers, and stream processing engines. Each component is associated with a configuration file, structured as a list of key-value pairs, specifying component-specific parameters.

Link Attributes: *streamline* allows the configuration of standard communication channel parameters on links (e.g., delay, bandwidth, and packet loss). The packet loss is useful to construct complex failure scenarios (e.g., gray failures[3]) and testing network congestion. Additionally, developers can specify the source and destination ports for each link, enabling precise control over the connection points between hosts.

4.3 Example

Figure 3a shows an example data processing pipeline that can be prototyped using *streamline*. This pipeline illustrates a document analytics application [36] and comprises a data source, which can read information from a file system or database, two stream processing jobs, and a data sink. The

3. Hardware malfunction that causes packet losses only for subsets of packets sent over a link [36].

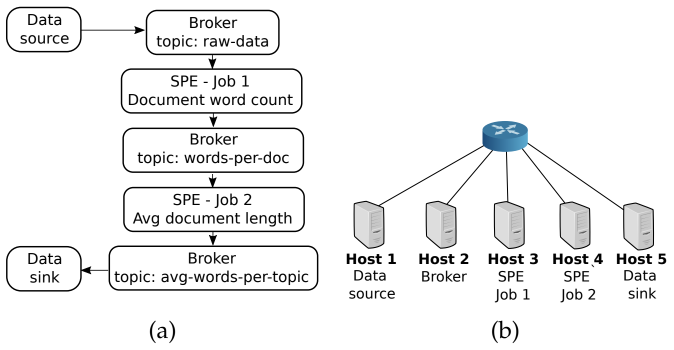


Fig. 3: a) Example data processing pipeline; b) Target pipeline allocation into the testing infrastructure.

```
1  <!-- Graph attributes -->
2  <graph edgedefault="undirected">
3    <data key="topicCfg"> topics.cfg </data>
4
5    <!-- Node attributes -->
6    <node id="h1">
7      <data key="prodType"> SFST </data>
8      <data key="prodCfg"> data-src.yaml </data>
9    </node>
10   <node id="h2">
11     <data key="brokerType"> KAFKA </data>
12     <data key="brokerCfg"> broker.yaml </data>
13   </node>
14   <node id="h3">
15      <data key="streamProcType"> SPARK </data>
16      <data key="streamProcCfg"> spe-1.yaml </data>
17   </node>
18   <node id="h4">
19      <data key="streamProcType"> SPARK </data>
20      <data key="streamProcCfg"> spe-2.yaml </data>
21   </node>
22   <node id="h5">
23      <data key="consType"> STANDARD </data>
24      <data key="consCfg"> data-sink.yaml </data>
25   </node>
26
27   <!-- Link attributes -->
28   <node id="s1"/>
29   <edge source="s1" target="h1">
30      <data key="st"> 1 </data>
31      <data key="dt"> 1 </data>
32      <data key="bw"> 1000 </data>
33      <data key="lat"> 50 </data>
34   </edge>
 ...
55 </graph>
```

Fig. 4: GraphML description for the data processing pipeline presented in Figure 3a. We omit some lines due to space constraints.

two stream processing jobs are responsible for counting the number of distinct words in a document and calculating the average document length based on their topic, respectively. The pipeline uses a message broker to stream data between processing and storage nodes, and each data migration happens on a different topic (i.e., "raw-data" and "avg-words-per-topic").

Figure 3b illustrates the target pipeline allocation into the tested infrastructure. Each component occupies a separate server reflecting a common scenario in which service

```
---
filePath : test-data.csv
topicName : raw-data
totalMessages : 1000
requestTimeout : 2000ms
bufferMemory : 32m
---
```

(a)

```
---
app : word-count.py
executorMemory : 1g
eventLog : true
---
```

(b)

Fig. 5: Example YAML configurations for the a) data source; and b) word count components of the data processing pipeline described in Figure 3a.

providers adopt dedicated, i.e., specialized, clusters [37]. We employ the "one-big-switch" abstraction [38] to model the network setup within streamline, simplifying the specification of the network while retaining essential communication channel details like delay and bandwidth. This abstraction transforms the complex network topology into a singular, large switch connecting all endpoints, focusing on endpoint connectivity and packet handling rules. The controller translates high-level policies into forwarding rules for data plane switch, optimizing rule space and reducing superfluous traffic. This approach simplifies network policy management, enhancing efficiency and clarity in network setup modeling.

Figure 4 illustrates how to describe our example data processing pipeline using *streamline*'s modeling language (i.e., GraphML). We start by setting up the configuration of *streamline*'s event streaming platform topic (line 3). Next, we specify the configuration of each pipeline component (lines 6-25). Note that each host (i.e., node) follows the target resource allocation previously discussed. Finally, we specify the networking setup for the communication channels between hosts in the cluster (lines 28-55). We use separate YAML files to specify the configuration of each application component. Figures 5a and 5b show two examples, which depict the configuration of the data source (or producer) and the word counting job of our example pipeline, respectively.

5 Implementation and Evaluation Setup

Implementation: *streamline* features two event streaming platforms: Apache Kafka 4.0.0 [2] and RabbitMQ 4.1.0 (rMQ) [39]. Similarly, we choose Apache Spark 3.5.5 [3] and Apache Flink 2.0.0 [4] as representative stream processing engines. Although rMQ is traditionally used for message queuing, we leverage its Streams plugin (introduced in version 3.9+) to enable stream processing and non-destructive message consumption capabilities[4]. While Apache Kafka, Spark and Flink are widely adopted for high-throughput stream analytics, rMQ is better suited for event-driven applications requiring reliable and ordered message delivery (e.g., tactical coordination) [40]. These platforms and engines are chosen due to their extensive documentation and broad adoption in academia and industry [41], reflecting their relevance in modern stream processing practices. Owing to modular design of *streamline*, developers can extend the framework by integrating new stream processing technologies with minimal effort, following the supplementary documentation provided in the open-source project repository [17].

4. https://www.rabbitmq.com/docs/streams

streamline enables communications among pipeline components over Mininet 2.3.0 [16], a popular network emulator chosen for its flexibility, ease of use, and ability to accurately model various network topologies and conditions (e.g., different latency, bandwidth, and packet loss). A software-define controller communicates over OpenFlow[5] 1.3 to monitor, configure, and manage network elements, including a lightweight switch control daemon (based on ovs-ofctl). We leverage this switch to minimize control plane overhead. OpenFlow statistics monitor network performance indicators such as bandwidth consumption and packet loss rates. Additionally, we employ the Python logging facility[6] to capture essential application events, including message timestamps (arrival and departure) and processing duration.

The implementation of *streamline* framework [17] comprising approximately 5,800 lines of Shell and Python code, utilizes Networkx 2.5.1 for parsing GraphML topology specifications and Matplotlib 3.3.4 for generating informative data visualizations. MySQL 8.0.30 is currently supported as an exemplary external data store component.

