# Move Fast and Meet Deadlines: Fine-grained Real-time Stream Processing with Cameo

Le Xu[*], Shivaram Venkataraman[2], Indranil Gupta[1], Luo Mai[3], and Rahul Potharaju[4]

[1]University of Illinois at Urbana-Champaign, [2]UW-Madison, [3]University of Edinburgh, [4]Microsoft

## Abstract

Resource provisioning in multi-tenant stream processing systems faces the dual challenges of keeping resource utilization high (without over-provisioning), and ensuring performance isolation. In our common production use cases, where streaming workloads have to meet latency targets and avoid breaching service-level agreements, existing solutions are incapable of handling the wide variability of user needs. Our framework called Cameo uses fine-grained stream processing (inspired by actor computation models), and is able to provide high resource utilization while meeting latency targets. Cameo dynamically calculates and propagates priorities of events based on user latency targets and query semantics. Experiments on Microsoft Azure show that compared to state-of-the-art, the Cameo framework: i) reduces query latency by 2.7×in single tenant settings, ii) reduces query latency by 4.6×in multi-tenant scenarios, and iii) weathers transient spikes of workload.

## 1 Introduction

Stream processing applications in large companies handle tens of millions of events per second [16, 68, 89]. In an attempt to scale and keep total cost of ownership (TCO) low, today's systems: a) parallelize operators across machines, and b) use multi-tenancy, wherein operators are collocated on shared resources. Yet, resource provisioning in production environments remains challenging due to two major reasons:

**(i) High workload variability.** In a production cluster at a large online services company, we observed orders of magnitude variation in event ingestion and processing rates, *across time*, *across data sources*, *across operators*, and *across applications*. This indicates that resource allocation needs to be dynamically tailored towards each operator in each query, in a nimble and adept manner at run time.

**(ii) Latency targets vary across applications.** User expectations come in myriad shapes. Some applications require quick responses to events of interest, i.e., short end-to-end latency. Others wish to maximize throughput under limited resources, and yet others desire high resource utilization. Violating such user expectations is expensive, resulting in breaches of service-level agreements (SLAs), monetary losses, and customer dissatisfaction.

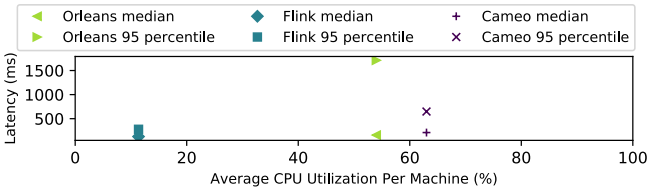



Figure 1: *Slot-based system (Flink), Simple Actor system (Orleans), and our framework Cameo.*

To address these challenges, we explore a new *fine-grained* philosophy for designing a multi-tenant stream processing system. Our key idea is to provision resources to each operator based solely on its *immediate* need. Concretely we focus on deadline-driven needs. Our fine-grained approach is inspired by the recent emergence of event-driven data processing architectures including actor frameworks like Orleans [10, 25] and Akka [1], and serverless cloud platforms [5, 7, 11, 51].

Our motivation for exploring a fine-grained approach is to enable resource sharing directly among operators. This is more efficient than the traditional *slot-based* approach, wherein operators are assigned dedicated resources. In the slot-based approach, operators are mapped onto processes or threads—examples include task slots in Flink [27], instances in Heron [57], and executors in Spark Streaming [90]. Developers then need to either assign applications to a dedicated subset of machines [13], or place execution slots in resource containers and acquire physical resources (CPUs and memory) through resource managers [8, 47, 84].

While slot-based systems provide isolation, they are hard to dynamically reconfigure in the face of workload variability. As a result it has become common for developers to "game" their resource requests, asking for over-provisioned resources, far above what the job needs [34]. Aggressive users starve other jobs which might need immediate resources, and the

[*]Contact author: Le Xu <lexu1@illinois.edu>

upshot is unfair allocations and low utilization.

At the same time, today's fine-grained scheduling systems like Orleans, as shown in Figure 1, cause high tail latencies. The figure also shows that a slot-based system (Flink on YARN), which maps each executor to a CPU, leads to low resource utilization. The plot shows that our approach, Cameo, can provide both high utilization and low tail latency.

To realize our approach, we develop a new priority-based framework for fine-grained distributed stream processing. This requires us to tackle several *architectural* design challenges including: 1) translating a job's performance target (deadlines) to priorities of individual messages, 2) developing interfaces to use real-time scheduling policies such as earliest deadline first (EDF) [65], least laxity first (LLF) [69] etc., and 3) low-overhead scheduling of operators for prioritized messages. We present *Cameo*, a new scheduling framework designed for data streaming applications. Cameo:

- *Dynamically* derives priorities of operators, using both: a) *static input*, e.g., job deadline; and b) *dynamic stimulus*, e.g., tracking stream progress, profiled message execution times.
- Contributes new mechanisms: i) *scheduling contexts*, which propagate scheduling states along dataflow paths, ii) a *context handling* interface, which enables pluggable scheduling strategies (e.g., laxity, deadline, etc.), and iii) tackles required scheduling issues including per-event synchronization, and semantic-awareness to events.
- Provides low-overhead scheduling by: i) using a stateless scheduler, and ii) allowing scheduling operations to be driven purely by message arrivals and flow.

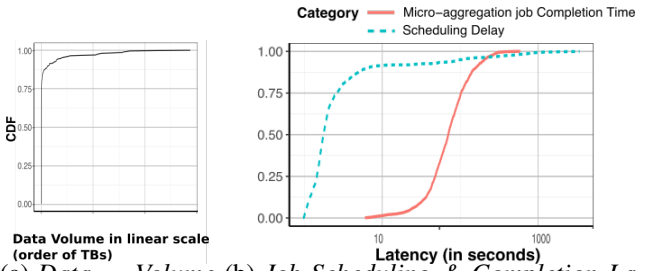
We build Cameo on Flare [68], which is a distributed data flow runtime built atop Orleans [10, 25]. Our experiments are run on Microsoft Azure, using production workloads. Cameo, using a laxity-based scheduler, reduces latency by up to 2.7× in single-query scenarios and up to 4.6× in multi-query scenarios. Cameo schedules are resilient to transient workload spikes and ingestion rate skews across sources. Cameo's scheduling decisions incur less than 6.4% overhead.

## 2 Background and Motivation

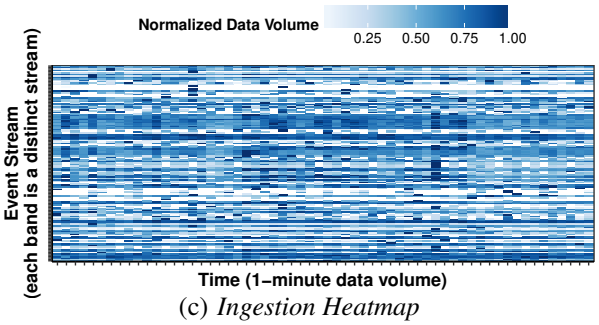
### 2.1 Workload Characteristics

We study a production cluster that ingests more than 10 PB per day over several 100K machines. The shared cluster has several internal teams running streaming applications which perform debugging, monitoring, impact analysis, etc. We first make key observations about this workload.

**Long-tail streams drive resource over-provisioning.** Each data stream is handled by a standing *streaming* query, deployed as a dataflow job. As shown in Figure 2(a), we first observe that 10% of the streams process a majority of the data. Additionally, we observe that a long tail of streams, each processing small amount data, are responsible for over-provisioning—their users rarely have any means of accurately gauging how many nodes are required, and end up over-provisioning for their job.




(a) *Data Volume Distribution* (b) *Job Scheduling & Completion Latencies*




(c) *Ingestion Heatmap*

Figure 2: *Workload characteristics collected from a production stream analytics system.*

**Temporal variation makes resource prediction difficult.** Figure 2(c) is a heat map showing incoming data volume for 20 different stream sources. The graph shows a high degree of variability across both sources and time. A single stream can have spikes lasting one to a few seconds, as well as periods of idleness. Further, this pattern is continuously changing. This points to the need for an agile and fine-grained way to respond to temporal variations, as they are occurring.

**Users already try to do fine-grained scheduling.** We have observed that instead of continuously running streaming applications, our users prefer to provision a cluster using external resource managers (e.g., YARN [2], Mesos [47]), and then run periodic micro-batch jobs. Their implicit aim is to improve resource utilization and throughput (albeit with unpredictable latencies). However, Figure 2(b) shows that this ad-hoc approach causes overheads as high as 80%. This points to the need for a common way to allow all users to perform fine-grained scheduling, without a hit on performance.

**Latency requirements vary across jobs.** Finally, we also see a wide range of latency requirements across jobs. Figure 2(b) shows that the job completion time for the micro-aggregation jobs ranges from less than 10 seconds up to 1000 seconds. This suggests that the range of SLAs required by queries will vary across a wide range. This also presents an opportunity for priority-based scheduling: applications have longer latency constraints tend to have greater flexibility in

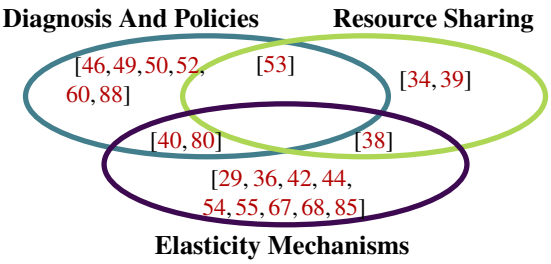



Figure 3: *Existing Dataflow Reconfiguration Solutions.*

terms of *when* its input can be processed (and vice versa).

## 2.2 Prior Approaches

**Dynamic resource provisioning for stream processing.** Dynamic resource provisioning for streaming data has been addressed primarily from the perspective of dataflow reconfiguration. These works fall into three categories as shown in Figure 3:
i) *Diagnosis And Policies*: Mechanisms for when and how resource re-allocation is performed;
ii) *Elasticity Mechanisms*: Mechanisms for efficient query reconfiguration; and
iii) *Resource Sharing*: Mechanisms for dynamic performance isolation among streaming queries.
These techniques make changes to the dataflows in reaction to a performance metric (e.g., latency) deteriorating.

Cameo's approach does not involve changes to the dataflow. It is based on the insight that the streaming engine can delay processing of those query operators which will not violate performance targets right away. This allows us to quickly prioritize and provision resources proactively for those other operators which could immediately need resources. At the same time, existing reactive techniques from Figure 3 are orthogonal to our approach and can be used alongside our proactive techniques.

