# StreamInsight: Characterization and Performance Modeling of Serverless and HPC Streaming Applications

Andre Luckow[1,2,3], 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*—Modern scientific experiments and industrial Internet-of-Things (IoT) applications increasingly rely on Experiment-in-the-Loop Computing (EILC). For this purpose, stream processing capabilities are crucial and provide the ability to analyze and derive real-time insights on incoming data feeds, simulations, and Internet-of-Things (IoT) sensors using state-of-the-art techniques, such as machine learning. Managing heterogenous infrastructure and applications to support this kind of processing ~~processing~~ cloud and HPC infrastructure is challenging. We present Pilot-Serverless for unified management of streaming workloads on serverless clouds and HPC. Understanding the performance and scaling characteristics to select the right resource configuration and estimate scaling limits is hindered by opaque infrastructure characteristics in particular due to shared infrastructure. StreamInsight provides the necessary performance understanding and aids developers to select the right infrastructure configuration and predict scaling limits of the application. Underlying StreamInsight is the universal scalability law, which permits us to accurately quantify the scalability properties of scientific stream applications. Using experiments on HPC and Lambda, we demonstrate that StreamInsight provides an accurate model for a variety of data rates, model sizes and resource configurations. Further, we derive interesting insights on scalability of streaming applications on Serverless Lambda and HPC showing e. g., that while the absolute performance achieved on HPC is better, the performance in the cloud is more predictable and reliable. StreamInsights will provide the foundation for new advanced middleware capabilities, such as predictive scaling.**

***Index Terms*—HPC, Cloud, Serverless, Streaming, Experiment-in-the-Loop, Machine Learning**

## I. Introduction

The integration of experimental devices, IoT sensors and computational HPC and cloud infrastructure becomes increasingly important to enable new scientific discoveries and advances. Experiment-in-the-Loop Computing (EILC) describes a new form of computing that addresses the dynamism of such environment that enables the coupling of data feeds from instruments with different computational capabilities on the edge and on HPC and cloud infrastructure enabling new forms of dynamic, closed-loop applications. However, managing compute, data and application models in such a heterogeneous environment is very challenging due to the lack of easy-to-use, compose-able abstractions for resource management and coordination that can address the compute continuum from IoT sensors to the cloud [1].

Stream processing capabilities are an instrumental part of Experiment-in-the-Loop Computing (EILC) and provide the critical capabilities for deriving real-time insights on experimental data using cloud and HPC resources and allows application to feed these insights back [2]. Applications range from autonomous vehicles, realtime analysis of astronomy data to other scientific experiments, such as light sources. For example, Synchrotron light source experiments, such as those at the National Synchrotron Light Sources II (NSLS-II) [3] or the X-Ray Free Electron Laser (XFEL) light sources. Streaming processing is crucial for detecting interesting events, preprocessing data and steer applications. An important technique is the usage of machine learning to detect outlier and select important events.

Infrastructures for supporting scientific applications are becoming increasingly complex and heterogeneous: modern applications can spawn from the edge, cloud to HPC. Infrastructures are usually complementary to each other, e. g. edge and cloud computing is well suited for realtime processing close to the data source. HPC environments are better suited for high-end computational and data-intensive tasks. One increasingly important infrastructure paradigm is *serverless computing*. Serverless refers to different higher-level abstractions for cloud infrastructures that abstract away most resource management concern, such as the allocation of nodes, containers and the management of the processing framework [4], [5]. However, many challenges related to the usage and integration of serverless computing remain:

**Development and Abstractions:** Serverless abstractions lack many capabilities of HPC abstractions (such as MPI) and Big Data abstractions (such as MapReduce). Application codes need to be careful partitioned and wrapped into the serverless APIs with limited abilities to express e. g. task-level parallelism, and the composition of functions [6]. While HPC infrastructures utilize batch resource management systems [7], serverless infrastructures typically rely on high-level resource constraints and SLAs for compute and data, such as memory and concurrency. Despite the convenient resource management and scaling, many low-level tasks remain, e. g. the need to configure the environment and implement parallelism at an appropriate level of concurrency. Also, abstractions and resource

specifications differ significantly between infrastructure providers making it difficult to spawn applications across different clouds and HPC.

**Deployment:** The deployment and integration of serverless application with HPC applications is difficult as runtimes between these heterogeneous environments are often not portable. Configurations are highly sensitive to infrastructure. While serverless promises to elevate manual resource management, it still requires the application to configure and adjust resource-related parameters, e. g., the degree of parallelism and the number of shards. Depending on the application characteristic and infrastructure the right level of parallelism must be determined and statically provided.

**Runtime and Performance:** The application and infrastructure needs to be monitored during runtime. Often, it is difficult to adjust parameters at runtime and enable dynamic scalability. To effectively tune applications a performance understanding is crucial. On both HPC and serverless infrastructures, there is an immense gap in understanding the performance of streaming applications and infrastructure. However, in particular in serverless computing, most details of the serverless runtime are abstracted and opaque to the application making the performance and scaling properties difficult to understanding. As a consequence users are often required to manually evaluate the systems under many different framework, infrastructure and resource configurations. Often, it is difficult to simulate operational environment, e. g., high load situations, and collect the right performance metrics and obtain an understanding of the scaling and performance properties of the system.