Evaluation Setup: We evaluate our experiments in three environments: *streamline*, hardware testbed, and cloud environment. For *streamline* (i.e., emulation results), we utilize a machine equipped with an Intel® Core i7-3770 CPU @ 3.40GHz, 16GB RAM, and 2TB of storage, running Ubuntu 22.04.3 LTS with kernel version 5.15.0-79-generic. Testbed deployment consists of a 4-node cluster with two 10-core Intel Xeon Silver 4210R at 2.40 GHz with 32 GB of memory and two 8-core Intel Core i7-9700 at 3.00 GHz with 16 GB of memory servers. The Xeon servers are equipped with a 25 Gbps Mellanox BlueField SmartNIC and run Ubuntu 20.04.1 LTS, while the Core i7 servers have a 40 Gbps Netronome Agilio LX SmartNIC and run Ubuntu 18.04.6 LTS. All servers have hyper-threading disabled and are connected to an Edgecore Wedge 100BF-32X switch with Intel Tofino ASIC. We leverage OpenStack [42] as the cloud environment due to its wide adoption, modular architecture, and extensive API support. The cloud consists of four virtual machines, each equipped with 8 virtual CPUs based on the Intel Xeon E312xx clocked at 2.00 GHz with 32 GB memory. All virtual machines run Ubuntu 20.04.1 LTS with hyper-threading disabled while communicating through a virtual switch.

6 USE CASES OF *streamline*

This section demonstrates *streamline* 's capabilities and applicability to real-world stream processing deployments through three sets of evaluations. In Section 6.1, we assess *streamline* 's ability to prototype a wide range of stream processing applications, illustrating its support for diverse workloads, deployment patterns, and integration configurations. In Section 6.2, we evaluate how *streamline* can emulate varied network conditions, such as fluctuating link delays and constrained bandwidth, and analyze their impact on end-to-end performance metrics. In Section 6.3,

TABLE 3: Example applications deployed on *streamline* and the framework capabilities tested.

Application	Application Characteristics	#Components	Tested *streamline* capabilities
Word count	Multiple stream processing jobs	5	Real-time stream processing Monitoring & logging
Ride selection	Structured data processing Stateful processing	5	Real-time stream processing
Sentiment analysis	Unstructured data processing	3	Real-time stream processing
Maritime monitoring	External persistent storage	4	Data store integration Topology configuration
Fraud detection	Machine learning prediction	5	Fault injection Monitoring & logging
Military coordination	Data replication Parameter tuning	4	Fault injection Benchmarking support Network emulation
Video analysis	Concurrent consumption	3	Scalability testing Monitoring & logging
Network traffic monitoring	Windowed aggregation	4	Cluster deployment Fault injection Reproducibility

we demonstrate *streamline* 's capability to test system reliability by injecting network- and platform-level faults, and analyzing their effects on message loss, latency, and recovery across both event streaming platforms and stream processing engines. Together, these evaluations highlight how *streamline* serves not only as a dependable platform for reproducing expected behaviors under controlled setups, but also as a powerful tool for uncovering non-trivial system behaviors under fault and stress scenarios.

6.1 Prototyping Stream Processing Applications

We implement and evaluate a range of stream processing applications using *streamline* to showcase its versatility across different processing needs and deployment settings. Table 3 summarizes these applications along with their characteristics and the specific *streamline* capabilities tested, such as real-time processing, fault injection, benchmarking, and cluster deployment. The number of components indicates how many modules (e.g., stream processors, message brokers, key-value stores) an application contains. More details on each application, including platform configurations and deployment topologies, are available in the public *streamline* repository [17].

`Word count` serves as a standard benchmark application for evaluating stream processing systems [3]. It collects textual data from a stream of files, splits it into words, and calculates the frequency of each word, eventually storing the results in a separate file. We implement the core logic into two distinct stream processing jobs: one for text splitting and another for frequency counting.

`Ride selection` application leverages structured data from a stream of taxi ride information, including geographical coordinates and fare values, to identify the best tipping areas in New York City. It utilizes the NYC Taxi dataset and GeoJSON data representing Manhattan neighborhoods to perform complex queries involving joins, groupby, and window operators. The implementation involves handling intermediate states efficiently to process these complex queries on streaming data.

`Sentiment analysis` computes the subjectivity and polarity, two common natural language processing (NLP) tasks, of each message in a Tweet stream. We employ the Python TextBlob[7] library to evaluate the subjectivity and polarity of tweets, focusing on their emotional and subjective content. The process involves collecting Twitter data on

5. https://opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.3.0.pdf
6. https://docs.python.org/3/library/logging.html
7. https://textblob.readthedocs.io/en/dev/

specific topics, followed by essential data cleaning to ensure quality input for analysis.

`Fraud detection` employs machine learning (specifically, an SVM algorithm) to detect anomalies and potential fraud in a stream of financial transactions. It encompasses the key stages of a machine learning workflow, including data preprocessing, feature extraction, model training, and evaluation, all organized within a data processing pipeline. The pipeline integrates a series of transformers and estimators to facilitate the seamless execution of these stages.

`Maritime monitoring` analyzes a stream of ship tracking reports (e.g., AIS messages) to count the number of ships heading to a set of desired ports in a given time window. The data processing pipeline uses an external key-value store, i.e., in addition to the one embedded in the stream processing engine, to store the results.

`Military coordination` simulates a common military network setup where multiple tactical teams need to stay operational by sharing and accessing critical data. It involves message brokers interconnected in a predefined topology, replicating messages produced into topics. Our implementation focuses on performance optimization by conducting a grid search across Apache Kafka's extensive parameter space to identify the ideal configuration for a military scenario.

`Network traffic monitoring` to evaluate the scalability of a stream processing-based traffic monitoring system for enterprise networks [43]. It takes a stream of network packets captured at different switches as input and computes key metrics like the number of active connections and bandwidth usage on a windowed basis. The system utilizes an event streaming platform to collect mirrored packets and a stream processing engine to calculate the desired metrics.

`Video analysis` application evaluates the performance of a stream processing framework for analyzing video traffic. It simulates a video streaming scenario and scales the number of consumers to assess the impact on system performance. The application utilizes *streamline*'s publish-subscribe messaging and scalability features to efficiently distribute and process the video stream data.

6.2 Emulating Networking Conditions

Cloud organizations have increasingly deployed geographically distributed services to reduce wide area network (WAN) traffic originating from data transmissions and minimize query response times. For example, many cloud providers use edge servers to partially aggregate data streams from multiple user (or IoT) devices before sending them to a data center for analytics [44]. The large variability of WAN bandwidth and latency though (up to 20x in production environments [45]) can directly affect the correctness and performance of stream processing applications, making it imperative for developers to fully understand the application's behavior under varying networking conditions.

Unfortunately, running a stream processing system in a real geo-distributed setup is challenging. It may require provisioning resources on several (edge) data centers and carefully crafting (or observing) desired running conditions, e.g., high-link delays and varying bandwidth. Furthermore,

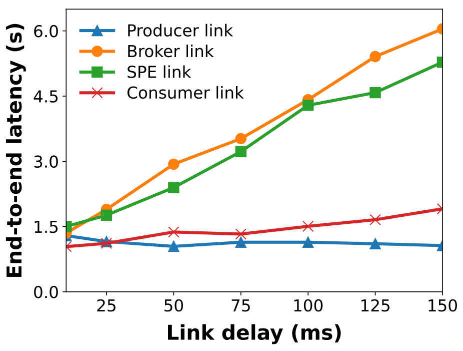


Fig. 6: End-to-end latency for the word count application when varying the link delay to reach out to each of its components. At each run, we increase the link delay of a single component and keep the remaining ones at a very low value (< 10 ms).

