Fast Prototyping of Distributed Stream Processing Applications with *stream2gym*

Md. Monzurul Amin Ifath, Miguel Neves, Israat Haque
Dalhousie University

***Abstract*—Stream processing applications have been widely adopted due to real-time data analytics demands, e.g., fraud detection, video analytics, IoT applications. Unfortunately, prototyping and testing these applications is still a cumbersome process for developers that usually requires an expensive testbed and deep multi-disciplinary expertise, including in areas such as networking, distributed systems, and data engineering. As a result, it takes a long time to deploy stream processing applications into production and yet users face several correctness and performance issues. In this paper, we present *stream2gym*, a tool for the fast prototyping of large-scale distributed stream processing applications. *stream2gym* builds on Mininet, a widely adopted network emulation platform, and provides a high-level interface to enable developers to easily test their applications under various operating conditions. We demonstrate the benefits of *stream2gym* by prototyping and testing several applications as well as reproducing key findings from prior research work in video analytics and network traffic monitoring. Moreover, we show *stream2gym* presents accurate results compared to a hardware testbed while consuming a small amount of resources (enough to be supported in a single commodity laptop even when emulating a dozen of processing nodes).**

I. Introduction

Stream processing applications have become prevalent in industry over the last decade due to growing demands for immediate decision-making upon massive scales of continuously arriving data. Indeed, more than 80% of the Fortune 100 companies currently use some stream processing platform, e.g., Apache Flink [1], Kafka Streams [2]. Modern stream processing applications can comprise several components, including processing engines, message brokers, and data stores, and often target a distributed cluster of servers to provide parallelism and replication in a scale-out model [3].

Despite the great success of the stream processing paradigm, testing a distributed stream processing application (particularly at scale) is still an intricate and often expensive process, time and money-wise. On one hand, developers and operators must cope with all challenges associated with deploying a networked system (e.g., routing, addressing, monitoring). On the other hand, they also need to rely on either costly testbeds (usually a cluster of servers organized into a pre-defined, i.e., fixed, topology) or complex cloud-based setups for running their experiments. Ultimately, these challenges can delay innovation, hide important software issues, and prevent minority groups from contributing to the community [4].

Existing approaches for prototyping stream processing applications fall short in several aspects. For instance, testbed environments (e.g., [5], [6]) can provide high-fidelity results, but they tend to only support small-scale experiments and require developers to instantiate complex distributed stream processing platforms almost from scratch. Simulation tools (e.g., [7], [8]) can easily scale to large systems, but they cannot fully represent real applications due to their reliance on computational models. Platform emulation can be a sweet spot for both testbeds and simulators. However, none of the current exemplars (e.g., [9]) focus on stream processing applications and thus require developers to deal with many low-level details, including network configurations. Finally, a few stream processing testing tools, e.g., [10], [11], are available to developers. Nevertheless, they are mostly focused on unit and integration testing and do not support the system level or end-to-end analyses that complex data pipelines require.

To address these issues, we propose *stream2gym*, a flexible and scalable prototyping environment targeted at stream processing applications. *stream2gym* uses network emulation to run real application code and provides a high-level API that developers can adopt to easily specify complex data processing pipelines. The tool supports a great set of monitoring tasks, including bandwidth and latency reports as well as event logs, and can be used to test applications under various operational conditions (e.g., network loads, failure models). We implemented *stream2gym* on top of a widespread network emulator, and made it open-source under an Apache License [12]. The tool currently supports a rich set of platforms commonly adopted in stream processing applications (e.g., Apache Kafka, Apache Spark, MySQL), and can run reasonably large setups (beyond 20 nodes) on a single commodity server.

In summary, our contributions are as follows:

- We design *stream2gym*, a modular, low-cost, and scalable prototyping environment for distributed stream processing applications. *stream2gym* provides a high-level API for developers to describe and test their data processing pipelines without knowing any detail about the low-level networked infrastructure. Ultimately, this decoupling simplifies application development and facilitates the re-usability of testing scenarios.
- We implement *stream2gym* on top of Mininet [13], a widely used network emulator, and make it open source.
- We deploy several applications using *stream2gym* and test their behavior under a variety of operational conditions, including different link delays as well as network failures, to demonstrate the relevance of our tool.
- We use *stream2gym* to reproduce experiments from published stream processing papers, including a video

analytics framework and a traffic monitoring system.

- We extensively evaluate *stream2gym* and show that it can match testbed results almost *exactly* while scaling to 10s of application components (e.g., message brokers, data consumers) with mere 8% and 25% increase in CPU and memory utilization, respectively.

II. Background and Motivation

A. Stream processing applications

Unlike batch processing, stream processing applications focus on real-time processing of a continuous stream of data [3]. They are typically developed in the context of a data processing pipeline, which may include data producers, message brokers, processing engines, storage, logging, and visualization stages. While the exact pipeline structure and computational tasks (e.g., data queries) may vary across applications, common deployment scenarios involve an ensemble of systems, combined together in a "system of systems". For instance, Uber's data analytics infrastructure [14] combines third-party tools such as Apache Kafka and Apache Flink for event streaming and stream processing, respectively, with their own customized workflow management solution. Meta [15] and Google [16] also have similar assets. These systems normally run on a distributed cluster of servers which relies on high-speed networking for coordination.

B. Testing approaches

Prototyping and testing stream processing applications have become a significant challenge for developers. In particular, the scale, complexity, and real-time nature of these applications pose stringent requirements on teams to identify and fix issues before they reach out to production. Unfortunately, there has been neither consensus about the best way to test a stream processing application nor consolidated tools for this task. However, the industry know-how in the area is vast and the literature documenting previous efforts is growing slowly (see Section VIII for more details).

Similarly to traditional software testing, developers must look at the stream processing testing problem at different granularities, including unit, integration, and system level testing [17]. While unit tests are widely adopted in practice and many stream processing systems may ship with their own unit testing modules (e.g., Kafka test-utils [18], Flink Spector [19]), integration and system testing are far less explored and often force developers to rely on custom ad-hoc solutions. A common technique for integration testing is mocking. Mocking frameworks (e.g., Mockito [20]) can imitate interactions among different components in a data processing pipeline. However, they are neither suitable for end-to-end, i.e., system level, testing nor offer control about the underlying infrastructure (e.g., networking delays, failure occurrences). We built *stream2gym* as a flexible and high-level prototyping environment to fill these gaps.

