Pilot-Streaming: A Stream Processing Framework for High-Performance Computing

Andre Luckow[1,2,3], George Chantzialexiou[1], Shantenu Jha[1,4]

[1]*RADICAL, ECE, Rutgers University, Piscataway,NJ 08854, USA*
[2]*Clemson University, Clemson, SC 29634, USA*
[3]*Ludwig Maximilian University, Munich, Germany*
[4]*Brookhaven National Laboratory, Upton, NY, USA*

***Abstract*—An increasing number of scientific applications utilize stream processing to analyze data feeds of scientific instruments, sensors, and simulations. In this paper, we study the streaming and data processing requirements of light source experiments, which are projected to generate data at 20 GB/sec in the near future. As beamtimes available to users are typically short, it is essential that processing and analysis can be conducted in a streaming mode. The development and deployment of streaming applications is a complex task and requires the integration of heterogeneous, distributed infrastructure, frameworks, middleware and application components written in different languages and abstractions. Streaming applications may be extremely dynamic due to factors, such as variable data rates, network congestions, and application-specific characteristics, such as adaptive sampling techniques and the different processing techniques. Consequently, streaming system are often subject to back-pressures and instabilities requiring additional infrastructure to mitigate these issues. We propose *Pilot-Streaming*, a framework for supporting streaming applications and their resource management needs on HPC infrastructure. Underlying Pilot-Streaming is a unifying architecture that decouples important concerns and functions, such as message brokering, transport and communication, and processing. Pilot-Streaming simplifies the deployment of stream processing frameworks, such as Kafka and Spark Streaming, while providing a high-level abstraction for managing streaming infrastructure, e. g. adding/removing resources as required by the application at runtime. This capability is critical for balancing complex streaming pipelines. To address the complexity in the development of streaming applications, we present the Streaming Mini-Apps, which supports different plug-able algorithms for data generation and processing, e. g., for reconstructing light source images using different techniques. We use the streaming Mini-Apps to evaluate the Pilot-Streaming framework demonstrating its suitability for different use cases and workloads.**

I. Introduction

Stream processing capabilities are increasingly important to analyze and derive real-time insights on incoming data from experiments, simulations, and Internet-of-Things (IoT) sensors [1]. Prominent examples are synchrotron light source experiments, such as those at the National Synchrotron Light Sources II (NSLS-II) or the X-Ray Free Electron Laser (XFEL) light sources. Some experiments at these light sources are projected to generate data at rates of 20 GB/sec [2]. This data needs to be processed in a time-sensitive if not real-time manner, to support steering of the experiments [3].

Further, an increasing number of scientific workflows integrate simulations either with data from experimental and observational instruments, or conduct real-time analytics of simulation data [4]. Workflows are stymied by the fact that capabilities to continuously process time-sensitive data on HPC infrastructures are underdeveloped while they require sophisticated approaches for resource management, data movement and analysis. The complex application and resource utilization patterns of streaming applications critically demand dynamic resource management capabilities. For example, minor changes in data rates, network bandwidths, and processing algorithms can lead to imbalanced and dysfunctional system.

We propose *Pilot-Streaming*, a framework designed to efficiently deploy and manage streaming frameworks for message brokering and processing, such as Kafka [5], Spark [6] and Dask [7], on HPC systems. Underlying Pilot-Streaming is a unifying architecture that decouples important concerns and functions, such as message brokering, transport and communication, and processing. Pilot-Streaming is based on the Pilot-Job concept and the Pilot-Abstraction [8]. Pilot-Streaming enables application and middleware developers to deploy, configure and manage frameworks and resources for complex streaming applications. Acquired resources can be dynamically adjusted at runtime – a critical capability for highly dynamic streaming applications. Further, Pilot-Streaming serves as unifying API layer for managing computational tasks in an interoperable, framework-agnostic way, i. e. it allows the implementation of streaming tasks that can run both in Spark Streaming, Dask or other frameworks.

To further address the development and deployment challenges of streaming apps, we develop the *Streaming Mini-Apps* framework based on a systematic analysis of different scientific streaming application [9]. The Mini-Apps provides the ability to quickly develop streaming applications and to gain an understanding of the performance of the pipeline, existing bottlenecks, and resource needs. We demonstrate the capabilities of Pilot-Streaming and the Streaming Mini-Apps by conducting a comprehensive set of experiments evaluating the processing throughput of different image reconstruction algorithms used in light source sciences.

This paper makes the following contributions: (i) It surveys the current state of message broker and streaming frameworks and their ability to support scientific streaming applications; (ii) It provides a conceptual framework for analyzing scientific streaming applications and applies it to a machine learning and light source analytics use case. The Mini-App framework provides a simple solution for simulating characteristics of these applications. (iii) It presents an abstraction and architecture for stream processing on HPC. Pilot-Streaming is

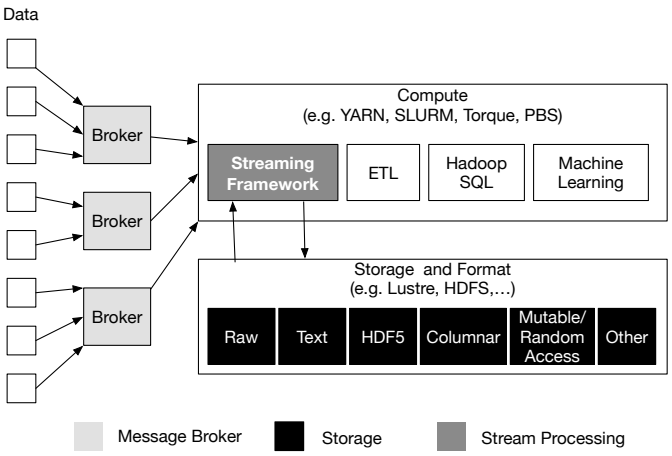

Fig. 1. **Streaming Applications Architecture:** The *message broker* decouples streaming applications from incoming data feeds and enables multiple applications to process the data. The *streaming framework* typically provides a windowing abstraction on which user-defined functions can be performed.

a reference implementation of that architecture, and (iv) It demonstrates and evaluates the described capabilities using a set of large-scale experiments on the XSEDE machine Wrangler, for streaming machine learning and different light source reconstruction algorithms.

This paper is structured as follows: In Section II we investigate the architectural components of a typical streaming infrastructures and applications and related work. We continue with an analysis of streaming applications in Section III. Section IV presents the architecture, capabilities and abstractions provided by Pilot-Streaming. The frameworks serves as basis for the Mini-Apps discussed in Section V. In Section VI we present an experimental evaluation of Pilot-Streaming.

II. Background and Related Work

We define a streaming application as an application that processes and acts on an unbounded stream of data close to real time. In this section we describe the current state of streaming middleware and infrastructure and related work. There is no consensus on software and hardware infrastructure for streaming applications, which increases the barrier for adoption of streaming technology in a broader set of application (see Fox et al. [1]). Notwithstanding the lack of consensus, in this paper we will explore the usage of the existing Pilot-Abstractions as a unified layer for the development of streaming applications.

A. Streaming Middleware and Infrastructure

The landscape of tools and frameworks for stream processing is heterogeneous (see [10] for survey). Figure 1 illustrates the main components of a stream system are: the message broker, the storage and the stream processing engine. We will investigate these in the following section.

Message Broker: The broker decouples data producers and consumers providing a reliable data storage and transport. By combining data transport and storage, the message broker can provide a durable, replay-able data source to streaming processing applications. For this purpose, the brokering system typically provides a publish-subscribe interface. The best throughputs are achieved by log-based brokering systems, such as Kafka [11]. Facebook Logdevice [12] provides a similar log abstraction, but with a richer API (record not byte based) and improved availability guarantees. Apache Pulsar is another distributed brokering system [13]. Other types of publish-subscribe messaging system exist, such as ActiveMQ and RabbitMQ, but are generally less scalable than distributed log-based services, such as Kafka [5]. A message broker enables application to observe a consistent event stream of data at its own pace executing complex analytics on that data stream. Kafka is one such distributed message broker optimized for large volume log files containing event streams of data. Amazon Kinesis [14] and Google Cloud Pub-Sub [15] are two message brokers offered as "platform as a service" in the cloud.