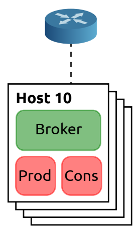
isolating the application's response to relevant events, e.g., a processing stall, from co-located services such as a co-hosted virtual machine may not be possible. *streamline* offers a simplified alternative to test geo-distributed environments. In the following, we present two scenarios for testing link delay and bandwidth (the most common metrics) over geo-distributed stream processing applications.

Varying Link Delay: We conduct delay assessment over the `Word count` application while systematically varying link delays. Specifically, we increase the link delay for communicating with a chosen host while keeping the delay for the remaining ones at a very low value (< 10 ms).

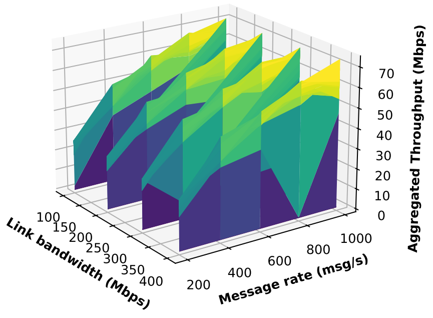
Figure 6 shows the end-to-end latency for processing a data unit (i.e., a text file) throughout the word count pipeline. Each point depicts the average latency of over 100 files. As expected, higher link delays impact the performance of all application components. Interestingly, the impact is more prominent when the data broker and the stream processing engine (e.g., Apache Spark) delays increase, up to 6x worse for a link delay of 150 ms. This highlights the fact that application components in a data processing pipeline may have distinct networking requirements and calls for a careful allocation of infrastructure resources. In particular, the data broker constantly communicates with *all* other components in our experiment and, therefore, is more susceptible to poor networking conditions.

Varying Link Bandwidth: We conduct the experiment over the `Military coordination` application, where 10 coordinating sites (i.e. hosts) are distributed over various locations (Figure 7a). There are 10 message brokers form a star topology and deal with 2 topics. Each host also runs: i) a data producer that randomly injects data into the topics; and ii) a consumer that collects data from the topics. This deployment scenario is inspired by standard streaming deployments [46], [47]. We systematically vary the available bandwidth for all brokers from 20 to 400 Mbps (following [45]), and for each bandwidth setting, we incrementally increase the message rate until the system's throughput reaches its limit.

Figure 7b illustrates the interplay between link bandwidth, message rate, and the resulting aggregated throughput. The plot shows that as link bandwidth increases, the



(a)



(b)

Fig. 7: *a)* Evaluation setup for military coordination application; *b)* Aggregated throughput with varying link bandwidth under different message rates.

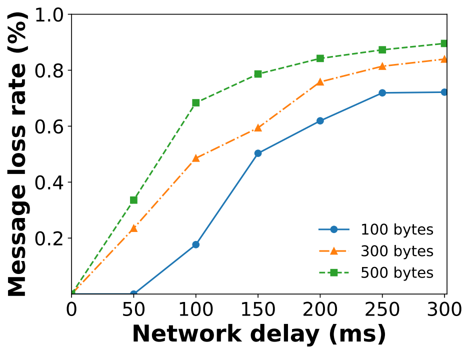


Fig. 8: Message loss rate vs. network delay for varying message sizes in military coordination application.

aggregated throughput also increases, especially at higher message rates. At the highest message rate of 1000 messages per second, the aggregated throughput increases sharply from around 10 Mbps to approximately 70 Mbps as the link bandwidth is increased from 100 Mbps to 400 Mbps. This demonstrates that a 4x increase in bandwidth leads to a 7x increase in aggregated throughput at high message rates, emphasizing how bandwidth and message rate together critically impact performance. The key takeaway is that bandwidth alone does not dictate throughput; the message rate also plays a crucial role. When the message rate is low, the system easily handles the traffic and additional bandwidth yields marginal benefits. In contrast, at higher rates, throughput scales non-linearly with bandwidth, revealing a saturation effect that emerges only under increased load. This controlled behavior establishes *streamline* 's accuracy under stress, which we build upon in Section 6.3 to explore less predictable behaviors under fault scenarios.

6.3 Testing Reliability

Network reliability is pivotal in distributed stream processing applications, particularly due to the inherent complexities and dynamic nature of distributed networks. Recent studies underscore the critical role of network quality in maintaining the efficiency and reliability of stream processing systems [48]. Alongside, production networks can experience network partitions as frequently as once a week, with repairs potentially taking hours [49]. Despite software and data redundancy being widespread on current stream processing systems, many of them still experience silent catastrophic failures when a network partition happens [50]. Reproducing, diagnosing, and hardening stream processing systems against such failures can be rather complicated due to the lack of proper tools. *streamline* fills this gap by allowing quickly and flexibly inject network-partitioning failures (e.g., after bringing network links down) into distributed stream processing systems.

To evaluate the reliability assessment capabilities of *streamline*, we conduct three sets of experiments. First, we examine how *streamline* can inject network faults and measure their impact on end-to-end data delivery. Second, we evaluate how it enables assessment of the fault tolerance behavior of event streaming platforms under failure scenarios like leader broker disconnection and high network delay. Third, we investigate *streamline* 's ability to assess stream processing engines (e.g., Apache Flink, Apache Spark) under worker node failures and their subsequent recovery.

Network Fault Injection: To evaluate the impact of network-induced faults on data delivery, we use the `Military coordination` application, where we emulate high network delays to simulate fault conditions. In this application, Kafka instances operate under at-most-once semantics, where messages are delivered no more than once, thereby reducing the potential for message duplication but increasing the risk of message loss in the case of network faults. This approach prioritizes avoiding message duplication at the cost of potential data loss. The producer generates a total of 350,000 messages. Also, it is set for zero acknowledgments to enforce at-most-once semantics, i.e., it does not wait for any confirmation from the server, enhancing the transmission speed but at the risk of message loss. We vary network delays from 1 ms to 300 ms prior to message transmission and maintain these conditions until all messages are sent.

Figure 8 shows the impact of network delay on message loss rates for different message sizes, where each point represents the average message loss rate at specific delays ranging from 0 to 300 ms for each message size. It highlights that larger messages experience significantly higher loss rates under increased delays, reaching up to 80% at 300 ms, compared to smaller messages, which show better resilience. This trend indicates that larger messages are more prone to buffer overflows and timeouts under high-delay conditions, emphasizing the need for optimizing message sizes and buffering strategies to maintain data reliability in latency-sensitive environments.

Event Streaming Platform Fault Tolerance: Here, we assess how event streaming platforms tolerate failures by injecting a broker disconnection fault and analyzing message delivery behavior in Apache Kafka. In the `Military coordination` application, we randomly disconnect the node hosting the leader broker for one of the two topics for 120 seconds (approximately 20% of the total experiment duration). Figure 9a shows the data delivery matrix for the producer that is co-located with the disconnected broker.