This paper proposes *Pilot-Serverless* as a common abstraction for resource management for streaming applications on HPC and serverless clouds. Pilot-Serverless extends Pilot-Streaming [8] and simplifies the development and deployment of streaming applications on heterogeneous infrastructure; the Pilot-Abstraction enables developer to efficiently express task-level parallelism overcoming many limitations of current serverless abstractions by providing, e. g., the ability to manage resources and compute tasks for batch and stream processing across different types of resources in a unified way. Further, we propose StreamInsight: StreamInsight enables the performance understanding of streaming applications and infrastructure. It is based on the Streaming Mini-App framework [8]. It follows a two-pronged approach of analytic performance modeling based on the Universal Scalability Model [9] and empirical data collection and analysis for validation of the model. We investigate the capabilities of the StreamInsights framework and performance modeling method using a realistic streaming pipeline that utilizes machine learning for detecting abnormal behavior in an incoming stream of data. In particular, we qualitatively and quantitatively compare and contrast two streaming infrastructures: (i) Serverless based on AWS Lambda [10] and Kinesis [11] and (ii) HPC based on Kafka [12] and Dask [13], and show that StreamInsight allows developers to understand and quantify the performance trade-offs between the different infrastructures.

This paper is structured as follow: In section II we provide important background and related work. We continue with a discussion of Pilot-Serverless in section III. In section IV, we present StreamInsight – our approach to analytical modeling augmented with empirical performance measurements. Further, we utilize StreamInsight to collect and describe the performance of streaming applications for different infrastructures and algorithm complexities. We conclude with a discussion of the results and future work in section V.

## II. Background and Related Work

In this section, we present important background and related work: After an introduction to serverless in section II-A, we present different related work: in section II-B we investigate related work dealing with benchmarking and performance modeling.

### A. Serverless Computing

Serverless infrastructures differ significantly from HPC and traditional cloud environments. Most commonly, the term *serverless* [5] is used for *Function-as-a-Service (FaaS)* infrastructures, such as AWS Lambda [10]. It is a computing paradigm that allows the scalable and fault-tolerant execution of functions in response to defined events without the need to consider low-level concerns, such as resource management, provisioning and scaling with respect the number of events. Serverless computing tightly integrates the abstraction and programming model with the runtime system and infrastructure. The function code is required to be stateless, i. e., the state needs to managed and stored outside the function. Further, the function is subject to different resource constraints: it has only access to a container with a single core, a defined amount of memory usage and is subject to a strict walltime (e. g. 15 minutes processing time per event for Lambda). The cloud provider provides a high-level SLA with respect these resources and fault tolerance.

Serverless computing is of increasingly interesting for scientific applications, especially in conjunction with HPC capabilities [14], [15], [16]. The event-driven nature of streaming applications make them ideally suited for the serverless paradigm. While the term serverless narrowly used for FaaS, e. g,, AWS Lambda [10], Google Cloud Functions and Azure Functions, more broadly it includes services, such as Kinesis [11] for message brokering and S3 [17] for storage.

### B. Streaming Performance and Modeling

While there is comprehensive work on benchmarking streaming systems [18], [19], typically these benchmarks are limited in their scope and often do not reflect the complex and highly dynamic requirements of real-world of scientific streaming applications. Seltzer et al. [20] emphasize the importance of application-specific benchmarks for achieving performance insights on real world applications. Fox et al. propose the concept of Big Data Ogres to describe well-understood application characteristics, which served as basis for the development of a

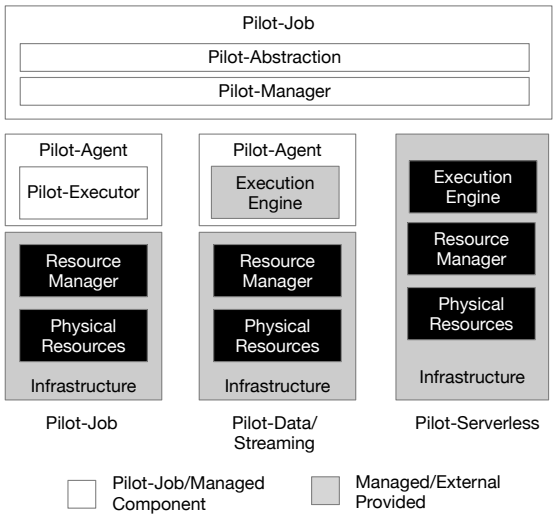


Fig. 1: **Pilot-Abstraction on Heterogeneous Infrastructures: HPC, Cloud and Serverless:** The Pilot-Abstraction can integrate with managed infrastructure services on various levels. Pilot-Serverless abstracts the serverless-specific resource allocation and execution model of the infrastructure provider providing a unified way to manage HPC and serverless resources.

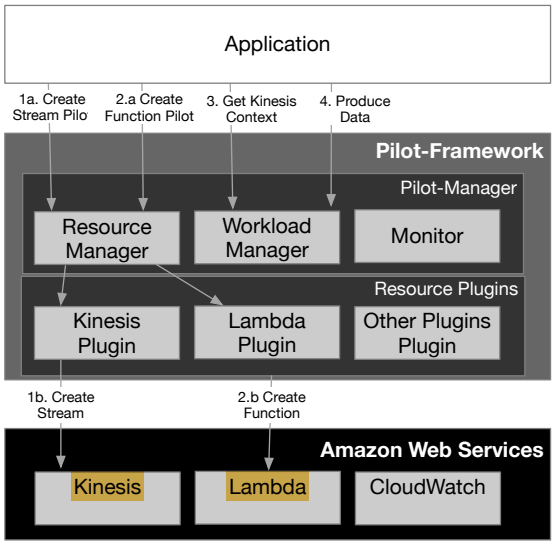


Fig. 2: **Pilot-Serverless High-Level Architecture and Interactions:** Pilot-Serverless enables the unified management of batch and streaming compute tasks on serverless infrastructure. It orchestrate the setup of Kinesis and Lambda resources and managed the streaming workload.

set of benchmarks that encapsulate commonalities of these applications [21]. The concepts of Mini-Apps was introduced in the domain of data-intensive HPC apps by Sukumar et al. [22]. Luckow et al. refined this concept for streaming apps [8].