III. *stream2gym* Design

In this section, we describe the design goals, the overall architecture, and the programming interface of *stream2gym*.

A. Design goals

We aim to build a flexible prototyping environment for stream processing applications with the following design goals:

Functional realism. It should faithfully mimic the functionality, scale and performance of production stream processing systems while executing exactly the same code as in a real deployment.

Topology flexibility. It should be easy to validate a stream processing application in a variety of network topologies and consider diverse operational conditions (e.g., route delays, link bandwidths) and data processing pipelines.

Developer friendliness. It should be easy for software/data engineers to test stream processing pipelines. This means no involvement with the low level details of distributed frameworks (e.g., network addressing/routing, platform interoperability concerns).

Low cost. It should be inexpensive to run extensive experiments, including testing (e.g., unit, integration, fault tolerance) and reproducibility campaigns. In particular, we look for an accessible alternative to the burdens of distributed testbeds (e.g., equipment costs, waiting times) and multi-node cloud setups (e.g., shared resources, vendor lock-in).

Achieving our design goals incurs a few challenges. First, there are multiple running tasks (e.g., network switches, stream processors, data loggers) that must be accommodated on the same host. Second, some of these tasks must meet stringent performance requirements to work properly. For example, a broker replica must reply to periodic messages on time to not be considered outdated by a leader node. Third, load can be significantly unbalanced, meaning a few tasks (e.g., the network control plane) may concentrate heavy load chunks. Lastly, some application components may not be compatible with each other off-the-shelf, meaning transparently connecting them may require a proxy or wrapper code.

We address these challenges by i) carefully tuning supported systems to reduce their default resource usage and thus increase the provided emulation scale. This includes, e.g., adopting sampling techniques and reducing default buffer sizes for data loggers and producers, respectively; ii) providing an interface for users to also tune application components according to their needs, including relaxing performance requirements, e.g., timeouts, that may be too strict; iii) supporting weighted resource allocation policies for emulated hosts, meaning hosts running heavier load chunks can access more resources; and iv) deploying wrappers to interface with incompatible components.

B. Architecture and workflow

Figure 1 shows *stream2gym*'s architecture. The tool takes as input a description of the emulation task containing: i) a set of stream processors (e.g., Spark 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

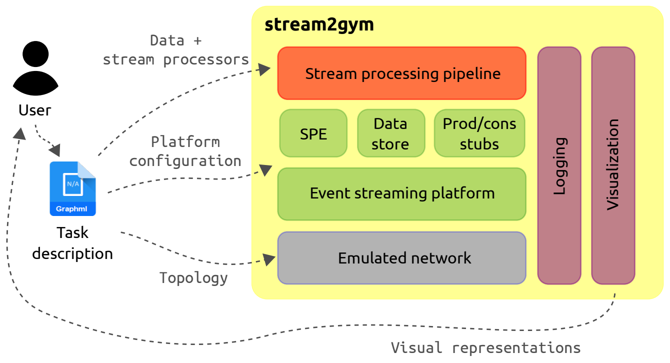


Fig. 1: *stream2gym* architecture and workflow. SPE = Stream Processing Engine.

TABLE I: *stream2gym* 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
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

platform, including its data stores, data producers, and message brokers in case these are present; and iii) a desired network topology to host the whole stream processing system. This design separates the application logic from its testing setup which enables re-using testing scenarios by modularly plugging different stream processors.

stream2gym instantiates the specified topology using a network emulator. Even though our tool focuses on single computers, it can be run with minimal modifications on distributed clusters (e.g., using [21] or [22]) if extreme-scale is needed. Once the network is set, *stream2gym* starts an event streaming platform to be used as a communication media among different application components. This is a common practice on current data processing pipelines (see Section II-A). *stream2gym* 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. Our tool 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 (e.g., producing each line of a file or each file in a directory as a data element). 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 facilitate debugging, *stream2gym* triggers a series of monitoring tasks that are responsible for logging relevant information from both the network and the application perspective (e.g., bandwidth measurements, timestamped events). Moreover, a visualization module presents a rich set of statistics to the user, which includes per-port throughput, message latency, and event ordering. Finally, *stream2gym* provides several parameters that can be tuned to model various operational conditions from production environments, including several routing algorithms and failure profiles.

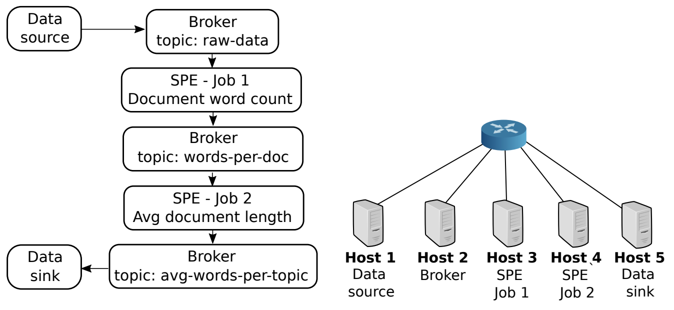
C. API

stream2gym provides a simple interface for modeling stream processing pipelines in terms of data flows, task allocations and network setups. The interface builds on GraphML [23], a widely adopted XML-based language for specifying generic graph structures and their node/link attributes. Table I lists the attributes *stream2gym* supports, which can either point to a configuration file or contain one or more user-specified values. Software/data engineers have the flexibility to specify any network topology or data flow graph they want. In this case, *stream2gym* uses its integrated event streaming platform to move data among network nodes based on a publish/subscribe model.

Graph attributes. Users can define a list of topics on which they want application components to produce/consume data. In this sense, *stream2gym* assumes the use of a broker-based messaging (or event streaming) system for transporting messages between components. For each topic, the user can also set a primary broker and a desired number of replicas. *stream2gym* provides a convenient API for emulating various failure scenarios, such as link failures, transient failures, and system crashes, which are helpful to test the reliability aspects of stream processing pipelines.

Node attributes. Network nodes can host several types of application components, including data stores, producers, consumers, message brokers, and stream processing engines. Each of these components has an associated configuration file, which contains component-specific parameters described as a list of key-value pairs. The choice of modular component configuration files makes it simpler to reuse component setups among different prototyping scenarios. *stream2gym* also offers the option for users to restrict the number of CPU cycles from the underlying server a host can get. In this way, they can easily allocate resources to the prototyping platform according to the expected load.