Each light color cell indicates whether a given consumer received a message. We observe no message loss during the disconnection period, even from the topic whose leader is temporarily unreachable. This behavior demonstrates the in-built fault resilience and consistency guarantees in Kafka. Specifically, Kafka's adoption of the KRaft mode in the recent versions and the new "Pre-Vote" mechanism [51] that helps reduce unnecessary leader elections by allowing nodes to assess their leadership viability before triggering the election process. This preemptive check significantly mitigates disruption during transient failures or network partitions, ensuring smoother recovery and more reliable message delivery.

We also measure the impact of network partitioning on message latency, i.e., the time for a published message to be available at a subscriber using *streamline*. Figure 9b shows the message latency at a random consumer (all consumers present a similar behavior). We classify messages according to their topic and order them based on their receiving time (older messages first). As we can observe, there are two latency spikes throughout the experiment, each affecting one of the topics. In both cases, the increased latency stems from the message commit process. For topic A (TA), whose leader got disconnected, all produced messages are put on hold until a new leader is elected, which then resumes accepting and delivering messages in place of the disconnected broker. Topic B (TB), on the other hand, only delays messages from the disconnected producer since the leader broker is available at all times. In this case, the disconnected producer tries to re-send messages until they are either accepted or a timeout occurs, and excessively long timeouts may incur on latency inflation.

We can also see the impact of network failures on the required bandwidth. In particular, Figure 9c shows the sending throughput of relevant hosts over time. After the disconnection (①), the TA leader stops serving requests and a replica broker assumes its role. We then observe two spikes on the required bandwidth: the first (②) comprises the new leader acknowledging and committing the backlog of messages that were produced during its election process while the second (③) involves serving the same backlog to the subscribed consumers. When the old leader reconnects, it eventually re-assumes topic A leadership (④) due to Kafka's preferred replica election mechanism.

Stream Processing Engine Recovery: We evaluate stream processing engines resilience under compute node failures. We incorporate controlled worker disconnection fault test in `Network traffic monitoring` application to enhance the reliability assessment of stream processing engines. This application involves selection, group-by, and windowed processing operations, and is configured to ensure at-least-once processing semantics. In this experiment, we simulate a worker node failure by disconnecting a single worker node from the cluster for a duration of 120 seconds. This approach aligns with chaos engineering principles [52], introducing deliberate disruptions to assess system resilience. After the two-minute interval, the worker node reconnects to the cluster. We perform these experiments on both Apache Flink and Spark frameworks. For Flink, we utilize its asynchronous barrier snapshotting mechanism for checkpointing, which allows the system to capture consistent snapshots of the application state without halting the data processing. For Spark, we enable checkpointing to persist the state of streaming computations, facilitating recovery in the event of failures. Throughout the experiments, we monitor key performance metrics, including throughput, latency, and recovery time. Throughput is measured as the number of records processed per second, latency as the time taken to process each record, and recovery time as the duration from the detection of the worker node failure to the point when the system resumed normal processing rates.

The results of fault injection experiments depicted in Figure 10 reveal distinct recovery behaviors between Flink and Spark. Following the two-minute disconnection, both frameworks encounter transient throughput drops and latency spikes. Flink promptly resumed stable operation, with throughput recovering in under 2 minutes and latency returning to baseline shortly thereafter. This rapid recovery can be attributed to Flink's fine-grained event-level checkpointing and backpressure mechanisms, which enable smooth state replay and minimize output disruption. Notably, no signs of message loss or duplication is observed, and the quick clearance of consumer lag further confirms the effectiveness of Flink's recovery process.

In contrast, Spark exhibits a more gradual recovery trajectory, with throughput stabilizing approximately three minutes after the worker rejoins. Although no record loss or duplication is detected, the recovery latency remained elevated for a longer duration. This is consistent with Spark's micro-batch-based execution and checkpointing model, where state recovery is triggered at batch boundaries. While this model ensures deterministic recovery, it introduces moderate delay and batch-level backpressure, resulting in a slower yet smoother recovery process. These observations underscore Flink's superior fault tolerance and recovery efficiency in handling transient worker failures, aligning with findings from recent studies on stream processing fault recovery performance [53].

7 Performance Evaluation of *streamline*

This section presents the performance of *streamline* by answering four questions: i) how accurate the framework is compared to testbed, virtualized cloud environment and state-of-the-art solution (Section 7.1); ii) how much resources it requires for running reasonably large experiments (Section 7.2); iii) how easy it is to deploy a cluster of workers in *streamline* and compare the results with a similar deployment (Section 7.3); and iv) how the framework can be used to benchmark different stream processing platforms (Section 7.4).

7.1 Dependable Accuracy

This evaluation aims to demonstrate the dependability of *streamline* as an alternative to hardware platforms. We also compare its performance against the OpenStack cloud setup. The chosen applications are `Word count` and `Military coordination`, as the former is a standard streaming application for performance assessment, and the latter is a complex one with intermittent network conditions.

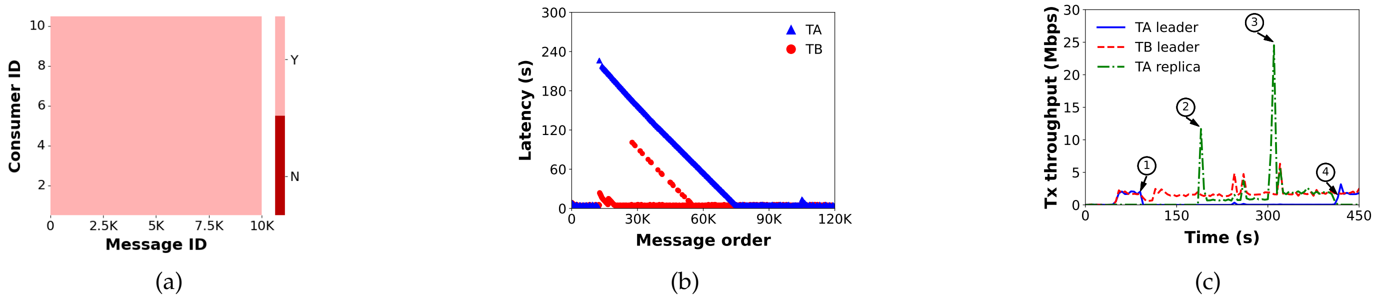


Fig. 9: *a)* Message delivery matrix for the co-located producer with a disconnected broker. Y = Message delivered. N = Message *not* delivered; *b)* Message latency at a consumer. TA/TB = Topic A/B; *c)* Sending throughput of designated hosts. Events of interest: ① = TA leader disconnection. ② = New leader election. ③ = Message backlog serving. ④ = Original leadership re-establishment. All results are for the military coordination application.