Increasingly statistical methods and machine learning approaches are used to optimize infrastructure configurations, e. g., by providing meaningful initial configurations or by predictively scaling up/down environments [23]. Kremer-Herman et al. [24] propose a model for recommending the optimal infrastructure configuration for master/worker applications. Ernest [25] is a system that combines analytical modeling and data derived from a small set of performance counters and sample runs. Cherrypick [26] uses a black-box model based on Bayesian optimization to find optimal resource configurations for Big data frameworks in the cloud. Caladrius [27] is a streaming forecasting system und provides a predictive model for traffic and an analytical model for the processing system. A main limitation of Calidarius is that it only investigates a small part of the stream processing pipeline and is constrained to Heron. The experimental validation is done using a simple word count example. More complex scientific applications are not investigated.

## III. Pilot-Serverless: Pilot-Abstraction for Serverless Computing

*Pilot-Serverless* aims to address the challenges related to development, deployment and management of streaming applications on serverless clouds. In particular, these challenges are: (i) the application code needs to wrapped using the serverless API. Commonly, the APIs differ between the cloud vendors, (ii) the infrastructure needs to be configured taking into the account the right level of parallelism and (iii) the application run needs to be monitored and allocated resources and configurations need to be adapted if required.

Pilot-Serverless provides a unified interface to serverless and HPC infrastructure. Underlying Pilot-Serverless is the Pilot-Abstraction, which provides two key capabilities: (i) to provide unified and infrastructure-agnostic access to resource and (ii) to enable applications to schedule tasks on the acquired resources [28], [29]. The abstraction consists of two key entities: the Pilot-Job, which represents a user-defined set of resources and the Compute-Unit, which is self-contained pieces of work, also referred to as tasks, that are executed on the Pilot-managed resources.

Figure 1 illustrates the different types of compute infrastructures supported by the Pilot-Abstraction. The original Pilot-Job focused on the provisioning and management of HPC and cloud resources (compute nodes, VMs), and the execution of the application workload on these. Pilot-Data [30] and Pilot-Streaming [8] extended the Pilot-Abstraction to externally provided distributed execution engines, e. g., Spark [31] and Dask [13], to support higher-level, data-intensive workloads that rely on the capabilities of these execution engines. With serverless the operational responsibility for the execution engine shifts from the application to the infrastructure provider. Pilot-Serverless enables the integration of serverless execution engines, such as AWS Lambda and Kinesis, into Pilot-based applications.

Several challenges are addressed by Pilot-Serverless: the functions code needs to be deployed on the infrastructure. Input data and streams need to managed to connected with the function code. For example, compute units need to be assigned to

the right size containers. As available resources per task is limited, this often means that the computational and data requirements need to aligned to meet the strict concerns of the serverless runtime. Other types of infrastructures, for example message brokers like Kinesis, require the manual management of shards and partitions. It is critical to allocate the right number of shards for a pipeline as this controls the I/O and parallelism, and thus, has a significant impact on the throughput. Over-provisioning results in unnecessary costs and worse performance.

Pilot-Serverless utilizes a modular architecture and supports the configuration/deployment and the execution of serverless streaming applications. Figure 2 illustrates the internals of Pilot-Serverless. The Pilot-Manager provides a unified interface for running Compute-Units on these frameworks, but also serves as orchestration manager for managing data and compute across multiple frameworks.

Managing a streaming application pipeline requires several infrastructure and application components, in particular a message broker (e. g., Kinesis) and a distributed execution engine (e. g., Lambda). The Pilot-Manager serves as interface for managing these frameworks and running Compute-Units on these frameworks. It also serves as orchestration manager for managing data and compute across multiple frameworks. In the first phase, Pilot-Streaming is used to allocate resources and deploy the application on these resources. For example, the application can use Pilot-Streaming, to initiate a Kinesis Pilot and allocate a stream with a defined number of shards (Step 1a and b in Figure 2). For this purpose, the user needs to create a Pilot-Description, which provides a normative way to specify resources for a streaming broker, e. g., the number of topic shards for Kinesis and Kafka can be specified using the same attribute. In addition, it support resource-specific extension fields. In cases where the message broker is outside of the managerial scope of the application this step is skipped, this step can be skipped.

The Lambda Pilot can then be created (Step 2a and b in Figure 2) Pilot-Streaming supports the execution of arbitrary Python code as Lambda function. Pilot-Streaming will serialize the function and dependency and ship it to Lambda. In addition, the usage of Lambda Layers is supported. Again, the Pilot-Description is used to specify and control the parallelism in a infrastructure-agnostic way, while allowing the support for infrastructure-specific capabilities, such as layers. Further, it the Pilot-Description is used to map the function to the data source, i. e., the Kinesis stream.

In summary, Pilot-Serverless simplifies the configuration, deployment and execution of serverless streaming applications. It provides a consistent way to allocate resources and deploy streaming application pipelines. The Pilot-Abstraction provides a well-established and well-designed API for task-based workloads that can be utilize to express different forms of parallelism, from bag-of-tasks, data-parallelism to streaming on heterogeneous infrastructure making it particularly well-suited for workloads required to run on a diverse set of infrastructure.