	Storm/Heron	Spark Streaming	Flink	Dask Streamz
Description	Java/C++ with Python API	Scala with Java, Python APIs	Java	Python
Architecture	Continuous	Mini-batch, Continuous	Continuous	Mini-batch
Windowing	Yes	Event time introduced with structured API	Yes with event/processing time	Fixed Time
Higher-Level APIs	Streamlet API (MapReduce)	Structured Streaming (DataFrames, SQL)	Data Tables	DataFrames (state-less)
Guarantees	Exactly once	Exactly once	Exactly once	No
Integration	Kafka	Kafka, Kinesis	Kafka	Kafka

TABLE I
Stream Processing Frameworks

Streaming Processing Frameworks: A heterogeneous landscape of infrastructures and tools supporting streaming needs on different levels emerged. Table I summarizes the properties of four important stream processing systems. Batch frameworks, such as Spark [6] and Dask [7], have been extended to provide streaming capabilities [16], [17], while different native streaming frameworks, such as Storm [18], Heron [19] and Flink [20] have emerged. Apache Beam [21] is high-level streaming abstraction that can be used together with Flink and Spark and is available as managed cloud service called Google Dataflow [22]. Apache Beam's abstraction is based on a rigorous model and provides well-defined and rich semantics for windowing, transformations and other operations. The different stream processing engines differs significantly in the ways they handle events and provide processing guarantees: Storm and Flink continuously process data as it arrives. Dask Streamz and Spark Streaming rely on micro-batches, i. e., incoming data is partitioned into batches according to a user-defined criteria (e. g. time window). The advantage of micro-batching is that it provides better fault tolerance, higher throughput and exactly-once processing guarantees, while native stream engines can provide lower latencies and more advanced windowing capabilities, e. g., tumbling and session-based windows.

Each of the described message brokers and stream processing frameworks provides unique capabilities, e. g., specific windows semantics, high-level APIs (such as streaming SQL), low latency. However, they do not address interoperability, deploy-

ment on HPC and resource management. While all frameworks provide an application-level scheduler, resource management is typically a second-order concern and not addressed in a generalized, holistic, framework-agnostic approach.

B. Related Work

There are several areas of related work: (i) frameworks that allow the interoperable use of streaming frameworks on HPC, (ii) the usage of HPC hardware features and frameworks (such as MPI) to optimize data streaming frameworks, and (iii) the exploration of data streaming in distributed applications.

Interoperable Streaming on HPC: Various tools have been proposed to support open source Big data frameworks, such as Hadoop and Spark on HPC environments on top of schedulers like SLURM, PBS/Torque etc [23], [24]. Other more streaming-oriented frameworks, such as Flink, Heron and Kafka are not supported on HPC out-of-the-box and require the manual implementation of job submission scripts.

While these script-based approaches is acceptable for small applications, it has severe limitations with respect to maintainability and support for more complex stream processing landscapes. For example, it is typically necessary to coordinate resources among several tools and frameworks, such as simulation and data acquisition, data message broker, and the actual stream processing framework. Also, streaming application are much more dynamic exhibiting varying data production and process rates, than traditional simulation and data analytics applications. Thus, in this paper we propose the usage of the Pilot-Abstraction as unifying layer for managing a diverse set of resources and stream processing frameworks.

Optimizing Streaming on HPC: The ability to leverage HPC hardware and software capabilities to optimize Big Data frameworks has been extensively explored. Kamburugamuve et al. [25] propose the usage of optimized HPC algorithms for low-latency communication (e. g. trees) and scheduling of tasks to enhance distributed stream processing in the Apache Storm framework [18]. In [26] they investigate the usage of HPC network technology, such as Infiniband and Omnipath, to optimize the interprocess communication system of Heron [19], the successor of Storm. Chaimov et al. [27] propose the usage of a file pooling layer and NVRAM to optimize Spark on top of Lustre filesystems. These approaches can complimentary to the high-level resource management approach proposed in this paper and can be used to optimize critical parts of a stream processing pipeline. These approaches mainly focus on low-level optimization of Big Data frameworks for HPC. Pilot-Streaming address critical gaps in the integration of these frameworks with the application and the ability to manage resources across these frameworks in a high-level and uniform way.

Streaming in Scientific Application: Fox et al. [10] identifies a broad set of scientific applications requiring streaming capabilities. Many aspects of these use cases have been explored: For example, Bicer et al. [28] investigates different light source reconstruction techniques on HPC. Du [29] evaluates streaming infrastructure for connected vehicle applications. Both approaches focus solely on a specific aspect of a single use cases, e. g., latencies or processing throughput. Proving a generalized architecture and solution for many use cases addressing important shared concerns, such as resource management, is not in scope of these approaches. Pilot-Streaming and the Streaming Mini-Apps provide a holistic approach for addressing a broad set of use cases end-to-end from data source, broker to processing on heterogeneous infrastructure.

The implementation of scientific streaming applications requires the integration of infrastructure, a diverse set of frameworks: from resource management, message brokering, data processing to advanced analytics. In most cases, the data source is external making it essential for streaming application to dynamically manage resources and frameworks.

III. Streaming Applications

Stream processing is becoming an increasingly important for scientific applications. While many streaming applications primarily perform simple analytics (smooth averages, max detection) on the incoming data, the computational demands are growing. For example, to run complex reconstruction algorithms for light source data streams or deep learning based computer-vision algorithms, such as convolutional neural networks, a vast amounts of scalable compute resources are required. In this section, we develop a taxonomy for classifying streaming applications. Further, we will discuss light source streaming as specific applications example.

A. Applications Characteristics

In the following we investigate different types of streaming applications in particular with respect to types data production (simulation, experiment) and processing:

Type 1 – Experimental Data Streaming: Experimental data generated by an instrument that is processed by a data analysis application and/or a simulation. An example are light source experiments (see section III-B).

Type 2 – Simulation Data Streaming: Simulation produces data that is processed by a data analysis application. This form of processing is referred to as in-situ processing. Different forms of in-situ analysis exist: the analysis tasks can e. g. run within the same HPC job or on a separate set of nodes coupled via shared storage and/or network. An example of co-analysis of molecular dynamics simulations data [4].

Type 3 – Streaming with Feedback/Control Loop: Data is processed with realtime feedback, i. e. output is used to steer simulation respectively experiment. Both type 1 and 2 applications typically benefit from the ability to integrate realtime insights into an experiment or simulation run.

Streaming applications involve the coupling a data source (simulation, experimental instrument), message broker and processing. In general, these components can be deployed across heterogeneous, distributed infrastructure. Often, it makes sense to run some pre-processing close to the data-source (on the edge), transmit selected data to the cloud and do global processing in the cloud. Resource needs are highly

dynamic and can change at runtime. Thus, an in-depth understanding of application and infrastructure characteristics is required.

The coupling between data source and processing can be (i) direct (e. g., using a direct communication channel, such as memory) or (ii) indirect via a brokering system. The direct couple is used when low latencies and realtime guarantees are required. The direct coupling approach is associated with several drawbacks: it involves a large amounts of custom code for interprocess communication, synchronization, windowing, managing data flows and different data production/consumptions rates (back-pressure) etc. Thus, it is in most cases advantageous to de-couple production and consumption using a message broker, such as Kafka. Another concern is the geographic distribution of data generation and processing: both can be co-located or geographically distributed. Further, the number of producer and consumers can vary.