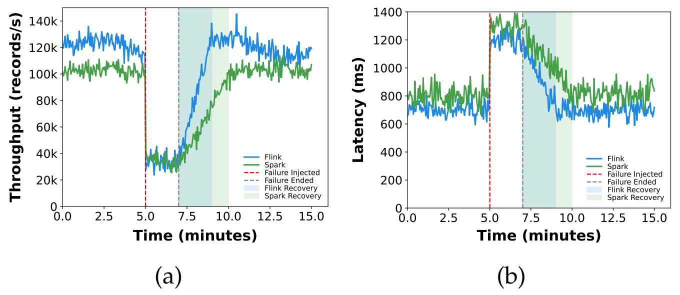


Fig. 10: Stream processing engine fault recovery under a two-minute worker disconnection in the network traffic analysis application. (a) Throughput (records/s) and (b) latency (ms) over time for Flink and Spark. Annotations indicate failure injection, end of failure, and recovery points.

We use Network Time Protocol (NTP) server[8] to synchronize clocks. Moreover, all measurements are collected after a 60-second warm-up interval to initialize every application component and configure the network routes. We use Linux traffic control utility, `tc`[9], to configure the link properties in the hardware setup. By modifying the queuing disciplines (qdiscs) of the network interface, `tc` can configure link properties for both inbound and outbound traffic. Qdiscs are elements that help in queuing and scheduling traffic transmission by a network interface.

The `Word count` application takes advantage of both event streaming (Apache Kafka) and a streaming processing engine (Apache Spark). Then, we inject a stream of data (i.e., text files) into its processing pipeline as quickly as possible. The `Military coordination` application utilizes Apache Kafka for event streaming as part of the pipeline, where each producer consistently generates data (usually position reports) at 30 Kbps, replicating the behaviors of entities like soldiers in a tactical unit. Also, the producer and consumer co-locate with the broker in the same node.

Figure 11 illustrates the end-to-end latency for both applications under varying link delays across different pipeline components. The results demonstrate a consistent pattern across all environments, validating *streamline*'s accuracy in replicating real-world scenarios. Notably, the cloud environment consistently exhibits higher latency compared to both *streamline* and the hardware setup. This overhead, ranging from 15-33%, is likely attributed to the inherent characteristics of cloud environments, such as network virtualization and resource sharing, which introduce additional layers of abstraction and potential contention for resources [54]. Based on this finding, it is recommended that for applications where precise latency measurements are crucial, and network overhead is a concern, a hardware testbed or *streamline* should be the preferred testing environment. However, a cloud-based environment like OpenStack is suitable if scalability is prioritized over precise latency control.

To further assess *streamline*'s accuracy in comparable realistic deployments, we conduct two additional experiments. First, we compare its performance against NAMB[10] [13], a state-of-the-art prototype generator framework for distributed stream processing systems. To evaluate the handling of bursty traffic and backpressure behavior, we configure both NAMB and *streamline* with identical pipeline that combined Kafka and Flink under acknowledgment-enabled conditions. In this setup, a producer issues tuples at a constant rate with periodic burst phases to simulate high-load scenarios. Kafka acts as the message broker and Flink processes the tuples using a stateless job. In NAMB, bursts are generated via a synthetic load generator with busy-wait loops, whereas *streamline* utilizes realistic message injection and rate control via Mininet-configured links. A burst in this context refers to a temporary increase in message arrival rate (e.g., 2x–3x the nominal rate), triggering flow control mechanisms in the pipeline. As shown in Figure 12, both frameworks maintain stable throughput under regular load. However, during burst periods, *streamline* consistently achieves higher throughput. This improvement is attributed to *streamline* 's resource isolation model, which reduces coordination overhead during peak loads.

Second, to approximate typical datacenter setups, we use our existing hardware testbed as a practical alternative, since we do not have access to a dedicated datacenter infrastructure. We adjust network parameters to reflect key

8. https://www.pool.ntp.org/zone/ca
9. https://man7.org/linux/man-pages/man8/tc.8.html

10. Despite our efforts, we could not compare *streamline* against TRAK [12], another state-of-the-art solution, due to the lack of artifact availability as well as dependency on outdated versions of Kafka and Docker.

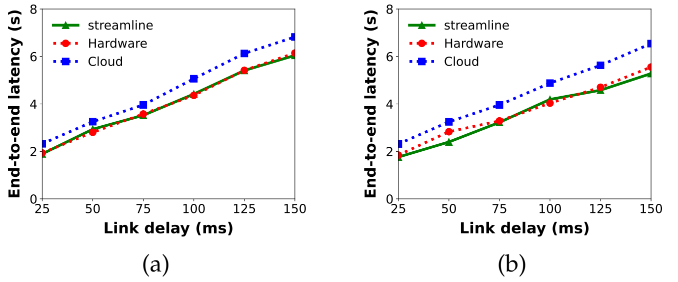


Fig. 11: Comparison among *streamline*, hardware testbed, and OpenStack while varying a) stream processing engine link delay for word count application and b) event streaming link delay for military coordination application.

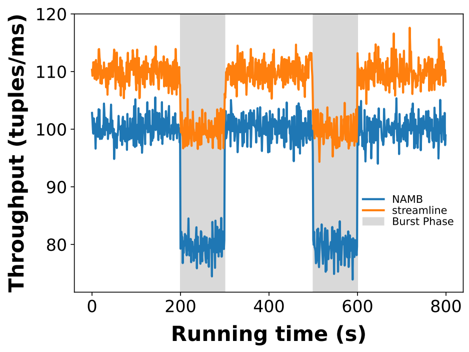


Fig. 12: Throughput comparison between *streamline* and NAMB during backpressure test with reliability mechanism enabled. Shaded regions represent burst phases.

datacenter characteristics by configuring latencies in the range of hundreds of microseconds [55] and ensuring no artificial faults are introduced. We then replicate these application setups within *streamline* using similar low-latency network configurations and no fault injections. Our results demonstrate that *streamline*'s performance closely matches the mimicked data center behavior, with less than 10% deviation in end-to-end latency under comparable conditions. This result aligns with our earlier result (Figure 11) and indicates that *streamline* effectively models real-world datacenter environments for performance assessment.

7.2 Resource Usage

This section presents the resource usage of *streamline* for the `Military coordination` application (uses the largest number of nodes in the pipeline). Specifically, we measure the CPU and memory utilization by snapshotting `/proc/stat/` and `/proc/meminfo/`, respectively, every 500 milliseconds. All measurements are collected after a 60 seconds warm-up interval.

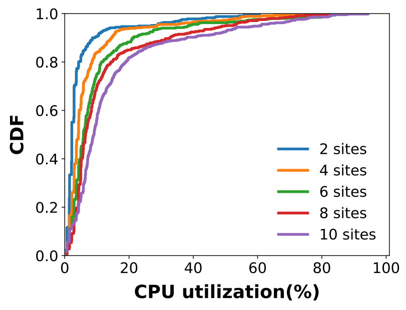
Figure 13a shows the cumulative distribution function (CDF) of the CPU utilization for different numbers of coordinating sites. Our analysis shows that the CPU utilization is reasonably low (less than 60%) most of the time (more than 90%), even when we have 10 coordinating sites. Each site hosts a message broker, a data producer and a consumer. In particular, most of the CPU demand stems from the system setup when *streamline* needs to initialize all application components. We also investigate how quickly the CPU utilization grows as we increase the number of sites. In Figure 13b, we plot the median CPU utilization for up to 10 sites. As we can observe, *streamline* scales to tens of application components (each coordinating site has three components) with a minimal 7% increase in CPU usage. Moreover, the overall CPU demand is low (around 10%) even for the largest scenario.

