# **SPSC**: Stream Processing Framework atop Serverless Computing for Industrial Big Data

Zinuo Cai, Zebin Chen, Xinglei Chen, Ruhui Ma, *Member, IEEE,* Haibing Guan, Rajkumar Buyya, *Fellow, IEEE,*

***Abstract*—With the advance of smart manufacturing and information technologies, the volume of data to process is increasing accordingly. Current solutions for big data processing resort to distributed stream processing systems such as Apache Flink and Spark. However, such frameworks face challenges of resource underutilization and high latency in big data application scenarios. In this paper, we propose SPSC, a serverless-based stream computing framework where events are discretized into the atomic stream and stateless Lambda functions are taken as context-irrelevant operators, achieving task parallelism and inherent data parallelism in processing. Also, we implement a prototype of the framework on AWS (Amazon Web Service) using AWS Lambda, AWS SQS and AWS DynamoDB. The evaluation shows that compared with Alibaba's real-time computing Flink version, SPSC outperforms by 10.12% when the overhead is close.**

***Index Terms*—Cloud Computing, Intelligent Industry, Big Data, Serverless Computing, Stream Processing**

## I. Introduction

NOWADAYS, large amounts of data are generated to process as the development of information and communication technologies, such as the Internet of Things, industrial sensors, sensor networks, etc., which impacts manufacturing profoundly. With the technologies, data generated from modern manufacturing systems are experiencing explosive growth, which has reached over 100 EB annually [1]. The manufacturing data contains rich knowledge to utilize, driving the transformation of the conventional manufacturing paradigm to the intelligent manufacturing paradigm. Smart manufacturing utilizes the concepts of cyber-physical systems with the Internet of Things (IoT), cloud computing, service-oriented computing, artificial intelligence and data science [2], which would be the hallmark of the next industrial revolution.

To deal with big data processing tasks, researchers resort to distributed stream processing systems where data is generated as streams and processed by distributed and low-latency computational frameworks on a continuous basis. The common distributed data stream processing frameworks include open-source frameworks such as Storm, Spark Streaming, Flink and Kafka Streams and proprietary frameworks such as IBM Streams. To process the data streams with short delays and deal with the large volume of data, researchers have proposed combining cloud and edge computing with stream processing systems. Apache Flink and Spark are general-purpose streaming data processing frameworks.

However, existing methods are not suitable for the increasing amount of data. On the one hand, an abundance of computing infrastructure and resources remain underused with the increasing growth of the Internet of Things and edge computing. On the other hand, due to the latency issues and networking overhead, today's cloud models suffer from high latency and response time when processing large volumes and varieties of data. The responsibilities of managing underlying infrastructure optimally are heavy for developers, and the process is mainly manual, task-specific and error-prone [3].

Fortunately, we identify that serverless computing [4, 5, 6, 7, 8], an emerging cloud computing paradigm, can effectively solve the difficulties faced by cloud computing when applied to collecting and processing industrial big data. Serverless computing does not mean that there are no servers in the cloud. Still, the operations of servers, such as application and release, scaling up and down, are handled by cloud vendors other than users. Serverless computing is also known as function as a service (FaaS) because cloud computing users only need to write code and complete the running logic of the application. Cloud computing providers should meet other requirements for code operation, such as lightweight virtualization to execute functions, external databases to save the status of functions, and monitoring and log services. These backend services are the key to ensuring the safe and correct execution of stateless function instances. Therefore, serverless computing is featured as function-as-a-service (FaaS) for cloud computing users and backend-as-a-service (BaaS) for providers.

Introducing serverless computing to the processing of streaming industrial data solves the two challenges mentioned above. First of all, the resource utilization efficiency of the system can be effectively improved. Serverless computing can automatically scale up and down, rather than the traditional cloud computing model, which requires reserving sufficient computing resources to cope with the surge of industrial data. Moreover, because serverless computing takes functions as the smallest resource application unit, lightweight virtualization is more agile than virtualization schemes represented by virtual machines, reducing the execution delay of the stream processing system.

Therefore, we propose **SPSC**, a stream computing framework to handle industrial data atop serverless computing platforms in this paper. We conceptually divide the events of the processing process into several subsets and call them atoms. The lowest level of operation of the framework is the atomic level, and each computing unit is also designed to perform atomic processing. In other words, the framework workflow is a computational diagram of implementing transactional microservices on the atomic flow using data flow semantics. Therefore, Lambda functions become operators in stream computing, and users only need to pay attention to atomic-level transaction business logic when coding. Then, we use

