Operator as a Service: Stateful Serverless Complex Event Processing

Manisha Luthra*, Sebastian Hennig*, Kamran Razavi*
*Technical University of Darmstadt, Germany
{firstname.lastname}@kom.tu-darmstadt.de
razavi@tk.tu-darmstadt.de

Lin Wang*†, Boris Koldehofe*‡
†Vrije Universiteit Amsterdam, Netherlands
lin.wang@vu.nl
‡University of Groningen, Netherlands
b.koldehofe@rug.nl

***Abstract*—Complex Event Processing (CEP) is a powerful paradigm for scalable data management that is employed in many real-world scenarios such as detecting credit card fraud in banks. The so-called *complex events* are expressed using a *specification language* that is typically implemented and executed on a specific *runtime system*. While the tight coupling of these two components has been regarded as the key for supporting CEP at high performance, such dependencies pose several inherent challenges as follows. (1) Application development atop a CEP system requires extensive knowledge of how the runtime system operates, which is typically highly complex in nature. (2) The specification language dependence requires the need of domain experts and further restricts and steepens the learning curve for application developers.**

In this paper, we propose CEPLESS, a scalable data management system that decouples the specification from the runtime system by building on the principles of *serverless computing*. CEPLESS provides "operator as a service" and offers flexibility by enabling the development of CEP application in *any* specification language while abstracting away the complexity of the CEP runtime system. As part of CEPLESS, we designed and evaluated novel mechanisms for *in-memory processing* and *batching* that enable the stateful processing of CEP operators even under high rates of ingested events. Our evaluation demonstrates that CEPLESS can be easily integrated into existing CEP systems like Apache Flink while attaining similar throughput under high scale of events (up to 100K events per second) and dynamic operator update in ˜238 ms.

***Index Terms*—Complex Event Processing; Serverless computing; Function as a Service; Internet of Things**

I. Introduction

Complex event processing (CEP) is a data management paradigm used in a wide range of applications to efficiently detect interesting event patterns in event streams. Such event patterns often named *complex events*, allow applications upon their detection to adapt to situational changes, such as the detection of fraud in credit card payments [2] and deriving tweet trends in Twitter [20]. The strengths of CEP reside in the simple specification of complex events by means of a query language and the support for efficient and distributed execution of the event detection logic.

Therefore, almost every CEP system provides two key components: (i) the *specification language* used to define event patterns and (ii) *the runtime system* to execute the event detection logic. Typically, these two components are highly intertwined. The constructs that describe the event detection logic are mapped to specific, at times infrastructure dependent, operator implementations, e.g., the algorithms for detecting sequences of events in a time-based window. Driven by preferences of programmers and the underlying systems infrastructure, many distinct CEP systems have been proposed [20], [8], [3], [11], offering each very specific features having specific programming models and infrastructures in mind. For example, classic CEP programming models such as CQL [6] and SASE [28] are based on SQL-like semantics, and hence, they also share many limitations of SQL. In particular, it is very difficult to express complex business logic using these programming models. Rather in current practices such as Google Dataflow [4], Millwheel [3], and Flink [8], object-oriented languages are often used to express complex business logic in the form of user-defined functions (UDFs). With UDFs an operator can encapsulate any business logic and hence can be customized as per user needs.

While the expressiveness is significantly improved with UDFs, existing CEP system still fall short in several aspects. One of them is the lack of runtime independence. Although within each CEP system, queries can be specified in a highly composable manner or even altered, there is little support to benefit from reusing development effort from one CEP-application to another. Simply, rewriting the query specification from one system to another is difficult since the way CEP queries are written has specific execution semantics in mind which are known to diverge between multiple systems [10]. Furthermore, the support for dynamic updates to the UDFs is weak. Thus, changes to the implementation of specific operators or the definition of new functionality often require a restart and new deployment of the operators [8], which is problematic in many applications like fraud detection where system availability is critically important.

In this paper, we aim for a better understanding of how one can benefit from the diversity of different CEP systems and enhance their applicability to a wide spectrum of different infrastructures. By proposing a data management system that builds on the principles of serverless computing architecture [15], we aim to enhance the reuse and integration of CEP operators. Furthermore, by proposing methods for adding new functionalities and altering the operator logic at run-time, CEP systems can adapt their processing logic dependent on the application context as well as the features

of the underlying infrastructure. This way, the diversity of operator implementations becomes no longer an obstacle in the development cycle, but a feature that allows CEP systems to evolve to new requirements and adapt to contextual changes at run-time.

While existing serverless computing platforms [26], [12] provide important concepts for the scalable execution of operator implementations, the extension of CEP systems to a serverless platform imposes many challenges. First, CEP operators often perform stateful processing such as detection of correlated events within a time-based window of the data stream [10]. Stateful processing is not easily possible in a serverless model because conventionally each function or operator execution is required to be isolated and ephemeral in current platforms. For example in AWS Lambda, the functions have a limited lifetime of up to 15 minutes, which is the maximum among several serverless platforms. For the functions or operators, it is assumed that the state is not recoverable across invocations. This stays in contradiction to the lifetime of a CEP operator which is required to be running as long as the query is executed [14]. Second, the execution mechanisms in current CEP systems perform many optimizations such as flow control, backpressure, and in-memory network buffers to guarantee low latency and high throughput. Achieving equal performance in an existing serverless platform is currently not possible especially due to missing the aforementioned optimizations and slow communication through storage. For example, in AWS Lambda [26] it is only possible to communicate between two lambdas using S3 that is extremely slow and more expensive than point-to-point networking [17].