The third component is the actual stream data processing: in simply cases the application utilizes non-complex analytics on the incoming data, e. g. for averaging, scoring, classification or outlier detection. Typically, streaming applications utilize less complex analytics and operate on smaller amounts of data, a so-called streaming window. There are multiple types of windowing, e. g. a fixed, sliding or session window. Commonly the streaming windows is either defined based on processing time or event time. More complex application involve combine analytics with state and model updates, e. g. the update of a machine learning model using incoming and historical data. This processing type requires that the model state is retained. Further, access to additional data is often required.

The main difference between streaming applications with traditional, data- intensive batch applications is that streaming data sources are unbounded. While this impacts some aspects of an applications, such as the runtime and the potentially need to carefully reason about ordering and time constraints, other factors remain the same, e. g., the computational complexity of the processing algorithms. In the following, we utilize the following sub-set of properties to characterize streaming applications:

Data Source and Transfer: describe the location of the data source in relation to the stream processing application. The data source can be external (e. g., an experimental instrument) or internal to the application (e. g., the coupling of a simulation and analysis application on the same resource). Output data is typically written to disk or transferred via a networking interface. Message brokers can serve as intermediate decoupling production and consumption.

Latency is defined as the time between arrival of new data and its processing.

Throughput describes the capacity of the streaming system, i. e. the rate at which the incoming data is processed.

Lifetime: Streaming applications operate on unbounded data streams. The lifetime of a streaming application is often dependent on the data source. In most cases it is not infinite and limited to e.g., the simulation or experiment runtime.

Time/Order Constraints defines the importance of order while processing events.

Dynamism: is variance of data rates and processing complexity observed during the lifetime of a streaming application.

Processing: This characteristics describes the complexity of data processing that occurs on the incoming data. It depends e. g. on the amount of data being processed (window size, historic data) and the algorithmic complexity.

B. Streaming Application Examples

In the following we utilize the defined streaming application characteristics to analyze two example use cases: (i) a generic streaming analytics application (Type 1 or 2), and a more specific use case (ii) light sources analytics (Type 1). Table II summarizes different characteristics of these applications.

1) Streaming Analytics: Use cases, such as Internet-of-Things, Internet/Mobile clickstreams, urban sensor networks, co-analysis of simulation data, demand the timely processing of data feeds using different forms of analysis [1], [30]. For example, an increasing number of scientific applications require streaming capabilities: cosmology simulations require increasing amounts of data analytics to digest simulation data, environmental simulation require the integration of remote sensing capabilities, etc. Depending on the nature of the data source, this type of application can be classified as type 1 or 2 application. The number of type 3 application is still comparable low. This can be attributed to the lack of sufficient middleware to support such complex architectures.

While the general problem architecture of data analytics and machine learning are similar to those of batch application, there are some subtle differences: typically the amount of data processed at a time is small compared to batch workloads. While the problem architecture of many machine learning algorithms remains the same, different techniques for updating the model using the new batch of data are used (e.g., averaging using a decay factor).

2) Light Source Sciences: X-Ray Free Electron Laser (XFEL) are a class of scientific instruments that have become instrumental for understanding fundamental processes in domains such as physics, chemistry and biology [31], [32]. Such light sources can reveal the structural properties of proteins, molecular and other compounds down to the atomic levels. The light source emits hundreds to thousands of x-ray pulses per second. Each pulse produces an image of the diffraction pattern as results. These images can then be combined and reconstructed into a 3-D model of the compound serving as the basis for a later analysis. Light sources can be used to exactly observe what is happening during chemical reactions and natural processes, such protein folding.

Example for light sources are the Linac Coherent Light Source (LCLS) [33] at SLAC, the National Synchroton Light Source II (NSLS II) [2] at Brookhaven, and the European XFEL light sources [34]. LCLS-I averages a throughput of 0.1-1 GB/sec with peaks at 5 GB/s utilizing 5 PB of storage and up to 50 TFlops processing [3]. The European XFEL produces 10-15 GB/sec per detector [34]. In the future even higher data rates

are expected: LCLS-II is estimated to produce data at a rate of more than 20 GB/sec. In the following, we focus on NSLS-II. NSLS-II consists of 29 operational beamlines. Thirty more beamlines are in development. Each beamline has different data characteristics, therefore the need for developing management tools that acquires the data from the beamlines and analyzes them is evident. As the beamtimes available to the user are typically short, it essential that processing and analysis can be conducted in a timely manner. Thus, streaming data analysis is an important capability to optimize the process. This ensures that scientists can adjust the settings on the beamline and optimize their experiment.

The Complex Materials Scattering (CMS) beamline is an NSLS-II beamline, which generates 8 MB images at a rate of 10 images/minute. While this production data rate is not very high, a single CMS experiment generates more than 17,000 images a day, equivalent to ∼140 GB of data. It is required that this data be processed within 6 hours, to prepare for the experiments the following day. The Coherent Hard X-ray (CHX) beamline [35] is dedicated to studies of nanometer-scale dynamics using X-ray photon correlation spectroscopy can produce data at much higher rates of ∼4.5GB/s [36].

Light source applications are a Type 1 application. In most cases, the instrument is co-located with some compute resources. However, scientists often rely on additional compute resource and also may need to integrate data from several instruments. Thus, the ability to manage geographically distributed resources is important. Currently, data analysis is often decoupled from the experiments. With increased sophistication of the instruments, the demand for steering capabilities will grow evolving this type of application toward Type 3.

The processing pipeline for light source data comprises of three stages: pre-processing, reconstruction and analysis [37]. Pre-processing can includes e. g. normalization of the data, filtering and the correction of errors. Various reconstruction with different properties, e. g. computational requirements and quality of the output, exist: GridRec [38] is based on a Fast-Fourier transformation and is less computational intensive and thus, fast. Iterative methods can provide a better fidelity. An example of an iterative method is Maximum likelihood expectation maximization (ML-EM) reconstruction [39]. A broad set of analytics methods can be applied to the reconstructed image, e. g. image segmentation and deep learning methods. For the CMS experiment, simple statistical algorithms, such as the computation of a circular average and peak finding is used.

3) Discussion: The requirements of streaming applications vary: For use cases involving physical instruments with potential steering requirement, e. g., X-Ray Free Electron Laser, both latency and throughput are important. Other use cases e. g. the coupling of simulation and analysis have less demanding latency and throughput requirements. The lifetime of scientific streaming applications is often coupled to the lifetime of the data source. Time and message ordering is in contrast to transactional enterprise applications not important for many scientific applications. With respect to the data transfer and processing requirements, the need to support different frameworks in a plug-able and interoperable way is apparent. Another important difference is that streaming applications are typically runtime constrained, i. e. they must process the incoming data at a certain rate to keep the system balanced. Thus, a good understanding of application characteristics is even more critical for streaming applications. Minor changes in the data rates, the processing approach (e. g. change of the processing window, sampling approaches or the need to process additional historic data or available resources) can lead to imbalance and a dysfunctional system. Thus, the ability the dynamically allocate additional resources to balance the system is critical. We use the characteristics identified in this section to design the Streaming Mini-Apps that aids the evaluation of complex streaming systems (see section V).

	Streaming Analytics: K-Means	**Light Source**
Data Source	external or internal	external
Latency	medium/high latencies	medium latencies
Throughput	medium	high
Duration	data source runtime	experiment runtime
Time/Order	not important	not important
Dynamism	varying data rate	varying data rate
Processing	**Model score:** Assign data to centroids/class $O(num\_points \cdot num\_clusters)$. **Model update:** Update centroids with in-coming mini-batch of data. **Model size:** small (O(number clusters))	**Reconstruction:** Reconstruction techniques with different complexities (GridRec, ML-EM). **Analysis:** data analysis techniques, such as peak finding, image processing models utilizing GPUs.