Link attributes. *stream2gym* allows the configuration of common communication channel parameters on emulated links, including delay, bandwidth, and packet loss. In particular, the latter is useful for constructing more complicated failure scenarios (e.g., gray failures [24]) as well as emulating network congestion. Users can also determine the source and


Data source
Broker topic: raw-data
SPE - Job 1 Document word count
Broker topic: words-per-doc
SPE - Job 2 Avg document length
Broker topic: avg-words-per-topic
Data sink
Host 1 Data source
Host 2 Broker
Host 3 SPE Job 1
Host 4 SPE Job 2
Host 5 Data sink

(a) (b)

Fig. 2: a) Example data processing pipeline; b) Target pipeline allocation into the emulated infrastructure.

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

(a)

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

(b)

Fig. 3: Example YAML configurations for the a) data source; and b) word count components of the data processing pipeline described in Figure 2a.

destination ports at which they want each link to be connected to the respective hosts.

D. Example

Figure 2a shows an example data processing pipeline that can be prototyped using *stream2gym*. This pipeline illustrates a document analytics application [24] and comprises a data source, which can read information from a file system or database, two stream processing jobs, and a data sink. The 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 2b illustrates the target pipeline allocation into the emulated infrastructure. Each component occupies a separate server reflecting a common scenario in which service providers adopt dedicated, i.e., specialized, clusters [25]. The example also considers a “one-big-switch” abstraction [26] to model the desired network setup, which simplifies its specification while still encompassing most of the communication channel details (e.g., delay, bandwidth).

Figure 4 illustrates how to describe our example data processing pipeline using *stream2gym*’s modeling language (i.e., GraphML). We start by setting up the configuration of *stream2gym*’s event streaming platform (line 3). Next, we specify the configuration of each pipeline component (lines 6-24). Note that each host (i.e., node) follows the target

```xml
1  <!-- Data processing pipeline -->
2  <graph edgedefault="undirected">
3   <data key="topicCfg"> topics.cfg </data>
4
5   <!-- Cluster allocation -->
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="brokerCfg"> broker.yaml </data>
12  </node>
13  <node id="h3">
14    <data key="streamProcType"> SPARK </data>
15    <data key="streamProcCfg"> spe-1.yaml </data>
16  </node>
17  <node id="h4">
18    <data key="streamProcType"> SPARK </data>
19    <data key="streamProcCfg"> spe-2.yaml </data>
20  </node>
21  <node id="h5">
22    <data key="consType"> STANDARD </data>
23    <data key="consCfg"> data-sink.yaml </data>
24  </node>
25
26  <!-- Network setup -->
27  <node id="s1"/>
28  <edge source="s1" target="h1">
29    <data key="st"> 1 </data>
30    <data key="dt"> 1 </data>
31    <data key="lat"> 50 </data>
32  </edge>
 ...
53 </graph>
```

Fig. 4: GraphML description for the data processing pipeline presented in Figure 2a. We omit some lines due to space constraints.

resource allocation previously discussed. Finally, we specify the networking setup for the communication channels between hosts in the cluster (lines 27-32). We use separate YAML files [27] to specify the configuration of each application component. Figures 3a and 3b show two examples, which depict the configuration of the data source (or producer) and the word counting job of our example pipeline, respectively.

IV. Implementation

We implement *stream2gym* on top of Mininet 2.3.0 and Apache Kafka 2.8.0. Our system comprises approximately 2.8K lines of Shell and Python code. We use Networkx 2.5.1 and Matplotlib 3.3.4 to parse topology specifications (expressed as GraphML files) and present data visualizations to the user, respectively. *stream2gym* currently supports Apache Spark 3.2.1 and MySQL 8.0.30 as example SPE and data store components. The emulated network is proactively configured using a lightweight switch control daemon (based on ovs-ofctl) to bind the control plane overhead. We use OpenFlow 1.3 statistics to monitor network performance indicators (e.g., bandwidth consumption) and the Python logging facility[1] to

[1]https://docs.python.org/3/library/logging.html

store relevant application events such as processing checkpoints. For experimenting with our tool, we use one machine equipped with an Intel® Core i7-3770 CPU @ 3.40GHz, 16GB RAM, and 2TB of storage. The server runs Ubuntu 20.04.4 LTS with kernel version 5.15.0-52-generic.

V. Use cases

A. Testing stream processing applications

Testing stream processing applications is a significant challenge for software engineers. In particular, there are a few unique aspects that make this task considerably harder compared to traditional application testing. First, as described in Section II-B, stream processing pipelines frequently involve multiple components (e.g., messaging platforms, machine learning training and inference systems, key-value stores), which complicates end-to-end debugging. Second, recent stream processing frameworks (e.g., [28]) often do not store or even enqueue incoming data. While this design choice helps to save memory for more pressing tasks (e.g., hosting a deep learning model), it forces stream processors to compute queries over samples and/or under tight latency constraints. As a result, developers have to fully characterize performance boundaries before the pipeline reaches a production environment which involves experimenting with several configurations in a controlled setup. Finally, streaming events can be out-of-order due to both task and data parallelism (e.g., multi-threading, distributed data ingestion) [29]. Ultimately, this makes query outputs non-deterministic and requires extensive monitoring and logging to identify anomalous behaviors.

stream2gym can assist developers to quickly prototype and test their applications from an end-to-end perspective. Focusing on integration and *system level testing* (i.e., testing multiple system components and/or services interacting altogether), the tool enables deploying complex data processing pipelines at almost no cost. Moreover, its extensive monitoring and logging capabilities provide a detailed analysis of the system behavior, which can be used to speed up several debugging tasks.

To assess the effort required for prototyping a stream processing application using *stream2gym*, we implemented a large set of diverse applications on the tool. Table II summarizes them. Number of components indicates how many modules (e.g., stream processors, message brokers, key-value stores) the application contains, while the features column depicts specific features each application deploys. Due to space constraints, we provide brief descriptions of each application below. More information, including the exact data processing pipeline, executed queries, and platform configurations can be found in the public *stream2gym* repository [12].