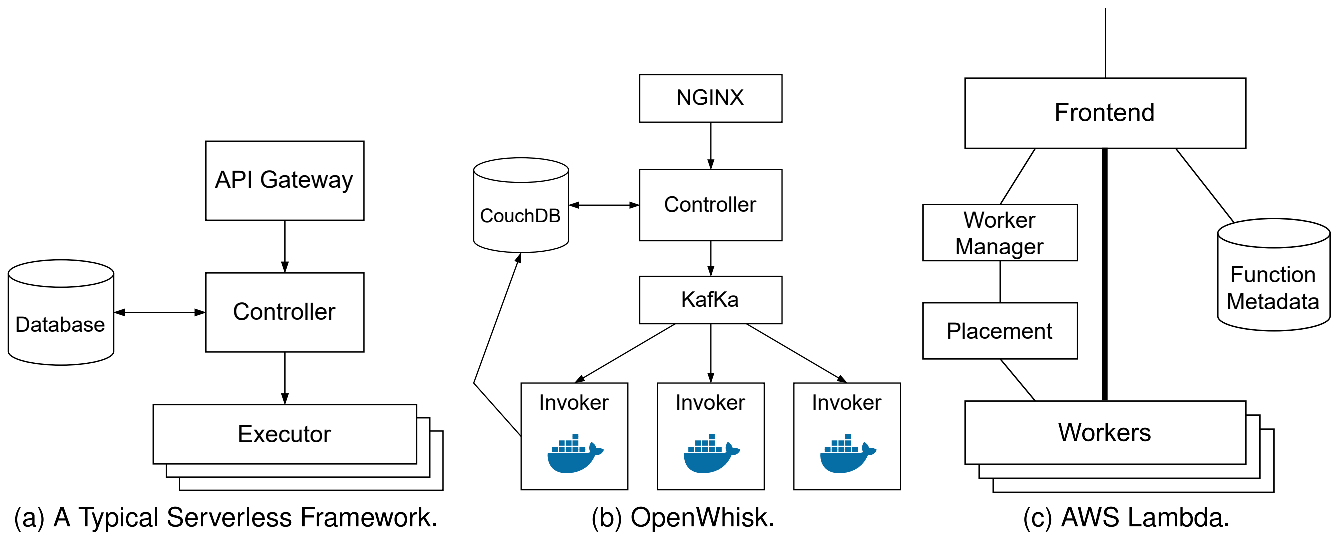


**Fig. 1: Architectures of Serverless Platforms.**

AWS SQS as a message queue to realize the communication between operators and the storage of intermediate states. The visibility of SQS ensures the At Least Once mechanism of the framework. We also use AWS DynamoDB as the framework's persistent storage solution for state storage so that users can quickly expand the database and do not need to design data relationships.

In this paper, we have made the following contributions:

- We propose a stream computing framework atop serverless architecture. Our approach combines the fundamental idea of the stateful data flow model with the serverless architecture, divides the event into atoms, and then uses the stateless Lambda function to realize the operator in the stream computing. We also use AWS SQS and AWS DynamoDB as message queues and persistent storage solutions.
- Based on the designed framework, we propose a parallel computing method. By discretizing the processing process into an atomic stream, operators are context-irrelevant, thus achieving task parallelism. The automatic adjustment of the concurrency of Lambda instances and the polling mechanism of message queues make data parallelism an inherent attribute of the framework.
- We have implemented a prototype of the framework and evaluated the performance of our implementation. According to our experimental results, compared with Alibaba's real-time computing Flink version, which is charged according to the lease duration, our framework can save 10.8% of the cost on average under the same computing tasks. Our framework can improve the performance by 10.12% on average when the overhead is close.

## II. Background & Motivation

### A. Serverless Computing

Serverless computing is a new paradigm of cloud computing. Its typical feature is function-as-a-service (FaaS), which is evolved from infrastructure-as-a-service (IaaS) [9] and platform-as-a-service (PaaS) [10]. Compared with the traditional cloud computing paradigm, serverless computing has the following advantages in the processing and collecting industrial big data. First, serverless computing provides more cost-efficient cloud services because of its "pay as you use" billing mode. Users only need to pay for how many cloud computing resources they use. On the contrary, in the platform-as-a-service mode, users need to charge for the occupied resources, even if the resources are not used. Second, serverless computing provides an excellent ability to automatically scale up and down, effectively responding to the change and frequency of industrial data volume. Because the generation of industrial data is a process of time series change, it is necessary to automatically increase or decrease the function instances according to the amount of data. Because serverless computing provides a lightweight application execution environment, container creation or destruction is faster than virtual machines. Third, serverless computing can reduce the time and experience of operation and maintenance personnel. Serverless computing is not featured as function-as-a-service but also as backend-as-a-service, including object storage, load balancing, resource scheduling and other backend services, to meet the needs of serverless computing.

Figure 1a shows the typical framework design of a serverless computing platform. Among the components of a serverless platform, Gateway is usually the platform's entrance responsible for receiving user requests. Controller is usually the core of the framework and undertakes the functions of user request processing, load balancing, function instance creation and destruction, etc. Executor is the execution environment of functions, usually lightweight containers or virtual machines, such as Docker, gVison or Firecracker [11]. Open-source serverless computing platforms include OpenWhisk [12], OpenFaas [13] and Fission [14]. Figure 1b shows the framework of OpenWhisk. Its Gateway uses the open-source reverse proxy server NGINX, and the Pod is composed of a Docker container in the Kubernetes cluster corresponding to the executor. Commercial serverless computing frameworks include Amazon's AWS Lambda, Google Cloud and Alibaba Cloud's Function Compute. The architecture of AWS Lambda is shown in Figure 1c. After the frontend receives the user's request, the worker manager will schedule and start the function instance. The function metadata will be sent to the corresponding worker for execution. Our subsequent work will be based on AWS Lambda.

### *B. Data Stream Processing*

Stream processing is the processing of data produced as a stream of events in motion. Unlike traditional batch processing, where static data is stored in a database, a file system, or other forms of mass storage and handled as needed, stream processing processes dynamic or continuous data as an event upon receiving one from the stream. A stream is an unbounded sequence of events generated continuously in time from the source to the sink. Stream processing pipelines often involve multiple operations such as filters, aggregations, analytics, transformations, enrichment, branching, joining, etc. As unbounded and global datasets are increasingly getting common and essential in day-to-day business [15], the majority of data are born as continuous streams such as sensor measurements, Weblogs, mobile usage statistics and financial trades. The stream processing market is experiencing exponential growth with applications relying heavily on real-time analytics, inferencing and monitoring, such as telecommunications, smart cities, health care, transportation, retail, manufacturing, advertising, cyber security, and finance.

Data processing systems are evolving to be more stream-oriented, where each data record is continuously processed as it arrives by distributed and low-latency computational frameworks. Currently, multiple distributed data stream processing frameworks, open source (Storm, Spark Streaming, Flink, Kafka Streams) and commercial (IBM Streams), exist for ingesting, processing, storing, indexing, and managing streaming data. Research on data stream processing engines has diverged into four directions: query-based systems such as NiagaraCQ [16], TelegraphCQ [17], and AsterixDB [18]; online distributed machine learning systems such as Scalable Advanced Massive Online Analysis (SAMOA) [19]; streaming graph analytics systems such as GraphJet [20]; general purpose streaming data processing frameworks such as Flink and Spark Streaming, with low-latency and a distributed parallel processing architecture. Apache Flink is an open-source distributed stream processing framework for stateful computations over unbounded and bounded data streams. Spark is a unified analytics engine for large-scale data processing supporting high-level APIs and general execution graphs.