Contributions: To overcome these flexibility limitations of current systems, we propose a new CEP system named CEPLESS based on the *function as a service (FaaS)* concept of *serverless computing* [26]. In this system, we provide CEP *operators as a service* which means that the operator thus can be specified independently of the underlying runtime system without any side-effects to the other system dependent operators running in the CEP system. The approach enables *specification language independence* from the CEP runtime system for the specified operators, which essentially means that the operators can be developed in *any* programming language desired by the developer. Furthermore, we contribute to CEPLESS with two mechanisms (i) supporting the stateful processing of events, which poses a challenge in existing serverless platforms and (ii) meeting the latency and throughput requirements of the CEP applications. CEPLESS addresses these challenges through *in-memory queue management*, which maintains the state in an in-memory queue for stateful operators like time-based windows. Furthermore, the *batching mechanism* aids in providing high throughput and low latency in event processing. Overall, CEPLESS introduces a high degree of flexibility for CEP systems with the involvement of FaaS in conjunction with our advanced design. In summary, this paper makes the following contributions:

1) We propose mechanisms for *in-memory queue management* and *batching*, enabling stateful processing and ensure correctness and fast delivery of events which are extremely important for CEP systems.
2) We define a unified *programming interface* that enables the specification of *novel* user-defined operators which are independent of the runtime system and the specification language of the runtime. Most importantly, using this interface, the operators can be specified in *any* existing programming language and by a single click, operators can be executed and deployed on any infrastructure.
3) We introduce a simple *user-defined operator interface* that allows the integration of highly diverse CEP runtime systems into CEPLESS system and benefits from them.
4) We implement and evaluate CEPLESS on two state-of-the-art CEP systems *Apache Flink* [8] and *CEP-P* [21] using an open and anonymous credit card transaction dataset [22]. Results show that CEPLESS enables runtime system independent updates of user-defined operators while attaining equal throughput and preserving low latency overhead (˜1.9 ms) under high scale of event rates of up to 100K events per second.

The rest of the paper is structured as follows. Section II presents a motivational example that better explains the challenges. Section III presents the system model introducing the important system entities. Section IV describes CEPLESS system design. Section V shows the evaluation. Section VI discusses the related approaches and Section VII concludes the paper.

II. Problem Statement Using Fraud Detection Example

A financial institution wants to detect payment frauds in the real-time credit card transactions of its customers. The fraud detection algorithm includes complex business logic such as machine learning models. Typically, this algorithm is required to be dynamically updated to incorporate newly observed transaction patterns of fraud. The department has its proprietary machine learning library that is implemented in a highly efficient and scalable language such as Rust, or a proprietary language developed within the department. The fraud department uses a CEP system for detecting fraud in sub-second latency and for a high number of users, resulting in a high scale of incoming transactions to be processed. Current CEP systems, however, fail to fulfill the above requirements of expressing a complex business logic as an operator without any prior knowledge of the CEP runtime system, while using the language preference of the department for the specification of complex events.

To better understand the requirements, consider an example specification of fraud detection in a widely used CEP system Apache Flink [8] (cf. Figure 1(a)). Here, the `highlighted` code refers to the specification of the operator or business logic that uses a pre-trained machine learning model to detect fraud. In the example, we have kept the business logic rather simple

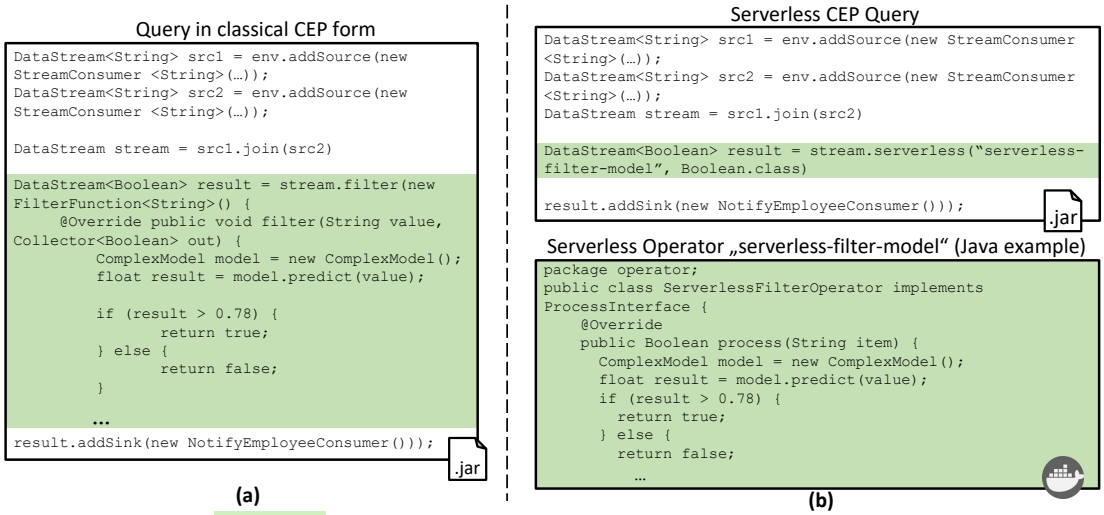

Fig. 1: An example of a simple `filter` application for fraud detection (a) in traditional CEP systems and (b) in the CEPLESS system. Here, the `serverless-filter-model` in (b) is decoupled from the CEP runtime and can be updated on the fly.

for better understandability. We can see that the specification is tightly coupled to the runtime APIs like `DataStream` and `FilterFunction`. Furthermore, the query operator `filter` is ossified and cannot be updated after deployment. Besides, the language of preference cannot be used due to missing APIs. This dependency on a specific runtime system can be problematic because of several reasons: (1) CEP systems are extremely complex by nature and the respective department for the development of fraud detection operators in a financial institution may not have the expertise to deal with such complex systems. (2) Fraud detection might require extending the existing runtime with external modules, e.g., machine learning libraries which can be very cumbersome in current CEP systems, if not impossible. (3) Different financial institutions might have individual specification language preferences for fraud detection operators. This is also not easily possible without extending the complete CEP system for a different specification language or rewrite of the operators.

On the right side Figure 1 (b) is our proposal, which segregates the business logic of the operator `serverless-filter-model` from the CEP runtime by implementing the operator as a serverless function containerized in a virtualization environment. Hence, the independent `ServerlessFilterOperator` can be reused for different CEP runtime environment. This segregation leads to several research challenges that we address in this work.

1) How to allow dynamic updates of an operator, while guaranteeing high performance in the delivery of events?
2) How to design user-defined operators, such as one for fraud detection, independent from the underlying CEP runtime, and the programming language?

III. CEPLESS MODEL

CEPLESS is able to consume continuous data streams (D) from a set of event producers (P) such as the Internet of Things devices. A set of event consumers (C) express interest in inferring a complex event such as *fraud detection* in the form of a query q. A query q induces a directed acyclic *operator graph* $G = (\Omega \cup P \cup C, D)$, where a vertex represents an operator $\omega \in \Omega$ and an edge represents the flow of events based on data streams, s.t., $D \subseteq (P \cup \Omega) \times (C \cup \Omega)$. Each operator ω dictates a processing logic f_ω.

CEPLESS provides a set of operators $\Omega = \{\Omega_S, \Omega_{UD}\}$ where an operator can be either a *system-defined operator* $\omega_S \in \Omega_S$ or a *user-defined operator* $\omega_{UD} \in \Omega_{UD}$. We define them as follows.