`Word count` is a standard benchmarking application for stream processing systems. It collects textual data from a stream of files, splits it into words, and stores word frequencies into another file. We implement text split and frequency counting as separate stream processing jobs. `Ride selection` leverages structured data (e.g., geographical coordinates, fare values) from a stream of taxi ride information to compute the best tipping areas in a city. The processed query includes a combination of join, groupby, and window operators, which requires dealing with an intermediate state. `Sentiment analysis` computes the subjectivity and polarity, two common natural language processing tasks [30], of each message in a Tweet stream and thus involves manipulating unstructured data. `Maritime monitoring` analyzes a stream of ship tracking reports (e.g., AIS messages [31]) to count the number of ships heading to a set of desired ports in a given time window. Its 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. `Fraud detection` runs a machine learning algorithm (SVM) to predict anomalies in a stream of financial transactions.

TABLE II: Example applications deployed on *stream2gym*.

Application	Components	Features	LoC
Word count	5	Multiple stream processing jobs	167
Ride selection	5	Structured Data, Stateful Processing	142
Sentiment analysis	3	Unstructured Data	72
Maritime monitoring	4	Persistent storage	162
Fraud detection	5	Machine learning prediction	185

Does *stream2gym* have tangible benefits? Testing is one of the "main elephants in the room" when one talks about stream processing applications, and we designed *stream2gym* specifically to lighten this burden. In particular, there is no need for developers to understand low-level networking concepts, no upfront cost with infrastructure setup, easy platform reconfiguration, and extensive monitoring capabilities, among other advantages while using our tool. In our subjective assessment (we make no claim of statistical significance), using *stream2gym* indeed improved our productivity when deploying stream processing tasks. First, it took us 10-100x longer to deploy our example applications (see Table II) on a hardware testbed, with software installations and networking setup consuming most of our time (approximately two days). Second, platform re-configurations, particularly those involving new networking conditions such as changing link delays or scaling up/down the cluster size, turned out to be cumbersome and error-prone processes that included several low-level parameter changes. Third, we had to manually instrument code to identify the causes of latent issues on hardware. For example, the event streaming system we used in our experiments was silently discarding messages upon a network partition (see Section V-B for more details), which we later discovered had been observed in the literature before. Finally, we had to run all our tests sequentially in the testbed as each test required the whole set of servers reserved for our experiments.

B. Emulating Networking Conditions

Varying link delay. Cloud organizations have increasingly deployed geographically distributed services to reduce 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 users

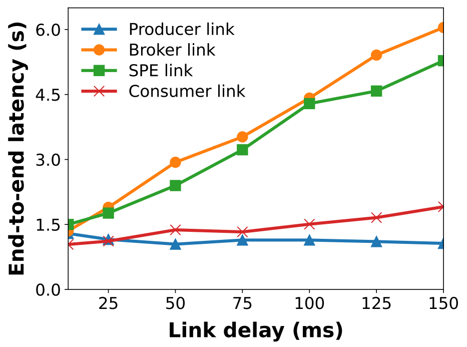


Fig. 5: End-to-end latency for the word count application when varying the link delay to reach out to each of its components. At each run, we increase the link delay of a single component and keep the remaining ones at a very low value (<10ms).

(or IoT) devices before sending them to a data center for analytics [32]. When transmitting aggregated data is still prohibitive, providers tend to execute queries geo-distributedly at the data generating sites [33]. The large variability of WAN bandwidth and latency though (up to 10x in production environments [34]) 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 network delays. Unfortunately, running a stream processing system in a real geo-distributed setup is challenging. First, it may require provisioning resources on several (edge) data centers and carefully crafting (or observing) desired running conditions, e.g., high-link delays, varying bandwidth. Also, it may not be possible to isolate the application's response to relevant events, e.g., a processing stall, from that of co-located services such as a co-hosted virtual machine.

stream2gym offers a simplified alternative to mock geo-distributed environments. In particular, its scenarios can be re-used and easily customized across different applications. To illustrate these benefits, we evaluate our word count application from Section V-A in a "one big switch" topology (i.e., similar to the one depicted in Figure 2a) while systematically varying the link delay for reaching out each application component. More specifically, in each experiment, we increase the link delay for communicating with a chosen host while keeping the delay for the remaining ones at a very low value (<10ms).

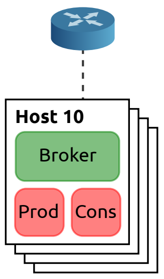
Figure 5 shows the end-to-end latency for processing a data unit (i.e., a text file) throughout the word count pipeline. Each point depicts the average latency of over 100 files. As expected, higher link delays impact the performance of all application components. Interestingly, the impact was more prominent when the data broker and the stream processing engine (Apache Spark in this case) 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 communicated with *all* other components in our experiment and therefore was more susceptible to poor networking conditions.

Network partitioning. Failure analysis is another scenario in which developers can benefit from using *stream2gym*, particularly network-partitioning failures. Recent studies indicate that network partitions happen as often as once a week in production networks and may take hours to repair [35]. 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 [36]. Reproducing, diagnosing, and hardening stream processing systems against such failures can be rather complicated due to the lack of proper tools. *stream2gym* can help to fill this gap by allowing developers to quickly and flexibly inject network-partitioning failures (e.g., after bringing network links down) into distributed stream processing systems.

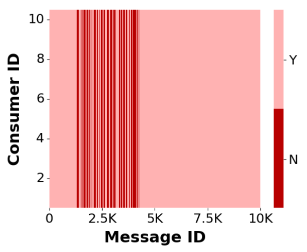
To illustrate *stream2gym*'s failure analysis capabilities, we set up a mid-scale experiment involving replicated brokers and concurrent data production/consumption in Apache Kafka. More specifically, we use *stream2gym* to deploy the scenario depicted in Figure 6a, where 10 message brokers are interconnected in a star topology and replicate messages produced into 2 topics. Each end host also runs: i) a data producer that randomly injects data into the two topics at a 30 Kbps rate; and ii) a consumer that collects data from both topics. According to our industry partners from the military sector, this is a common setup in their networks and reflects a scenario in which all tactical teams must remain operational (e.g., feeding historical data to fresh members) even in case of a disconnection. To test the system behavior under network-partitioned conditions, 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). We were able to deploy this scenario in *stream2gym* in less than 250 lines of GraphML and YAML code.

Figure 6b shows the data delivery matrix for the producer that is co-located with the disconnected broker. Each cell indicates whether a message was received by a given consumer light color) or not (dark color). We can observe intermittent losses for messages produced during the disconnection period (dark vertical bars). Moreover, all lost messages come from the topic whose leader got disconnected. This is in line with previous results found in the literature [36] and is due to the ZooKeeper (the distributed coordination service used by Apache Kafka) data consolidation mechanism, which may discard data (or pull it from an outdated log) during the partition merging process after a re-connection. We were *not* able to observe a similar behavior in the more recent Raft-based Kafka [37].