Under several emerging application scenarios, such as operational monitoring of extensive infrastructure, IoT (Internet of Things) and smart cities, data streams must be processed under very short delays, and the data volume is enormous. These data stream processing frameworks have to be scalable and efficient. To meet these challenges, architecture has been proposed to use cloud computing to enable data stream processing as the resource elasticity and fault tolerance features of cloud computing. Here describe several public cloud solutions for processing streaming data. Amazon Kinesis Streams [21] is a service that enables continuous data intake and processing for several types of applications, such as data analytics and reporting, infrastructure log processing, and complex event processing. Google Cloud Dataflflow [22] is a programming model and managed service for developing and executing a variety of data processing patterns such as Extract, Transform, and Load (ETL) tasks, batch processing, and continuous computing. Azure Stream Analytics (ASA) enables real-time analysis of streaming data from several sources such as devices, sensors, websites, social media, applications, and infrastructures, among other sources [23].

### *C. Amazon Cloud Services*

Since serverless computing is featured as FaaS for cloud customers and BaaS for cloud vendors, we adopt the following three backend services in addition to AWS Lambda. We use Amazon S3 to store the raw data, AWS SQS as the communication channel between lambdas and AWS DynamoDB to persist the results.

**Amazon S3.** Amazon Simple Storage Service (Amazon S3) is an object storage service allowing users to store, protect and retrieve data from "buckets" at any time from anywhere. Amazon S3 focuses on two key components: buckets and objects that work together to create the storage system. Users create buckets to store objects in the cloud. Objects are data files, including documents, photos and videos, identified by a unique key each. The use cases of Amazon S3 include data lakes, mobile applications, IoT devices and big data analytics.

**AWS SQS.** AWS Simple Queue Service (SQS) is a managed service by AWS to handle message queueing, releasing developers from setting up and maintaining a queue system. AWS SQS is built on the broad mechanism of message queues and provides high-level APIs that developers can use to communicate with the service. SQS is frequently used to decouple distributed backend services or accommodate mismatches in service scalability.

**AWS DynamoDB.** DynamoDB is a NoSQL serverless database provided by AWS which follows a key-value store structure and adopts a distributed architecture for high availability and scalability. In DynamoDB, data is organized in tables containing items, and each item contains a set of key-value pairs of attributes. As in any serverless system, there's no infrastructure provisioning needed with DynamoDB.

## III. Design

This section gives a detailed overview of **SPSC**, a stream processing framework atop serverless computing platforms.

### *A. Overview*

The application mainly includes parallel workflow, communication and fault tolerance. Figure 2 shows the designed stream computing framework based on the services provided on AWS, which reduces the hardware requirements for users.

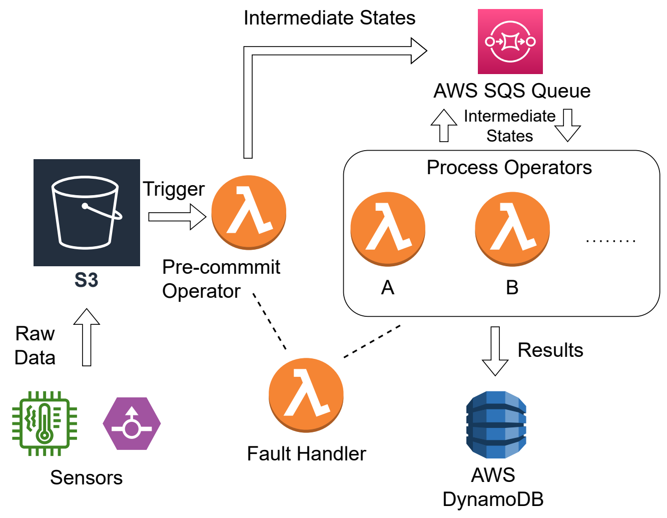


**Fig. 2: Overview of the Serverless stream computing framework.**

In the hypothetical industrial scenario, the intelligent sensors collect and upload the raw data to the low-cost persistent storage service S3 provided by AWS. Generally speaking, production data analysis in industrial scenarios does not require too high real-time, and the raw data is persistent and massive. Using S3 can save considerable storage costs. When the raw data is uploaded to S3, the Lambda instance of the pre-submit operator will be triggered to perform the initial processing of the data. Then the intermediate states of the data will be pushed into the queue of AWS SQS. The messaging mechanism of SQS itself can ensure that the intermediate state will only be processed by one Lambda instance, avoiding the additional cost caused by redundant processing. Similarly, the pushed message in the queue will trigger the start of the instance of the process operator. The process operator is the coding entry of the user program logic. The intermediate states between process operators are also transferred through SQS queues to ensure real-time and non-repetitive processing. The last process operator, which can be regarded as the exit of the user program, stores the results in the structured NoSQL database AWS DynamoDB, thus making the calculation results persistent. To deal with the unbounded input of stream computing, the framework needs to have the ability to deal with and recover from the failure of computing nodes. Hence, we design a Lambda function to handle failures in the computing procedure.

### B. Parallel Workflow

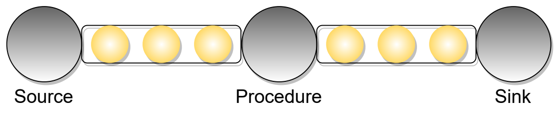
First of all, we need to declare that the framework abstracts the computing task into the concept of atomic stream. The level of processing is a low level. The data in the stream should be cut as small as possible to meet the concept of data elements. All kinds of operators in the framework operate based on the level of data elements, which are the lowest and indivisible processing level. In other words, data elements are atoms.

**Atomic stream.** Atomic stream is a relatively ordered, distributed and immutable atomic stream. Processors generate atomic streams, and consumers consume them. In order to ensure the indivisibility of atoms, we stipulate that consumer must complete all the work of consuming and processing the atom it holds before consuming and processing the next atom. Consumption must follow atomic order. Producers must produce atoms after consuming and processing atoms and before consuming and processing subsequent atoms. The resulting atomic production order will establish the atomic order. This enables us to safely combine the roles of producers and consumers. An operator can assume the roles of producers and consumers at the same time, reducing the difficulty of user coding while maintaining end-to-end fault-tolerance between micro-services.