**The promise of event-driven systems.** To achieve fine-grained scheduling, a promising direction is to leverage emerging event-driven systems such as actor frameworks [43, 74] and serverless platforms [24]. Unlike slot-based stream processing systems like Flink [27] and Storm [83], operators here are not mapped to specific CPUs. Instead event-driven systems maintain centralized queues to host incoming messages and dynamically dispatch messages to available CPUs. This provides an opportunity to develop systems that can manage a unified queue of messages across query boundaries, and combat the over-provisioning of slot-based approaches. Recent proposals for this execution model also include [11, 24, 26, 58].

Cameo builds on the rich legacy of work from two communities: classical real-time systems [63, 75] and first-generation stream management systems (DSMS) in the database community [14, 15, 31, 71]. The former category has produced rich scheduling algorithms, but unlike Cameo, none build a full working system that is flexible in policies, or support streaming operator semantics. In the latter category the closest to our work are event-driven approaches [14, 22, 28]. But these do not interpret stream progress to derive priorities or support trigger analysis for distributed, user-defined operators. Further, they adopt a centralized, stateful scheduler design, where the scheduler *always* maintains state for all queries, making them challenging to scale.

Achieving Cameo's goal of dynamic resource provisioning is challenging. Firstly, messages sent by user-defined operators are a black-box to event schedulers. Inferring their impact on query performance requires new techniques to analyze and re-prioritize said messages. Secondly, event-driven schedulers must scale with message volume and not bottleneck.

# 3 Design Overview

**Assumptions, System Model:** We design Cameo to support streaming queries on clusters shared by cooperative users, e.g., within an organization. We also assume that the user specifies a latency target at query submission time, e.g., derived from product and service requirements.

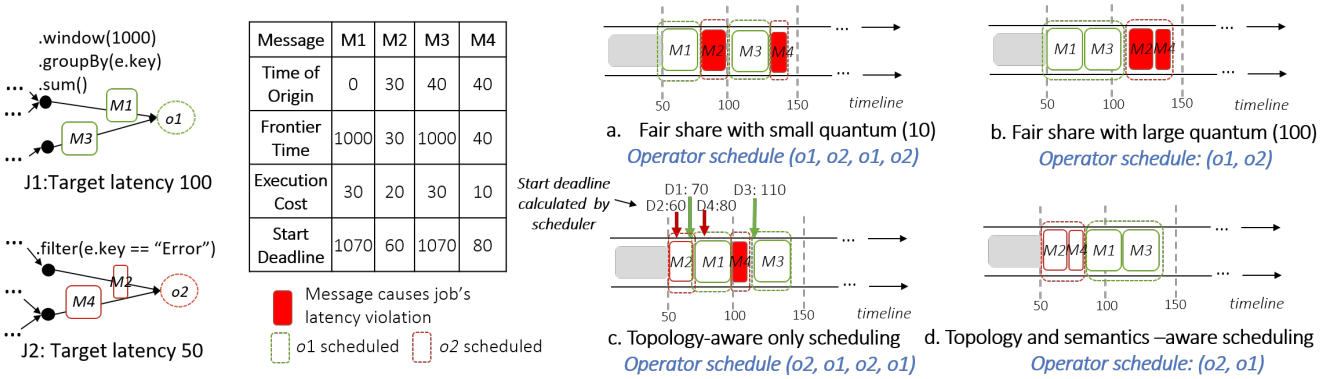
The architecture of Cameo consists of two major components: (i) a scheduling strategy which determines message priority by interpreting the semantics of query and data streams given a latency target. (Section 4), and (ii) a scheduling framework that 1. enables message priority to be generated using a pluggable strategy, and 2. schedules operators dynamically based on their current pending messsages' priorities (Section 5).

Cameo prioritizes operator processing by computing the *start deadlines* of arriving messages, i.e., latest time for a message to start execution at an operator without violating the downstream dataflow's latency target for that message. Cameo continuously reorders operator-message pairs to prioritize messages with earlier deadlines.

Calculating priorities requires the scheduler to continuously book-keep both: (i) per-job static information, e.g., latency constraint/requirement[1] and dataflow topology, and (ii) dynamic information such as the timestamps of tuples being processed (e.g., stream progress [19, 61]), and estimated execution cost per operator. To scale such a fine-grained scheduling approach to a large number of jobs, Cameo utilizes *scheduling contexts*— data structures attached to messages that capture and transport information required to generate priorities.

The scheduling framework of Cameo has two levels. The upper level consists of *context converters*, embedded into each operator. A context converter modifies and propagates scheduling contexts attached to a message. The lower level is a *stateless scheduler* that determines target operator's priority by interpreting scheduling context attached to the message. We also design a programmable API for a pluggable scheduling strategy that can be used to handle scheduling contexts. In summary, these design decisions make our scheduler scale to a large number of jobs with low overhead.

[1] We use latency constraint and latency requirement interchangeably.




| Message | M1 | M2 | M3 | M4 |
|---|---|---|---|---|
| Time of Origin | 0 | 30 | 40 | 40 |
| Frontier Time | 1000 | 30 | 1000 | 40 |
| Execution Cost | 30 | 20 | 30 | 10 |
| Start Deadline | 1070 | 60 | 1070 | 80 |

Figure 4: *Scheduling Example: J1 is batch analytics, J2 is latency-sensitive. Fair-share scheduler creates schedules "a" and "b". Topology-aware scheduler reduces violations ("c"). Semantics-aware scheduler further reduces violations ("d"). We further explain these examples in Section 4.2*

**Example.** We present an example highlighting our approach. Consider a workload, shown in Figure 4, consisting of two streaming dataflows $J1$ and $J2$ where $J1$ performs a batch analytics query and $J2$ performs a latency sensitive anomaly detection pipeline. Each has an output operator processing messages from upstream operators. The default approach used by actor systems like Orleans is to: i) order messages based on arrival, and ii) give each operator a fixed time duration (called "quantum") to process its messages. Using this approach we derive the schedule "a" with a small quantum, and a schedule "b" with a large quantum — both result in two latency violations for $J2$. In comparison, Cameo discovers the opportunity to postpone less latency-sensitive messages (and thus their target operators). This helps $J2$ meet its deadline by leveraging topology and query semantics. This is depicted in schedules "c" and "d". This example shows that *when* and *how long* an operator is scheduled to run should be dynamically determined by the priority of the next pending message. We expand on these aspects in the forthcoming sections.

## 4 Scheduling Policies in Cameo

One of our primary goals in Cameo is to enable fine-grained scheduling policies for dataflows. These policies can prioritize messages based on information, like the deadline remaining or processing time for each message, etc. To enable such policies, we require techniques that can calculate the priority of a message for a given policy.

We model our setting as a non-preemptive, non-uniform task time, multi-processor, real-time scheduling problem. Such problems are known to be NP-Complete offline and cannot be solved optially online without complete knowledge of future tasks [33, 81]. Thus, we consider how a number of commonly used policies in this domain, including Least-Laxity-First (LLF) [69], Earliest-Deadline-First (EDF) [65] and Shortest-Job-First (SJF) [82], and describe how such policies can be used for event-driven stream processing.We use the LLF policy as the default policy in our description below.

The above policies try to prioritize messages to avoid violating latency constraints. Deriving the priority of a message requires analyzing the impact of each operator in the dataflow on query performance. We next discuss how being deadline-aware can help Cameo derive appropriate priorities. We also discuss how being aware of query semantics can further improve prioritization.

| Symbol | Definition |
|---|---|
| $ID_M$ | ID of Message M. |
| $ddl_M$ | Message start deadline. |
| $o_M$ | target operator of $M$. |
| $C_{o_M}$ | Estimated execution cost of $M$ on its target operator. |
| $t_M$, and $p_M$ | Physical (and logical) time associated with the last event required to produce $M$. |
| $L$ | Dataflow latency constraint of the dataflow that $M$ belongs to. |
| $p_{M_F}$, and $t_{M_F}$ | Frontier progress, and frontier time. |

Table 1: *Notations used in paper for message M.*

### 4.1 Definitions and Underpinnings

**Event.** Input data arrives as *events*, associated with a *logical time* [30] that indicates the *stream progress* of these events in the input stream.

**Dataflow job and operators.** A dataflow job consists of a DAG of *stages*. Each stage operates a user-defined function. A stage can be parallelized and executed by a set of dataflow *operators*.

We say an operator $o_k$ is *invoked* when it processes its input message, and $o_k$ is *triggered* when it is invoked and leads to an output message, which is either passed downstream to further operators or the final job output.

Cameo considers two types of operators: i) regular operators that are triggered immediately on invocation; and ii) windowed operators [61] that partitions data stream into sec-

tions by logical times and triggers only when all data from the section are observed.

**Message timestamps.** We denote a message $M$ as a tuple $(o_M, (p_M, t_M))$, where: a) $o_M$ is the operator executing the message; b) $p_M$ and $t_M$ record the *logical* and *physical* time of the input stream that is associated with $M$, respectively. Intuitively, $M$ is influenced by input stream with logical time $\leq p_M$. Physical time $t_M$ marks the system time when $p_M$ is observed at a source operator.

We denote $C_{o_M}$ as the estimated time to process message $M$ on target operator $O$, and $L$ as the latency constraint for the dataflow that $M$ belongs to.

**Latency.** Consider a message $M$ generated as the output of a dataflow (at its sink operator). Consider the set of all events $E$ that influenced the generation of $M$. We define latency as the difference between the last arrival time of any event in $E$ and the time when $M$ is generated.

## 4.2 Calculating Message Deadline

We next consider the LLF scheduling policy where we wish to prioritize messages which have the least laxity (i.e., flexibility). Intuitively, this allows us to prioritize messages that are closer to violating their latency constraint. To do this, we discuss how to determine the *latest time* that a message $M$ can start executing at operator $O$ without violating the job's latency constraint. We call this as the *start deadline* or in short the *deadline* of the message $M$, denoted as $ddl_M$. For the LLF scheduler, $ddl_M$ is the message priority (lower value implies higher priority).

We describe how to derive the priority (deadline) using topology-awareness and then query (semantic)-awareness.