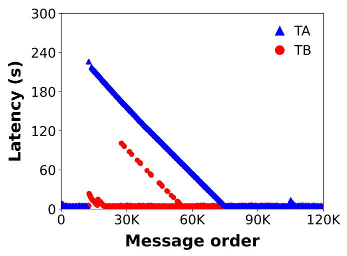
In addition to message loss, we also measure the impact of network partitioning failures on message latency, i.e., the time for a published message to be available at a subscriber, using *stream2gym*. Figure 6c shows the message latency at a random consumer (all consumers present a similar behavior). We classify messages according to their topic and order them



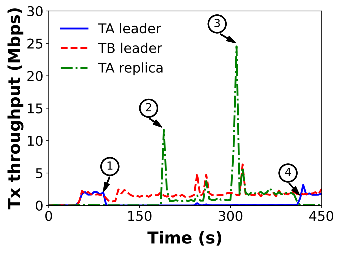
(a)



(b)



(c)



(d)

Fig. 6: *a)* Evaluation setup for network partitioning analysis; *b)* Message delivery matrix for the co-located producer with a disconnected broker. Y = Message delivered. N = Message *not* delivered; *c)* Message latency at a consumer. TA/TB = Topic A/B; *d)* 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.

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 6d 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 [38].

C. Reproducing Research Work

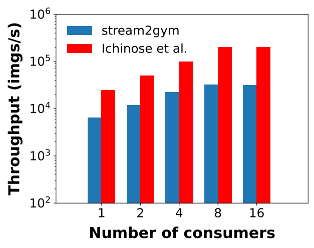
This section details experiments we perform to reproduce published stream processing research using *stream2gym*. Our main goal is to show *stream2gym* can qualitatively match the results generated on hardware by the original authors. Moreover, we envision our tool to be a helpful asset for other researchers to compare different proposals on similar ground.

Video analysis framework. In our first experiment, we use *stream2gym* to reproduce results measured by Ichinose et al. [39] to determine the performance of their proposed stream processing framework for analyzing video data. The framework comprises an event streaming cluster that transfers videos collected from multiple cameras (data producers) to a group of stream processing nodes (data consumers) that will analyze the video frames according to user-specified queries. As part of their evaluation, the authors investigate the performance of the event streaming cluster when producing frames to different numbers of consumers, all running on the same server.

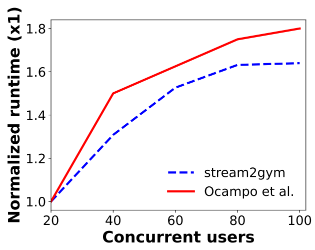
We replicate the experiment from Ichinose et al. using a single end host that runs a data pipeline containing one broker, one producer, and a varying number of consumers. Similarly to the original paper, we use a single topic to ingest data and produce a large number of MNIST images [40] before the first consumer subscribes to the topic to avoid data stalls. Figure 7a shows the transfer throughput (i.e., the rate at which data consumers can collect frames from the streaming cluster) for both *stream2gym* and the original paper as we vary the number of consumers. We can see that *stream2gym* results match those from Ichinose et al., showing an increase in the transfer throughput up to 8 consumers (same number of cores in the underlying host). Beyond that, increasing the number of consumers does not cause a significant impact on the observed throughput.

Traffic monitoring for enterprise networks. In our second experiment we reproduce the results obtained by Ocampo et al. [41] while evaluating the scalability of their stream processing-based traffic monitoring system. The proposed system takes a stream of network packets captured at different switches as input and computes a set of relevant metrics (e.g., number of active connections, bandwidth usage) on a windowed basis. The authors use an event streaming platform to collect mirrored packets from switches and a stream processing engine to compute the desired metrics. As part of their evaluation, they assess how the proposed system scales as the number of network users (i.e., traffic generators) increases. Each user generates traffic to a pre-defined set of services (e.g., FTP, Web, DNS) following a Poisson process.

We instantiate a scenario comprising a broker, a one-node Spark cluster, and a varying number of producers mimicking network users in *stream2gym*. As in Ocampo et al., traffic is processed in slots of one second. Figure 7b shows Spark mean

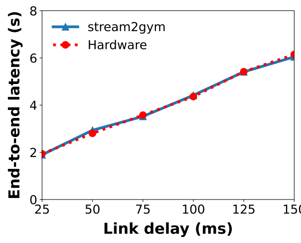


(a)

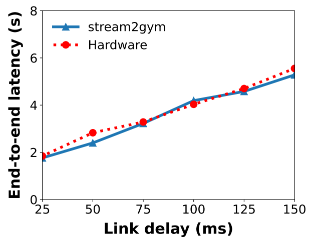


(b)

Fig. 7: Reproduced results for (a) Ichinose et al. [39] and (b) Ocampo et al. [41] using *stream2gym*. Check Section VII for details about the results magnitude variation.



(a)



(b)

Fig. 8: Comparison between *stream2gym* and testbed (i.e., hardware) results. Both environments run the word count application described in Section V-A with varying a) broker and b) SPE link delays.

execution time as we increase the number of network users, normalized by the results obtained for 20 users. *stream2gym* shows a similar increasing rate compared to Ocampo et al., with a bit more variation for large numbers of users (up to 20% for 100 users).

VI. Evaluation

In addition to the use cases described in the previous section, we also conducted an in-depth evaluation of *stream2gym* performance. More specifically, we explore: i) how accurate the tool is compared to testbed results (Section VI-B); and ii) how much resources it requires for running reasonably large experiments (Section VI-C). We start by describing our evaluation setup in the next section.

A. Setup

We run our experiments in two different environments. For *stream2gym* (i.e., emulation results), we use the same environment described in Section IV while testbed results were obtained using a 4-node cluster. The cluster has 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 [42] and run Ubuntu 20.04.1 LTS while the Core i7 servers have a 40 Gbps Netronome Agilio LX SmartNIC [43] 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 [44].

B. Accuracy

We now show *stream2gym* is accurate and can obtain realistic results compared to a hardware testbed. Here we use the word count application from Section V-A as a reference workload and inject a stream of data (i.e., text files) into its processing pipeline as quickly as possible. For the testbed results, we run the stream processing engine (Apache Spark 3.2.1) and message broker (Apache Kafka 2.8.0) on the two Xeon servers while the data producer and consumer execute on the Core i7 ones. We adopt a public NTP server [45] to synchronize clocks and perform latency measurements in the cluster. Moreover, all measurements are collected after a 60 seconds warm-up interval to initialize every application component and configure the network routes.