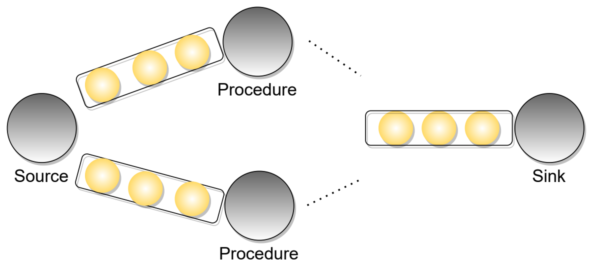
**Atom.** Conceptually, each atom contains a finite stream of events, and each event belongs to exactly one atom. Events and atoms are distinct with different purposes. Events represent input in the form of consumed and processed data, while atoms represent a group of events to be processed together in an atomic manner. States between instances of Serverless function cannot be simply shared, but the state is visible during the running of a single instance. We can combine a set of context-related event operations to form an operator. Events within the atom can be processed simultaneously. This difference is useful because starting the operation of an atom requires atomic-related costs (reflected in the generation of function instances in the Serverless architecture). Smaller atoms have higher relative costs. Users should determine the size of atoms according to their needs.

**Workflow.** Workflow (Figure 3) consumes atomic stream and generates atomic stream. Workflow is a directed acyclic graph (DAG) of source, procedures and links. Workflow has a source, which is the event intake point of workflow. Procedures are stateful operators that use atoms and generate atoms. Workflow has a sink point, which can be considered as the exit of workflow, that is, the exit of calculation results. The workflow needs to conform to the principles of atomic processing. That is, the workflow must always use and process one atom at a time, that is, it does not need to process two atoms at the same time. Nested workflows are not supported. Workflow has an input source and an output sink, just like the Source and Sink in other stream computing systems.

**Parallel Method.** To improve the throughput and efficiency of stream computing, common parallel methods are task parallelism and data parallelism. Task parallelism allows tasks from different operators to perform calculations on the same or different data in parallel to make better use of the computing resources of the cluster. Data parallelism performs the same operation in parallel on the data subset, allowing processing of large amounts of data and spreading the computing load to multiple computing nodes. In the Serverless architecture, the operators implemented by Lambda function and triggered by the specified data flow naturally has the task parallel attribute. By cutting the processing process into multiple context-independent processing operators, the framework realizes on-demand startup of operators, thus saving costs while ensuring throughput. The instantiation of lambda functions



(a) The simplest workflow.



(b) Slightly complex workflow.

**Fig. 3: Workflow examples in the framework. Figure 3a shows the simplest workflow structure. The gray nodes represent the user-defined processing operators, and the yellow nodes represent the atomic streams passed between operators. Figure 3b shows a slightly complex but DAG-compliant workflow. The processing process can be split according to the actual situation to generate multiple processing paths. However, it is recommended that only one Source and Sink exist for a workflow.**

will automatically obtain instance resources from the public reservation pool to perform computing tasks, and dynamically tilt computing resources for each function based on utilization. The feature of AWS SQS service that can only be seen by one person at the same time automatically manages the data subsets. The above features of AWS services enable the framework to automatically generate several operator instances to process atoms in parallel, while avoiding some common problems of data parallelism. That is to say, on the Serverless architecture, users can code highly reliable stream computing logic by applying the idea of task parallelism and data parallelism without having too much relevant knowledge.

## *C. State Storage*

The data processed by the stream processing system is often borderless: data will always be input from the data source, and users need to see the real-time results of SQL queries. At the same time, the computing nodes in the stream processing system may make errors and failures, and may expand and shrink in real time according to the user's needs. In this process, the system needs to be able to efficiently transfer the intermediate states of the calculation between nodes and persist the results to the external system, so as to ensure the uninterrupted calculation. Common state storage solutions, such as embedded storage, require the computing nodes to manage the state storage, which is obviously not applicable in the dynamically generated, stateless, and storage-and-calculation separation Serverless architecture. The stateless nature of Lambda functions makes the Serverless-based stream computing framework only adopt the architecture of storage and computing separation.

**Separation of Storage and Computing.** The storage responsibility and calculation responsibility of the system are separated. The storage node is only responsible for data storage, while the calculation node is only responsible for calculation, that is, to execute business logic. Such a design is called the separation of storage and computing. For the stateless computing instances generated by Lambda instantiation, each instance is the same and naturally supports horizontal expansion. The generated instances of the same type obtain the states by polling and then process them to easily achieve load balancing. Failover is also simpler and faster. If an instance fails, the computing task on it will be acquired by other instances. For developers, they can focus on the development of computing business logic without paying attention to such troublesome storage problems as data consistency, data reliability and data read and write performance, which greatly reduces the development difficulty and improves the development efficiency.

**Message Queue.** Stream computing systems usually do not need message queues, because they can communicate directly between functions, and end-to-end Exactly Once mechanism is also guaranteed in other ways. However, on the Serverless platform, there is no direct communication between instances, which also makes it difficult to implement the end-to-end Exactly Once mechanism. The message queue service provided by the service provider, such as AWS SQS, can ensure that each message pushed to the queue will be shared by only one object at the same time. After a period of time, if the object does not perform other operations on the message, the message will be released to other objects. Just set its visibility time to be longer than the instance lifetime in the framework, you can think that when a message is released to other objects, the previous instance has failed, thus realizing the failover. Therefore, in the design of the framework, we use AWS SQS as the message queue to complete the communication between operators and intermediate state storage.

**Persistent Storage.** The final results of stream computing, or some states that do not need immediate processing temporarily, need to be persisted to the external storage system. For example, if we want to count the production data in the past five minutes, and some of the earlier data arrive later than the later data due to network communication and other reasons, in this case, we can only store the states in the persistent storage database, and then sort it before processing. The disordered and temporarily stored message queue obviously does not support the above requirements. NoSQL, which does not need to design data relations in advance, is easy to expand and can be used on demand, is widely used in persistent storage systems of stream computing systems. AWS DynamoDB, a fully hosted cloud NoSQL database service provided by Amazon, can realize seamless expansion and automatically delete expired items from the table. We use it as a framework for persistent storage system.