Figure 13c shows the peak memory usage of *streamline* for different numbers of coordinating sites. We also consider two buffer sizes at data producers (16 and 32 MB) to assess the impact of application component configurations on the overall platform resource consumption. This buffer size reflects the amount of memory a producer reserves for queuing messages that are waiting to be sent to a broker (e.g., if the producer is sending messages faster than the broker can handle). We can see that the required memory grows linearly as the number of coordinating sites increases, yet the overall increment is low (less than 25% in total in our experiment). Moreover, the buffer size has a non-negligible impact on *streamline*'s memory consumption (as much as 16% in our test), which indicates the framework can be further optimized to accommodate bigger setups depending on how flexible it is to configure an application component. Likewise, we envision our framework can be used as a playground for automatically tuning stream processing system parameters [56]. We leave exploring both directions for future work. Finally, as *streamline* aims to provide a generalized framework for deploying and evaluating stream processing applications under realistic performance and resilience constraints, its CPU and memory usage is our primary focus without considering its cost. However, we plan to explore this cost assessment of *streamline* in supported applications and deployment scenarios.

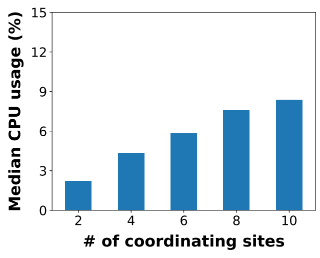
7.3 Support for Cluster Deployment

This section demonstrates *streamline*'s support for assessing applications in a cluster setup required in some applications for scalability and fault tolerance [53]. *streamline* supports clustering on a single node in standalone mode, i.e., independent of external resource managers like Apache Mesos or Hadoop YARN. We implement `Network traffic monitoring` application in this cluster setup and assess how the system scales as the number of Spark workers and network users increases. Each user generates traffic to a pre-defined set of services (e.g., FTP, Web, DNS) following a Poisson process. This configuration requires less than 100 lines of additional code (beyond the core application logic) in GraphML and YAML formats (example cluster configuration in Figure 14a).

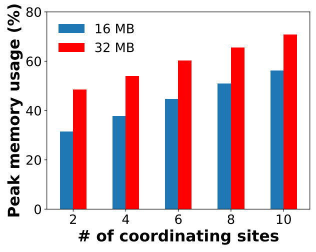
Figure 14b conveys two messages: *streamline*'s ability to reproduce prior research work and the performance benefits of cluster deployment. We successfully reproduced results from Ocampo *et al.* [43], indicating the research work reproducibility feature of *streamline*. Specifically, in a single worker scenario, the performance of *streamline* closely matches the reported results, even under varying concurrent user loads, except with 20% variations only at very high user counts (e.g., 100 concurrent users). The performance trend is similar with two and three Spark workers. Moreover,



(a)

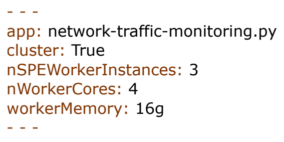


(b)



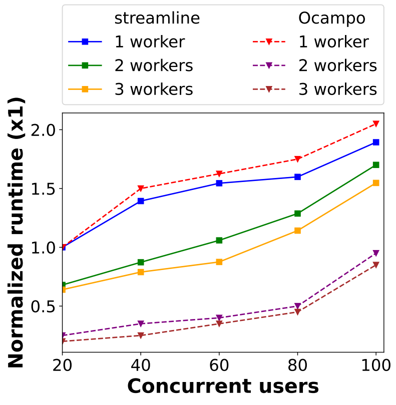
(c)

Fig. 13: *a)* CDF and *b)* median CPU utilization as we vary the number of coordinating sites, for the military coordination application described in Figure 7a; *c)* Peak memory usage for the same scenario considering different buffer sizes on data producers.



```
- - -
app: network-traffic-monitoring.py
cluster: True
nSPEWorkerInstances: 3
nWorkerCores: 4
workerMemory: 16g
- - -
```

(a)



(b)

Fig. 14: (a) Clustering YAML configurations in *streamline*; and (b) Comparison of normalized runtimes: *streamline* vs. Ocampo *et al.* for network traffic monitoring application in one to three workers setup with increasing concurrent users.

due to the additional computing resources and workers, we can now support more concurrent users within a certain time, highlighting the importance of cluster deployment in enhancing scale and speed.

7.4 Benchmarking Stream Processing Platforms

This evaluation again leverages the `Military coordination` application[11] to benchmark the performance of Apache Kafka and RabbitMQ (rMQ) as representative event streaming platforms. We choose these platforms as both Kafka and rMQ offer robust data access through extensive history logs, ensuring data remains available even after disconnections. Kafka's polling mechanism provides immediate feedback on connection status, although it may introduce some latency and additional network traffic. rMQ excels in complex messaging scenarios with flexible routing and delivery options, but this can impact its overall throughput.

We consider a cluster-based architecture where each tactical team operates a message broker, producer, and consumer. Log replication across all cluster nodes guarantees continued operation in case of tactical team disconnection. In essence, each broker must (i) sustain message forwarding within a team during disconnection and (ii) automatically synchronize intra-team messages with other units upon reconnection. We measure the total throughput (bandwidth demand) and the message latency. In our setup, each producer continuously generates data at 30 Kbps, simulating real-world entities within a tactical team. Data collection begins after a 60-second warm-up period to allow the system to stabilize. We adjust link delays between 1 and 500 ms in the latency experiment, reflecting representative network deployment. Furthermore, we modify both Kafka and rMQ connection timeouts to function correctly under these conditions.

Figure 15a illustrates the bandwidth demand for Kafka and RabbitMQ across varied cluster sizes. Notably, rMQ's bandwidth requirement surpasses that of Kafka by approximately 13x in a 10-node topology. This difference stems from Kafka's message compression and rMQ's sequential delivery (preventing batching), which leads to more overhead. Additionally, our study shows Kafka's bandwidth usage grows faster than linear growth when scaling from 10 to 20 nodes due to increased overhead from coordination, replication, and network traffic in larger clusters.

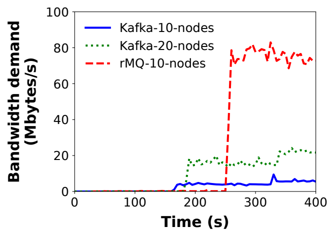
Figure 15b displays the cumulative distribution function (CDF) of message latency in Kafka and rMQ under different link delays. Overall, latency increases considerably (over 300% and 85% at the median for Kafka and rMQ, respectively) in a highly constrained network compared to a baseline scenario with negligible link delay. Furthermore, rMQ consistently underperforms Kafka. This is mainly due to two factors: (i) AMQP[12] connections necessitate multiple RTTs before message forwarding can begin, and (ii) rMQ brokers must await consumer acknowledgments before sending subsequent messages.