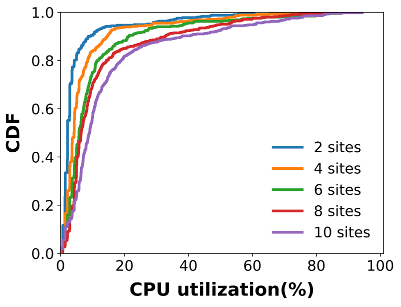
Figure 8 shows the end-to-end latency of the word count application as we vary the link delay of its message broker (Figure 8a) and stream processing engine (Figure 8b) components. We use `tc` to configure the link properties in the hardware setup. As we can observe, the results match almost *exactly* which demonstrates *stream2gym* correctness.

C. Resource usage

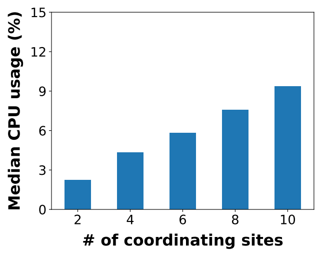
Next, we evaluate *stream2gym* scalability to large emulations. For that, we set up the same scenario from Figure 6a with a varying number of hosts (i.e., coordinating sites). Each site produces data at a 30 Kbps rate. We then measure the CPU and memory utilization of the underlying server by snapshotting `/proc/stat/` and `/proc/meminfo/`, respectively, every 500 milliseconds. All measurements are collected after a 60 seconds warm-up interval.

Figure 9a 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%) for the vast majority of 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 *stream2gym* needs to initialize all application components. We also investigate how quick the CPU utilization grows as we increase the number of sites. In Figure 9b, we plot the median CPU utilization for up to 10 sites. As we can observe, *stream2gym* scales to 10s of application components (each coordinating site has three components) with a minimal 8% increase in CPU usage. Moreover, the overall CPU demand is low (around 10%) even for the largest scenario.

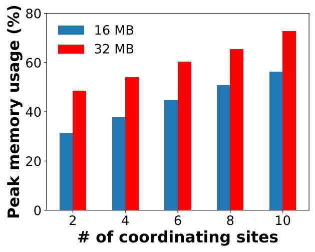
Finally, we analyze *stream2gym* memory consumption in large-scale emulations. More specifically, Figure 9c shows the peak memory usage of our tool for different numbers of coordinating sites. We also consider two buffer sizes at data producers (16 and 32 MB) in order to assess the impact of application component configurations on the overall platform resource consumption. This buffer size reflects the amount of memory a producer reserves for queuing messages that are



(a)



(b)



(c)

Fig. 9: *a)* CDF and *b)* median CPU utilization as we vary the number of coordinating sites, i.e., hosts, for the scenario described in Figure 6a; *c)* Peak memory usage for the same scenario considering different buffer sizes on data producers.

waiting to be sent to a broker (e.g., if the producer is sending messages faster than the broker can handle) [46]. 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 *stream2gym*'s memory consumption (as much as 18% in our test), which indicates the tool can be further optimized to accommodate bigger setups depending on how flexible it is to configure an application component. Likewise, we envision our tool can be used as a playground for automatically tuning stream processing system parameters [47]. We leave exploring both directions as future work.

VII. Discussion

Cloud-native setups. Container-based implementations of stream processing pipelines have gained popularity over the last years due to their improved management and scalability, particularly if implemented in the cloud. *stream2gym* is orthogonal to this approach and abstracts away the underlying environment from the application developer as much as possible. Moreover, unlike current container orchestration tools such as Kubernetes [48] which require users to configure the hosting infrastructure (e.g., by setting up the physical servers in the cluster), *stream2gym* can automatically provision all the resources necessary for running a stream processing application.

Infrastructure-as-code. Infrastructure-as-code (IaC) tools, e.g., Terraform [49], Ansible [50], Puppet [51], enable platform operators to manage their infrastructures via programmable configuration files. Ultimately, this facilitates the provision of scalable environments, their versioning, and patching. *stream2gym* draws inspiration from this idea to allow developers to describe and let others reproduce their setups using a standard configuration file. However, unlike IaC tools, the main purpose of *stream2gym* is application prototyping and testing, for which it provides a set of targeted features such as the emulation of specific networking conditions (e.g., packet drops, high link delays). Also, we leave integrating support for importing/exporting IaC scripts in *stream2gym* as future work.

Limitations. *stream2gym* can emulate diverse stream processing scenarios fairly well and therefore facilitate research comparison and dissemination as well as application testing. However, a few limitations must be taken into account. Due to its reliance on a network emulator, *stream2gym* is restricted by the capabilities of the underlying host (or cluster of servers). In particular, the server CPU must accommodate all running components (i.e., message brokers, data producers, stream processing nodes) which may impact accuracy in large-scale setups. Also, although *stream2gym* enables collecting meaningful *performance* results without setting up a hardware testbed, absolute values may vary (e.g., as in Figure 7) because of software limitations. For instance, current hardware switches can be more than one order of magnitude faster than software ones [52].

Other stream processing tools. Even though the current *stream2gym* implementation supports a limited set of stream processing tools, the high-level concepts presented in this work (e.g., specifying end-to-end tests for stream processing pipelines through a central configuration script) are generic and applicable to many other platforms such as Apache Flink or Kafka Streams. Likewise operators in orchestration solutions [53], we envision modularly supporting additional stream processing tools on *stream2gym* by adding support for plug-ins, and leaving the definition of specific interfaces as future work.

VIII. Related Work

Network testbeds. PlanetLab [5], Emulab [6], CloudLab [54] and Chameleon [55] are network testbeds providing large numbers of machines and network links that can be programmatically configured by users. Despite their high computational capabilities, they lack flexibility to customize topologies, forwarding behaviors and metrics though. Moreover, they require programmers to instantiate data processing platforms (e.g., stream processing engines, messaging systems) from scratch.

System simulation. NS-3 [7] and OMNeT++ [8] are popular simulators that enable users to model communication networks, multiprocessors, and other distributed or parallel systems. SimBricks [56] extends this concept and combines multiple simulators using a customized synchronization protocol to model different system components (e.g., gem5 for host simulation, FEMU for flash memories, NS-3 for networking).