Definition III.1. *System-defined operators* (Ω_S) is a set of standard CEP operators s.t. $\Omega_S \subseteq \Omega$ and $\omega_S \in \Omega_S$. Conventional CEP operators comprising of single-item such as *selection*, logical operators such as *conjunction*, window operators such as *sliding window*, and flow-management operators such as *join* [10].

Definition III.2. *User-defined operators* (Ω_{UD}) is a set of user-defined operators s.t. $\Omega_{UD} \subseteq \Omega$ and $\omega_{UD} \in \Omega_{UD}$. Unlike system-defined operators, it contains custom logic or user-defined code that typically cannot be expressed by system-defined operators. The flow of information from ω_{UD} is encapsulated in a *container*, which is detailed in the next section.

In a distributed setting, the operators are typically placed on a set N of nodes that are responsible to process the incoming data streams. The nodes hosting the ω_{UD} operator is managed by a *node manager* NM_n that handles all ω_{UD} operator incoming requests.

In Figure 2, we illustrate an overview of this CEPLESS system model comprising of three layers: the serverless layer, the execution layer, and the devices layer. The devices layer is composed of primary devices such as the Internet of Things, which generates continuous data streams that are to be processed. The execution layer comprises of a variety of current CEP systems, which processes the data streams to derive a complex event. Lastly, the serverless layer provides system for defining user-defined operators and the flexibility of multiple runtime environments. The operator graph mapping

Notation	Meaning
P	Set of event producers
C	Set of event consumers
D	Continuous data stream
G	Operator graph
Ω	Set of CEP operators ($\omega \in \Omega$)
Ω_S	Set of system-defined operators ($\omega_S \in \Omega_S$)
Ω_{UD}	Set of user-defined operators ($\omega_{UD} \in \Omega_{UD}$)
N	Set of nodes ($n \in N$)
NM_n	Node manager for each node n
E	Set of events ($e \in E$)
f_ω	Processing logic of an operator
C_ω	Operator container
$eq_{\omega_{UD}}$	Event queue for each ω_{UD}
b_s and b_r	Sent and received batch of events
$t_{backoff}$	Queue polling backoff time

TABLE I: Notations and their meaning.

to the layers in the figure comprises standard ω_S operators which are system-defined such as a stream operator (ω_{str}) comprising continuous time-series event tuples in the form of $< E >$. Here, E is a set of time-series event tuples $\{< ts_1, e_1 >, < ts_2, e_2 >, .. < ts_n, e_n >\}$. In the CEPLESS system, multiple operator graphs can co-exist with different CEP systems in execution at the same time. For example, an operator graph of CEP-P (with dotted lines) can co-exist with that of Flink (solid line) as represented in the figure.

A. Container Model

Operators containers provide the ability to encapsulate any processing logic f_ω to be used in CEP systems as a user-defined operator ω_{UD}.

Definition III.3. The *operator container* (C_ω) serves as an interface for interacting with ω_{UD} in order to have a unified interaction between all operators and user applications. Events received by the CEP system in the execution layer are forwarded through the container interface into ω_{UD} which provides an event entry point for f_ω.

While a user-defined operator continuously processes events, it can emit single or multiple events that are forwarded through the aforementioned interface back to the CEP system for further processing.

B. Queue Model

In order to provide a clean abstraction for ω_{UD} and enable flexibility of defining any processing logic f_ω as an operator, a messaging system needs to be established. To achieve this, we enable interactions between ω_{UD} and CEP systems using an *event queue* as an event exchange point. We consider different aspects relevant for processing a continuous event stream such as (i) statefulness, (ii) performance, and (iii) in-order processing. There are no restrictions in place regarding the size of the queue as it is dependent on the number of event items processed.

Definition III.4. An *event queue* ($eq_{\omega_{UD}}$) is an instance in a queuing system distributed alongside with the CEP system on every node that participates in the cluster. Each ω_{UD} maintains two queues, namely *Input* ($eq\_in_{\omega_{UD}}$) and *Output* ($eq\_out_{\omega_{UD}}$). The former contains events received by the CEP system's operators ω_S which are to be forwarded for processing to ω_{UD}. The latter contains the results of ω_{UD} execution, which are processed by the CEP system for the next operator in the operator graph. User-defined operators and CEP systems only communicate with the queue placed on their respective node. Therefore, the queue can be accessed by user-defined operators and the CEP system through the local networking interface on the host which provides a negligible overhead in communication.

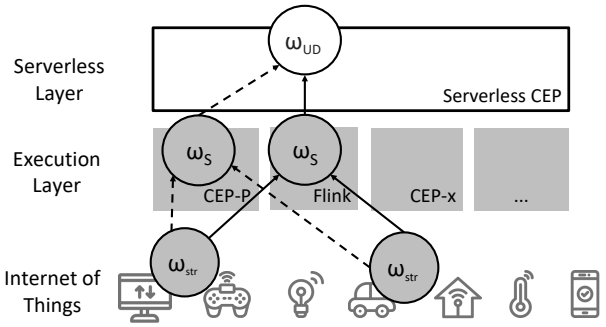

Fig. 2: The System model of CEPLESS. *The grey layers are shared.*

Definition III.5. An *event batch* (b_s and b_r) comprises of a subset of event tuples from E that are to be sent to (b_s) and from (b_r) the CEPLESS system through the event queues and the execution layer.

Definition III.6. The *back-off interval* ($t_{backoff}$) is the time to backoff from polling the event queue $eq_{\omega_{UD}}$ when no events are received. It is incremented linearly at each polling step. This effectively reduces the execution overhead when the queues are empty.

Definition III.7. *User-defined operator (UDO) interface.* Queues interact with the system layer using a UDO interface. We consider this entity implemented in each CEP system enabling communication with ω_{UD} over event queues $eq_{\omega_{UD}}$. Each CEP system that wants to use CEPLESS is required to have an implementation of the UDO interface.

To support multiple ω_{UD} on a single machine, each ω_{UD} gets assigned two distinct event queues $eq_{\omega_{UD}}$ by CEPLESS as defined in Definition III.4. Incoming events in the first queue get processed by the ω_{UD} in a sequence of their arrival. The result of the processed events gets added to the second queue which is processed by the CEP system again.