### 4.2.1 Topology Awareness

**Single-operator dataflow, Regular operator.** Consider a dataflow with only one regular operator $o_M$. The latency constraint is $L$. If an event occurs at time $t_M$, then $M$ should complete processing before $t_M + L$. The start deadline, given execution estimate $C_{o_M}$, is:

$$ddl_M = t_M + L - C_{o_M} \tag{1}$$

**Multiple-operator dataflow, Regular operator.** For an operator $o$ inside a dataflow DAG that is invoked by message $M$, the start deadline of $M$ needs to account for execution time of downstream operators. We estimate the maximum of execution times of critical path [49] from $o$ to any output operator as $C_{path}$. The start deadline of $M$ is then:

$$ddl_M = t_M + L - C_{O_M} - C_{path} \tag{2}$$

Schedule "c" of Figure 4 showed an example of topology-aware scheduling and how topology awareness helps reduce violations. For example, $ddl_{M2} = 30 + 50 - 20 = 60$ means that $M2$ is promoted due to its urgency. We later show that even when query semantics are not available (e.g., UDFs), Cameo improves scheduling with topology information alone. Note that upstream operators are not involved in this calculation. $C_{O_M}$ and $C_{path}$ can be calculated by profiling.

### 4.2.2 Query Awareness

Cameo can also leverage dataflow semantics, i.e., knowledge of user-specified commands inside the operators. This enables the scheduler to identify messages which can tolerate further delay without violating latency constraints. This is common for windowed operations, e.g., a `WindowAggregation` operator can tolerate delayed execution if a message's logical time is at the start of the window as the operator will only produce output at the end of a window. Window operators are very common in our production use cases.

**Multiple-operator dataflow, Windowed operator.** Consider $M$ that targets a windowed operator $o_M$, Cameo is able to determine (based on dataflow semantics) to what extent $M$ can be delayed without affecting latency. This requires Cameo to identify the minimum logical time ($p_{M_F}$) required to trigger the target window operator. We call $p_{M_F}$ *frontier progress*. Frontier progress denotes the stream progress that needs to be observed at the window operator before a window is complete. Thus a windowed operator will not produce output until frontier progresses are observed at all source operators. We record the system time when all frontier progresses become available at all sources as *frontier time*, denoted as $t_{M_F}$.

Processing of a message $M$ can be safely delayed until all the messages that belong in the window have arrived. In other words when computing the start deadline of $M$, we can extend the deadline by $(t_{M_F} - t_M)$. We thus rewrite Equation 2 as:

$$ddl_M = \mathbf{t_{M_F}} + L - C_{O_M} - C_{path} \tag{3}$$

An example of this schedule was shown in schedule "d" of Figure 4. With query-awareness, scheduler derives $t_{M_F}$ and postpones $M1$ and $M3$ in favor of $M2$ and $M4$. Therefore operator $o2$ is prioritized over $o1$ to process $M2$ then $M4$.

The above examples show the derivation of priority for a LLF scheduler. Cameo also supports scheduling policies including commonly used policies like EDF, SJF etc. In fact, the priority for EDF can be derived by a simple modification of the LLF equations. Our EDF policy considers the deadline of a message prior to an operator executing and thus we can compute priority for EDF by omitting $C_{O_M}$ term in Equation 3. For SJF we can derive the priority by setting $ddl_M = C_{O_M}$—while SJF is not deadline-aware we compare its performance to other policies in our evaluation.

## 4.3 Mapping Stream Progress

For Equation 3 frontier time $t_{M_F}$ may not be available until the target operator is triggered. However, for many fixed-sized window operations (e.g., `SlidingWindow`, `TumblingWindow`, etc.), we can *estimate* $t_{M_F}$ based on the message's logical time

$p_M$. Cameo performs two steps: first we apply a TRANSFORM function to calculate $p_{M_F}$, the logical time of the message that triggers $o_M$. Then, Cameo infers the frontier time $t_{M_F}$ using a PROGRESSMAP function. Thus $t_{M_F}$ = PROGRESSMAP(TRANSFORM($p_M$)). We elaborate below.

**Step 1 (Transform):** For a windowed operator, the completion of a window at operator $o$ triggers a message to be produced at this operator. Window completion is marked by the increment of window ID [61,62], calculated using the stream's logical time. For message $M$ that is sent from upstream operator $o_u$ to downstream operator $o_d$, $p_{M_F}$ can be derived using $p_M$ using on a TRANSFORM function. With the definition provided by [62], Cameo defines TRANSFORM as:

$$p_{M_F} = \text{TRANSFORM}(p_M) = \begin{cases} (p_M / S_{o_d} + 1) \cdot S_{o_d} & S_{o_u} < S_{o_d} \\ p_M & \text{otherwise} \end{cases}$$

For a sliding window operator $o_d$, $S_{o_d}$ refers to the *slide size*, i.e., value step (in terms of logical time) for each window completion to trigger target operator. For the tumbling window operation (i.e., windows cover consecutive, non-overlapping value step), $S_{o_u}$ equals the window size. For a message sent by an operator $o_u$ that has a shorter slide size than its targeting operator $o_d$, $p_{M_F}$ will be increased to the logical time to trigger $o_d$, that is, $= (p_M / S_{o_d} + 1) \cdot S_{o_d}$.

For example if we have a tumbling window with window size 10 s, then the expected frontier progress, i.e., $p_{M_F}$, will occur every 10th second (1, 11, 21 ...). Once the window operator is triggered, the logical time of the resultant message is set to $p_{M_F}$, marking the latest time to influence a result.

**Step 2 (ProgressMap):** After deriving the frontier progress $p_{M_F}$ that triggers the next dataflow output, Cameo then estimates the corresponding frontier time $t_{M_F}$. A temporal data stream typically has its logical time defined in one of three different time domains:

(1) *event time* [3,6]: a totally-ordered value, typically a timestamp, associated with original data being processed;

(2) *processing time*: system time for processing each operator [19]; and

(3) *ingestion time*: the system time of the data first being observed at the entry point of the system [3,6].

Cameo supports both event time and ingestion time. For processing time domain, $M$'s timestamp could be generated when $M$ is observed by the system.

To generate $t_{M_F}$ based on progress $p_{M_F}$, Cameo utilizes a PROGRESSMAP function to map logical time $p_{M_F}$ to physical time $t_{M_F}$. For a dataflow that defines its logical time by data's ingestion time, logical time of each event is defined by the time when it was observed. Therefore, for *all* messages that occur in the resultant dataflow, logical time is assigned by the system at the origin as $t_{M_F} = \text{PROGRESSMAP}(p_{M_F}) = p_{M_F}$.

For a dataflow that defines its logical time by the data's event time, $t_{M_F} \neq p_{M_F}$. Our stream processing run-time provides channel-wise guarantee of in-order processing for all target operators. Thus Cameo uses linear regression to map $p_{M_F}$ to $t_{M_F}$, as: $t_{M_F} = \text{PROGRESSMAP}(p_{M_F}) = \alpha \cdot p_{M_F} + \gamma$, where $\alpha$ and $\gamma$ are parameters derived via a linear fit with running window of historical $p_{M_F}$'s towards their respective $t_{M_F}$'s. E.g., For same tumbling window with window size 10s, if $p_{M_F}$ occurs at times $(1, 11, 21 \ldots)$, with a 2s delay for the event to reach the operator, $t_{M_F}$ will occur at times $(3, 13, 23 \ldots)$.

We use a linear model due to our production deployment characteristics: the data sources are largely real time streams, with data ingested soon after generation. Users typically expect events to affect results within a constant delay. Thus the logical time (event produced) and the physical time (event observed) are separated by only a small (known) time gap. When an event's physical arrival time cannot be inferred from stream progress, we treat windowed operators as regular operators. Yet, this conservative estimate of laxity does not hurt performance in practice.

## 5 Scheduling Mechanisms in Cameo

We next present Cameo's architecture that addresses three main challenges:

1 How to make static and dynamic information from both upstream and downstream processing available during priority assignment?

2 How can we efficiently perform fine-grained priority assignment and scheduling that scales with message volume?

3 How can we admit pluggable scheduling policies without modifying the scheduler mechanism?

Our approach to address the above challenges is to separate out the priority assignment from scheduling, thus designing a two-level architecture.This allows priority assignment for user-defined operators to become programmable.To pass information between the two levels (and across different operators) we piggyback information atop messages passed between operators.

More specifically, Cameo addresses challenge 1 by propagating *scheduling contexts* with messages. To meet challenge 2, Cameo uses a two-layer scheduler architecture. The top layer, called the *context converter*, is embedded into each operator and handles scheduling contexts whenever the operator sends or receives a message. The bottom layer, called the *Cameo scheduler*, interprets message priority based on the scheduling context embedded within a message and updates a priority-based data structure for both operators and operators' messages. Our design has advantages of: (i) avoiding the bottleneck of having a centralized scheduler thread calculate priority for each operator upon arrival of messages, and (ii) only limiting priority to be per-message. This allow the operators, dataflows, and the scheduler, to all remain stateless.

To address 3 Cameo allows the priority generation process to be implemented through the context handling API. A context converter invokes the API with each operator.

## 5.1 Scheduling Contexts

Scheduling contexts are data structures attached to messages, capturing message priority, and information required to perform priority-based scheduling. Scheduling contexts are *created*, *modified*, and *relayed* alongside their respective messages. Concretely, scheduling contexts allow capture of scheduling states of both upstream and downstream execution. A scheduling context can be seen and modified by both context converters and the Cameo scheduler. There are two kinds of contexts:

1. **Priority Context** (PC): PC is necessary for the scheduler to infer the priority of a message. In Cameo PCs are defined to include local and global priority as (*ID*, $PRI_{local}$, $PRI_{global}$, $Dataflow\_DefinedField$). $PRI_{local}$ and $PRI_{global}$ are used for applications to enclose message priorities for scheduler to determine execution order, and $Dataflow\_DefinedField$ includes upstream information required by the pluggable policy to generate message priority.

A PC is attached to a message before the message is sent. It is either created at a source operator upon receipt of an event, or inherited and modified from the upstream message that triggers the current operator. Therefore, a PC is seen and modified by all executions of upstream operators that lead to the current message. This enables PC to address challenge 1 by capturing information of dependant upstream execution (e.g., stream progress, latency target, etc.).

2. **Reply Context** (RC): RC meets challenge 1 by capturing periodic feedback from the downstream operators. RC is attached to an acknowledgement message [2], sent by the target operator to its upstream operator after a message is received. RCs provide processing feedback of the target operator and all its downstream operators. RCs can be aggregated and relayed recursively upstream through the dataflow.