8 Conclusion

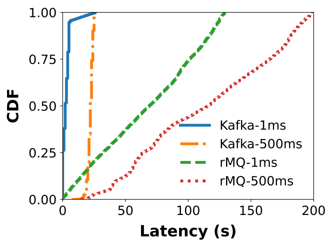
streamline stands out as a holistic approach for the prototyping and testing stream processing applications, effectively

11. While we utilize the Military Coordination application for benchmarking the performance of Apache Kafka and RabbitMQ, *streamline*'s modular design allows researchers to easily integrate and utilize other applications for benchmarking additional streaming platforms.

12. Advanced Message Queuing Protocol (AMQP) is the backbone application layer protocol used by rMQ.



(a)



(b)

Fig. 15: (a) Total bandwidth demand from both Kafka and rMQ for different cluster sizes; and (b) Cumulative distribution function (CDF) of the message latency from both Kafka and rMQ for different link delays.

addressing the complex challenges within real-time data analytics. Using a developer-friendly API, it enables seamless setup of network and stream processing configurations. We have demonstrated that *streamline* is dependable and efficient by deploying and assessing a number of popular streaming applications. Furthermore, its performance is comparable to a hardware testbed with efficient resource usage, utilizing less than 10% of server CPU resources even when testing complex component networks. These benefits, achieved using accessible computing resources, make *streamline* a cost-effective tool. Overall, *streamline* is pioneering automated *end-to-end* testing and configuration of stream processing systems, promoting innovation and enhancing system performance and reliability in distributed stream processing environments. As part of future work, we will extend the cluster deployment capability of *streamline* with external resource managers (e.g., Hadoop YARN) for additional capabilities of assessment and benchmarking. We will also replicate *streamline* in datacenter environments with empirically measured network characteristics in capturing real-world performance variations.

References

[1] X. Liu, N. Iftikhar, and X. Xie, "Survey of real-time processing systems for big data," in *Proc. of the IDEAS*, 2014.
[2] Apache kafka. Accessed: 2025-04-29. [Online]. Available: https://kafka.apache.org/.
[3] Structured streaming programming guide - spark 3.5.5 documentation. Accessed: 2025-04-29. [Online]. Available: https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html/.
[4] Structured streaming programming guide - spark 3.5.5 documentation. Accessed: 2025-04-29. [Online]. Available: https://nightlies.apache.org/flink/flink-docs-release-2.0/docs/dev/datastream/overview/.
[5] P. Carbone, M. Fragkoulis, V. Kalavri, and A. Katsifodimos, "Beyond analytics: The evolution of stream processing systems," in *Proc. of the ACM SIGMOD*, 2020.
[6] ns-3 network simulator. Accessed: 2023-01-06. [Online]. Available: https://www.nsnam.org.
[7] Omnet++ discrete event simulator. Accessed: 2023-01-06. [Online]. Available: https://omnetpp.org/.
[8] H. H. Liu, Y. Zhu, J. Padhye, J. Cao, S. Tallapragada, N. P. Lopes, A. Rybalchenko, G. Lu, and L. Yuan, "Crystalnet: Faithfully emulating large production networks," in *Proc. of the ACM SOSP*, 2017.
[9] J. Cao, Y. Liu, Y. Zhou, L. He, and M. Xu, "Turbonet: Faithfully emulating networks with programmable switches," *IEEE/ACM TON*, 2022.
[10] L. Peterson, A. Bavier, M. E. Fiuczynski, and S. Muir, "Experiences building planetlab," in *Proc. of the USENIX OSDI Symp.*, 2006.
[11] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar, "An integrated experimental environment for distributed systems and networks," in *Proc. of the USENIX OSDI Symp.*, 2002.
[12] H. Wu, Z. Shang, and K. Wolter, "Trak: A testing tool for studying the reliability of data delivery in apache kafka," in *IEEE ISSREW*, 2019.
[13] A. Pagliari, F. Huet, and G. Urvoy-Keller, "Namb: A quick and flexible stream processing application prototype generator," in *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. IEEE, 2020, pp. 61–70.
[14] Confluent — apache kafka® reinvented for the cloud. Accessed: 2024-08-30. [Online]. Available: https://www.confluent.io/.
[15] The data and ai company — databricks. Accessed: 2024-08-30. [Online]. Available: https://www.databricks.com/.
[16] Mininet: An instant virtual network on your laptop (or other pc). Accessed: 2023-01-16. [Online]. Available: http://mininet.org/.
[17] streamline code repository. [Online]. Available: https://github.com/PINetDalhousie/streamline
[18] M. M. Amin Ifath, M. Neves, and I. Haque, "Fast prototyping of distributed stream processing applications with stream2gym," in *Proc. of the IEEE ICDCS Conf.*, 2023.
[19] Y. Fu and C. Soman, "Real-Time Data Infrastructure at Uber," in *Proc. of the ACM SIGMOD Conf.*, 2021.
[20] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb *et al.*, "The design and operation of {CloudLab}," in *Proc. of the USENIX ATC*, 2019.
[21] K. Keahey, J. Anderson, Z. Zhen, P. Riteau, P. Ruth, D. Stanzione, M. Cevik, J. Colleran, H. S. Gunawi, C. Hammock, J. Mambretti, A. Barnes, F. Halbach, A. Rocha, and J. Stubbs, "Lessons Learned from the Chameleon Testbed," in *Proc. of the USENIX ATC*, 2020.
[22] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown, "Reproducible Network Experiments Using Container-Based Emulation," in *Proc. of the ACM CoNEXT*, 2012.
[23] A. Vianna, W. Ferreira, and K. Gama, "An exploratory study of how specialists deal with testing in data stream processing applications," in *Proc. of the ACM/IEEE ESEM*. IEEE, 2019, pp. 1–6.
[24] Testing kafka streams. Accessed: 2023-01-06. [Online]. Available: https://kafka.apache.org/20/documentation/streams/developer-guide/testing.html.
[25] Framework for apache flink unit tests. Accessed: 2023-01-06. [Online]. Available: https://github.com/ottogroup/flink-spector.
[26] S. Mostafa and X. Wang, "An empirical study on the usage of mocking frameworks in software testing," in *Proc. of the IEEE QSIC Conf.*, 2014, pp. 127–132.
[27] K. Kallas, F. Niksic, C. Stanford, and R. Alur, "Diffstream: differential output testing for stream processing programs," *Proc. of the ACM Program. Lang.*, vol. 4, no. OOPSLA, pp. 1–29, 2020.
[28] E. Asyabi, Y. Wang, J. Liagouris, V. Kalavri, and A. Bestavros, "A new benchmark harness for systematic and robust evaluation of streaming state stores," in *Proc. of the ACM EuroSys Conf.*, 2022.
[29] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and V. Markl, "Benchmarking distributed stream data processing systems," in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, 2018, pp. 1507–1518.
[30] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng *et al.*, "Benchmarking streaming computation engines: Storm, flink and spark streaming," in *Proc. of the IEEE IPDPSW*, 2016.
[31] S. Henning, A. Vogel, M. Leichtfried, O. Ertl, and R. Rabiser, "Shufflebench: A benchmark for large-scale data shuffling operations with distributed stream processing frameworks," in *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering*, 2024, pp. 2–13.
[32] G. Di Lena, A. Tomassilli, D. Saucez, F. Giroire, T. Turletti, and C. Lac, "Distrinet: A mininet implementation for the cloud," *in ACM SIGCOMM Comput. Commun. Rev.*, 2021.
[33] Mininet cluster edition. Accessed: 2023-01-06. [Online]. Available: https://github.com/mininet/mininet/wiki/Cluster-Edition-Prototype.
[34] The graphml file format. Accessed: 2023-01-06. [Online]. Available: http://graphml.graphdrawing.org/.
[35] The official yaml web site. Accessed: 2023-01-19. [Online]. Available: https://yaml.org/.