IV. THE CEPLESS SYSTEM DESIGN

Figure 3 shows an overview of the CEPLESS design on a single node[1]. On the left side, we show the CEPLESS system components, which allow efficient integration of multiple CEP runtimes and their reuse while ensuring runtime updates of ω_{UD} operator. We show developer-centric components on the right side, which allows developers to specify and submit ω_{UD}

[1]In multiple nodes CEPLESS system is distributed (shown by the dotted line).

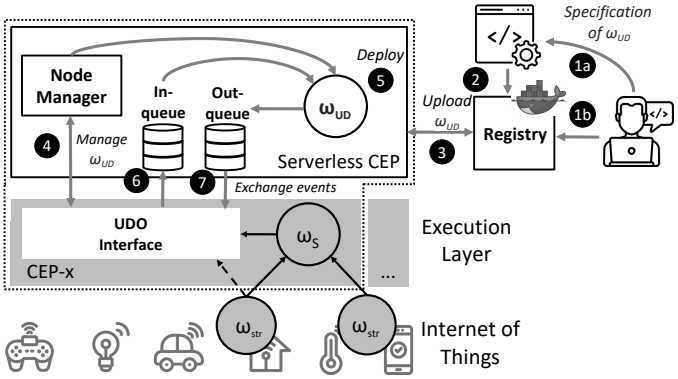

Fig. 3: Overview of a node in the dotted line using CEPLESS. *White layers are* CEPLESS *entities.*

operators in their preferred programming language. We show the flow of defining, deploying,and processing ω_{UD} using CEPLESS major components as follows.

We propose a programming interface to define and deploy specification language and CEP runtime independent user-defined operators ω_{UD} to the CEPLESS system. It is important to note that at this point developers do not need to have any knowledge of the CEP runtime where the ω_{UD} operators will be deployed. ❶a The developers can either program them using our interface or ❶b reuse operators by packaging them into an executable operator container C_ω and submit to the CEPLESS registry. ❷ The programming interface wraps the user-defined operator code into a container and pushes into the CEPLESS registry. ❸ The CEPLESS system maintains all the ω_{UD} operators in a central registry to facilitate the reuse of operators.

To deploy a ω_{UD} operator, the CEPLESS system invokes the internal *UDO interface* component of a specific CEP runtime. ❹ The UDO interface requests the deployment of the corresponding operator at a *Node Manager* NM_n instance, which is responsible for the deployment and registration of operator container C_ω on a node. ❺ After a NM_n instance downloaded the requested C_ω from the registry, it is started on the respective node for execution. The CEP runtime or system communicates with the ω_{UD} over the ❻ input event queue $eq\_in_{\omega_{UD}}$ and ❼ output event queue $eq\_out_{\omega_{UD}}$ as soon as the deployment by NM_n has finished. In this way, CEPLESS system provides a consistent functionality to other serverless systems [26], [12] towards the user. In the next subsections, we show the design of our system in terms of system components and their deployment. Furthermore, we introduce the interactions between CEPLESS and the provided execution layer, i.e., the CEP system. For the design of CEPLESS, we are focused on the following research questions identified before.

1) How to allow updates of an operator at system runtime, while guaranteeing high performance in the delivery of events? (Section IV-A)
2) How to design user-defined operators ω_{UD} independent from the underlying CEP runtime and the programming language? (Section IV-B)

A. Serverless Platform for CEP

In consistent with the principles of serverless computing, the serverless platform of CEPLESS acts as a bridge between the user and the different CEP runtime environments. The platform simplifies the deployment and configuration for the user while still preserving the correct execution of the ω_{UD} operators on the underlying IoT resources. It manages every compute resource in the network and is therefore not bound to a specific system physical location, e.g., in co-location with a CEP runtime. Besides providing simple communication between the users and the CEP system, the platform also acts as a central point of knowledge in our deployment design. The platform keeps information of all the submitted ω_{UD} operators and operator execution requests by the CEP systems. In the following subsections, we detail on the sub-components used in the serverless platform of CEPLESS for deployment and runtime management of ω_{UD} operators.

Listing 1: User-Defined Operator (UDO) Interface.

```
1 trait UserDefinedOperatorInterface {
2   def requestOperator(operatorName: String, cb:
        OperatorAddress => Unit): Unit
3   def sendEvent(e: Event, address:
        OperatorAddress): Unit
4   def addListener(address: OperatorAddress, cb:
        Any => Unit): Boolean
5   def removeListener(address: OperatorAddress):
        Boolean
6 }
```

User-Defined Operator Interface: We enable user-defined operators ω_{UD} to be utilized in CEP systems by abstracting the processing logic f_ω from the CEP system base. Operators can be implemented, used, and modified using any language that is executable by the virtualization layer. With this design choice, we give users the ability to use effectively any program that is executable in a virtualized container, with only minor modifications. For a CEP system to be able to communicate with this abstracted operator, it needs to implement an interface that handles events to and from the ω_{UD}, equal to state-of-the-art serverless systems. Therefore, we propose the *User-Defined Operator (UDO) Interface* which handles communication between the ω_{UD} and CEP system. Because every CEP system provides different language semantics, the implementation of this interface differs slightly between CEP systems, however, it is designed to provide only minimal overhead to the existing codebase.

Listing 1 shows the required functionality that every UDO Interface needs in a CEP system to be usable with our extension. Line 2 is the first request issued by a CEP system that initiates the deployment of an operator by CEPLESS. Parameter *operatorName* is a unique identifier for ω_{UD} to be deployed. The second parameter is a function invoked as soon as ω_{UD} deployment was initiated and returns an operator address at which it is reachable. The operator address is used for routing events from the CEP system to ω_{UD} and has to be used at any interaction with the interface. Line 3 shows the function that needs to be invoked to send an event to

a user-defined operator. The parameter *Event* is specific to the respective runtime system. The UDO interface is required to serialize the given event object into a readable format for the operator. For the CEP system to be able to also receive resulting events from the deployed operator, it can add and remove a listener as seen in Line 4 and 5. This listener expects a function that is called every time a new event was received by CEPLESS from a ω_{UD} that has a listener registered.

B. In-memory Queue Management

The serverless platform provides in-memory queues $eq_{\omega_{UD}}$ for communication between ω_{UD} operators and the underlying execution layer comprising of different CEP runtimes. The internal management of in-memory queue achieves three design goals (i) in order and stateful processing of events, (ii) runtime independent transfer of events to and from the platform and the execution layer and (iii) high performance in the delivery of events, which are extremely important for CEP applications. In our design the CEP system and the CEPLESS are residing on the same computational resource, thus, have only minimal latency impact on the streaming system by avoiding network latency. Of course, the design also allows us to deploy multiple user-defined operators as well in a distributed setting.