Cameo provides a programmable API to implement these scheduling contexts and their corresponding policy handlers in context converters. API functions include:

1. **function** BUILDCXTATSOURCE(EVENT *e*) that creates a PC upon receipt of an event *e*;

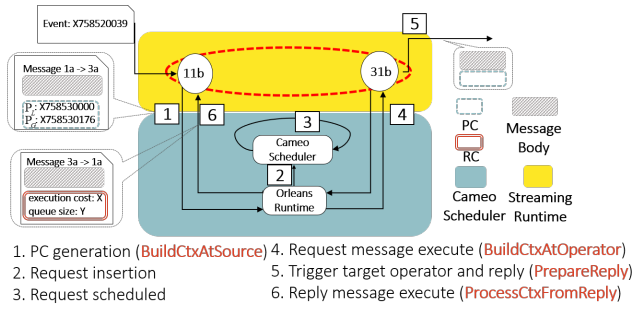
2. **function** BUILDCXTATOPERATOR(MESSAGE *M*) that modifies and propagates a PC when an operator is invoked (by *M*) and ready to send a message downstream;

3. **function** PROCESSCTXFROMREPLY(MESSAGE *r*) that processes RC attached to an acknowledgement message *r* received at upstream operator; and

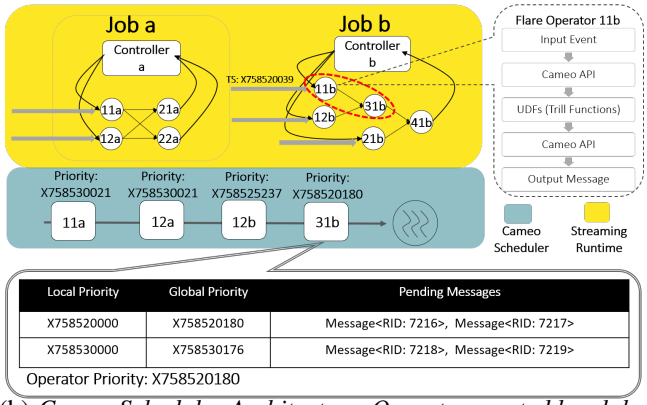
4. **function** PREPAREREPLY(MESSAGE *r*) that generates RC containing user-defined feedbacks, attached to *r* sent by a downstream operator.

## 5.2 System Architecture

Figure 5(a) shows context converters at work. After an event is generated at a source operator 1*a* (step 1), the converter creates a PC through BUILDCXTATSOURCE and sends the message to Cameo scheduler. The target operator is scheduled (step 2) with the priority extracted from the PC, before it is executed. Once the target operator 3*a* is triggered (step 4), it calls BUILDCTXATOPERATOR, modifying and relaying *PC* with its message to downstream operators. After that 3*a* sends an acknowledgement message with an RC (through PREPAREREPLY) back to 1*a* (step 5). RC is then populated by the scheduler with runtime statistics (e.g, CPU time, queuing delays, message queue sizes, network transfer time, etc.) before it is scheduled and delivered at the source operator (step 6).




(a) *Scheduling contexts circulating between two operators.*




(b) *Cameo Scheduler Architecture. Operators sorted by global priority. Messages at an operator sorted by local priority.*

Figure 5: *Cameo Mechanisms.*

Cameo enables scheduling states to be managed and transported alongside the data. This allows Cameo to meet challenge 2 by keeping the scheduler away from centralized state maintenance and priority generation. The Cameo scheduler manages a two level priority-based data structure, shown in Figure 5(b). We use $PRI_{local}$ to determine *M*'s execution priority within its target operator, and $PRI_{global}$ of the next message in an operator to order all operators that have pending messages. Cameo can schedule at either message granularity or a coarser time quanta. While processing a message, Cameo *peeks* at the priority of the next operator in the queue. If the next operator has higher priority, we swap with the current operator after a fixed time quantum (tunable).

[2] A common approach used by many stream processing systems [27, 57, 83] to ensure processing correctness

**Algorithm 1** Priority Context Conversion

**function** BUILDCXTATSOURCE(EVENT $e$) ▷ Generate PC for message $M_e$ at source triggered by event $e$
  $\texttt{PC}(M_e) \leftarrow$ INITIALIZEPRIORITYCONTEXT()
  $\texttt{PC}(M_e).(PRI_{local}, PRI_{global}) \leftarrow (e.p_e, e.t_e)$
  $\texttt{PC}(M_e) \leftarrow$ CONTEXTCONVERT($\texttt{PC}(M_e), \texttt{RC}_{\texttt{local}}$)
  **return** $\texttt{PC}(M_e)$
**function** BUILDCXTATOPERATOR(MESSAGE $M_n$) ▷ Generate PC for message $M_d$ at an intermediate operator triggered by upstream message $M_u$
  $\texttt{PC}(M_d) \leftarrow \texttt{PC}(M_u)$
  $\texttt{PC}(M_d).(PRI_{local}, PRI_{global}) \leftarrow \texttt{PC}(M_u).(p_{M_F}, t_{M_F})$
  $\texttt{PC}(M_d) \leftarrow$ CONTEXTCONVERT($\texttt{PC}(M_d), \texttt{RC}_{\texttt{local}}$)
  **return** $\texttt{PC}(M_d)$
**function** CXTCONVERT($\texttt{PC}(M), \texttt{RC}$) ▷ Calculating message priority based on $\texttt{PC}(M), \texttt{RC}$ provided
  $p_{M_F} \leftarrow$ TRANSFORM($\texttt{PC}(M).p_M$)
  $t_{M_F} \leftarrow$ PROGRESSMAP($p_{M_F}$) ▷ As in Section 4.3
  **if** $t_{M_F}$ defined in stream event time **then**
    PROGRESSMAP.UPDATE($\texttt{PC}.t_M, \texttt{PC}.p_M$) ▷ Improving prediction model as in Section 5.3
  $\texttt{PC}(M).p_M, \texttt{PC}(M).t_M \leftarrow p_{M_F}, t_{M_F}$
  $ddl_M \leftarrow t_{M_F} + \texttt{PC}(M).L - \texttt{RC}.C_m - \texttt{RC}.C_{path}$
  $\texttt{PC}(M).(PRI_{local}, PRI_{global}) \leftarrow (p_{M_F}, ddl_M)$
**function** PROCESSCTXFROMREPLY(MESSAGE $r$) ▷ Retrieve reply message's RC and store locally
  $\texttt{RC}_{\texttt{local}}.update(r.\texttt{RC})$
**function** PREPAREREPLY(MESSAGE $r$) ▷ Recursively update maximum critical path cost $C_{path}$ before reply
  **if** SENDER($r$) = Sink **then**
    $r.\texttt{RC} \leftarrow$ INITIALIZEREPLYCONTEXT()
  **else** $r.\texttt{RC}.C_{path} \leftarrow \texttt{RC}.C_m + \texttt{RC}.C_{path}$

## 5.3 Implementing the Cameo Policy

To implement the scheduling policy of Section 4, a PC is attached to message $M$ (denoted as $PC(M)$) with these fields:

| $ID$ | $PRI_{local}$ | $PRI_{global}$ | $Dataflow-DefinedField$ |
|---|---|---|---|
| $ID_M$ | $p_{M_F}$ | $ddl_{M_F}$ | $(p_{M_F}, t_{M_F}, L)$ |

The core of Algorithm 1 is CXTCONVERT, which generates PC for downstream message $M_d$ (denoted as $\texttt{PC}(M_d)$), triggered by $\texttt{PC}(M_u)$ from the upstream triggering message. To schedule a downstream message $M_d$ triggered by $M_u$, Cameo first retrieves stream progress $p_{M_u}$ contained in $\texttt{PC}(M_u)$. It then applies the two-step process (Section 4.3) to calculate frontier time $t_{M_F}$ using $p_{M_u}$. This may extend a message's deadline if the operator is not expected to trigger immediately (e.g., windowed operator). We capture $p_{M_F}$ and estimated $t_{M_F}$ in PC as message priority and propagate this downstream. Meanwhile, $p_{M_u}$ and $t_{M_u}$ are fed into a linear model to improve future prediction towards $t_{M_F}$. Finally, the context converter computes message priority $ddl_{M_u}$ using $t_{M_F}$ as described in Section 4.

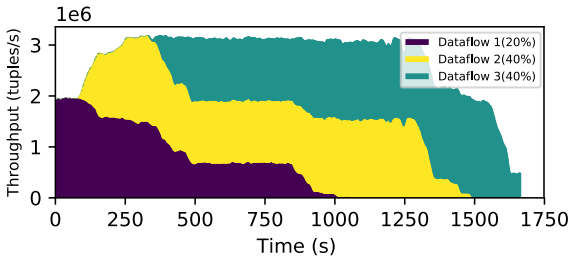


Figure 6: *Proportional fair sharing using Cameo.*

Cameo utilizes RC to track critical path execution cost $C_{path}$ and execution cost $C_{o_M}$. RC contains the processing cost (e.g., CPU time) of the downstream critical path up to the current operator, obtained via profiling.

## 5.4 Customizing Cameo: Proportional Fair Scheduling

We next show how the pluggable scheduling policy in Cameo can be used to support other performance objectives, thus satisfying 3. For instance, we show how a token-based rate control mechanism works, where token rate equals desired output rate. In this setting, each application is granted tokens per unit of time, based on their target sending rate. If a source operator exceeds its target sending rate, the remaining messages (and all downstream traffic) are processed with operator priority reduced to minimum. When capacity is insufficient to meet the aggregate token rate, all dataflows are downgraded equally. Cameo spreads tokens proportionally across the next time interval (e.g., 1 sec) by tagging each token with the timestamp at each source operator. For token-ed messages, we use token tag $PRI_{global}$, and interval ID as $PRI_{local}$. Messages without tokens have $PRI_{global}$ set to MIN_VALUE. Through PC propagation, all downstream messages are processed when no tokened traffic is present.

Figure 6 shows Cameo's token mechanism. Three dataflows start with 20% (12), 40% (24), and 40% (24) tokens as target ingestion rate per source respectively. Each ingests 2M events/s, starting 300 s apart, and lasting 1500 s. Dataflow 1 receives full capacity initially when there is no competition. The cluster is at capacity after Dataflow 3 arrives, but Cameo ensures token allocation translates into throughput shares.