### D. Fault Tolerance

The problem of unbounded input stream in the stream computing system has brought many new challenges to Fault Tolerance, such as low latency, Exactly Once mechanism, and so on. Many stream computing tasks are $7 \times 24$ hours without interruption, with end-to-end second delay. It is extremely difficult to quickly recover to normal in case of unexpected problems such as network flash, machine failure, and so on, without affecting the correctness of the calculation results. Moreover, the statelessness and separation of components of the Serverless architecture make the fault-tolerant design of this framework different from the traditional stream computing system.

**Timeout.** Unlike traditional stream computing system operators, which will exist for a long time once generated, Lambda instances have a limited and short survival time. To save the cost of repeatedly generating instances, the framework is designed to continuously process atoms once an operator instance is generated until atoms cannot be obtained or times out. Therefore, it can be considered that the instance running timeout will be a common exception in the running process. Considering the startup costs that can be saved, we think it is acceptable that a very small amount of data is delayed due to timeout exceptions. We set a timeout exception handling function. When the instance senses that it is about to timeout, it will throw a timeout exception and invoke the function. The exception handler invokes the operator function again according to the received event. In fact, because of the high substitutability of operator instances, even if the function is not restarted, other working instances will poll the processing atoms. However, restarting it makes the number of concurrent instances of the operator stable and can reduce the fluctuation of throughput as much as possible.

**At Least Once.** Due to the statelessness and non-direct communication of Lambda instance running, the snapshot recovery mechanism commonly used in streaming computing systems such as Flink is obviously difficult to implement. That is to say, it is very difficult to achieve Exactly Once on the Serverless architecture which is hard to perceive and maintain the running state. However, the design of storing intermediate states through message queues and serving as communication channels between operators makes it easy for the framework to implement At Least Once. AWS SQS guarantees that a message will only be owned by one object at the same time. When the visibility time setting is exceeded, if the message has not been processed by the owner, it will be returned to the queue and opened to other objects. By setting the visibility time slightly longer than the instance survival time, we can ensure that each atom will be processed at least once.

## IV. Evaluation

In order to verify the performance of the prototype of our framework, we compare its performance with that of the Serverless real-time computing platform launched by Alibaba Cloud in this section.

### A. Environment

We deploy the prototype of the framework on the AWS platform and use the services provided by the supplier. As designed in the framework, we created AWS SQS queues and AWS DynamoDB tables for the prototype. Then, we created S3 buckets to store randomly generated source data. Next, we deployed the prototype code on AWS Lambda and allocated a concurrent quota of 40 instances. Among them, in order to improve the throughput of reading data from slow S3 storage, 10 reserved concurrency is allocated for the pre-commit operator of the source. In other words, there are still 30 instances left in the public reservation pool.

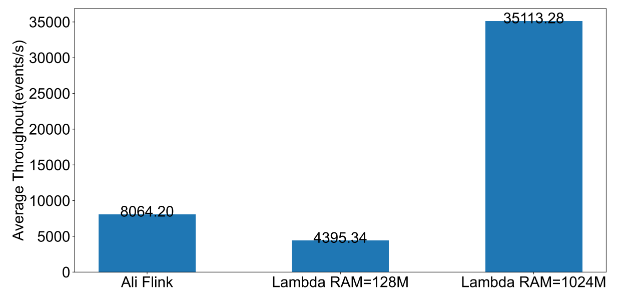
For the Flink version of Alibaba Cloud's real-time computing platform, we have rent it in a pay-as-you-go way. According to the official manual, 1 CU=1 core CPU+4 GB memory. CU corresponds to the CPU computing capacity of the underlying system. When creating a workspace, the system deploys a development console for each cluster. Each development console and its necessary components require about 2 CU of control resources. So we rented 6 CU, 2 for controlling resources and 4 for actual calculation. Other configurations, such as the datebase RDS, use the default configuration 4 RCU.

### B. Performance Metrics

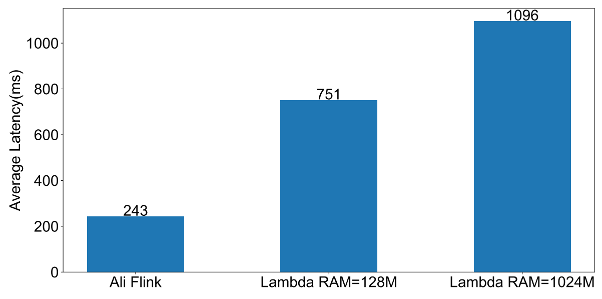
We have carried out the same experiments on the prototype and the leased Alibaba Cloud real-time computing platform, and the evaluation metrics of their performance are as follows:

**Throughout.** The number of events successfully transmitted by the computing framework in unit time. The unit of throughput in this experiment is: events/second. Throughput reflects the load capacity of the system and how much data the system can process per unit time under the corresponding resource conditions. Throughput is often used for resource planning, but also to help analyze system performance bottlenecks, so as to make corresponding resource adjustments to ensure that the system can meet the processing capacity required by users. Suppose that the merchant can make 20 lunches per hour (throughput: 20 lunches per hour), and a delivery boy can only deliver two lunches per hour (throughput: 2 lunches per hour). The bottleneck of this system is in the delivery of the boys, which can arrange ten delivery boys for the merchant.

**Latency.** The time of event from entering the system to exiting the system. The unit of latency in this experiment is: microsecond. Latency reflects the real-time processing of the system. A large number of real-time computing services such as financial transaction analysis have high requirements for latency. The lower the latency, the stronger the real-time of data. Suppose that it takes 5 minutes for the merchant to make a lunch and 25 minutes for the brother to deliver. In this



(a) Throughput of Alibaba Flink and prototype.



(b) Latency of Alibaba Flink and prototype.