In order and stateful processing of events. We use FIFO queuing mechanisms to ensure that the given data stream ingested to the CEP system in the execution layer keeps the order for processing of the operator. Equally, the data stream generated from ω_{UD} is handled in the same manner. In this way, CEPLESS keeps the order of events, however, CEPLESS is dependent on the execution layer ordering mechanisms, e.g., Flink ensures ordering using watermarks, to ensure correctness in ordering such that there are no false positives and negatives. We store the incoming events from the execution layer (to be sent to the ω_{UD} operator), outgoing events from the ω_{UD} operator (to be sent to the execution layer) as well as the intermediate state of operators in in-memory queues. These are maintained for every ω_{UD} operator by the serverless layer.

A universal message format encapsulates the events which are passed into the queue. This format allows us to interact with different CEP runtimes without any dependence on their semantics. As the events are received from the execution layer, the serverless layer communicates with the in-memory queue for sending and receiving the events. The in-memory queue handles this communication using a client-server model. When the client residing at the serverless layer creates a request to add a new event, the event is appended to the tail of a specified operator queue $eq_{\omega_{UD}}$. The communication is managed within the queue by issuing commands such as `push` and `pop` received in the form of TCP requests whenever the events are to be sent and received, respectively. The queues store the events and their state in memory without having to persist them on external storage, which could induce high transfer and I/O latency. In this way, we provide fast but stateful processing of events, which is extremely important for CEP applications to deal with operators such as *time-based windows*.

Algorithm 1: Event queue command batching and flushing

```
   Data:
   b_s            ← Batch of events sent (and processed) to ω_UD;
   b_r            ← Batch of events received from ω_UD;
   t_backoff      ← Back-off interval increment;
   t_sleep        ← Total current back-off time;
   sendBuffer     ← Internal Buffer for command compilation;
   cmds           ← Event Queue client commands;
   inBatchSize    ← Batch size of commands to be received;
   outBatchSize   ← Batch size of commands to be sent;
   // receives events from the execution layer (ω_S)
1  function receiveEvent
2      b_s.push(event);
   // sends events to ω_UD
3  function backgroundThreadSend
4      t_sleep ← 0;
5      while true do
6          if b_s.size = 0 then
7              t_sleep ← t_sleep + t_backoff ;
8              sleep(t_sleep);
9          else
10             if b_s.size > outBatchSize then
11                 sendBuffer ← b_s.pop(0, outBatchSize);
12             else
13                 sendBuffer ← b_s;
14                 b_s.clear();
15             if sendBuffer.size > 0 then
16                 cmds ← sendBuffer.compileCommands();
17                 cmds.flush();
18             t_sleep ← 0;
   // receives events from ω_UD
19 function backgroundThreadReceive
20     t_sleep ← 0;
21     while true do
22         b_r ← range(0, inBatchSize) ;
23         if b_r.size = 0 then
24             t_sleep ← t_sleep + t_backoff ;
25             sleep(t_sleep);
26         forwardEvents(b_r) ;
27         t_sleep ← 0;
```

However, CEP systems often observe high rates of incoming events. For the client-server model as above, this would mean many TCP requests written to the network layer for each incoming event, which can be highly inefficient. We provide a solution to this problem as follows.

Guarantee high performance in the delivery of events. To handle the high rates of incoming events, we parallelize the process of event transfer and provide a batching mechanism that aids in processing a high rate of incoming events. Both mechanisms provide advantages to process more events per time unit and therefore aid in provisioning high throughput in delivery. With batching we manage the command flushing mechanism to the TCP server, which is a known control mechanism of in-memory queues [5]. This effectively means a suitable time when commands can be written to the network layer, i.e., the time when the commands are started. With automatic command flushing commands are always issued as soon as they are invoked at the client. This results in many commands (i.e., requests) being written to the network sequen-

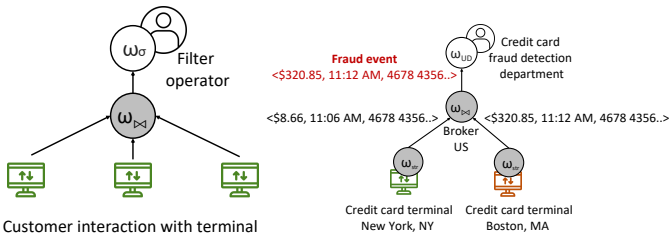

Fig. 4: Credit card fraud detection query that is used for evaluating our system.

tially, resulting in many TCP requests opened and closed per time unit. To resolve this issue, we send commands by issuing command flushes at specified intervals, independently from the query process. With command flushs, the TCP connection is only opened once per batch, all the commands are sent and the result for *all* commands is received. In Algorithm 1, we present this process. On the main thread (Line 2) we collect the received events from the execution layer (or user-defined operator ω_{UD}) and return immediately for fast processing in the execution layer. We utilize a background thread (Line 3–18) immediately initialized in the beginning, which continuously processes the collected events by flushing the commands (Line 17). The background thread avoids any blocking of the main thread which continuously receives the events. Instead of sending one request per event, we use a batching mechanism to collect multiple events and flush a batch b_s of commands to the in-memory queue server after a certain threshold is exceeded (Line 10). This essentially avoids the slowing down of the query as detailed above. The background thread runs after a backoff time $t_{backoff}$ if previous batch b_s was empty (Line 6). The total backoff time t_{sleep} gets linearly increased by $t_{backoff}$ with every iteration (Line 7) that contains an empty batch b_s. This process avoids any wastage of resources as there is no use of sending a request when the batch is empty. As soon as there are events in the batch, those are processed and the backoff interval is reset (Line 18).

To receive events after they were processed from the ω_{UD} operator, we initialize another background thread (Line 19–27). Here also, we take advantage of command batching for a range query (Line 22). Much like database range queries, the in-memory queue in CEPLESS provides the ability to define a start index and length to fetch items from the event queue. Using this command, we effectively reduce the number of requests to the event queue and fast-forwarding. Similar to the send thread, the receive thread utilizes a linearly increased backoff time to avoid polluting an empty queue with requests (Line 24).

The combination of these mechanisms: parallel processing, manual flushing, and command batching, CEPLESS showed significant performance improvements in terms of throughput while still preserving low processing latency, which is evaluated in Section V.