# 6 Experimental Evaluation

We next present experimental evaluation of Cameo. We first study the effect of different queries on Cameo in a single-tenant setting. Then for multi-tenant settings, we study Cameo's effects when:

- **Varying environmental parameters** (Section 6.2): This includes: a) workload (tenant sizes and ingestion rate), and b) available resources, i.e., worker thread pool size, c) workload bursts.
- **Tuning internal parameters and optimization** (Section 6.3): We study: a) effect of scheduling granularity, b)

frontier prediction for event time windows, and c) starvation prevention.

We implement streaming queries in Flare [68] (built atop Orleans [10, 25]) by using Trill [30] to run streaming operators.We compare Cameo vs. both i) default **Orleans** (version 1.5.2) scheduler, and ii) a custom-built **FIFO** scheduler. By default, we use the 1 ms minimum re-scheduling grain (Section 5.2). This grain is generally shorter than a message's execution time. Default Orleans implements a global run queue of messages using a ConcurrentBag [9] data structure. ConcurrentBag optimizes processing throughput by prioritizing processing thread-local tasks over the global ones. For the FIFO scheduler, we insert operators into the global run queue and extract them in FIFO order. In both approaches, an operator processes its messages in FIFO order.

**Machine configuration.** We use DS12-v2 Azure virtual machines (4 vCPUs/56GB memory/112G SSD) as server machines, and DS11-v2 Azure virtual machines (2 vCPUs/14GB memory/28G SSD) as client machines [12]. Single-tenant scenarios are evaluated on a single server machine. Unless otherwise specified, all multi-tenant experiments are evaluated using a 32-node Azure cluster with 16 client machines.

**Evaluation workload.** For the multi-job setting we study performance isolation under concurrent dataflow jobs. Concretely, our workload is divided into two control groups:

- **Latency Sensitive Jobs (Group 1 ):** This is representative of jobs connected to user dashboards, or associated with SLAs, ongoing advertisement campaigns, etc. Our workload jobs in Group 1 have sparse input volume across time (1 msg/s per source, with 1000 events/msg), and report periodic results with shorter aggregation windows (1 second). These have strict latency constraints.
- **Bulk Analytic Jobs (Group 2):** This is representative of social media streams being processed into longer-term analytics with longer aggregation windows (10 seconds). Our Group 2 jobs have input of both higher and variable volume and high speed, but with lax latency constraints.

Our queries feature multiple stages of windowed aggregation parallelized into a group of operators. Each job has 64 client sources. All queries assume input streams associated with event time unless specified otherwise.

**Latency constraints.** In order to determine the latency constraint of one job, we run multiple instances of the job until the resource (CPU) usage reaches 50%. Then we set the latency constraint of the job to be twice the tail (95th percentile) latency. This emulates the scenario where users with experience in isolated environments deploy the same query in a shared environment by moderately relaxing the latency constraint. Unless otherwise specified, a latency target is marked with grey dotted line in the plots.

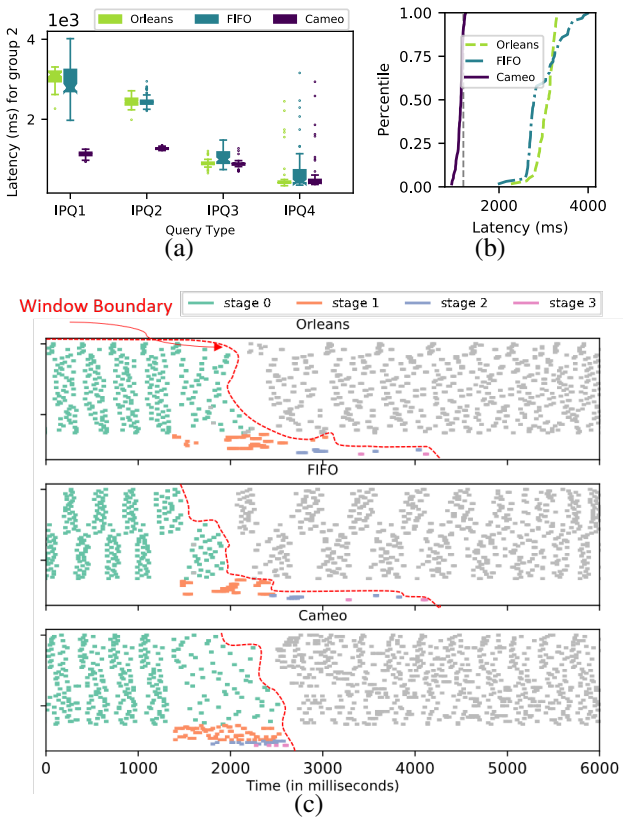


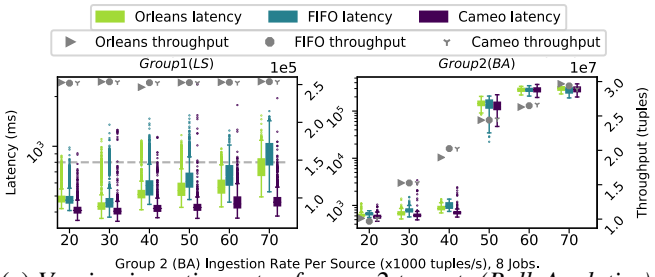

Figure 7: *Single-Tenant Experiments: (a) Query Latency. (b) Latency CDF. (c) Operator Schedule Timeline: X axis = time when operator was scheduled. Y axis = operator ID color coded by operator's stage. Operators are triggered at each stage in order (stage 0 to 3). Job latency is time from all events that belong to the previous window being received at stage 0, until last message is output at stage 3.*

## 6.1 Single-tenant Scenario

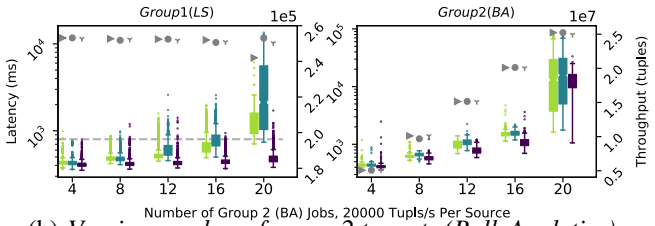
In Figure 7 we evaluate a single-tenant setting with 4 queries: IPQ1 through IPQ4. IPQ1 and IPQ3 are periodic and they respectively calculate sum of revenue generated by real time ads, and the number of events generated by jobs groups by different criteria. IPQ2 performs similar aggregation operations as IPQ1 but on a sliding window (i.e., consecutive window contains overlapped input). IPQ4 summarizes errors from log events via running a windowed join of two event stream, followed by aggregation on a tumbling window (i.e., where consecutive windows contain non-overlapping ranges of data that are evenly spaced across time).

From Figure 7(a) we observe that Cameo improves median latency by up to 2.7×and tail latency by up to 3.2×.We also observe that default Orleans performs almost as well as Cameo for IPQ4. This is because IPQ4 has a higher execution time with heavy memory access, and performs well when pinned to a single thread with better access locality.

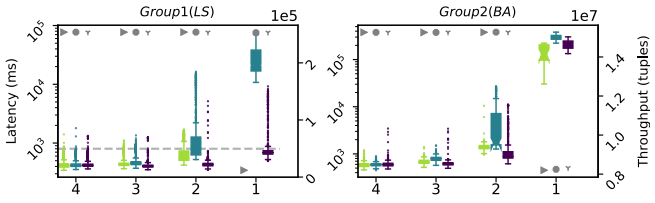
**Effect on intra-query operator scheduling.** The CDF in




(a) *Varying ingestion rate of group 2 tenants (Bulk Analytics).*




(b) *Varying number of group 2 tenants (Bulk Analytics).*




(c) *Varying worker thread pool size.*

Figure 8: *Latency-sensitive jobs under competing workloads.*

Figure 7(b) shows that Orleans' latency is about 3× higher than Cameo. While FIFO has a slightly lower median latency, its tail latency is as high as in Orleans.

Cameo's prioritization is especially evident in Figure 7(c), where dots are message starts, and red lines separate windows. We first observe that Cameo is faster, and it creates a clearer boundary between windows. Second, messages that contribute to the first result (colored dots) and messages that contribute to the second result (grey dots) do not overlap on the timeline. For the other two strategies, there is a *drift* between stream progress in early stages vs. later stages, producing a prolonged delay. In particular, in Orleans and FIFO, early-arriving messages from the next window are executed *before* messages from the first window, thus missing deadlines.

## 6.2 Multi-tenant Scenario

Figure 8 studies a control group of latency-constrained dataflows (group 1 LS jobs) by fixing both job count and data ingestion rate. We vary data volume from competing workloads (group 2 BA jobs) and available resources. For LS jobs we impose a latency target of 800 ms, while for BA jobs we use a 7200s latency constraint.

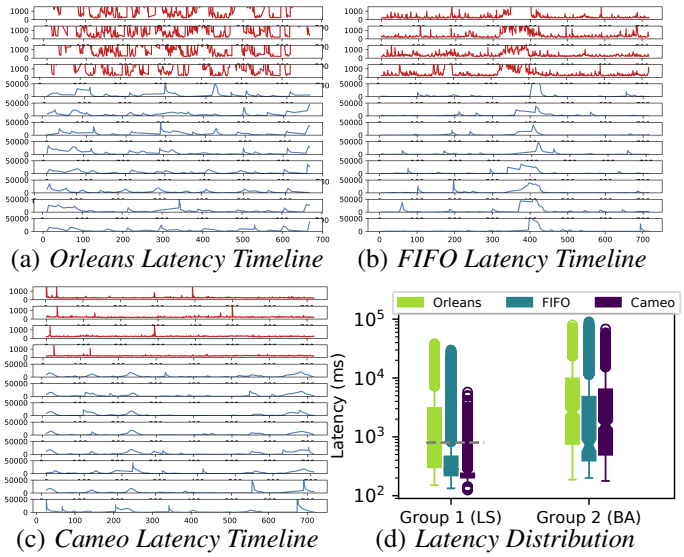
**Cameo under increasing data volume.** We run four group 1 jobs alongside group 2 jobs. We increase the competing group 2 jobs' traffic, by increasing the ingestion speed (Figure 8(a)) and number of tenants (Figure 8(b)). We observe that all three strategies (Cameo, Orleans, FIFO) are comparable up to per-source tuple rate of 30K/s in Figure 8(a), and up to twelve group 2 jobs in Figure 8(b). Beyond this, overloading causes massive latency degradation, for group 1 (LS) jobs at median and 99 percentile latency (respectively): i) Orleans is worse than Cameo by up to 1.6 and 1.5× in Figure 8(a), up to 2.2 and 2.8× in Figure 8(b), and ii) FIFO is worse than Cameo by up to 2 and 1.8× in Figure 8(a), up to 4.6 and 13.6× in Figure 8(b). Cameo stays stable. Cameo's degradation of group 2 jobs is small— with latency similar or lower than Orleans and FIFO, and Cameo's throughput only 2.5% lower.