TABLE II
STREAMING APPLICATION PROPERTIES

IV. PILOT-STREAMING: ABSTRACTIONS, CAPABILITIES AND IMPLEMENTATION

Pilot-Streaming addresses the identified challenges and gaps related to deploying and managing streaming frameworks and applications on HPC infrastructure. Pilot-Streaming makes two key contributions: (i) it defines a high-level abstractions that provide sufficient flexibility to the application while supporting the resource management and performance needs of streaming applications are essential, and (ii) the reference implementation supports different stream processing and brokering frameworks on HPC resources in a plug-able and extensible way.

Pilot-Streaming provides a well-defined abstraction, i. e. a simplified and well-defined model that emphasizes some of the system's details or properties while suppressing other [40], for managing HPC resources using Pilot-Jobs and deploy streaming frameworks on these. The Pilot-Stream abstraction is based on the Pilot-Job abstraction. A Pilot-Job is a system that generalizes the concept of a placeholder job to provide multi-level scheduling to allow application-level control over the system scheduler via a scheduling overlay [8]. Pilot-Jobs have been proven to provide efficient mechanisms for managing data and compute across different, possibly distributed resources. The Pilot-Abstraction is heavily used by many HPC application for efficiently implementing task-level parallelism, but also advanced execution modes, such as processing of DAG-based task graphs. Examples for using the Pilot-Abstraction are molecular dynamics simulations [41] and high energy application [42]. Further, we have explored the applicability of the Pilot-Abstraction [43] to data-intensive applications on HPC and Hadoop environments [44], [45].

The Pilot-Streaming reference implementation allows the management and deployment of different message brokers and stream processing frameworks, currently Spark, Dask and Kafka, as well as its ability to serve as unified access layer to run tasks across these in an interoperable way. Further, these frameworks can be deployed side-by-side on the same or different distributed resources a capabilities which is critical for many streaming pipelines. The framework is designed in extensible way and can easily be extended to support Flink, Heron and other stream processing frameworks. Another key capability is the ability to dynamically scale these frameworks by adding resources. This is essential to deal with varying data rates and compute requirements. Further, framework continuously monitors the applications and thus, provides an enhanced level of fault tolerance, which is essential as stream applications typically run longer than batch jobs. We continue with a discussion of the Pilot-Streaming abstraction in section IV-A and the reference implementation in section IV-B.

A. Pilot-Abstractions and Capabilities

In this section, we describe the provided abstraction from developer point of view. The abstraction is based on the Pilot-abstraction, which provides two key abstractions: a Pilot represents a placeholder job that encapsulates a defined set of user-requested resources. Compute-Units are self-contained pieces of work, also referred to as tasks, that are executed on these resources. Pilot-Streaming utilizes multi-level scheduling and can manage Compute-Units in a framework-agnostic way. For this purpose, Pilot-Streaming interfaces with the schedulers of the different frameworks, e. g. the Spark scheduler, which then manage the further execution of the Compute-Units. The key features of Pilot-Streaming are:

Unified and Programmatic Resource Management: The Pilot-Abstraction provides a unified resource management abstraction to manage streaming frameworks for processing and message brokering on HPC environments. It allows the orchestration of compute and data across different frameworks.

Streaming Data Sources: While our previous work focused on integration static datasets and compute units managed by Pilot-Jobs [44], Pilot-Streaming extends this ability to streaming data sources, such as Kafka topics.

Interoperable Streaming Data Processing: For the processing of streaming data applications can utilize the Pilot-API for defining Compute-Units. Compute-Units can either rely on native HPC libraries and applications or can integrate with stream processing frameworks, such as Spark-Streaming. This enables applications to utilize the different capabilities of these frameworks in a unified way.

Extensibility and Scalability: Pilot-Streaming is extensible and can easily be extended to additional message brokers and streaming frameworks. It is architected to scale to large (potentially distributed) machines both at deploy and runtime.

The framework exposes two interfaces: (i) a command-line interface and (ii) the Pilot-API for programmatic access. The API is based on a well-defined conceptual model for Pilot-Jobs [8]. The Pilot-API allows reasoning about resources and performance trade-off associated with streaming applications. It provides the means necessary to tune and optimize application execution by adding/removing resources at runtime. Listing 1 shows the initialization of a Pilot-managed Spark cluster. The user simply provides a pilot compute description object, which is a simple key/value based dictionary.

```
from pilot.streaming.manager import PilotComputeService
spark_pilot_description1 = {
    "service_url":
           "slurm+ssh://login1.wrangler.tacc.utexas.edu",
    "number_cores": 48,
    "type":"spark"
}
pilot1 = PilotComputeService.create_pilot(
     spark_pilot_description)
```

Listing 1. Pilot-Streaming: Creation of Spark Cluster

A key capability of Pilot-Streaming is the ability to dynamically add/remove resources to the streaming cluster by just referencing a parent cluster in the Pilot-Description. If the resources are not needed anymore, the pilot can be stopped and the cluster will automatically resize. This capability not only allows application to respond to varying resource needs, but also provides the ability to work around maximum job size limitations imposed by many resource providers.

Pilot-Streaming provides several hooks to integrate with the managed streaming frameworks. It supports custom configurations, which can be provided in their framework native form (e.g., spark-env format etc.) and can easily be managed on per machine basis. This ensures that machine-specific aspects, e.g., amount of memory, the usage SSD and parallel filesystems, network configurations, can optimally be considered.

Pilot-Streaming supports interoperability on several levels. The API provides a unified way to express stream computations agnostic to specific framework. Listing 2 illustrates how to execute a Python function can be executed as a Compute-Unit in a interoperable way. This is suitable for simple stream-processing tasks, such as tasks that can be expressed as map-only job. Using the unified API, functions can easily be run across frameworks, e. g. to utilize advanced, framework-specific capabilities, such as parallel processing, windowing or ordering guarantees. For more complex tasks, the API provides the ability to access the native API of each framework allowing the implementation of complex processing DAGs.

```
def compute(x): return x*x
compute_unit = pilot.submit(compute, 2)
compute_unit.wait()
```

Listing 2. Pilot-Streaming: Interoperable Compute Unit

Listing 3 illustrates how the Context-API provides the ability to interface with the native Python APIs from these frameworks. The context object exposes the native client application, i. e., the Spark Context, Dask Client or Kafka Client object. Having obtained the context object, the user can then utilize the native API, e.g., the Spark RDD, DataFrame and Structured Streaming API.

```
sc = spark_pilot1.get_context()
rdd = sc.parallelize([1,2,3])
rdd.map(lambda x: x*x).collect()
```

Listing 3. Pilot-Streaming: Native Spark API Integration

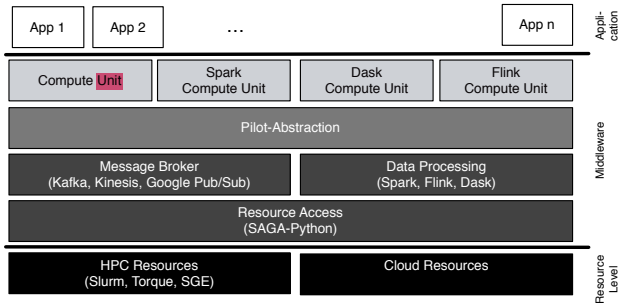

Fig. 2. **Pilot-Streaming Architecture:** Pilot-Streaming allows the management of message brokers and stream processing frameworks on HPC.