V. EVALUATION

In the evaluation we intend to answer the following questions towards CEPLESS:

Simulation time t_s	20 min
Warmup time t_w	60 s
Number of runs	30
Number of operators	3
Number of producers	1 - 3
Number of brokers and consumers	1
Number of serverless operators	1
Back-off interval increment $t_{backoff}$	1 ns
Input event rate	1000, 10,000 and 100,000
CEP systems	Apache Flink [8], CEP-P [21]
Queries	Fraud detection and forward

TABLE II: Configuration parameters for the evaluation. *Default or mostly used parameters are underlined.*

1) How to provide dynamic operator updates using CEPLESS without runtime dependence? (Section V-B)
2) What is the performance impact in terms of latency and throughput of implementing a user-defined operator using CEPLESS in comparison to a direct implementation in CEP systems? (Section V-C)

We answer the above questions in a threefold evaluation. In Section V-B, we evaluate the ability of CEPLESS to dynamically update an operator in comparison to the baseline of state-of-the-art CEP systems using throughput metrics. In section V-C, we provide a performance comparison of CEPLESS with baseline CEP systems Flink and CEP-P in terms of throughput (Section V-C1) and latency (Section V-C2).

A. Evaluation Setup

In the following, we describe the evaluation platform, the CEPLESS implementation, dataset, and queries used in the evaluation.

Evaluation platform. For evaluating CEPLESS, we utilize Docker version 19.03 running on a server with Intel(R) Xeon(R) CPU E5-2630 v2 @ 2.60GHz processors with Ubuntu 18.04 installed. The server has 128 GB RAM and 24 cores of Intel CPU. We will use two different setup modes: (i) running direct implementation of f_ω in CEP system (baseline) and (ii) running the CEP system in combination with CEPLESS and a user-defined operator (ω_{UD}). For the former setup, we will execute the CEP system in a docker container which runs a given query. For the latter setup, the CEP system and CEPless will also be executed in two separate docker containers running the same query. The CEPless setup will be running the event queue, a user-defined operator ω_{UD}, and the node manager on the same node, as proposed by our design. Furthermore, the docker containers are not restricted in their respective resource usage and therefore able to utilize the complete available node resources.

Dataset. We use a real-world dataset containing financial transactions [22] to detect credit card fraud for evaluation. The anonymized dataset contains 284,807 rows of credit card transaction data. As a data producer, we used Apache Kafka running directly on the same computational resource.

Queries. The query used for the evaluation is shown in Figure 4. In our evaluations, we will replace the *Filter* (ω_σ) by

a user-defined operator realized in CEPLESS (figure at right). This operator performs a simple filter operation for filtering out transactions as fraud in the dataset. It will be evaluated using only the natively provided operator directly in the CEP system specification language and written as a ω_{UD} operator in our system. Furthermore, to evaluate the performance impact of CEPLESS onto existing CEP systems, we use a forward query that forwards the event tuple or data stream directly to the consumer (cf. Section V-C).

We consider credit card terminals as producers (e.g. at supermarkets). Every card terminal produces an event as soon as a transaction process has started, i.e. the customer has presented the card at the terminal. An emitted event contains the following tuples [22]: *timestamp*, *amount*, *cardId*, *terminalId*. Events sent by card terminals serve as input for a CEP query that detects fraudulent transactions. Terminals are connected directly through a wired connection to the network which enables stable connections to brokers.

A summary of the configuration parameters we used in the evaluation can be found in Table II. *Backoff time* describes the interval in which the aforementioned batches are written to the network layer. In the next section, we go into the performance evaluation of CEPLESS.

B. Evaluation of dynamic operator updates

To understand how CEP systems benefit from CEPLESS operator updates, we show evaluations regarding the explicit time when updating an operator that is currently in execution. For this, we utilize two metrics: i) downtime, ii) update time and (iii) throughput. *Downtime* represents the amount of time where no events were received at the consumer side, while *update time* gives the amount of time it took to update the operator to the new version. *Throughput* is defined as the number of output events received at the consumer for the input events ingested by the producer. Such operator updates are necessary for applications like fraud detection, earlier motivated in Section II. Small down and update times are considered to be good. We evaluated CEPLESS in comparison to Apache Flink by performing an operator update using our presented mechanism for the former and by deploying a query with the new operator for the latter. The business logic of the operator gets updated to detect *more* fraud patterns hence resulting in a higher throughput of events after the update. Ideally, the throughput should remain at a constant rate to not have performance peaks in resource usage of the given server. Both systems share the same business logic inside the operator before the update and get updated with the same business logic.

The evaluated throughput of the experiment is shown in Figure 5. We repeated the experiment 30 times and plot the 95% confidence interval (shown as shadow in the line plot) of the observations. The operator update was issued at t = 290 in both systems. While the update is proceeding in Flink a downtime is observed, because of the deployment of the new query. This downtime is introduced by the distribution and execution of the used JAR-file for the query. The mean

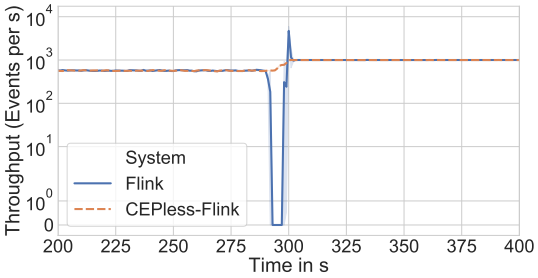

Fig. 5: Apache Flink experiences a downtime while CEPLESS steadily updates the operator while providing optimal throughput.

downtime of the query was approximately **8.6 s**, which is equivalent to the update time, and resulted in a throughput decline and spike at approximately t = 298. This spike is caused by the events that were not processed while the query was in an update state. As soon as the new query is executing again, it begins processing from the last saved checkpoint at the data generator. In comparison, CEPless showed no downtime at all, only a mean update time of **238 ms** without a throughput decline or spike. In fact, the CEPLESS achieves a throughput of the new query very steadily at t = 300.

We achieved this relatively small update time with the mechanism of updating the operator containers instead of complete queries. While this update occurs, the state of the query currently in execution is kept in the presented message queues.

C. Performance Evaluation

To understand how the CEPLESS system influences a CEP system we analyze both implementations (Apache Flink and CEP-P) in terms of *end-to-end latency* and *throughput*. These two are the most important metrics for a CEP system as well as the financial fraud detection example. We evaluate a forward query introduced in Section V-A to observe whether CEPLESS achieves optimal throughput and latency while forwarding the data streams. We evaluate the following three system configurations:

1) Running only the native CEP system implementation without our extension (baseline).
2) Running the CEP system with a user-defined operator, Redis [5] as an in-memory queue with and without batching.