## IV. StreamInsight: Performance Characterization and Modeling

The characteristics of streaming applications are complex and vary dynamic. Understanding their performance characteristics and requirements is necessary to ensure that sufficient resources have been allocated, frameworks and infrastructure have been properly configured to meet the demands of time-sensitive streaming applications. Often, it is difficult to discern the impact of a single parameter, e. g. the amount of cores or memory, on the performance of the stream processing system. The large combinatorial space of parameters makes it impossible to investigate the entire space, and thus it is critical to use a good experimental design. StreamInsight supports the full lifecycle of performance experimentation and modeling: from the design, automation, analysis, modeling to the explanation of experiments. It enables users to identify and quantify bottlenecks, test the streaming system under various traffic loads, fine-tune system configurations, optimize resource allocations and predict the performance of applications. Further, the system can serve as building block for higher-level functions, such as predictive autoscaling.

StreamInsight uses a two-pronged approach: (i) We use the Universal Scalability Model [9] to understand and predict the scalability of streaming infrastructure and applications; (ii) we utilize data collected using the Mini-App framework to estimate the parameters of the model.

### A. Modeling Method

USL defines the following model for the throughput $T$ of a system:

$$T(n) = \frac{N}{1+\sigma(N-1)+\kappa N(N-1)}$$

The model is controlled by two parameters: $\sigma$ measures the serial overhead in the system, while $\kappa$ captures the coherence between all processors. For example, $\sigma$ can be used to quantify overheads, such as serialization; $\kappa$ quantifies all-to-all communication, such as sharing model parameters across all processors.

The usage of the USL has many advantages: it provides the right level of granularity while maintaining a high degree of interpretability. The model does not require low-level timing to model the system. Gunther [32] showed that USL generalizes Amdahl's laws [33] and adds meaningful extensions to explain, e. g., degradations.

There are two primary metrics relevant for stream processing: (i) latency and (ii) throughput. The latency $l$ is defined as time difference between two states, e. g. $t_{px}$ is defined as time the between arrival and processing of the data. The throughput $T$ is the defined as the number of events (e. g., messages) a system handles in a certain amount of time. In the following, throughput refers to the maximum sustained throughput of a configuration, i. e., the optimal load a streaming system can handle without deterioration in the performance (e. g., due to back-pressure) [34]. In the following, we particular focus on modeling $T^{px}$.

There are two primary components of a streaming system: the message broker ($br$) and the streaming processing system ($px$). We refer the throughput of the broker $T^{br}$ and the throughput of the stream processing system as $T^{px}$. The parallelism in both systems is controlled by the number of partitions $N(p)$: $N^{br}(p)$ refers to the broker partitions, $N^{px}(p)$ to the processing partitions. $N^{br}(p)$ refers to the number partitions for Kafka and to the number shards for Kinesis. $N^{br}(p)$ need to be carefully chosen taking into the account the available number of nodes, the cores per nodes, the disk I/O per node and the desired processing parallelism. For Kinesis only $N^{br}(p)$ can be specified.

Further, broker and stream processing system can be distributed across a multiple broker nodes $N^{br}(n)$ and processing nodes $N^{px}(p)$. The processing throughput $T_{px}$ primarily depends on the parallelism of the stream processing system. As described, the primary dependent parameters are: $N^{br}(p)$, $N^{px}(p)$, $N^{br}(n)$, and $N^{px}(n)$. The ratio between $N^{px}(n)$ and $N^{px}(p)$ determines the amount of memory per core, available, which is a significant factor in particular for machine learning applications as this determines the memory available for storing model parameters and data. For simplicity, we keep the ratio $N^{px}(n)$ to $N^{px}(p)$, and $N^{br}(n)$ to $N^{br}(p)$ fixed. Further, we use the same number of broker and processing nodes: $N(n)^{px} = N(n)^{br}$ referred to as $N(n)$.

In general, parameters can be classified into two classes: infrastructure-related parameters and user/application dependent parameters. Examples for infrastructure parameters are: the number of broker nodes, partitions, memory, CPU cores and network speed. Depending on the environment, the user has a different level of control of these parameters. For example, while HPC environment typically allow the selection of the number of nodes, on IaaS infrastructure the use can chose different types of VMs, storage and networks. Serverless environments are most constrained to very few options, e. g., the number of containers and container memory. User/application parameters include the input data rate and the processing complexity (with respect to CPU, memory and communication). In contrast to batch applications, which are often exploratory, the processing complexity typically remains the same as the same algorithm is applied to the incoming stream of data.

### B. Streaming Mini-App: Experiment Automation

Collecting performance data for characterization of distributed systems and in particular streaming system is challenging: The number and properties of the involved systems and frameworks is high, which results in heterogeneous data, which is difficult to understand, normalize and compare. The Streaming Mini-App framework [8] addresses these challenges and allows the user to simulate complex streaming applications, execution frameworks and infrastructures. Using this capability, application developers are able to automate the exploration of a large parameter space of different frameworks and application configurations.

The Mini-App framework is build on the Pilot-Streaming to manage resources, stream processing runtimes, and the message broker. We extended the framework to support the collection of comprehensive performance data on serverless infrastructure in addition to HPC infrastructure and provide the ability to trace data across all components, i. e., data source, brokering and processing system. For this purpose, the framework assigns a unique run id, which is propagated to all involved components. This way events can be attributed to certain runs and analyzed in a consistent way. The instrumentation system is architected in a modular way allowing the developer to easily add/remove metrics and events for all components, e. g, metrics collected by the Pilot-System and processing framework, application log files, and cloud-based log services, such as AWS CloudWatch for serverless Lambda applications. The framework is architected in a modular way, i. e., the data collector can easily be extended to support new systems and infrastructure. The framework can simulate different data rates and input data characteristics and allows, e. g., the variation of the data production rate aiding the developer to identify and understand the maximum sustained throughput before back-pressure occurs. To conduct measurements at the maximum sustained throughput, the framework utilizes an intelligent back-off strategy during data production.