[36] E. C. Molero, S. Vissicchio, and L. Vanbever, "Fast in-network gray failure detection for isps," in *Proc. of the ACM SIGCOMM Conf.*, 2022.
[37] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro *et al.*, "Applied machine learning at facebook: A datacenter infrastructure perspective," in *Proc. of the IEEE HPCA Conf.*, 2018.
[38] N. Kang, Z. Liu, J. Rexford, and D. Walker, "Optimizing the "one big switch" abstraction in software-defined networks," in *Proc. of the ACM CoNEXT Conf.*, 2013.
[39] Rabbitmq: One broker to queue them all — rabbitmq. Accessed: 2025-04-29. [Online]. Available: https://www.rabbitmq.com/.
[40] M. M. A. Ifath, M. Neves, B. Bremner, J. White, T. Szeredi, and I. Haque, "Are data streaming platforms ready for a mission critical world?" *Authorea Preprints*, 2024.
[41] G. Van Dongen and D. Van den Poel, "Evaluation of stream processing frameworks," *IEEE TPDS*, 2020.
[42] Open source cloud computing infrastructure - openstack. Accessed: 2024-07-18. [Online]. Available: https://www.openstack.org/.
[43] A. F. Ocampo Palacio, T. Wauters, B. Volckaert, and F. De Turck, "Scalable distributed traffic monitoring for enterprise networks with spark streaming," in *Proc. of the ECCWS*, 2018.
[44] D. Kumar, S. Ahmad, A. Chandra, and R. K. Sitaraman, "Aggnet: Cost-aware aggregation networks for geo-distributed streaming analytics," in *Proc. of the IEEE/ACM SEC Conf.*, 2021.
[45] F. Lai, M. Chowdhury, and H. Madhyastha, "To relay or not to relay for Inter-Cloud transfers?" in *Proc. of the USENIX HotCloud Conf.*, 2018.
[46] L. Rosa, W. Song, L. Foschini, A. Corradi, and K. Birman, "Derechodds: Strongly consistent data distribution for mission-critical applications," in *IEEE MILCOM*, 2021.
[47] J. Miao, N. Lv, Q. Gao, K. Chen, and X. Wang, "Fault-tolerant embedding algorithm for node failure in airborne tactical network virtualization," *IEEE Access*, 2022.
[48] C. Stanford, K. Kallas, and R. Alur, "Correctness in stream processing: Challenges and opportunities," in *Proc. of the CIDR*, 2022.
[49] R. Govindan, I. Minei, M. Kallahalla, B. Koley, and A. Vahdat, "Evolve or die: High-availability design principles drawn from googles network infrastructure," in *Proc. of the ACM SIGCOMM Conf.*, 2016.
[50] A. Alquraan, H. Takruri, M. Alfatafta, and S. Al-Kiswany, "An analysis of network-partitioning failures in cloud systems," in *Proc. of the USENIX OSDI Conf.*, 2018.
[51] Kip-996: Pre-vote - apache kafka - apache software foundation. Accessed: 2025-04-29. [Online]. Available: https://cwiki.apache.org/confluence/display/KAFKA/KIP-996%3A+Pre-Vote/.
[52] A. Basiri, N. Behnam, R. De Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal, "Chaos engineering," *IEEE Software*, vol. 33, no. 3, pp. 35–41, 2016.
[53] A. Vogel, S. Henning, E. Perez-Wohlfeil, O. Ertl, and R. Rabiser, "A comprehensive benchmarking analysis of fault recovery in stream processing frameworks," *arXiv preprint arXiv:2404.06203*, 2024.
[54] F. Xu, F. Liu, H. Jin, and A. V. Vasilakos, "Managing performance overhead of virtual machines in cloud computing: A survey, state of the art, and future directions," *Proc. of the IEEE*, 2014.
[55] R. Iyer, M. Unal, M. Kogias, and G. Candea, "Achieving microsecond-scale tail latency efficiently with approximate optimal scheduling," in *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023, pp. 466–481.
[56] M. Bilal and M. Canini, "Towards automatic parameter tuning of stream processing systems," in *Proc. of the ACM SoCC*, 2017.

Md. Monzurul Amin Ifath is a Doctoral student at Dalhousie University working on distributed systems, computer networks, and machine learning. He holds a Bachelor's degree from BUET, Bangladesh, and has three years of experience working with DRDC and GDMS-C on collaborative projects.

Tommaso Melodia is the William Lincoln Smith Professor with the Department of Electrical and Computer Engineering at Northeastern University in Boston. He is also the Founding Director of the Institute for the Wireless Internet of Things and the Director of Research for the PAWR Project Office. He received his Laurea (integrated BS and MS) from the University of Rome - La Sapienza and his Ph.D. in Electrical and Computer Engineering from the Georgia Institute of Technology in 2007. He is an IEEE Fellow, an ACM Distinguished Member, and a recipient of the National Science Foundation CAREER award. He received several best paper awards, including at IEEE Infocom 2022. Prof. Melodia is the Editor in Chief for Computer Networks and a co-founder of the 6G Symposium, and served as the Technical Program Committee Chair for IEEE Infocom, and General Chair for ACM MobiHoc, among others. Prof. Melodia's research on modeling, optimization, and experimental evaluation of wireless networked systems has been funded by many US government and industry entities.

Israat Haque is an Associate Professor in the Faculty of Computer Science at Dalhousie University, where she leads the Programmable and Intelligent Networking (PINet) Lab. Her expertise is in systems and security. Specifically, she leverages network programmability to develop cutting-edge, high-performance, secure, and dependable systems for AI/ML, Big Data, cloud, telecommunication, and IoT systems. She is also interested in applying data-driven approaches to solve practical and relevant problems. Dr. Haque received her PhD from the Department of Computing Science at the University of Alberta. Subsequently, she held an NSERC post-doctoral position at the Department of Computer Science and Engineering at the University of California, Riverside, before joining Dalhousie University. She was recognized as an ACM/IEEE N2Women Rising Star in 2021 for her research and leadership contributions. Subsequently, she received Digital Nova Scotia's Thinking Forward Award 2022 for training the next generation of tech talents. In 2024, she received the Alumni Honour Award from the University of Alberta.