The impact of throughput and latency are evaluated on Flink and CEP-P that showed similar observations as detailed in the following. This shows the universal applicability and behavior of different CEP systems.

1) Impact on throughput: An *optimal* throughput is achieved when the output events exactly match or supersedes the input event rate. We collected throughput measurements for the above three configurations for 20 minutes and other parameters as presented in Table II. We repeated the experiment 30 times to provide 90, 95, 99% confidence interval of throughput for the different input rates, respectively. In Table III , we present the mean throughput measurements for the baseline Flink and our extension CEPLESS with Flink ingested with

	1,000 events/s				10,000 events/s				100,000 events/s			
System	*mean*	*min*	*max*	*quantiles (90, 95, 99)*	*mean*	*min*	*max*	*quantiles (90, 95, 99)*	*mean*	*min*	*max*	*quantiles (90, 95, 99)*
Flink	1047	108	1093	1001, 1002, 1002	10475	2180	19990	10010, 10011, 10020	100123	16770	200168	100098,100136,100797
CEPless-Flink	**1070**	**378**	**1190**	**1003, 1004, 1004**	**10477**	**1174**	**20007**	**10019, 10027, 12081**	**100353**	**18353**	**249689**	**100623,100815,101806**
CEP-P	1021	396	2292	1001, 1002, 1002	10022	2660	45185	10010, 10015, 10110	-	-	-	-
CEPless-CEPP	**1000**	**214**	**4927**	**1001, 1002, 1004**	**10002**	**580**	**32091**	**10015, 10025, 10163**	-	-	-	-

TABLE III: Throughput measurements: mean, min, max, and quantiles (90,95,99) for the forward operator.

	1,000 events/s				10,000 events/s				100,000 events/s			
System	*mean*	*min*	*max*	*quantiles (90, 95, 99)*	*mean*	*min*	*max*	*quantiles (90, 95, 99)*	*mean*	*min*	*max*	*quantiles (90, 95, 99)*
Flink	0.78	1.0	31.46	0.98, 1.06, 1.55	1.71	1.0	341.61	2.09, 2.22, 3.11	1.74	1.0	87.91	2.29, 2.45, 2.88
CEPless-Flink	**3.19**	**1.0**	**26.19**	**4.10, 4.35, 4.9**	**3.63**	**1.15**	**503.50**	**4.21, 4.48, 5.15**	**4.61**	**1.82**	**956.31**	**5.71, 6.11, 7.80**
CEP-P	1.14	1.0	27.89	1.45, 1.57, 1.82	4.21	1.0	496.63	2.54, 2.74, 24.32	-	-	-	-
CEPless-CEPP	**4.94**	**0.72**	**21.27**	**6.10, 6.37, 6.86**	**5.21**	**1.97**	**488.84**	**6.25, 6.75, 10.38**	-	-	-	-

TABLE IV: Latency measurements: mean, min, max, and quantiles (90,95,99) for the forward operator (in ms).

an input rate of 1,000, 10,000, and 100,000 events per second, respectively from left to right. In comparison, our extension of Flink with CEPLESS achieves matching optimal throughput per time unit for the given input rate. Clearly, CEPLESS-Flink matches the baseline in all the results. We observed similar results for our extension on CEP-P for throughput. CEP-P baseline was not able to perform well under a higher scale (100K events per second) because of the absence of back pressure and flow control mechanisms in the system. Hence we do not report the results for the 100K event rate. For 1K and 10K events, CEPLESS with CEP-P performs equally well and attains optimal throughput. Henceforth, our extension *does not* introduce any overhead in terms of throughput and can easily deal with a high scale of ingested events as seen in the evaluations.

To achieve matching throughput for high event rates 100K events we set the input batch size to 10,000. As described in Section IV-B, this implicitly increases the size of a range query in Redis leading to lower queue processing overhead due to fewer requests and internal network round-trips. We further elaborate on the batch size in the last subsection.

Table III in some cases also shows higher throughput results for CEPLESS-Flink than Apache Flink (baseline), which is an interesting observation. We analyzed this behavior and found out that since we process events in batches due to the batching mechanism introduced in Algorithm 1, the events are queued up in the batch instead of getting processed directly. Moreover, Flink implements flow control and backpressure mechanisms, e.g., credit based flow control [8] to deal with such situations. The throughput for CEPLESS-Flink is higher since the system handles events that were previously received but backpressured by the Flink engine. Therefore, when looking at the evaluated values the throughput per second might seem higher but at minimal latency cost, evaluated in the next subsection.

2) Impact on latency: The *end to end latency* is defined as the time taken to retrieve an event from the producer until it reaches the consumer from the CEP engine. Similar to the throughput evaluations, we collected latency measurements for different input event rates. In Table IV , we present the latency measurement observed for the Flink baseline and our extension. Our CEPLESS system atop Flink attains good performance while introducing only a minimal overhead in terms of latency. We observe only a mean overhead in latency of **1.92 ms** for 10,000 events per second and **2.87 ms** for 100,000 events per second (calculated as *mean latency of* CEPLESS– *mean latency of Flink*). We also present the latency measurements observed for CEP-P (baseline) and with our extension. Similar to Apache Flink, our extension here as well induce a minimal overhead of **1 ms** for 10K events. The overhead is minimal because we have parameterized the batch sizes for a suitable configuration as seen in the next subsection. The overhead comprises of two factors: i) the translation in the universal message format of every event and ii) the round-trip-time (RTT) of the message queue to the CEP system in use.

VI. RELATED WORK

In the following, we present previous work in terms of flexible operator deployment and unifying CEP systems and serverless frameworks.

CEP Systems and Programming Models. Apache Beam [11] developed by Google provides a unified programming model as an abstraction layer above multiple different CEP systems including Apache Flink [8] and Storm [20]. This work confirms the need to have a unified programming model for deploying queries on different CEP systems. However, with operators being bound to the respective execution environments – as proposed in Beam – CEP systems fail to fulfill the flexibility of complex applications that require different systems to interact together. Recently, Bartnik et al. [7], proposed an extension of Flink with runtime updates of operators, however, their proposal is Flink's runtime dependent and not applicable to other CEP runtime systems. Some classical systems like Amit [1] and Apama [2] allow dynamic updates of CEP operators, however, with a strict dependency on the runtime. Many CEP languages have been developed in the past years such as CQL [6], SASE [28], TESLA [9], besides the domain-specific ones proposed in the above programming models. However, as per our knowledge, none of the above programming models provide a powerful abstraction on multiple runtime environments as we do.