### C. Performance Characterization

In this section, we present an in-depth characterization of streaming applications on AWS serverless and HPC infrastructure. We use the Mini-App framework on different infrastructures. For HPC, we use the XSEDE machines Stampede2 and Wrangler. Each Wrangler nodes provides 48 cores and 128 GB RAM. On Stampede2 the Knights Landing nodes with 68 cores, 96 GB RAM were used. For serverless, we use AWS Lambda and Kinesis with different container sizes and number of partitions.

We investigate the performance of two stream processing systems: (i) Kinesis/Lambda serverless processing on AWS and (ii) Kafka/Dask stream processing on HPC. In both cases, we use the clustering algorithm KMeans [35] as a representative workloads. K-Means is well understood and commonly used in streaming context to detect abnormal behavior. KMeans has a complexity of $O(nc)$ with $n$ as the number of points and $c$ as the number of cluster centroids. The algorithm comprises of two phases: In the first step, K-Means computes the euclidian distance between all points ($n$) and the centroids ($c$) resulting in a complexity of $O(nc)$. In the second step the new centroid positions are computed by averaging the positions of all points assigned to a centroid.

In our implementation, we use the MiniBatch K-Means of scikit-learn [36]. Using the mini-batch algorithm, the K-Means model is continuously updated using the incoming data. For this purpose, the model is shared across tasks using a file storage (i. e., S3 respectively the Lustre filesystem). In our experiments, we vary the number of centroids to simulate workloads with different computational complexity. We utilize different number of centroids to simulate different sizes of models: the larger the number of centroids that more compute is necessary. Further, the I/O for writing/reading the model is impacted.

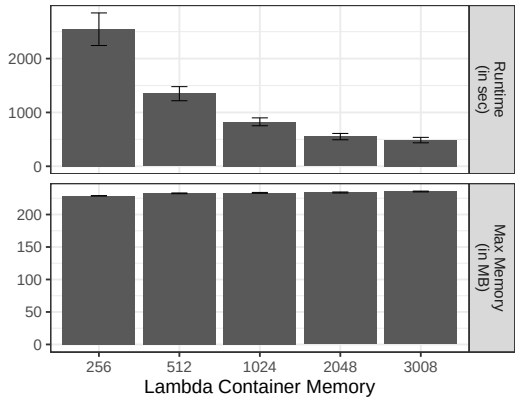


Fig. 3: **Lambda Container Memory for 8,000 points and 1,024 centroids:** Lambda containers with larger amount of memory provide more compute capacity and thus, enable shorter runtimes. The fluctuation in the data is significant lower for larger container sizes.


Fig. 4: **Message Processing Time for KMeans on AWS Lambda and HPC:** Broken down by Number Partitions, Message Size and Model Complexity. The processing times increase with the message size and the model complexity. While for Lambda the processing times remain constant with increasing parallelism, we observe a negative impact for Dask/Kafka on HPC likely due to the need to use shared resources, e. g., Lustre.

For both Dask and Lambda, we investigated different message sizes from 8,000 to 26,000 points. The message size is on average 296 kb for 8,000 points, 592 kb for 16,000 points and 962 kb for 26,000 points. Further, we use the number of centroids of the K-Means model to vary the model complexity from 128 to 8192 centroids.

*1) Lambda Memory:* An important configuration for Lambda applications is the amount of memory and the concurrency. In the following, we utilize the standard concurrency for Lambda, i. e., the Lambda execution engine determines how many containers are allocated. We use up to two `m5.4xlarge` nodes to produce K-Means data with a different number of centroids and different message sizes.

Figure 3 illustrates the runtime of the KMeans Lambda function in dependence to the requested a lambda container memory. We investigate the memory usage for a message size of 8K points and 1024 clusters. Even though, the actual used memory for the lambda function nearly remains constant, the runtime decreases with larger memory runtimes (currently max 3,008 MB) indicating that AWS scales the CPU allotment proportional to the memory.

*2) Lambda vs. Dask:* Figure 4 investigates the processing time per message for different sizes of messages und clustering models with different complexities. For Dask, we use up to 128 processes distributed across up to nine 48 core nodes. For Lambda, we use 3 GB RAM containers. The processing times increase with the number of points and centroids for both Dask and Lambda due to the increase of the computational, IO and memory demand of the processing function. For Lambda, the processing times remain stable with higher parallelisms, i. e., higher partition counts. This demonstrates that the Lambda containers are well isolated providing a predictable performance. In contrast to Lambda, for Dask the processing increase with the number of partitions indicating a memory and/or IO contention.

Further, we investigate the behavior of Lambda, in particular the parallelism of the system during the test run. The degree of parallelism, i. e., the number of Lambda containers assigned by the AWS Lambda execution engine is mainly determined by the Kinesis partitions, i. e., Lambda will not generate more concurrent containers than partitions. Cloudwatch creates one log stream per container, which corresponds to the number of containers. The numbers of containers is mainly determined by the number of partitions in the Kinesis stream. The workload of the lambda function, determined by the message size and number of clusters has a smaller impact on the number of containers – surprisingly Lambda allocated fewer containers for for more compute intensive tasks with larger cluster and message sizes. The maximum number of containers allocated by Lambda is 30. Also, we observed several irregularities visible in the high deviation in the data. We attribute this to the different load situations on AWS.