Although these tools are fully flexible and scale well to large systems, they cannot provide accurate functionality due to their reliance on computational models rather than real software.

Emulation platforms. CrystalNet [9] is a cloud-based network emulator targeting the emulation of large-scale networks (thousands of devices). The tool focuses on the emulation of the network control plane and does not support configuring data plane parameters (e.g., link delay and bandwidth). TurboNet [57] uses programmable switches to emulate both the network data and control planes. However, it is limited by the switch capabilities and thus cannot run end host applications without requiring users to set up additional servers. Digibox [58] is a prototyping environment for IoT applications. The framework enables mocking several IoT devices (including their communications). Closest to our work, Mininet [59] uses containers to emulate software-defined networks, including real applications running on end hosts. In common, none of these tools focus on stream processing applications, leaving it entirely to developers to deal with their deployment and testing complexities.

Stream processing testing. There is limited work in testing for stream processing applications. In addition to the efforts described in Section II-B, Kallas et al. [29] propose a library (DiffStream) for differential testing in stream processing systems. TRAK [10], on its turn, is a tool for testing the reliability of event streaming platforms, particularly Apache Kafka. Gadget [11] is a framework for benchmarking embedded data stores on stateful stream processing engines. Karimov et al. [60] and Chintapalli et al. [61] also propose benchmarking tools for stream processing systems. Unlike *stream2gym*, none of them provide end-to-end testing for complex stream processing pipelines containing several application components (e.g., stream processing engines, message brokers, data stores).

IX. Conclusion

This work proposes *stream2gym*, a tool to facilitate the fast prototyping of stream processing applications in a distributed environment. *stream2gym* uses a network emulation platform and a high level API to carry out the network and stream processing setup on behalf of the application developer. We present a detailed design, workflow, and implementation of the proposed tool, and investigate its benefits in a number of use cases including application testing and the reproducibility of research work. Our evaluation shows that *stream2gym* can provide accurate results compared to a hardware testbed while using less than 10% of the underlying server CPU even when emulating 10s of application components (i.e., message brokers, data producers, consumers). This work opens up a new direction for the automated *end-to-end* testing and configuration of stream processing pipelines. Moreover, we made our contribution open source [12] to facilitate its adoption by the stream processing community.

Acknowledgements. We would like to thank the anonymous reviewers for their valuable feedback. This research is supported in part by NSERC under an Alliance Grant and CFI under a JELF grant.

References

[1] Apache flink: Stateful computations over data streams. Accessed: 2023-01-16. [Online]. Available: https://flink.apache.org/

[2] Apache kafka. Accessed: 2023-01-16. [Online]. Available: https://kafka.apache.org/documentation/streams/

[3] P. Carbone, M. Fragkoulis, V. Kalavri, and A. Katsifodimos, "Beyond analytics: The evolution of stream processing systems," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 2651–2658.

[4] J. Mirkovic and P. Pusey, "User experiences on network testbeds," in *Cyber Security Experimentation and Test Workshop*, ser. CSET '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 72–82.

[5] L. Peterson, A. Bavier, M. E. Fiuczynski, and S. Muir, "Experiences building planetlab," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, ser. OSDI '06. USA: USENIX Association, 2006, p. 351–366.

[6] 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 *OSDI 02*. Boston, MA: USENIX Association, Dec. 2002, pp. 255–270.

[7] ns-3 network simulator. Accessed: 2023-01-06. [Online]. Available: https://www.nsnam.org.

[8] Omnet++ discrete event simulator. Accessed: 2023-01-06. [Online]. Available: https://omnetpp.org/.

[9] 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 *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 599–613.

[10] H. Wu, Z. Shang, and K. Wolter, "Trak: A testing tool for studying the reliability of data delivery in apache kafka," in *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2019, pp. 394–397.

[11] 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 *Proceedings of the Seventeenth European Conference on Computer Systems*, ser. EuroSys '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 559–574.

[12] stream2gym code repository. [Online]. Available: https://github.com/PINetDalhousie/stream2gym

[13] Mininet: An instant virtual network on your laptop (or other pc). Accessed: 2023-01-16. [Online]. Available: http://mininet.org/

[14] Y. Fu and C. Soman, "Real-time data infrastructure at uber," in *Proceedings of the 2021 International Conference on Management of Data*, ser. SIGMOD '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 2503–2516.

[15] G. J. Chen, J. L. Wiener, S. Iyer, A. Jaiswal, R. Lei, N. Simha, W. Wang, K. Wilfong, T. Williamson, and S. Yilmaz, "Realtime data processing at facebook," in *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 1087–1098.

[16] A. Gupta and J. Shute, "High-availability at massive scale: Building google's data infrastructure for ads," in *Workshop on Business Intelligence for the Real Time Enterprise (BIRTE)*, 2015.

[17] A. Vianna, W. Ferreira, and K. Gama, "An exploratory study of how specialists deal with testing in data stream processing applications," in *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2019, pp. 1–6.

[18] Testing kafka streams. Accessed: 2023-01-06. [Online]. Available: https://kafka.apache.org/20/documentation/streams/developer-guide/testing.html.

[19] Framework for apache flink unit tests. Accessed: 2023-01-06. [Online]. Available: https://github.com/ottogroup/flink-spector.

[20] S. Mostafa and X. Wang, "An empirical study on the usage of mocking frameworks in software testing," in *2014 14th International Conference on Quality Software*, 2014, pp. 127–132.

[21] G. Di Lena, A. Tomassilli, D. Saucez, F. Giroire, T. Turletti, and C. Lac, "Distrinet: A mininet implementation for the cloud," *SIGCOMM Comput. Commun. Rev.*, vol. 51, no. 1, p. 2–9, mar 2021.

[22] Mininet cluster edition. Accessed: 2023-01-06. [Online]. Available: https://github.com/mininet/mininet/wiki/Cluster-Edition-Prototype.

[23] The graphml file format. Accessed: 2023-01-06. [Online]. Available: http://graphml.graphdrawing.org/.

[24] E. C. Molero, S. Vissicchio, and L. Vanbever, "Fast in-network gray failure detection for isps," in *Proceedings of the ACM SIGCOMM 2022 Conference*, ser. SIGCOMM '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 677–692.

[25] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, J. Law, K. Lee, J. Lu, P. Noordhuis, M. Smelyanskiy, L. Xiong, and X. Wang, "Applied machine learning at facebook: A datacenter infrastructure perspective," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 620–629.