Serverless Frameworks. In recent years, many commercial serverless platforms evolved e.g., AWS Lambda [26], Google Cloud Functions [12], Azure Functions [24], and IBM OpenWhisk [16]. Most of these providers often support the streaming of input data towards serverless functions, making it possible to also execute continuous data flows. However, often those services are limited to the provider-specific streaming so-

lutions for example Kinesis [25] by Amazon. An open-source alternative to AWS Lambda is Kubeless [19], which provides multiple different runtime environments supporting different languages, but also the ability to provide a custom runtime using a custom image. Programming models for serverless [18], [23] on top of AWS and Microsoft Azure, respectively have been proposed as well. However, an integrated solution, with CEP systems, e.g., Flink, is still missing which we see as a gap in current serverless execution paradigms. Our system could be used as another serverless service by cloud providers to provide real serverless operators in combination with existing streaming systems. This service could be especially useful when looking at CEP systems.

VII. Conclusion

In this work, we proposed CEPless, a CEP system based on *serverless computing*, which provides flexibility in developing *new* user-defined operators in any programming language and updating them at runtime. This is highly beneficial for applications that require dynamic changes such as fraud detection in financial context and Internet of Things applications. To this end, we (i) ease development, (ii) provide dynamic operator updates and (iii) improve the flexibility of operators in CEP system while preserving the statefulness and meeting the performance requirements in terms of throughput (up to 100K events per second) and latency (˜1.9 ms). This work essentially takes an important step towards *serverless computing* for CEP systems. Furthermore, it can be highly beneficial to develop user-defined operator placement algorithms that are independent of the execution environment. Besides, distributed in-memory queues such as Anna [27] can be used to achieve a higher scale in stateful processing for instance for bigger window sizes. Finally, CEPless provides a foundation for developing advanced serverless features like just-in-time billing and auto-scaling on cloud platforms. For example, the programming interface can provide the information required for billing operators and container technology is at the core of CEPless can be easily used for auto-scaling. Sophisticated scaling strategies for CEP [13] can also be integrated as a runtime in our execution layer.

References

[1] A. Adi and O. Etzion. Amit - the situation manager. *The VLDB Journal*, 13:177–203, 2004.

[2] S. AG. Apama cep. http://www.apamacommunity.com/, 2018. [Accessed on 21.5.2020].

[3] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. Millwheel: fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment*, 6(11):1033–1044, 2013.

[4] 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. *Proceedings of the VLDB Endowment*, 8(12):1792–1803, 2015.

[5] Antirez. Redis: In-memory database. https://redis.io/, 2019. [Accessed on 3.4.2020].

[6] A. Arasu, S. Babu, and J. Widom. The cql continuous query language: Semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, June 2006.

[7] A. Bartnik, B. D. Monte, T. Rabl, and V. Markl. On-the-fly reconfiguration of query plans for stateful stream processing engines. In *Datenbanksysteme für Business, Technologie und Web (BTW), 18. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme (DBIS) Proceedings*, volume P-289, pages 127–146. Gesellschaft für Informatik, Bonn, 2019.

[8] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink: Stream and batch processing in a single engine. *IEEE Data Engineering Bulletin*, 38, 2015.

[9] G. Cugola and A. Margara. Tesla: A formally defined event specification language. In *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems (DEBS)*, page 50–61, New York, NY, USA, 2010. Association for Computing Machinery.

[10] G. Cugola and A. Margara. Processing Flows of Information: From Data Stream to Complex Event Processing. *ACM Computing Surveys*, 44(3):1–62, 2012.

[11] T. A. S. Foundation. Beam. https://beam.apache.org/, 2019. [Accessed on 3.4.2020].

[12] Google. Google cloud functions. https://cloud.google.com/functions/, 2019. [Accessed on 3.4.2020].

[13] V. Gulisano, R. Jiménez-Peris, M. Patiño-Martínez, C. Soriente, and P. Valduriez. Streamcloud: An elastic and scalable data streaming system. *IEEE Transactions on Parallel and Distributed Systems*, 23(12):2351–2365, 2012.

[14] 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. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2020.

[15] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Serverless computation with openlambda. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2016.

[16] IBM. Ibm openwhisk. https://developer.ibm.com/open/projects/openwhisk/, 2019. [Accessed on 3.4.2020].

[17] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht. Occupy the cloud: Distributed computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing (SOCC)*, page 445–451, New York, NY, USA, 2017. Association for Computing Machinery.

[18] S. Joyner, M. MacCoss, C. Delimitrou, and H. Weatherspoon. Ripple: A Practical Declarative Programming Framework for Serverless Compute. In *arXiv:2001.00222*, January 2020.

[19] Kubeless. Kubeless. https://kubeless.io/, 2019. [Accessed on 3.4.2020].

[20] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja. Twitter heron: Stream processing at scale. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 239–250. ACM, 2015.

[21] M. Luthra, B. Koldehofe, P. Weisenburger, G. Salvaneschi, and R. Arif. Tcep: Adapting to dynamic user environments by enabling transitions between operator placement mechanisms. In *Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems*, DEBS '18, pages 136–147, 2018.

[22] U. Machine Learning Group. Credit card fraud detection: Anonymized credit card transactions labeled as fraudulent or genuine. https://www.kaggle.com/mlg-ulb/creditcardfraud, 2017. [Accessed on 3.4.2020].

[23] G. McGrath and P. R. Brenner. Serverless computing: Design, implementation, and performance. In *Proceedings of 37th IEEE International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 405–410, June 2017.

[24] Microsoft. Microsoft azure functions. https://azure.microsoft.com/en-us/services/functions/, 2019. [Accessed on 3.4.2020].

[25] A. W. Services. Aws kinesis. https://aws.amazon.com/kinesis/, 2019. [Accessed on 3.4.2020].

[26] A. W. Services. Aws lambda. https://aws.amazon.com/lambda/, 2019. [Accessed on 3.4.2020].

[27] C. Wu, J. Faleiro, Y. Lin, and J. Hellerstein. Anna: A kvs for any scale. In *Proceedings of the 34th IEEE International Conference on Data Engineering (ICDE)*, pages 401–412, April 2018.

[28] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, page 407–418, New York, NY, USA, 2006. Association for Computing Machinery.