**Effect of limited resources.** Orleans' [74] underlying SEDA architecture [86] resizes thread pools to achieve resource balance between execution steps, for dynamic re-provisioning. Figure 8(c) shows latency and throughput when we decrease the number of worker threads. Cameo maintains the performance of group 1 jobs except in the most restrictive 1 thread case (although it still meets 90% of deadlines). Cameo prefers messages with impending deadlines and this causes back-pressure for jobs with less-restrictive latency constraints, lowering throughput. Both Orleans and FIFO observe large performance penalties for group 1 and 2 jobs (higher in the former). Group 2 jobs with much higher ingestion rate will naturally receive more resources upon message arrivals, leading to back-pressure and lower throughput for group 1 jobs.

**Effect of temporal variation of workload.** We use a Pareto distribution for data volume in Figure 9, with four group 1 jobs and eight group 2 jobs. (This is based on Figures 2(a), 2(c), which showed a Power-Law-like distribution.) The cluster utilization is kept under 50%.

High ingestion rate can suddenly prolong queues at machines. Visualizing timelines in Figures 9(a), 9(b), and 9(c) shows that for latency-constrained jobs (group 1), Cameo's latency is more stable than Orleans' and FIFO's. Figure 9(d) shows that Cameo reduces (median, 99th percentile) latency by (3.9×, 29.7×) vs. Orleans, and (1.3×, 21.1×) vs. FIFO. Cameo's standard deviation is also lower by 23.2× and 12.7× compared to Orleans and FIFO respectively. For group 2, Cameo produces smaller average latency and is less affected by ingestion spikes. Transient workload bursts affect many jobs, e.g., all jobs around $t = 400$ with FIFO, as a spike at one operator affects all its collocated operators.

**Ingestion pattern from production trace.** Production workloads exhibit high degree of skew across data sources. In Figure 10 we show latency distribution of dataflows consuming two workload distributions derived from Figure 2(c): Type 1 and 2. Type 1 produces twice as many events as Type 2. However, Type 2 is heavily skewed and its ingestion rate varies by 200× across sources. This heavily impacts operators that are collocated. The success rate (i.e., the fraction of outputs that meet their deadline) is only 0.2% and 1.5% for Orleans and 7.9% and 9.5% for FIFO. Cameo prioritizes critical messages, maintaining success rates of 21.3% and 45.5% respectively.




(a) *Orleans Latency Timeline* (b) *FIFO Latency Timeline*
(c) *Cameo Latency Timeline* (d) *Latency Distribution*

Figure 9: *Latency under Pareto event arrival.*

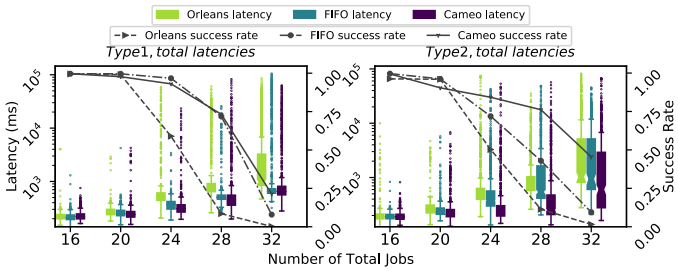



Figure 10: *Spatial Workload Variation.*

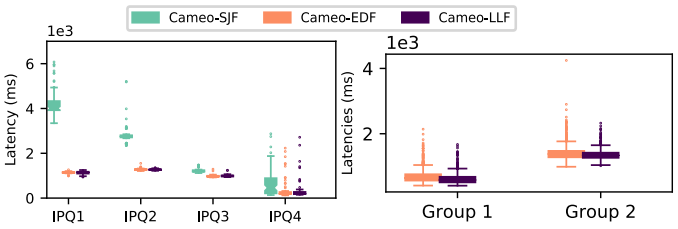



Figure 11: *Cameo Policies. Left: Single query latency distribution. Right: Multi-Query Latency Distribution.*

## 6.3 Cameo: Internal Evaluation

We next evaluate Cameo's internal properties.

**LLF vs. EDF vs. SJF.** We implement three scheduling policies using the Cameo context API and evaluate using Section 6.1's workload. The three scheduling policies are: Least Laxity First (LLF, our default), Earliest Deadline First (EDF), and Shortest Job First (SJF). Figure 11 shows that SJF is consistently worse than LLF and EDF (with the exception of query IPQ4– due to the lack of queuing effect under lighter workload). Second, EDF and LLF perform comparably.

In fact we observed that EDF and LLF produced similar schedules for most of our queries. This is because: i) our operator execution time is consistent within a stage, and ii) operator execution time is $\ll$ window size. Thus, excluding operator cost (EDF) does not change schedule by much.

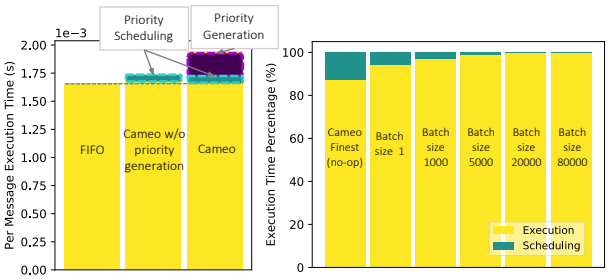



Figure 12: *Cameo Scheduling Overhead.*

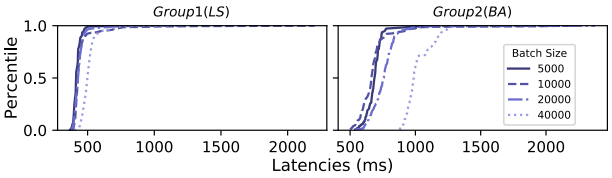



Figure 13: *Effect of Batch Size.*

**Scheduling Overhead.** To evaluate Cameo with many small messages, we use one thread to run a no-op workload (300-350 tenants, 1 msg/s/tenant, same latency needs). Tenants are increased to saturate throughput.

Figure 12 (left) shows breakdown of execution time (inverse of throughput) for three scheduling schemes: FIFO, Cameo without priority generation (overhead only from priority scheduling), and Cameo with priority generation and the LLF policy from Section 4 (overhead from both priority scheduling and priority generation). Cameo's scheduling overhead is $< 15\%$ of processing time in the worst case, comprising of 4% overhead from priority-based scheduling and 11% from priority generation.

In practice, Cameo encloses a columnar batch of data in each message like Trill [30]. Cameo's overhead is small compared to message execution costs. In Figure 12 (right), under Section 6's workload, scheduling overhead is only 6.4% of execution time for a local aggregation operator with batch size 1. Overhead falls with batch size. When Cameo is used as a generalized scheduler and message execution costs are small (e.g., with $< 1$ ms), we recommend tuning scheduling quantum and message size to reduce scheduling overhead.

In Figure 13, we batch more tuples into a message, while maintaining same overall tuple ingestion rate. In spite of decreased flexibility available to the scheduler, group 1 jobs' latency is unaffected up to 20K batch size. It degrades at higher batch size (40K), due to more lower priority tuples blocking higher priority tuples. Larger messages hide scheduling overhead, but could starve some high priority messages.

To evaluate the effect of increasing the scheduling quantum, we evaluate Cameo's performance under varying scheduling quantum (Section 5.2) using workload described in Figure 10. Figure 14 (Left) shows the latency distribution with

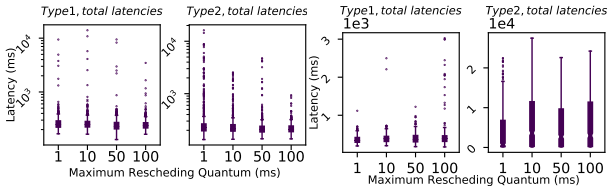



Figure 14: *Effect of Varying Scheduling Quantum. Left: Jobs Triggered By Clustered Stream Progress. Right: Jobs Triggered By Interleaved Stream Progress.*

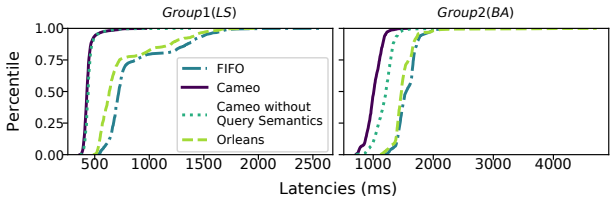



Figure 15: *Benefit of Query Semantics-awareness in Cameo.*

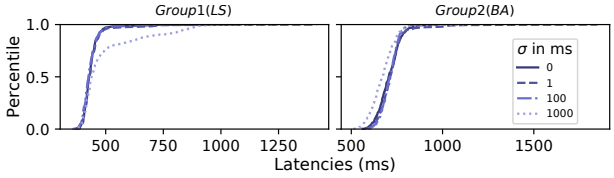



Figure 16: *Profiling Inaccuracy. Standard deviation in ms.*

*all* Type 1 and Type 2 jobs trigger dataflow output on the *same* stream progress (e.g., 10, 20, 30s ...), whereas Figure 14 (Right) shows the result with jobs's output triggered on *interleaved* stream progress (e.g., job 1 on 10, 20, 30s etc., job 2 on 12, 22, 32s, etc.). The left figure reveals the potential benefits of using coarser scheduling quantum, as resources are contended for by many high priority messages, using finest scheduling granularity causes longer latency tail due to frequent context switches. However, a very large scheduling quantum (100ms) can hurt Cameo's performance by prohibiting the scheduler from preempting low-priority operators that arrive early, creating head-of-line blocking for high priority messages.

**Varying Scope of Scheduler Knowledge.** If Cameo is unaware of query semantics (but aware of DAG and latency constraints), Cameo conservatively estimates $t_{M_F}$ without deadline extension for window operators, causing a tighter $ddl_M$. Figure 15 shows that Cameo performs slightly worse without query semantics (19% increase in group 2 median latency). Against baselines, Cameo still reduces group 1 and group 2's median latency by up to 38% and 22% respectively. Hence, even without query semantic knowledge, Cameo still outperforms Orleans and FIFO.