[26] N. Kang, Z. Liu, J. Rexford, and D. Walker, "Optimizing the "one big switch" abstraction in software-defined networks," in *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 13–24.

[27] The official yaml web site. Accessed: 2023-01-19. [Online]. Available: https://yaml.org/.

[28] Y. Li, Y. Shen, and L. Chen, "Camel: Managing data for efficient stream learning," in *Proceedings of the 2022 International Conference on Management of Data*, ser. SIGMOD '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1271–1285.

[29] K. Kallas, F. Niksic, C. Stanford, and R. Alur, "Diffstream: Differential output testing for stream processing programs," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, nov 2020.

[30] B. Liu, *Sentiment Analysis and Subjectivity*, 2nd ed. Chapman and Hall/CRC., 2010.

[31] A. Harati-Mokhtari, A. Wall, and J. Wang, "Automatic identification system (ais): Data reliability and human error implications," *The Journal of Navigation*, vol. 60, no. 3, pp. 373–389, 2007.

[32] D. Kumar, S. Ahmad, A. Chandra, and R. K. Sitaraman, "Aggnet: Cost-aware aggregation networks for geo-distributed streaming analytics," in *2021 IEEE/ACM Symposium on Edge Computing (SEC)*, 2021, pp. 297–311.

[33] Q. Pu, G. Ananthanarayanan, P. Bodik, S. Kandula, A. Akella, P. Bahl, and I. Stoica, "Low latency geo-distributed data analytics," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, ser. SIGCOMM '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 421–434.

[34] B. Zhang, X. Jin, S. Ratnasamy, J. Wawrzynek, and E. A. Lee, "Awstream: Adaptive wide-area streaming analytics," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 236–252.

[35] R. Govindan, I. Minei, M. Kallahalla, B. Koley, and A. Vahdat, "Evolve or die: High-availability design principles drawn from googles network infrastructure," in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 58–72.

[36] A. Alquraan, H. Takruri, M. Alfatafta, and S. Al-Kiswany, "An analysis of network-partitioning failures in cloud systems," in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'18, 2018, p. 51–68.

[37] Apache kafka. Accessed: 2023-01-19. [Online]. Available: https://kafka.apache.org/31/documentation.html.

[38] N. Narkhede, G. Shapira, and T. Palino, *Kafka: The Definitive Guide Real-Time Data and Stream Processing at Scale*, 1st ed. O'Reilly Media, Inc., 2017.

[39] A. Ichinose, A. Takefusa, H. Nakada, and M. Oguchi, "A study of a video analysis framework using kafka and spark streaming," in *2017 IEEE International Conference on Big Data (Big Data)*, 2017, pp. 2396–2401.

[40] L. Deng, "The mnist database of handwritten digit images for machine learning research," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[41] A. F. Ocampo Palacio, T. Wauters, B. Volckaert, and F. De Turck, "Scalable distributed traffic monitoring for enterprise networks with spark streaming," in *ECCWS2018, the 17th European Conference on Cyber Warfare and Security*, 2018, pp. 563–569.

[42] Nvidia mellanox bluefield smartnic mode - winof-2 v2.50 - nvidia networking docs. Accessed: 2023-01-19. [Online]. Available: https://docs.nvidia.com/networking/display/winof2v250/NVIDIA+Mellanox+BlueField+SmartNIC+Mode.

[43] Agilio lx smartnics - netronome. Accessed: 2023-01-19. [Online]. Available: https://www.netronome.com/products/agilio-lx/.

[44] Intel® tofino™ series programmable ethernet switch asic. Accessed: 2023-01-19. [Online]. Available: https://www.intel.ca/content/www/ca/en/products/network-io/programmable-ethernet-switch/tofino-series.html.

[45] pool.ntp.org: Ntp servers in canada, ca.pool.ntp.org. Accessed: 2023-01-19. [Online]. Available: https://www.pool.ntp.org/zone/ca.

[46] Advanced kafka producer configurations. Accessed: 2023-01-06. [Online]. Available: https://www.conduktor.io/kafka/other-advanced-kafka-producer-configurations.

[47] M. Bilal and M. Canini, "Towards automatic parameter tuning of stream processing systems," in *Proceedings of the 2017 Symposium on Cloud Computing*, ser. SoCC '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 189–200.

[48] Kubernetes. Accessed: 2023-01-19. [Online]. Available: https://kubernetes.io/.

[49] Terraform by hashicorp. Accessed: 2023-01-19. [Online]. Available: https://www.terraform.io/.

[50] Ansible is simple it automation. Accessed: 2023-01-19. [Online]. Available: https://www.ansible.com/.

[51] Puppet infrastructure and it automation at scale — puppet by perforce. Accessed: 2023-01-19. [Online]. Available: https://www.puppet.com/.

[52] F. Dürr, T. Kohler *et al.*, "Comparing the forwarding latency of openflow hardware and software switches," *Fakultät Informatik, Elektrotechnik Informationstechnik, Univ. Stuttgart, Stuttgart, Germany, Tech. Rep. TR*, vol. 4, p. 2014, 2014.

[53] Welcome to operatorhub.io. Accessed: 2023-01-19. [Online]. Available: https://operatorhub.io/.

[54] 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 *2019 USENIX annual technical conference (USENIX ATC 19)*, 2019, pp. 1–14.

[55] 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 *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC'20. USA: USENIX Association, 2020.

[56] H. Li, J. Li, and A. Kaufmann, "Simbricks: End-to-end network system evaluation with modular simulation," in *Proceedings of the ACM SIGCOMM 2022 Conference*, ser. SIGCOMM '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 380–396.

[57] J. Cao, Y. Liu, Y. Zhou, L. He, and M. Xu, "Turbonet: Faithfully emulating networks with programmable switches," *IEEE/ACM Transactions on Networking*, vol. 30, no. 3, pp. 1395–1409, 2022.

[58] S. Fu, H. Zhang, S. Ratnasamy, and I. Stoica, "The internet of things in a laptop: Rapid prototyping for iot applications with digibox," in *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*, ser. HotNets '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 24–30.

[59] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown, "Reproducible network experiments using container-based emulation," in *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 253–264.

[60] 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.

[61] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng, and P. Poulosky, "Benchmarking streaming computation engines: Storm, flink and spark streaming," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2016, pp. 1789–1792.