Figure 5 illustrates the throughput and speedup. For scenarios with higher compute to I/O ratio, i. e. in particular larger model sizes, a small speedup is observable for Dask until 4 partitions.

### *D. Performance Modeling*

We apply USL to model the throughput for different configurations. We assume a stable system, i. e., the system reached its maximum sustained throughput. The Mini-App frameworks ensures using an intelligent backoff function on the consumer function that the system operates at a stable data rate. Further, overheads for initialization of the streaming application and infrastructures, i. e., the time for starting the streaming framework via the Pilot-Framework, are not considered. We focus

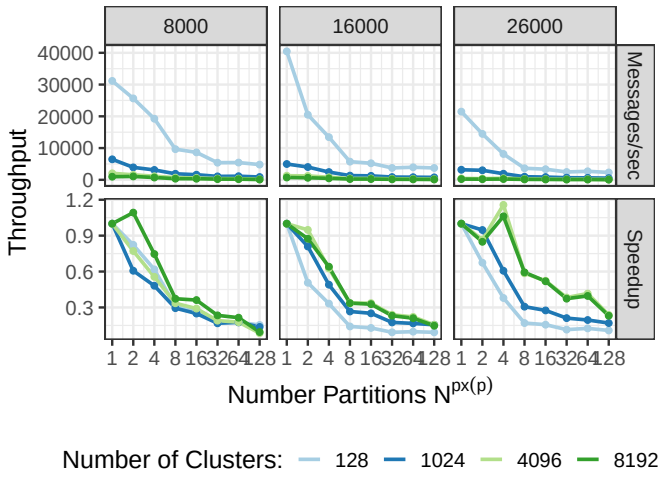


Fig. 5: **Dask Throughput and Speedup:** The increased processing times also impact the throughput and speedup. For scenarios with higher compute to I/O ratio a small speedup is observable for Dask until 4 partitions.

on model the processing throughput of the system $T_{processing}$ – also referred to as $T$. We fit the data to the USL model using a non-linear regression model provided by the USL package in R [37].

Figure 6 illustrates the results. Streaming applications are in general mostly loosely coupled, all tasks process a partition of data independently with minor synchronization for model sharing. Further, there are shared IO resources used by the broker and the processing engine. These characteristics are reflected in the $\sigma$ and $\kappa$ coefficients of the USL model. $\sigma$ quantifies the serial overhead, while $\kappa$ measures the cross-talk between all nodes.

It shows that the scalability depends greatly on the infrastructure. While HPC provides the better absolute performance, cloud infrastructures are more predictable. For HPC the peak performance is already reached using a single partition. The system performance degrades with increased parallelisms due to contention and coherencies overheads. Lambda and Kinesis provide better resource isolation, which is instrumental for ensuring SLAs on serverless infrastructure.

For Lambda, the throughput increases with the number of partitions thanks to the greater parallelism. Smaller message and model sizes have a higher throughput. USL shows that $\sigma$ is close to zero and $\kappa$ is zero demonstrating an almost optimal scalability. Lambda/Kinesis scale at almost optimal efficiency demonstrating that AWS provides very predictable SLAs for its services and that Lambda containers are well isolated providing a predictable performance.

For Dask, a $\sigma$ between 0.6 and 1 indicates that the majority of the reduced scalability is caused by contention, e. g., through a memory and/or IO contention at the shared filesystem. However, $\kappa$ indicates that some significant coherency due to cross-communication between all processors, e. g., the synchronization of the model updates via the shared filesystem. Thus, the peak scalability of the system is already reached with a single partition. Further, it must be noted that Dask/Kafka on HPC was carefully fine-tuned with respect to the memory/-core ratio, location of the Kafka data log files etc. to ensure an optimal execution. However, the amount of shared resources that impact performance is significant larger than in a carefully SLA-managed serverless environment on AWS.

### *E. Model Evaluation*

To evaluate the model, we investigate the model fit as well as the performance of the model on unseen data. We observed a training $R^2$ between 0.85 and 0.98 indicating that the model is able to capture a great proportion of the variance within the data. For unseen data, we split the benchmark data into a training and test set. We utilize a different number of training configurations to create a performance model. We investigate the root mean squared error of the predictions on the unseen test data of the remaining configurations.

Figure 7 illustrates the results. A small number of observations is sufficient to create a well-performing model. In general, the Lambda/Kinesis is more predictable than the Dask/Kafka model. For Dask, we observe a higher RSME in particular for short running tasks, i. e., smaller message and model sizes.

In summary, we demonstrated that StreamInsight allows the quantification of the overheads and scalability for streaming applications and infrastructure using only very few data points. Due to the small amount of data it can easily be used to identify optimal configurations for production systems. In the future, we will use the model to dynamically scale streaming infrastructures. The performance models allow us to explain and quantify different properties, such as linear scalability, diminishing returns, bottleneck limitations, but also retrograde performance. StreamInsight will provide the basis for more advanced capabilities, such as predictive scaling.

## V. Conclusion and Future Work

Serverless environments are becoming an increasingly important runtime environment for different scientific applications including streaming application that an important component for experiment-in-the-loop-computing. The Pilot-Abstraction addressed critical challenges related to the development, deployment and resource management of such streaming applications. In this paper, we demonstrated that the Pilot-Abstractions' task model is well suited to express and manage serverless streaming workloads. The Pilot-Abstraction fills a critical gap and enhances the expressiveness of serverless abstractions, e. g., by adding the ability to coordinate across multiple tasks required for many applications, such as machine learning. Further, the Pilot-Abstraction enables the interoperable use of serverless, cloud and HPC resources.