B. Reference Implementation: Architecture and Interactions

Figure 2 illustrates the high-level architecture of Pilot-Streaming. Pilot-Streaming provides a unified access to both HPC and cloud infrastructure. For resource access we utilize the SAGA Job API [46], a lightweight, standards-based abstraction to resource management systems, such as SLURM, SGE and PBS/Torque. The framework provides two key capabilities: the management of message broker on HPC and the management of distributed data processing Engines on HPC. These two capabilities are encapsulated in the message broker and data processing module. The interface to the framework is the Pilot-Abstraction [8], a proven API for supporting dynamic resource management on top of HPC machines. The application logic is expressed using so-called Compute-Unit, which can be executed in either (i) a task-parallel processing engine, such as Pilot-Jobs (e. g., RADICAL-Pilot [47], BigJob [43] or Dask), or (ii) a streaming framework, such as Spark Streaming. Case (i) typically requires the manual implementation of some capabilities, e. g. the continuous polling of data. In case (ii) the developer can rely on the streaming framework for implementing windowing. Both scenarios have trade-offs: while scenario (i) allows the interoperable execution of CUs across frameworks, scenario (ii) is often faster to implement. Pilot-Streaming supports both cases.

Figure 3 shows the interaction diagram for Pilot-Streaming. In the first step the application requests the setup of Spark, Dask or Kafka cluster using a Pilot-Description as specification. Then the Pilot-Manager initiates a new Pilot-Job, a placeholder job for the data processing or message broker cluster, via the local resource manager. The component running on resource is referred to as Pilot-Streaming-Agent (PS-Agent). After the job and framework has been initialized, the application can start to submit Compute-Units or initiative interactions with the native framework APIs via the context object.

Pilot-Streaming is an extensible framework allowing the simple addition of new streaming data sources and processing frameworks. By encapsulating important components of streaming applications into a well-defined component and API, different underlying frameworks can be used supporting a wide variety of application characteristics. It utilizes the SAGA-Python [48] implementation to provision and manage resources on HPC machines.

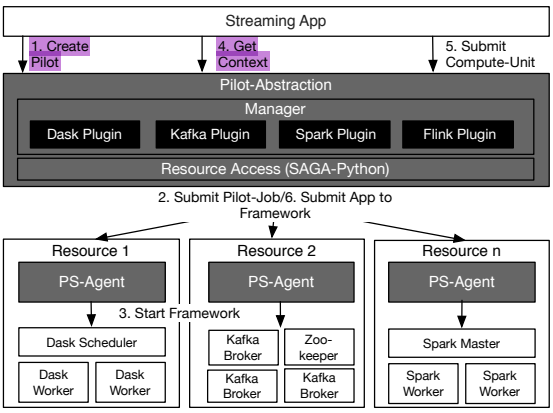

Fig. 3. **Pilot-Streaming Interaction Diagram:** The figure shows the control flow used by Pilot-Streaming to manage frameworks and applications.

Data Source	Adhoc deployment of broker and processing close to data
Latency	Framework selection, co-location of data/compute resource
Throughput	Framework selection, optimization of resource configuration to data rate
Fault Tolerance	Monitoring of Jobs through Pilot-Job Management and Agent
Time/Ordering	Orderting, Windowing mechanism of underlying framework
Dynamism	Add/Remove resources at runtime via Pilot-Job Mechanism

TABLE III
STREAMING CHALLENGES ADDRESSED BY PILOT-STREAMING

The streaming frameworks specifics are encapsulated in a plugin. A framework plugin comprises a simple service provider interface (SPI) and a bootstrap script executed on the resource. As depicted in Listing 4, the interface has six functions, e. g., to start/extend a cluster, to retrieve cluster information, such as state and connection details.

```
class ManagerPlugin():
    def __init__(self, pilot_compute_description)
    def submit_job(self)
    def wait(self)
    def extend(self)
    def get_context(self, configuration)
    def get_config_data(self)
```

Listing 4. Pilot-Streaming Plugin Interface

Discussion: Data and streaming applications are more heterogeneous and complex than compute-centric HPC applications. Pilot-Streaming allows the usage of different message brokers and data processing engines in an interoperable way on HPC infrastructures. Table III summarizes how Pilot-Streaming addresses the requirements of streaming applications.

Pilot-Streaming removes the need for application developers to deal with low-level infrastructure, such as resource management systems. Running Spark, Kafka and Dask clusters across a flexible number of Pilot-Jobs provides the ability to dynamically adjust resources during runtime. Further, the framework provides a common abstraction to execute compute tasks and integrate these with streaming data. It supports the interoperable execution of these CU across different frameworks. In addition, Pilot-Streaming provides the ability to also utilize the higher-level APIs provided by the frameworks. Currently, Pilot-Streaming supports Kafka, Spark, and Dask. It can be extended via a well-documented plugin-interface. Pilot-Streaming is open-source, maintained by an active developer community and available on Github [49].

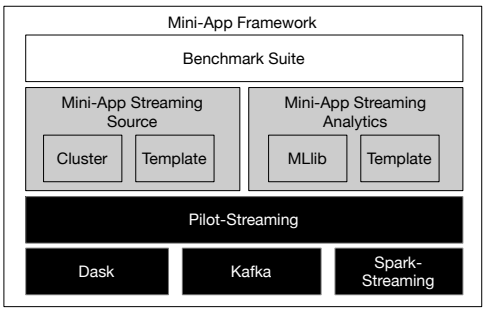

Fig. 4. Streaming Mini-Apps: The framework is based on Pilot-Streaming and provides two components: the *MASS (MiniApp for Stream Source)* emulates different streaming data sources and the *MASA (MiniApp for Streaming Analysis)* provides different synthetic processing workloads.

V. Streaming Mini-Apps

Developing streaming application pipelines is a complex task as it requires multiple parts: data source, broker and processing component. Every one of these components typically relies on different programming and middleware systems making it highly complex to develop such pipelines. During development process the real data source is often not available. Often developers have to rely on a static dataset, which results in significant efforts for setting setup a real test and development environment that is capable of mimicking non-bounded datasets as well as non-functional requirements, such as different data rates, message sizes, serialization formats and processing algorithms. If available, real applications are often not as parameterizable and tunable to characterize and optimize application, middleware and infrastructure configurations.

The Streaming Mini-Apps [50] addresses these challenges. Figure 4 shows the architecture of the framework. The framework is based on Pilot-Streaming, which provides the ability to rapidly allocate different size of cluster environments. The core of the framework consists of two main components: (i) the *MASS (Mini-App for Stream Source)* can emulate a streaming data source, which can be tuned to produce streams with different characteristics: data rates, messages size. (ii) the *MASA (Mini-App for Streaming Analysis)* provides a framework for evaluating different forms of stream data processing.

The MASS app includes a pluggable data production functions. The current framework provides two types of functions: A cluster source generates random data points following certain structures, e. g., for evaluation of streaming cluster analysis algorithms. The second type: template produces an unbounded stream based on a static template dataset. Data rates, message sizes etc. can be controlled via simple configuration options. Using these two base data source the majority of streaming applications can be emulated. For example, KMeans or other cluster algorithms for detecting outliers in data streams can be developed and tested with the cluster source. The template algorithms is great for migrating batch workloads to streaming. It can be used to emulate important application, such as light sources.

Similarly, the MASA app enables the user utilize machine learning algorithms from MLlib [51] or to provide custom data processing functions. Currently, it is based on Spark Streaming, but the framework can easily ported to other streaming frameworks as it is based on Pilot-Streaming. The processing function is data-parallel by nature. The machine learning algorithms provided by MLlib are capable of utilizing distributed resources supporting both data and model parallelism. In particular, we provide pre-configured support for KMeans clustering [52] and for reconstructing light source data. The K-Means algorithm has a complexity of $O(cn)$ where c is the number of cluster centroids and n is the number of data points. The light source reconstructing algorithm is based on Tomopy [37], a framework that is commonly used for pre-processing raw light source data, e. g., image reconstructions, and for further analysis. Different reconstruction algorithms are supported by the Mini-Apps, e. g., GridRec [38] and ML-EM [39].