**Effect of Measurement Inaccuracies.** To evaluate how Cameo reacts to inaccurate monitoring profiles, we perturb measured profile costs ($C_{O_M}$ from Equation 3) by a normal distribution ($\mu$=0), varying standard deviation ($\sigma$) from 0 to 1 s. Figure 16 shows that when $\sigma$ of perturbation is close to window size (1 s), latency is: i) stable at the median, and ii) modestly increases at tail, e.g., only by 55.5% at the 90th percentile. Overall, Cameo's performance is robust when standard deviation is $\leq$ 100ms, i.e., when measurement error is reasonably smaller than output granularity.

## 7 Related Work

**Streaming system schedulers.** The first generation of Data Stream Management Systems (DSMS) [15, 32], such as Aurora [28], Medusa [23] and Borealis [14], use QoS based control mechanisms with load shedding to improve query performance at run time. These are either centralized (single-threaded) [28], or distributed [14, 23] but do not handle timestamp-based priorities for partitioned operators. TelegraphCQ [31] orders input tuples before query processing [21, 79], while Cameo addresses operator scheduling within and across query boundaries. Stanford's STREAM [71] uses chain scheduling [22] to minimize memory footprints and optimize query queues, but assumes all queries and scheduler are execute in a single-thread. More recent works in streaming engines propose operator scheduling algorithms for query throughput [20] and latency [41, 64]. Reactive and operator-based policies include [20, 64], while [41] assumes arrivals are periodic or Poisson—however, these works do not build a framework (like Cameo), nor do they handle per-event semantic awareness for stream progress.

Modern stream processing engines such as Spark Streaming [90], Flink [27], Heron [57], MillWheel [18], Naiad [72], Muppet [59], Yahoo S4 [73]) do not include *native* support for multi-tenant SLA optimization. These systems also rely on coarse-grained resource sharing [13] or third-party resource management systems such as YARN [84] and Mesos [47].

**Streaming query reconfiguration.** Online reconfiguration has been studied extensively [48]. Apart from Figure 3, prior work addresses operator placement [39, 76], load balancing [56, 66], state management [29], policies for scale-in and scale-out [44–46, 67]. Among these are techniques to address latency requirements of dataflow jobs [44, 67], and ways to improve vertical and horizontal elasticity of dataflow jobs in containers [87]. The performance model in [60] focuses on dataflow jobs with latency constraints, while we focus on interactions among operators. Online elasticity was targeted by System S [40, 80], StreamCloud [42] and TimeStream [78]. Others include [35, 53]. Neptune [38] is a proactive scheduler to suspend low-priority batch tasks in the presence of streaming tasks. Yet, there is no operator prioritization *within* each application. Edgewise [37] is a queue-based scheduler based on operator load but not query semantics. All these works are orthogonal to, and can be treated as pluggables in, Cameo.

**Event-driven architecture for real-time data processing.** This area has been popularized by the resource efficiency of serverless architectures [4, 5, 7]. Yet, recent proposals [17, 26, 58, 70] for stream processing atop event-based frameworks

do not support performance targets for streaming queries.

## 8 Conclusion

We proposed Cameo, a fine-grained scheduling framework for distributed stream processing. To realize flexible per-message scheduling, we implemented a stateless scheduler, contexts that carry important static and dynamic information, and mechanisms to derive laxity-based priority from contexts. Our experiments with real workloads, and on Microsoft Azure, showed that Cameo achieves $2.7\times - 4.6\times$ lower latency than competing strategies and incurs overhead less than 6.4%.

## Acknowledgements

We thank Matei Zaharia and our anonymous referees at NSDI 2020 for their reviews and help with improving the paper. We thank Kai Zeng for providing feedbacks for initial ideas. This work was supported in part by the following grants: NSF IIS 1909577, NSF CNS 1908888, NSF CNS 1319527, NSF CNS 1838733, a Facebook faculty research award, the Helios project at Microsoft [77], and another generous gift from Microsoft. Shivaram Venktaraman is also supported by the Office of the Vice Chancellor for Research and Graduate Education with funding from the Wisconsin Alumni Research Foundation. We are grateful to the Cosmos, Azure Data Lake, and PlayFab teams at Microsoft.

## References

[1] Akka. https://akka.io/.

[2] Apache Hadoop. https://hadoop.apache.org/.

[3] Apache Kafka Core Concepts. https://kafka.apache.org/11/documentation/streams/core-concepts.

[4] AWS Lambda. https://aws.amazon.com/lambda/.

[5] Azure Functions. https://azure.microsoft.com/en-us/services/functions/.

[6] Flink Time Attribute. https://ci.apache.org/projects/flink/flink-docs-release-1.7/dev/event_time.html.

[7] Google Cloud Functions. https://cloud.google.com/functions.

[8] Kubernetes: Production-Grade Container Orchestration. https://kubernetes.io/.

[9] .NET ConcurrentBag. https://docs.microsoft.com/en-us/dotnet/api/system.collections.concurrent.concurrentbag-1?view=netframework-4.8.

[10] Orleans. https://dotnet.github.io/orleans/.

[11] Serverless Streaming Architectures and Best Practices, amazon web services. https://d1.awsstatic.com/whitepapers/Serverless_Streaming_Architecture_Best_Practices.pdf.

[12] Sizes for Windows virtual machines in Azure. https://docs.microsoft.com/en-us/azure/virtual-machines/windows/sizes.

[13] Storm Multitenant Scheduler. https://storm.apache.org/releases/current/javadocs/org/apache/storm/scheduler/multitenant/package-summary.html.

[14] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, et al. The design of the Borealis stream processing engine. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, volume 5, pages 277–289, 2005.

[15] Daniel J Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: a new model and architecture for data stream management. *Proceedings of the VLDB Endowment*, 12(2):120–139, 2003.

[16] Lior Abraham, John Allen, Oleksandr Barykin, Vinayak Borkar, Bhuwan Chopra, Ciprian Gerea, Daniel Merl, Josh Metzler, David Reiss, Subbu Subramanian, et al. Scuba: diving into data at facebook. *Proceedings of the VLDB Endowment*, 6(11):1057–1067, 2013.

[17] Adil Akhter, Marios Fragkoulis, and Asterios Katsifodimos. Stateful functions as a service in action. *Proceedings of the VLDB Endowment*, 12(12):1890–1893, 2019.

[18] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. Millwheel: fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment*, 6(11):1033–1044, 2013.

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

[20] Lisa Amini, Navendu Jain, Anshul Sehgal, Jeremy Silber, and Olivier Verscheure. Adaptive control of extreme-scale stream processing systems. In *Proceedings of the IEEE 26th International Conference on Distributed Computing Systems (ICDCS)*, pages 71–71. IEEE, 2006.

[21] Ron Avnur and Joseph M Hellerstein. Eddies: Continuously adaptive query processing. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 261–272, 2000.

[22] Brian Babcock, Shivnath Babu, Rajeev Motwani, and Mayur Datar. Chain: Operator scheduling for memory minimization in data stream systems. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 253–264. ACM, 2003.

[23] Magdalena Balazinska, Hari Balakrishnan, and Michael Stonebraker. Load management and high availability in the medusa distributed stream processing system. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 929–930. ACM, 2004.

[24] Philip A Bernstein, Todd Porter, Rahul Potharaju, Alejandro Z Tomsic, Shivaram Venkataraman, and Wentao Wu. Serverless event-stream processing over virtual actors. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2019.

[25] Sergey Bykov, Alan Geller, Gabriel Kliot, James R Larus, Ravi Pandya, and Jorgen Thelin. Orleans: Cloud computing for everyone. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, page 16. ACM, 2011.

[26] Paris Carbone, Marios Fragkoulis, Vasiliki Kalavri, and Asterios Katsifodimos. Beyond Analytics: the Evolution of Stream Processing Systems. In *Proceedings of the 2020 ACM SIGMOD international conference on Management of data*, 2020.

[27] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache Flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[28] Don Carney, Uğur Çetintemel, Alex Rasin, Stan Zdonik, Mitch Cherniack, and Mike Stonebraker. Operator scheduling in a data stream manager. In *Proceedings of the VLDB Endowment*, pages 838–849. VLDB Endowment, 2003.

[29] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *Proceedings of the 2013 ACM SIGMOD international conference on Management of data*, pages 725–736. ACM, 2013.

[30] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, Robert DeLine, Danyel Fisher, John C Platt, James F Terwilliger, and John Wernsing. Trill: A high-performance incremental query processor for diverse analytics. *Proceedings of the VLDB Endowment*, 8(4):401–412, 2014.

[31] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J Franklin, Joseph M Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R Madden, Fred Reiss, and Mehul A Shah. Telegraphcq: continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 668–668. ACM, 2003.

[32] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Donald Carney, Ugur Cetintemel, Ying Xing, and Stanley B Zdonik. Scalable distributed stream processing. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, volume 3, pages 257–268, 2003.

[33] Edward G Coffman, Jr, Michael R Garey, and David S Johnson. An application of bin-packing to multiprocessor scheduling. *SIAM Journal on Computing*, 7(1):1–17, 1978.

[34] Avrilia Floratou, Ashvin Agrawal, Bill Graham, Sriram Rao, and Karthik Ramasamy. Dhalion: Self-regulating stream processing in heron. *Proceedings of the VLDB Endowment*, August 2017.

[35] Avrilia Floratou, Ashvin Agrawal, Bill Graham, Sriram Rao, and Karthik Ramasamy. Dhalion: self-regulating stream processing in Heron. *Proceedings of the VLDB Endowment*, 10(12):1825–1836, 2017.

[36] Tom ZJ Fu, Jianbing Ding, Richard TB Ma, Marianne Winslett, Yin Yang, and Zhenjie Zhang. DRS: dynamic resource scheduling for real-time analytics over fast streams. In *Proceedings of the 35th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 411–420. IEEE, 2015.

[37] Xinwei Fu, Talha Ghaffar, James C Davis, and Dongyoon Lee. Edgewise: a better stream processing engine for the edge. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 929–946, 2019.

[38] Panagiotis Garefalakis, Konstantinos Karanasos, and Peter Pietzuch. Neptune: Scheduling suspendable tasks for unified stream/batch applications. In *Proceedings*

*of the ACM Symposium on Cloud Computing (SoCC)*, pages 233–245, 2019.