Understanding and describing the performance of streaming infrastructure is crucial, e. g., for the optimization of infrastructure configurations for certain workloads and applications and for auto-tuning approaches, such as predictive scaling.

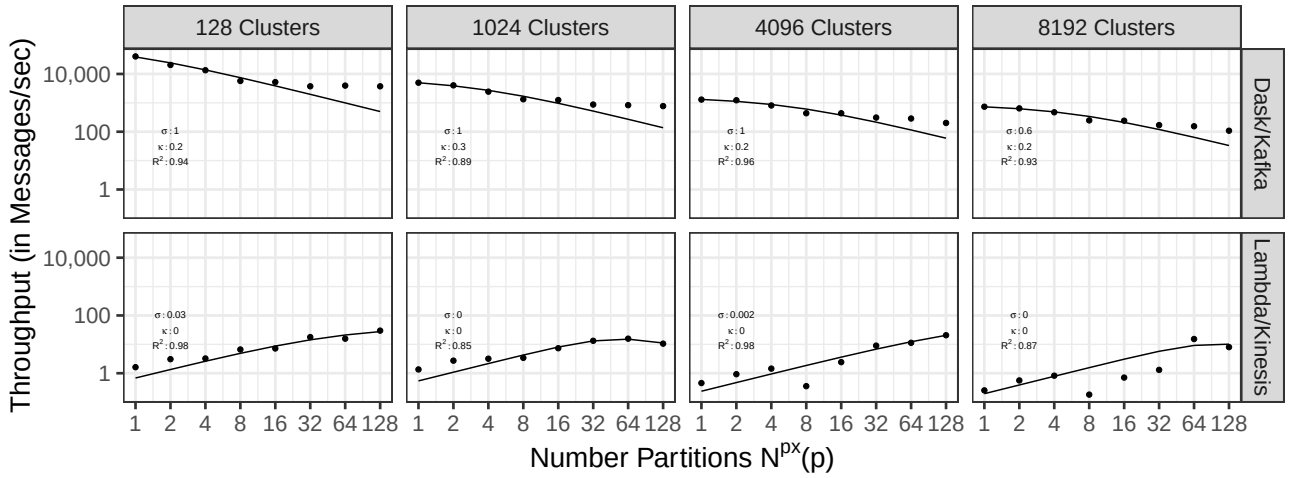


Fig. 6: **Model Fit Universal Scalability Law applied to Lambda and Dask Streaming:** The figure shows how the USL model was fitted to different scenarios. In all scenario the message size is 16,000 points. USL is suitable to describe the performance of streaming applications and enables StreamInsight to quantify overheads.

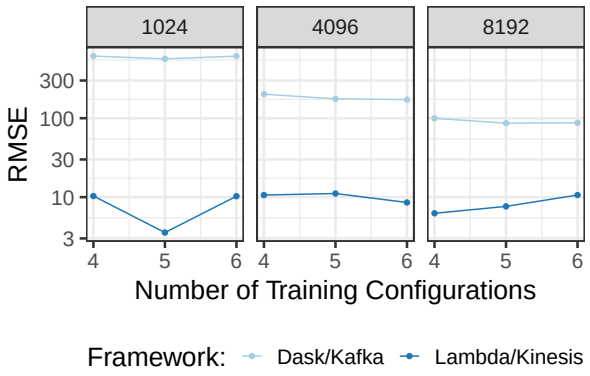


Fig. 7: RSME for Different Sizes of Training Data

StreamInsight demonstrated how the USL can be used to predict the scaling properties of a streaming applications using a minimal amount of data. We evaluated StreamInsight using different complex machine learning tasks. It enables application to characterize and understand application and infrastructure behavior, to identify bottlenecks and make prediction. We found that using Kafka/Dask based stream processing on HPC provides a better performance than Kinesis/Lambda. However, we were impressed by the scalability and predictability of serverless services, such as Kinesis and Lambda. Kinesis/Lambda provide a reliable, scalable solution for streaming tasks, which is however subject to several limitations. Most notable is the runtime, which is constrained to 15 minutes and the storage size of the deployment package of 250 MB. Also, currently no GPU are supported. Thus, for high-end workloads that require complex and large dependencies, e. g., TomoPy for light sources sciences, Lambda's are not suitable.

In the future, we will utilize StreamInsight to support more advanced experimentation-in-the-loop-computing scenarios making use of complex edge, fog and cloud infrastructure and support greater dynamism on all levels. Further, we will enhance the Pilot-Abstraction to better support FaaS workloads on HPC and also on edge devices. In addition, we use StreamInsight ton enhance the management capabilities of the Pilot-Abstraction, e. g., predictive auto-scaling, i. e., the ability to adapt the resource allocation and configuration with changes to the incoming data rate. For this purpose, the Pilot-Abstraction and StreamInsight provide two instrumental building blocks: dynamic resource management and a predictive model to guide resource allocation. Another use case is to determine the amount of throttling that needs to be applied to a data source to ensure processing. Further, we plan to extend this work to edge and fog computing. By moving serverless functions to the edge and thus, data, further optimizations are possible, however, it increases the demand and complexity of resource management.

*Acknowledgements:* We acknowledge support from NSF DIBBS 1443054 and NSF CAREER OAC 1253644. XSEDE computational resources were made available via XRAC allocation TG-MCB090174.