In summary, the Streaming Mini-Apps provide optimal customizability with the ability to plug in custom data production and processing functions and control various configuration parameters, such as data rates, message sizes, etc. The framework provides comprehensive performance analysis options, e. g. it includes standard profiling probes that enables to measure common metrics, such as production and consumption rate allowing the benchmark of application and streaming middleware components making it easy to understand performance bottlenecks as well as the impact of changes. This is an essential capability to develop, test and tune streaming pipelines under complex, real world loads. In particular components like the message broker are difficult to analysis as the write/read load can vary significantly depending on the number consumers and producers. Further, the Mini-Apps allow for easy reproducibility of such experiments. The Streaming Mini-Apps provide a powerful tool to develop, optimize applications, and empirically evaluate streaming frameworks and infrastructure. In contrast to other approaches [53], the streaming mini app framework focuses on data-related characteristics, in particular the need to produce, transport and process data at different rates. In addition, the framework can emulate the application characteristics of K-Means application.

VI. Experiment and Evaluation

The aim of this section is to investigate different infrastructure configuration with respect to their ability to fulfill defined application requirements in terms of latency and throughput. For this purpose, we use the Mini-Apps to simulate different data production and processing characteristics. All experiments are conducted on Wrangler, an XSEDE machine designed for data-intensive processing. Each Wrangler nodes has 128 GB of memory and 24 cores.

A. Startup Overhead

There are two main steps for setting up Spark and Kafka on HPC: (i) Running the batch job that sets up the Kafka/Spark cluster and (ii) initiating an actual session with the broker respectively starting a Spark job by initializing a Spark session. Figure 5 compares the startup times for different size Kafka, Spark and Dask clusters. The startup time for Kafka increase significantly with the number of nodes indicating that some

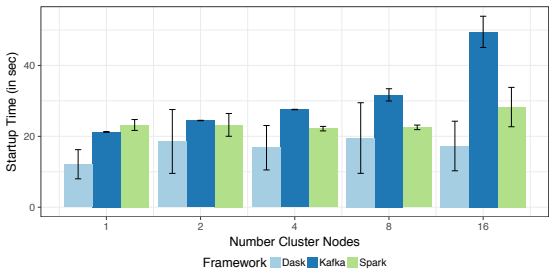

Fig. 5. **Kafka, Spark, and Dask Startup Time on Wrangler:** Kafka start involves the startup of both Zookeeper and the Kafka brokers and thus, is most of the times longer than Spark. Dask has the shortest startup times. For Kafka, the startup time increase with the number of nodes. The Spark and Dask startup times did not significantly change for larger clusters.

optimizations are necessary for larger clusters. Spark and Dask utilize parallelism to startup the cluster and thus, show no significant increase.

The measured startup times are short compared to the overall runtime of streaming application. In particular, considering the benefits of Pilot-Streaming: improved isolation of application components, the ability to independently scale parts of the streaming pipeline to the application needs, better diagnoseability, debug-ability and predictability of the application, this is an acceptable overhead.

B. Producer Throughput

In this section, we analyze the performance for publishing data into the Kafka system using the MASS app. The produces batches of random 3-D points, which are serialized to a string and pushed to Kafka using PyKafka [54]. We utilize different data source types: (i) KMeans: every message consists of 5,000 randomly generated double precision points. The average serialized size of message is 0.32 MB; (ii) Lightsource Micro-Tomography (Light-MT): every message consists of raw input dataset in the APS data format and an average encoded message size of 2 MB. (iii) Lightsource CMS (Light-CMS): every message consists of one image generated from the CMS Beamline. The size of each image is 8 MB (HDF5) and 18 MB (serialized). The scenarios were chosen to demonstrate the variety characteristics with respect to number messages and message sizes streaming application can exhibit. We investigate the throughput and its relationship to different MASS types and configurations as well as to different Kafka broker cluster sizes. For the experiment, we utilize different resource configuration parameters determined in a set of micro-experiments: the number partitions is fixed at 12 per node. On every producer node, we run 8 producer processes in Dask. While each node possesses 24 cores, the performance per node deteriorated drastically when using more producers/node due to network and I/O bottlenecks. We evaluate four scenarios: KMeans-Random, KMeans-Static, Light-MT and Light-CMS. The KMeans-Random scenario uses the cluster MASS plugin to generate points randomly distributed around a defined number of centroids. Kmeans-Static and both light scenarios use a static message at a configured rate.

Figure 6 shows the results. The KMeans-Random configuration is bottlenecked by the random number generator. Thus, the

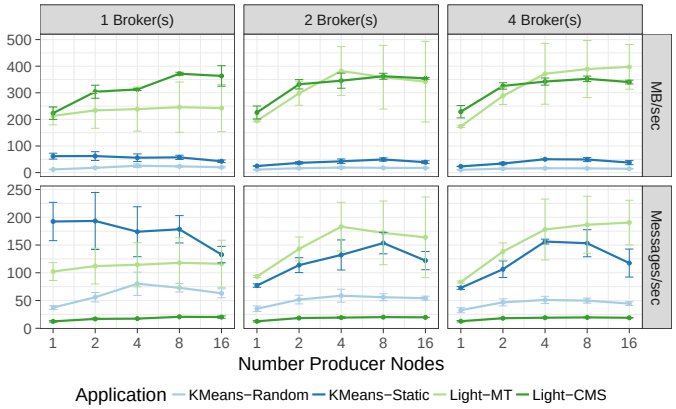

Fig. 6. **MASS Producer Throughput for Different Data Sources Types and Resource Configurations:** We utilize up to 16 producer nodes with 8 processes/node and 4 Kafka nodes. The achievable throughput depends on the message size: KMeans: 0.3 MB, Light-MT: 2 MB and for Light-CMS: 18 MB.

KMeans-Static setup has on average a 1.6x higher throughput than KMeans-Random. The light scenarios show a significant MB/sec throughput mainly due to larger message sizes: Light-CMS uses a much larger message size (18 MB) compared to Light-MT (2 MB), thus the throughput is in many cases higher for Light-CMS than for Light-MT. As expected, the message throughput is lower for Light-CMS due to the larger message sizes. Both the message throughput and high variance in the measured bandwidth indicate that the performance is network bound. Also, it must be noted that the network is a shared resource and external factors likely lead to the high variance in the measured bandwidths for Light-CMS and Light-MT. The usage of more brokers does not improve the performance in all scenario due to the overhead associated with accessing a multi-node Kafka cluster, e. g. concurrent connections and partitioning overhead. A multi-node Kafka cluster is particular advantageous when a larger number of medium-sized messages need to be handled, such as for Light-MT.

C. Processing Throughput

We use the MASA Mini-App to investigate the throughput of three different processing algorithms: a streaming KMeans that trains a model with 10 centroids and makes a prediction on the incoming data, and two light source reconstruction algorithms: GridRec and ML-EM. We use the distributed KMeans implementation of MLlib and the GridRec, ML-EM of TomoPy. In the experiment we utilize the MASS Mini-App with 1 node and 8 producer processes to continuously produce messages of 0.3 MB/5000 points for KMeans and 2 MB/1 point for the light source scenarios. This way are able to simulate a complex read/write workloads on the Kafka broker. We use 12 partitions/node for the Kafka topic. The Mini-App uses Spark Streaming with a mini-batch window of 60 sec.