[39] Panagiotis Garefalakis, Konstantinos Karanasos, Peter R Pietzuch, Arun Suresh, and Sriram Rao. Medea: scheduling of long running applications in shared production clusters. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys)*, pages 4–1, 2018.

[40] Buğra Gedik, Scott Schneider, Martin Hirzel, and Kun-Lung Wu. Elastic scaling for data stream processing. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1447–1463, 2014.

[41] Yu Gu, Ge Yu, and Chuanwen Li. Deadline-aware complex event processing models over distributed monitoring streams. *Mathematical and Computer Modelling*, 55(3-4):901–917, 2012.

[42] Vincenzo Gulisano, Ricardo Jimenez-Peris, Marta Patino-Martinez, Claudio Soriente, and Patrick Valduriez. Streamcloud: An elastic and scalable data streaming system. *IEEE Transactions on Parallel and Distributed Systems*, 23(12):2351–2365, 2012.

[43] Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2-3):202–220, 2009.

[44] Thomas Heinze, Zbigniew Jerzak, Gregor Hackenbroich, and Christof Fetzer. Latency-aware elastic scaling for distributed data stream processing systems. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, pages 13–22. ACM, 2014.

[45] Thomas Heinze, Valerio Pappalardo, Zbigniew Jerzak, and Christof Fetzer. Auto-scaling techniques for elastic data stream processing. In *Proceedings of the IEEE Data Engineering Workshops (ICDEW)*, pages 296–302. IEEE, 2014.

[46] Thomas Heinze, Lars Roediger, Andreas Meister, Yuanzhen Ji, Zbigniew Jerzak, and Christof Fetzer. Online parameter optimization for elastic data stream processing. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, pages 276–287. ACM, 2015.

[47] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, 2011.

[48] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. A catalog of stream processing optimizations. *ACM Computing Surveys (CSUR)*, 46(4):46, 2014.

[49] Moritz Hoffmann, Andrea Lattuada, John Liagouris, Vasiliki Kalavri, Desislava Dimitrova, Sebastian Wicki, Zaheer Chothia, and Timothy Roscoe. Snailtrail: Generalizing critical paths for online analysis of distributed dataflows. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 95–110, 2018.

[50] Moritz Hoffmann, Andrea Lattuada, and Frank McSherry. Megaphone: latency-conscious state migration for distributed streaming dataflows. *Proceedings of the VLDB Endowment*, 12(9):1002–1015, 2019.

[51] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. Cloud programming simplified: A Berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383*, 2019.

[52] Vasiliki Kalavri, John Liagouris, Moritz Hoffmann, Desislava Dimitrova, Matthew Forshaw, and Timothy Roscoe. Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 783–798, 2018.

[53] Faria Kalim, Le Xu, Sharanya Bathey, Richa Meherwal, and Indranil Gupta. Henge: Intent-driven multi-tenant stream processing. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, pages 249–262. ACM, 2018.

[54] Evangelia Kalyvianaki, Themistoklis Charalambous, Marco Fiscato, and Peter Pietzuch. Overload management in data stream processing systems with latency guarantees. In *Proceedings of the 7th IEEE International Workshop on Feedback Computing*, 2012.

[55] Evangelia Kalyvianaki, Marco Fiscato, Theodoros Salonidis, and Peter Pietzuch. Themis: Fairness in federated stream processing under overload. In *Proceedings of the 2016 International Conference on Management of Data*, pages 541–553. ACM, 2016.

[56] Wilhelm Kleiminger, Evangelia Kalyvianaki, and Peter Pietzuch. Balancing load in stream processing with the cloud. In *Proceedings of the 27th IEEE International Conference on Data Engineering Workshops*, pages 16–21. IEEE, 2011.

[57] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthikeyan Ramasamy, and Siddarth Taneja. Twitter Heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD international conference on Management of data*, 2015.

[58] Avinash Kumar, Zuozhi Wang, Shengquan Ni, and Chen Li. Amber: a debuggable dataflow system based on the actor model. *Proceedings of the VLDB Endowment*, 13(5):740–753, 2020.

[59] Wang Lam, Lu Liu, STS Prasad, Anand Rajaraman, Zoheb Vacheri, and AnHai Doan. Muppet: Mapreduce-style processing of fast data. *Proceedings of the VLDB Endowment*, 5(12):1814–1825, 2012.

[60] Boduo Li, Yanlei Diao, and Prashant Shenoy. Supporting scalable analytics with latency constraints. *Proceedings of the VLDB Endowment*, 8(11):1166–1177, 2015.

[61] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A Tucker. Semantics and evaluation techniques for window aggregates in data streams. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 311–322. ACM, 2005.

[62] Jin Li, Kristin Tufte, Vladislav Shkapenyuk, Vassilis Papadimos, Theodore Johnson, and David Maier. Out-of-order processing: a new architecture for high-performance stream systems. *Proceedings of the VLDB Endowment*, 1(1):274–288, 2008.

[63] Na Li and Qiang Guan. Deadline-aware event scheduling for complex event processing systems. In *Proceedings of the International Conference on Intelligent Data Engineering and Automated Learning*, pages 101–109. Springer, 2013.

[64] Xin Li, Zhiping Jia, Li Ma, Ruihua Zhang, and Haiyang Wang. Earliest deadline scheduling for continuous queries over data streams. In *Proceedings of the IEEE International Conference on Embedded Software and Systems*, pages 57–64. IEEE, 2009.

[65] Chung Laung Liu and James W Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.

[66] Simon Loesing, Martin Hentschel, Tim Kraska, and Donald Kossmann. Stormy: an elastic and highly available streaming service in the cloud. In *Proceedings of the 2012 Joint EDBT/ICDT Workshops*, pages 55–60. ACM, 2012.

[67] Björn Lohrmann, Peter Janacik, and Odej Kao. Elastic stream processing with latency guarantees. In *Proceedings of the 35th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 399–410. IEEE, 2015.

[68] Luo Mai, Kai Zeng, Rahul Potharaju, Le Xu, Steve Suh, Shivaram Venkataraman, Paolo Costa, Terry Kim, Saravanan Muthukrishnan, Vamsi Kuppa, et al. Chi: a scalable and programmable control plane for distributed stream processing systems. *Proceedings of the VLDB Endowment*, 11(10):1303–1316, 2018.

[69] Aloysius Ka-Lau Mok. *Fundamental design problems of distributed systems for the hard-real-time environment*. PhD thesis, Massachusetts Institute of Technology, 1983.

[70] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. Ray: A distributed framework for emerging AI applications. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 561–577, 2018.

[71] Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Manku, Chris Olston, Justin Rosenstein, and Rohit Varma. Query processing, resource management, and approximation in a data stream management system. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2003.

[72] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455. ACM, 2013.

[73] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. S4: Distributed stream computing platform. In *Proceedings of the IEEE Data Mining Workshops (ICDMW)*, pages 170–177. IEEE, 2010.

[74] Andrew Newell, Gabriel Kliot, Ishai Menache, Aditya Gopalan, Soramichi Akiyama, and Mark Silberstein. Optimizing distributed actor systems for dynamic interactive services. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 38. ACM, 2016.

[75] Zhengyu Ou, Ge Yu, Yaxin Yu, Shanshan Wu, Xiaochun Yang, and Qingxu Deng. Tick scheduling: A deadline based optimal task scheduling approach for real-time data stream systems. In *Proceedings of the International Conference on Web-Age Information Management*, pages 725–730. Springer, 2005.

[76] Peter Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh, and Margo Seltzer. Network-aware operator placement for stream-processing systems. In *Proceedings of the 22nd International Conference on Data Engineering, (ICDE)*, pages 49–49. IEEE, 2006.

[77] Rahul Potharaju, Terry Kim, Wentao Wu, Vidip Acharya, Steve Suh, Andrew Fogarty, Apoorve Dave, Sinduja Ramanujam, Tomas Talius, Lev Novik, and Raghu Ramakrishnan. Helios: Hyperscale indexing for the cloud & edge. *Proceedings of the VLDB Endowment*, 13(12), 2020.

[78] Zhengping Qian, Yong He, Chunzhi Su, Zhuojie Wu, Hongyu Zhu, Taizhi Zhang, Lidong Zhou, Yuan Yu, and Zheng Zhang. Timestream: Reliable stream computation in the cloud. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 1–14. ACM, 2013.

[79] Vijayshankar Raman, Bhaskaran Raman, and Joseph M Hellerstein. Online dynamic reordering for interactive data processing. In *VLDB*, volume 99, pages 709–720, 1999.

[80] Scott Schneider, Henrique Andrade, Bugra Gedik, Alain Biem, and Kun-Lung Wu. Elastic scaling of data parallel operators in stream processing. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–12. IEEE, 2009.

[81] John A Stankovic, Marco Spuri, Marco Di Natale, and Giorgio C Buttazzo. Implications of classical scheduling results for real-time systems. *Computer*, 28(6):16–25, 1995.

[82] Andrew S Tanenbaum and Herbert Bos. *Modern operating systems*. Pearson, 2015.

[83] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 147–156, 2014.

[84] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache Hadoop Yarn: Yet another resource negotiator. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, page 5. ACM, 2013.

[85] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Michael Armbrust, Ali Ghodsi, Michael J Franklin, Benjamin Recht, and Ion Stoica. Drizzle: Fast and adaptable stream processing at scale. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 374–389. ACM, 2017.

[86] Matt Welsh, David E. Culler, and Eric A. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the 18th ACM Symposium on Operating System Principles SOSP*, pages 230–243. ACM, 2001.

[87] Yingjun Wu and Kian-Lee Tan. ChronoStream: Elastic stateful stream computation in the cloud. In *Proceedings of the 31st IEEE International Conference on Data Engineering (ICDE)*, pages 723–734. IEEE, 2015.

[88] Le Xu, Boyang Peng, and Indranil Gupta. Stela: Enabling stream processing systems to scale-in and scale-out on-demand. In *Proceedings of the IEEE International Conference on Cloud Engineering (IC2E)*, pages 22–31. IEEE, 2016.

[89] Da Yu, Yibo Zhu, Behnaz Arzani, Rodrigo Fonseca, Tianrong Zhang, Karl Deng, and Lihua Yuan. Dshark: A general, easy to program and scalable framework for analyzing in-network packet traces. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, NSDI'19, page 207–220, USA, 2019.

[90] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, pages 423–438. ACM, 2013.