## References

[1] Pete Beckman, Jack Dongarra, Nicola Ferrier, Geoffrey Fox, Terry Moore, Dan Reed, and Micah Beck. Harnessing the Computing Continuum for Programming Our World. http://dsc.soic.indiana.edu/publications/paper_computing_continuum-2.pdf, 2019.

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

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

[4] Ioana Baldini, Paul C. Castro, Kerry Shih-Ping Chang, Perry Cheng, Stephen J. Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric M. Rabbah, Aleksander Slominski, and Philippe Suter. Serverless computing: Current trends and open problems. *CoRR*, abs/1706.03178, 2017.

[5] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Menezes Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. Cloud programming simplified: A berkeley view on serverless computing. Technical Report UCB/EECS-2019-3, EECS Department, University of California, Berkeley, Feb 2019.

[6] Ioana Baldini, Perry Cheng, Stephen J. Fink, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Philippe Suter, and Olivier Tardieu. The serverless trilemma: Function composition for serverless computing. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2017, pages 89–103, New York, NY, USA, 2017. ACM.

[7] Shantenu Jha, Judy Qiu, André Luckow, Pradeep Kumar Mantha, and Geoffrey Charles Fox. A tale of two data-intensive paradigms: Applications, abstractions, and architectures. *Proceedings of 3rd IEEE Internation Congress of Big Data*, abs/1403.1528, 2014.

[8] André Luckow, George Chantzialexiou, and Shantenu Jha. Pilot-streaming: A stream processing framework for high-performance computing. *Proceedings of 14th IEEE eScience*, 2018.

[9] Neil J. Gunther. A simple capacity model of massively parallel transaction systems. In *Int. CMG Conference*, 1993.

[10] AWS. Lambda: Run code without thinking about servers. https://aws.amazon.com/lambda/, 2019.

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

[12] 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.

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

[14] Geoffrey C. Fox, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. Status of serverless computing and function-as-a-service(faas) in industry and research. *CoRR*, abs/1708.08028, 2017.

[15] Eric Jonas, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the cloud: Distributed computing for the 99%. *CoRR*, abs/1702.04024, 2017.

[16] Vaishaal Shankar, Karl Krauth, Qifan Pu, Eric Jonas, Shivaram Venkataraman, Ion Stoica, Benjamin Recht, and Jonathan Ragan-Kelley. numpywren: serverless linear algebra. *CoRR*, abs/1810.09679, 2018.

[17] Amazon S3 Web Service. http://s3.amazonaws.com.

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

[19] 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)*, volume 00, pages 1789–1792, May 2016.

[20] Margo Seltzer, David Krinsky, Keith Smith, and Xiaolan Zhang. The case for application-specific benchmarking. In *Proceedings of the The Seventh Workshop on Hot Topics in Operating Systems*, HOTOS '99, pages 102–, Washington, DC, USA, 1999. IEEE Computer Society.

[21] Geoffrey C. Fox, Shantenu Jha, Judy Qiu, Saliya Ekanazake, and Andre Luckow. Towards a Comprehensive Set of Big Data Benchmarks, 2015.

[22] S. R. Sukumar, M. A. Matheson, R. Kannan, and S. Lim. Mini-apps for high performance data analysis. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 1483–1492, Dec 2016.

[23] Geoffrey Fox, James A. Glazier, Jcs Kadupitiya, Vikram Jadhao, Minje Kim, Judy Qiu, James Sluka, Endre Somogyi, Madhav Marathe, Abhijin Adiga, Jiangzhuo Chen, Oliver Beckstein, and Shantenu Jha. Learning everywhere: Pervasive machine learning for effective high-performance computation, 02 2019.

[24] Nathaniel Kremer-Herman, Benjamin Tovar, and Douglas Thain. A lightweight model for right-sizing master-worker applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18, pages 39:1–39:13, Piscataway, NJ, USA, 2018. IEEE Press.

[25] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. Ernest: Efficient performance prediction for large-scale advanced analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 363–378, Santa Clara, CA, 2016. USENIX Association.

[26] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 469–482, Boston, MA, 2017. USENIX Association.

[27] Faria Kalim, Thomas Cooper, Huijun Wu, Yao Li, Ning Wang, Neng Lu, Maosong Fu, Xiaoyao Qian, Hao Luo, Da Cheng, et al. Caladrius: A performance modelling service for distributed stream processing systems. *Technical Report University of Illinois*, 2019.

[28] 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.

[29] 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.

[30] Andre Luckow, Mark Santcroos, Ashley Zebrowski, and Shantenu Jha. Pilot-Data: An Abstraction for Distributed Data. *Journal Parallel and Distributed Computing*, October 2014. http://dx.doi.org/10.1016/j.jpdc.2014.09.009.

[31] 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.

[32] Neil J. Gunther. Unification of amdahl's law, logp and other performance models for message-passing architectures. In *International Conference on Parallel and Distributed Computing Systems, PDCS 2005, November 14-16, 2005, Phoenix, AZ, USA*, pages 569–576, 2005.

[33] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[34] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, and Volker Markl. Benchmarking distributed stream processing engines. *IEEE 34th International Conference on Data Engineering (ICDE)*, 2018.

[35] J. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics*, pages 281–297, Berkeley, Calif., 1967. University of California Press.

[36] Scikit-Learn Community. Mini batch kmeans. https://scikit-learn.org/stable/modules/generated/sklearn.cluster.MiniBatchKMeans.html, 2019.

[37] Neil J. Gunther and Stefan Moeding. usl: Analyze system scalability with the universal scalability law. https://cran.r-project.org/web/packages/usl/index.html, 2017.