Figure 7 shows the results of the experiment. The processing throughput depends on various aspects, such as the bandwidth to the message broker, computational complexity, and the scalability of the processing algorithm. The KMeans application shows the highest throughput. It scales both increasing number of processing nodes. For example, it is apparent that in the

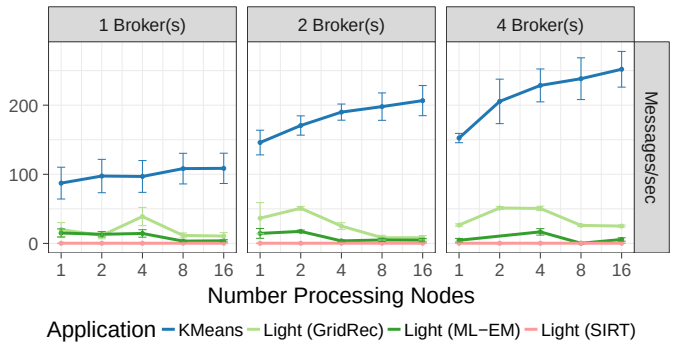

Fig. 7. **MASA Throughput for KMeans and two Light Source Reconstruction Algorithms:** KMeans scales well with increasing numbers of nodes. GridRec shows a higher throughput than ML-EM as it less computation complex. Scaling of both reconstruction algorithms is limited by I/O contention.

1 and 2 broker scenario, the I/O to the broker constraints the performance. With additional broker nodes, the available bandwidth and parallelism increasing. Spark Streaming assigns 1 task per Kafka partition. This is visible in a significant increase in throughput. With KMeans we were able to achieve a maximum throughput of 277 messages/sec and thus, were easily able to sustain the generated data rate.

The throughput of the light source reconstruction algorithms is significantly worse with maximum 63 message/sec for GridRec and 22 messages/sec for ML-EM. As describe iterate algorithms, such as ML-EM are more demanding than GridRec. Additional broker nodes yielded in significant performance improvements. Additional processing nodes improved the performance as long the bandwidth to the resource broker was able to keep up with the additional processing resources. The amount of data transferred is with 2 MB/message significant larger than in the KMeans scenario. Further, we observed some resource contentions caused by running multiple instances of the algorithm on the same node and the need to buffer a significant number of messages. The results show the importance of resource management - only if the bandwidth and read-parallelism to the data source or broker is large enough additional compute resources are beneficial.

Discussion: As demonstrated, the overhead for Pilot-Streaming is small: the startup time for dynamically starting Kafka, Dask and Spark clusters is outweighed by the benefits of improved flexibility, resource isolation (per application components), and the ability to scale components independently (at runtime if needed). We demonstrated the scalability of the framework by managing large streaming landscapes of Dask, Spark and Kafka concurrently on up to 32 nodes, 1536 virtual cores, and 4 TB of memory achieving throughputs of up to 390 MB/sec for the lightsource scenario. This throughput is large enough to sustain the LCLS-I data stream with a high enough sampling rate. At the current setup, the processing side is the bottleneck. We are only able to process a fraction of the data. Scaling stream processing is more difficult than scaling batch analytics workload as it requires a careful balance of bandwidth to/from the data source respectively the broker and compute resources. In particular, it can be difficult to diagnose bottlenecks in the broker, as the varying mixture of write/read I/O makes the performance often unpredictable. Pilot-Streaming provides the necessary abstractions to manage resources effectively at runtime on application-level.

The Streaming Mini-Apps simplify streaming application development and performance optimizations. Using the Streaming Mini-Apps, we were able to emulate various complex application characteristics. It is apparent that the different frameworks and application components each have unique scaling characteristics and resource needs. Even for optimization of just one component a large number of combinations of experiments is required. On streaming application-level this leads to a combinatorial explosion of configurations. The Streaming Mini-Apps and Pilot-Streaming provide essential tools for automating this process. In the future, we will use both frameworks as foundation for higher-level performance optimization approaches, e. g., modeling the performance of each component, the usage of experimental design and machine learning techniques for performance predictions.

VII. Conclusion and Future Work

Pilot-Streaming fills an important gap in supporting stream processing on HPC infrastructure by providing the ability to on-demand deploy and manage streaming frameworks and applications. This capability is crucial for an increasing number of scientific applications, e. g., light source sciences, to generate timely insights and allow steering. The landscape of tools and frameworks for message brokering, data storage, processing and analytics is diverse. Pilot-Streaming currently integrates with Kafka, Spark Streaming, Dask and Flink. Its flexible, plug-in architecture allows the simple addition of new frameworks. Streaming applications can have unpredictable and often, external induced resource needs, e. g. driven by the data production rate. Pilot-Streaming addresses these needs with a well-defined resource model and abstraction that allows the adjustments of the allocated resources for each component at runtime. Another important contribution are the Streaming Mini-Apps, which simplifies the development of streaming pipelines with the ability to emulate data production and processing. We demonstrated the variety of features of this framework with several experiments using a streaming KMeans and different light source analysis algorithms.

This work represents the starting point for different areas of research: We will extend Pilot-Streaming to support highly distributed scenarios enabling applications to push compute closer to the edge for improved data locality. The Streaming Mini-Apps will be the basis for the development and characterization of new streaming algorithms, e. g. additional reconstruction algorithms and deep learning based object classification algorithms. We will explore the usage of accelerators (such as GPUs) to support compute-intensive deep learning workloads. Another area of research are steering capabilities. Further, we will continue to utilize the Streaming Mini-Apps to improve our understanding of streaming systems and embed this into performance models that can inform resource and application schedulers about expected resource needs.

Acknowledgements: We thank Stuart Campbell and Julien Lhermitte (BNL) for guidance on the light source application. This work is funded by NSF 1443054 and 1440677. Computational resources were provided by NSF XRAC award TG-MCB090174.

REFERENCES

[1] Geoffrey Fox, Shantenu Jha, and Lavanya Ramakrishnan. Stream 2015 final report. http://streamingsystems.org/finalreport.pdf, 2015.

[2] Brookhaven National Laboratory. National synchrotron light source ii. https://www.bnl.gov/ps/, 2017.

[3] Amedeo Perazzo. Lcls data analysis strategy. https://portal.slac.stanford.edu/sites/lcls_public/Documents/LCLSDataAnalysisStrategy.pdf, 2016.

[4] Preeti Malakar, Venkatram Vishwanath, Christopher Knight, Todd Munson, and Michael E. Papka. Optimal execution of co-analysis for large-scale molecular dynamics simulations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 60:1–60:14, Piscataway, NJ, USA, 2016.

[5] J. Kreps, N. Narkhede, and J. Rao. Kafka: A distributed messaging system for log processing. In *Proceedings of 6th International Workshop on Networking Meets Databases (NetDB), Athens, Greece*, 2011.

[6] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[7] Dask Development Team. Dask: Library for dynamic task scheduling. http://dask.pydata.org, 2016.

[8] Andre Luckow, Mark Santcroos, Andre Merzky, Ole Weidner, Pradeep Mantha, and Shantenu Jha. P*: A model of pilot-abstractions. *IEEE 8th International Conference on e-Science*, pages 1–10, 2012. http://dx.doi.org/10.1109/eScience.2012.6404423.

[9] Geoffrey C. Fox, Devarshi Ghoshal, Shantenu Jha, Andre Luckow, and Lavanya Ramakrishnan. Streaming computational science: Applications, technology and resource management for hpc. http://dsc.soic.indiana.edu/publications/streaming-nysds-abstract.pdf, 2017.

[10] Supun Kamburugamuve and Geoffrey Fox. Survey of distributed stream processing. Technical report, Indiana University, Bloomington, IN, USA, 2016.

[11] Guozhang Wang, Joel Koshy, Sriram Subramanian, Kartik Paramasivam, Mammad Zadeh, Neha Narkhede, Jun Rao, Jay Kreps, and Joe Stein. Building a replicated logging system with apache kafka. *PVLDB*, 8(12):1654–1665, 2015.