**Fig. 4: The Average results obtained by Counting Several Random Windows.**

**TABLE I: Cost Calculation.**

| Scenario | Cost of Running for 1 hour(USD) | Cost without Writing to DB(USD) | Cost per million Events(USD) |
|---|---|---|---|
| Ali Flink | 35.588 | 1.028 | 0.037 |
| Lambda RAM=128M | 18.482 | 0.483 | 0.034 |
| Lambda RAM=1024M | 161.492 | 3.994 | 0.032 |

process, the user feels a latency of 30 minutes. If the latency becomes 60 minutes after changing the delivery plan, and the food is cold when it is delivered, the new plan is unacceptable.

**Cost.** General ledger of all cloud services leased. Since the cloud service leased in this experiment comes from two cloud service providers in different regions, and the currency charged is their local official currency, we have converted the two according to the exchange rate at the time of the experiment, which is about 6.7 yuan to 1 USD. The charging standard of Alibaba Cloud real-time computing platform is 0.133 USD/CU/hour, and the bill is calculated from the time the workspace is generated. It also includes the SLB service and database service it provides. For SLB, the unit price of each instance is 0.01 USD per hour. For databases, RDS is used by default, and its price is 0.055 USD/hour/RCU. And the price per million write request units is 1.20 USD. The billing of AWS cloud services includes the billing of Lambda, SQS and DynamoDB. For Lambda, when the memory is 128M, the price for each instance to run 1ms is 0.0000000021 USD. When the memory is expanded proportionally, the price can also be seen as increasing proportionally. For SQS, the first 1 million requests are free, and the price of 1 million to 100 billion requests is 0.40 USD per million requests. We believe that the number of demand requests for our framework should be within the latter range. For DynamoDB, the price per million write request units is 1.25 USD.

## *C. Performance*

In order to test the performance of the framework itself, we conducted input-output tests on both. Specifically, in the prototype, we set two operators, one is the pre-commit operator and the other is the processing and sink operator. The pre-commit operator randomly generates message and records the generated timestamp, and then pushes it into the SQS queue. The processing operator writes the processing time after receiving the message from the queue, which is actually the output time, and then writes it to the table. The same is true for Alibaba Cloud real-time computing platform, where a job that generates data and writes it to the database is published. Then, we randomly grab several five-minute windows and count the average throughput and average latency of the two during this period.

Figure 4 shows the throughput and latency of the experimental results. As can be seen from Figure 4a, the throughput of the prototype we built has no major defects compared with Ali Flink, and it can achieve a considerable increase in throughput with the increase of allocated RAM. Throughput does not increase linearly. It is speculated that communication with SQS limits its linear growth. Therefore, the relationship between communication queues and throughput can be studied in the future. Compared with Ali Flink, the latency of the prototype has increased significantly, which is understandable. The time cost of communicating with external cloud services such as SQS must be higher than the internal communication of main memory in Ali Flink. The comparison of performance between different configurations is not significant, but it proves that our framework can realize the functions of stream computing system.

In order to compare the advantages of our framework, we calculated the costs that customers usually care about most, as shown in Table I. A lot of overhead comes from reading and writing data to the database, because persistent storage is always expensive. However, generally speaking, the request to write records to the database is far lower than the request in the input-output experiment, so the cost calculation of database storage should be removed. From the perspective of processing costs per million businesses, our framework has a slight advantage over Ali Flink, that is, it can save 10.8 % of the cost on average when processing the same business volume. Although throughput and cost are not strictly linear, we can still estimate the approximate throughput of the prototype at the same cost by linear interpolation. According to our estimation, if Lambda's overhead cost is 1.028 USD, its estimated throughput is 8812 events/s. That is to say, our framework is expected to improve the performance by 10.12 % at the same cost.

## V. Related Work

*a) Serverless Computing and its Application:* Serverless computing [24, 25, 26, 27, 28] is a promising paradigm of next-generation cloud computing. It has been widely used in many fields thanks to its more lightweight virtualization runtime and faster startup and destruction time than virtual machines. In distributed machine learning, the parallel ability that serverless computing can provide can optimize the training speed of data parallelism or pipeline parallelism [29, 30, 31, 32, 33, 34]. LambdaML [32] is a general machine learning training platform atop serverless computing, and Jiang et al. conduct a comprehensive study on the different aspects, like communication channel, synchronization and cost efficiency. Siren [34] also employs AWS Lambda, the most representative serverless computing platform, for distributed model training and designs a reinforcement learning algorithm to guide the resource configuration for functions. In addition, serverless computing can also be applied to the deployment of the pre-training model [4, 5, 6, 7, 8] to deal with the inference task of changing requests. Because serverless computing has good scalability and an efficient startup rate, the operation and maintenance personnel do not need to consider the system's load balancing and concurrent processing when deploying model reasoning tasks. Infless [7] designs an automatic resource scheduling and configuration controller for a deep learning model serving atop OpenFaaS, an open-source serverless computing platform. Yang et al. propose a long-short-term-histogram (LSTH) algorithm to resolve the cold start challenge in serverless serving. In addition to the training and deployment tasks of machine learning, other high-performance computing tasks can also benefit from serverless computing platforms, such as video processing [35, 36, 37, 38], high-dimensional matrix operations [39, 40, 41], workflow tasks [42, 43, 44, 45], etc.

*b) Serverless Computing for Data Processing:* [46] introduces a serverless architecture for big data analysis. As the size of data is increasing day by day, it is tough and complex to design the exact architecture for data analysis, including server management, storage, clustering, algorithm deployment, etc. The misconfiguration would lead to the underuse of resources and infrastructure and unnecessarily high costs. With the challenges of scalability and efficiency that big data processing systems face, researchers have turned to serverless (Fuction-as-a-Service) to strengthen big data analysis. Stefan [3] proposed a serverless real-time data analytics platform for edge computing. As the user-defined functions are seamlessly and transparently hosted and managed by the serverless platform, it releases the developers of the burden to resort to optimal management of underlying infrastructure on the edge side. Mijanur [46] presented the serverless architecture for big data analytics with a serverless big data application on AWS (Amazon Web Service). With the serverless paradigm, developers can concentrate on the implementation of data analytics applications rather than underlying infrastructure and pay only for constant execution rather than particular server components. Portals [47] is a serverless, distributed programming model that blends the exactly-once processing guarantees of stateful dataflow streaming frameworks with the message-driven compositionality of actor frameworks. With Portals, the decentralized application can be built dynamically and scale on demand with the guarantee of strict atomic processing.

## VI. Conclusion

In this paper, we proposed **SPSC**, a stream computing framework built on serverless architecture to cope with industrial data. By dividing and abstracting the events into atoms and atomic streams, **SPSC** realizes task and data parallelism and achieves high throughput and efficiency of stream computing. Combined with AWS components such as AWS SQS and AWS DynamoDB, **SPSC** achieves abilities of At Least Once guarantee and persistent storage for stream computing applications in the serverless computing environment. Through extensive evaluation, we show that **SPSC** exceeds Alibaba's real-time computing Flink version by 10.12% in performance.

## Acknowledgment

This work was supported in part by Pilotage Program–Special Subject on Internet of Things, Shanghai Key Laboratory of Scalable Computing and Systems, China Institute of IOT (Wuxi), Wuxi IoT Innovation Promotion Center, and Open Project from Jiangsu Key Laboratory of Media Design and Software Technology (Jiangnan University) under Grant 21ST0101. Ruhui Ma (ruhuima@sjtu.edu.cn) is the corresponding author.

## References

[1] W. Li, Y. Liang, and S. Wang, *Data driven smart manufacturing technologies and applications*. Springer, 2021.

[2] A. Kusiak, "Smart manufacturing," *International Journal of Production Research*, vol. 56, no. 1-2, pp. 508–517, 2018.

[3] S. Nastic, T. Rausch, O. Scekic, S. Dustdar, M. Gusev, B. Koteska, M. Kostoska, B. Jakimovski, S. Ristov, and R. Prodan, "A serverless real-time data analytics platform for edge computing," *IEEE Internet Computing*, vol. 21, no. 4, pp. 64–71, 2017.

[4] A. Ali, R. Pinciroli, F. Yan, and E. Smirni, "Optimizing inference serving on serverless platforms," *Proceedings of the VLDB Endowment*, vol. 15, no. 10, pp. 2071–2084, 2022.

[5] K. Mahajan and R. Desai, "Serving distributed inference deep learning models in serverless computing," in *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*. IEEE, 2022, pp. 109–111.

[6] J. Li, L. Zhao, Y. Yang, K. Zhan, and K. Li, “Tetris: Memory-efficient serverless inference through tensor sharing,” in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022.

[7] Y. Yang, L. Zhao, Y. Li, H. Zhang, J. Li, M. Zhao, X. Chen, and K. Li, “Infless: a native serverless system for low-latency, high-throughput inference,” in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 768–781.

[8] T. P. Bac, M. N. Tran, and Y. Kim, “Serverless computing approach for deploying machine learning applications in edge layer,” in *2022 International Conference on Information Networking (ICOIN)*. IEEE, 2022, pp. 396–401.

[9] S. S. Manvi and G. K. Shyam, “Resource management for infrastructure as a service (iaas) in cloud computing: A survey,” *Journal of network and computer applications*, vol. 41, pp. 424–440, 2014.

[10] R. Yasrab, “Platform-as-a-service (paas): the next hype of cloud computing,” *arXiv preprint arXiv:1804.10811*, 2018.

[11] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, “Firecracker: Lightweight virtualization for serverless applications.” in *NSDI*, vol. 20, 2020, pp. 419–434.

[12] “Openwhisk,” https://openwhisk.apache.org/, accessed Jan, 2023.

[13] “Openfaas,” https://www.openfaas.com/, accessed Jan, 2023.

[14] “Fission,” https://fission.io/, accessed Jan, 2023.

[15] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt *et al.*, “The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing,” 2015.

[16] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang, “Niagaracq: A scalable continuous query system for internet databases,” in *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, 2000, pp. 379–390.

[17] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah, “Telegraphcq: continuous dataflow processing,” in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, 2003, pp. 668–668.

[18] S. Alsubaiee, Y. Altowim, H. Altwaijry, A. Behm, V. Borkar, Y. Bu, M. Carey, I. Cetindil, M. Cheelangi, K. Faraaz *et al.*, “Asterixdb: A scalable, open source bdms,” *arXiv preprint arXiv:1407.0454*, 2014.

[19] G. D. F. Morales and A. Bifet, “Samoa: scalable advanced massive online analysis.” *J. Mach. Learn. Res.*, vol. 16, no. 1, pp. 149–153, 2015.

[20] A. Sharma, J. Jiang, P. Bommannavar, B. Larson, and J. Lin, “Graphjet: Real-time content recommendations at twitter,” *Proceedings of the VLDB Endowment*, vol. 9, no. 13, pp. 1281–1292, 2016.

[21] “Amazon kinesis streams,” https://aws.amazon.com/cn/kinesis/data-streams/, accessed Jan, 2023.

[22] “Google cloud dataflow,” https://cloud.google.com/dataflow/, accessed Jan, 2023.

[23] “Azure stream analytics,” https://azure.microsoft.com/en■us/services/stream-analytics/, accessed Jan, 2023.

[24] Z. Li, L. Guo, J. Cheng, Q. Chen, B. He, and M. Guo, “The serverless computing survey: A technical primer for design architecture,” *ACM Computing Surveys (CSUR)*, vol. 54, no. 10s, pp. 1–34, 2022.

[25] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski *et al.*, “Serverless computing: Current trends and open problems,” *Research advances in cloud computing*, pp. 1–20, 2017.

[26] H. Shafiei, A. Khonsari, and P. Mousavi, “Serverless computing: a survey of opportunities, challenges, and applications,” *ACM Computing Surveys*, vol. 54, no. 11s, pp. 1–32, 2022.

[27] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar *et al.*, “Cloud programming simplified: A berkeley view on serverless computing,” *arXiv preprint arXiv:1902.03383*, 2019.

[28] J. M. Hellerstein, J. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu, “Serverless computing: One step forward, two steps back,” *arXiv preprint arXiv:1812.03651*, 2018.