[12] Mark Marchukov. Logdevice: a distributed data store for logs. https://code.facebook.com/posts/357056558062811/logdevice-a-distributed-data-store-for-logs/, 2017.

[13] Joe Francis and Matteo Merli. Open-sourcing pulsar, pub-sub messaging at scale. https://yahooeng.tumblr.com/post/150078336821/open-sourcing-pulsar-pub-sub-messaging-at-scale, 2016.

[14] Amazon kinesis. https://aws.amazon.com/kinesis/, 2017.

[15] Google pub/sub. https://cloud.google.com/pubsub/, 2017.

[16] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 423–438, New York, NY, USA, 2013. ACM.

[17] Matthew Rocklin. Dask streamz. https://streamz.readthedocs.io/en/latest/, 2018.

[18] Twitter. Storm: Distributed and fault-tolerant realtime computation. http://storm-project.net/.

[19] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 239–250, New York, NY, USA, 2015. ACM.

[20] Apache Flink. https://flink.apache.org/, 2018.

[21] Apache Beam. https://beam.apache.org/, 2018.

[22] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment*, 8:1792–1803, 2015.

[23] Sriram Krishnan, Mahidhar Tatineni, and Chaitanya Baru. Myhadoop - hadoop-on-demand on traditional hpc resources. Technical report, San Diego Supercomputer Center, 2011.

[24] Ekasitk. Spark-on-hpc. https://github.com/ekasitk/spark-on-hpc, 2016.

[25] Supun Kamburugamuve, Saliya Ekanayake, Milinda Pathirage, and Geoffrey Fox. Towards high performance processing of streaming data in large data centers. In *HPBDC 2016 IEEE International Workshop on High-Performance Big Data Computing in conjunction with The 30th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2016), Chicago, Illinois USA, Friday*, 2016.

[26] Supun Kamburugamuve, Karthik Ramasamy, Martin Swany, and Geoffrey Fox1. Low latency stream processing: Twitter heron with infiniband and omni-path. In *Technical Report*, 2017.

[27] Nicholas Chaimov, Allen Malony, Shane Canon, Costin Iancu, Khaled Z. Ibrahim, and Jay Srinivasan. Scaling spark on hpc systems. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '16, pages 97–110, New York, NY, USA, 2016. ACM.

[28] T. Bicer, D. Gursoy, R. Kettimuthu, I. T. Foster, B. Ren, V. De Andrede, and F. De Carlo. Real-time data analysis and autonomous steering of synchrotron light source experiments. In *2017 IEEE 13th International Conference on e-Science (e-Science)*, pages 59–68, Oct 2017.

[29] Y. Du, M. Chowdhury, M. Rahman, K. Dey, A. Apon, A. Luckow, and L. B. Ngo. A distributed message delivery infrastructure for connected vehicle technology applications. *IEEE Transactions on Intelligent Transportation Systems*, PP(99):1–15, 2017.

[30] Dennis Gannon. Observations about streaming data analytics for science. https://esciencegroup.com/2016/05/23/observations-about-streaming-data-analytics-for-science/, 2016.

[31] Wolfgang Eberhardt and Franz Himpsel. Next-generation photon sources for grand challenges in science and energy. https://science.energy.gov/~/media/bes/pdf/reports/files/Next-Generation_Photon_Sources_rpt.pdf, 2009.

[32] Eric Hand. X-ray free-electron lasers fire up. *Nature*, 461(7265):708–709, oct 2009.

[33] Stanford. Linac coherent light source. https://portal.slac.stanford.edu/sites/lcls_public/Pages/Default.aspx, 2017.

[34] A Münnich, Steffen Hauf, Burkhard Heisen, Friederike Januschek, Markus Kuster, Philipp Micheal Lang, Natascha Raab, Tonn Rüter, Jolanta Sztuk-Dambietz, and Monica Turcato. Integrated detector control and calibration processing at the european xfel, 10 2015.

[35] CHX team. Chx.pdf. https://www.bnl.gov/nsls2/beamlines/files/pdf/CHX.pdf.

[36] Mark Sutton. Streaming data analysis tools to study structural dynamics of materials. https://www.bnl.gov/nysds16/files/pdf/talks/NYSDS16%20Abeykoon.pdf, August 2016.

[37] D. Gursoy, F. De Carlo, X. Xiao, and F. Jacobsen. Tomopy: A framework for the analysis of synchrotron tomographic data. *Journal of Synchrotron Radiation*, 21, Aug 2014.

[38] Betsy A. Dowd, Graham H. Campbell, Robert B. Marr, Vivek V. Nagarkar, Sameer V. Tipnisand Lisa Axe, and D. Peter Siddons. In *Developments in synchrotron x-ray computed microtomography at the National Synchrotron Light Source*, volume 3772, pages 3772 – 3772 – 13, 1999.

[39] J. Nuyts, C. Michel, and P. Dupont. Maximum-likelihood expectation-maximization reconstruction of sinograms with arbitrary noise distribution using nec-transformations. *IEEE Transactions on Medical Imaging*, 20(5):365–375, May 2001.

[40] Mary Shaw. The impact of modelling and abstraction concerns on modern programming languages. In Book: On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases, and Programming Languages, Springer New York, 1984.

[41] Vivekanandan Balasubramanian, Iain Bethune, Ardita Shkurti, Elena Breitmoser, Eugen Hruska, Cecilia Clementi, Charles Laughton, and Shantenu Jha. Extasy: Scalable and flexible coupling of md simulations and advanced sampling techniques. In *e-Science (e-Science), 2016 IEEE 12th International Conference on*, pages 361–370. IEEE, 2016.

[42] Matteo Turilli, Mark Santcroos, and Shantenu Jha. A comprehensive perspective on pilot-job systems. *ACM Comput. Surv.*, 51(2):43:1–43:32, April 2018.

[43] Andre Luckow, Lukas Lacinski, and Shantenu Jha. SAGA BigJob: An Extensible and Interoperable Pilot-Job Abstraction for Distributed Applications and Systems. In *The 10th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 135–144, 2010.

[44] Andre Luckow, Mark Santcroos, Ashley Zebrowski, and Shantenu Jha. Pilot-data: an abstraction for distributed data. *Journal of Parallel and Distributed Computing*, 79:16–30, 2015.

[45] Andre Luckow, Ioannis Paraskevakos, George Chantzialexiou, and Shantenu Jha. Hadoop on HPC: integrating Hadoop and pilot-based dynamic resource management. In *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*, pages 1607–1616. IEEE, 2016.

[46] Andre Merzky, Ole Weidner, and Shantenu Jha. Saga: A standardized access layer to heterogeneous distributed computing infrastructure. *SoftwareX*, 1:3–8, 2015.

[47] Andre Merzky, Matteo Turilli, Manuel Maldonado, Mark Santcroos, and Shantenu Jha. Using Pilot Systems to Execute Many Task Workloads on Supercomputers. *JSSPP 2018 (in conjunction with IPDPS'18)*, 2018. http://arxiv.org/abs/1512.08194.

[48] SAGA-Python. http://saga-project.github.io/saga-python/, 2018.

[49] Pilot-streaming: Managing stream processing on hpc. https://github.com/radical-cybertools/pilot-streaming, 2018.

[50] Streaming mini-apps. https://github.com/radical-cybertools/streaming-miniapps, 2018.

[51] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. Mllib: Machine learning in apache spark. *J. Mach. Learn. Res.*, 17(1):1235–1241, January 2016.

[52] Spark-streaming: K-means. https://spark.apache.org/docs/2.1.0/mllib-clustering.html, 2017.

[53] A. Merzky and S. Jha. Synapse: Synthetic application profiler and emulator. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1259–1268, May 2016.

[54] Andrew Montalenti. Pykafka: Fast, pythonic kafka, at last! http://blog.parsely.com/post/3886/pykafka-now/, 2016.