[29] Y. Liu, B. Jiang, T. Guo, Z. Huang, W. Ma, X. Wang, and C. Zhou, “Funcpipe: A pipelined serverless framework for fast and cost-efficient training of deep learning models,” *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 6, no. 3, pp. 1–30, 2022.

[30] P. G. Sarroca and M. Sánchez-Artigas, “Mlless: Achieving cost efficiency in serverless machine learning training,” *arXiv preprint arXiv:2206.05786*, 2022.

[31] P. Gimeno Sarroca and M. Sánchez-Artigas, “Mlless: Achieving cost efficiency in serverless machine learning training,” *arXiv e-prints*, pp. arXiv–2206, 2022.

[32] J. Jiang, S. Gan, Y. Liu, F. Wang, G. Alonso, A. Klimovic, A. Singla, W. Wu, and C. Zhang, “Towards demystifying serverless machine learning training,” in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 857–871.

[33] F. Xu, Y. Qin, L. Chen, Z. Zhou, and F. Liu, “$\lambda$dnn: Achieving predictable distributed dnn training with serverless architectures,” *IEEE Transactions on Computers*, vol. 71, no. 2, pp. 450–463, 2021.

[34] H. Wang, D. Niu, and B. Li, “Distributed machine learning with a serverless architecture,” in *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 2019, pp. 1288–1296.

[35] L. Ao, L. Izhikevich, G. M. Voelker, and G. Porter, “Sprocket: A serverless video processing framework,” in *Proceedings of the ACM Symposium on Cloud Computing*, 2018, pp. 263–274.

[36] M. Zhang, Y. Zhu, C. Zhang, and J. Liu, "Video processing with serverless computing: A measurement study," in *Proceedings of the 29th ACM workshop on network and operating systems support for digital audio and video*, 2019, pp. 61–66.

[37] F. Romero, M. Zhao, N. J. Yadwadkar, and C. Kozyrakis, "Llama: A heterogeneous & serverless framework for auto-tuning video analytics pipelines," in *Proceedings of the ACM Symposium on Cloud Computing*, 2021, pp. 1–17.

[38] M. Zhang, F. Wang, Y. Zhu, J. Liu, and B. Li, "Serverless empowered video analytics for ubiquitous networked cameras," *IEEE Network*, vol. 35, no. 6, pp. 186–193, 2021.

[39] S. Werner, J. Kuhlenkamp, M. Klems, J. Müller, and S. Tai, "Serverless big data processing using matrix multiplication as example," in *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, 2018, pp. 358–365.

[40] V. Shankar, K. Krauth, K. Vodrahalli, Q. Pu, B. Recht, I. Stoica, J. Ragan-Kelley, E. Jonas, and S. Venkataraman, "Serverless linear algebra," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, 2020, pp. 281–295.

[41] V. Gupta, S. Kadhe, T. Courtade, M. W. Mahoney, and K. Ramchandran, "Oversketched newton: Fast convex optimization for serverless systems," in *2020 IEEE International Conference on Big Data (Big Data)*. IEEE, 2020, pp. 288–297.

[42] B. Carver, J. Zhang, A. Wang, A. Anwar, P. Wu, and Y. Cheng, "Wukong: A scalable and locality-enhanced framework for serverless parallel computing," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, 2020, pp. 1–15.

[43] A. Mahgoub, E. B. Yi, K. Shankar, E. Minocha, S. Elnikety, S. Bagchi, and S. Chaterji, "Wisefuse: Workload characterization and dag transformation for serverless workflows," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 6, no. 2, pp. 1–28, 2022.

[44] A. Mahgoub, K. Shankar, S. Mitra, A. Klimovic, S. Chaterji, and S. Bagchi, "Sonic: Application-aware data passing for chained serverless applications," in *USENIX Annual Technical Conference (USENIX ATC)*, 2021.

[45] A. Mahgoub, E. B. Yi, K. Shankar, S. Elnikety, S. Chaterji, and S. Bagchi, "Orion and the three rights: Sizing, bundling, and prewarming for serverless dags," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022, pp. 303–320.

[46] M. M. Rahman and M. H. Hasan, "Serverless architecture for big data analytics," in *2019 Global Conference for Advancement in Technology (GCAT)*. IEEE, 2019, pp. 1–5.

[47] J. Spenger, P. Carbone, and P. Haller, "Portals: An extension of dataflow streaming for stateful serverless," in *Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, 2022, pp. 153–171.

**Zinuo Cai** is currently a graduate student in Computer Science at Shanghai Jiao Tong University, China. He obtained the bachelor's degree in Software Engineering at Shanghai Jiao Tong University. His research interests are focused on resource schedule and system security in cloud computing.

**Zebin Chen** is currently a graduate student in the Department of Computer Science and Engineering at Shanghai Jiao Tong University, China. His research interests center around the applications of distributed systems.

**Xinglei Chen** is currently a graduate student in Computer Science at Shanghai Jiao Tong University, China. His research interests are focused on system security in cloud computing.

**Ruhui Ma** is currently an associate professor in the Department of Computer Science and Engineering at Shanghai Jiao Tong University. He received his Ph.D. degree in computer science from Shanghai Jiao Tong University. His research interests include cloud computing systems, AI systems, and machine learning.

**Haibing Guan** is currently a professor in the School of Electronic Information and Electronic Engineering, Shanghai Jiao Tong University, and the director of the Shanghai Key Laboratory of Scalable Computing and Systems. He received his Ph.D. degree from Tongji University in 1999. His research interests include cloud/distributed computing and machine learning.

**Rajkumar Buyya** (Fellow, IEEE) is a redmond barry distinguished professor and director of the Cloud Computing and Distributed Systems (CLOUDS) Laboratory, University of Melbourne, Australia. He has authored more than 625 publications and seven text books. He is one of the highly cited authors in computer science and software engineering worldwide (h-index=153, g-index=324, more than 121,200 citations). Microsoft Academic Search Index ranked him as #1 author in the world (2005-2016) for both field rating and citations evaluations in the area of distributed and parallel computing. He is recognized as a "Web of Science Highly Cited Researcher" during 2016-2021 by Thomson Reuters.