Thesis Proposal

Hai Duc Nguyen

April 14, 2023

Abstract

Today's Function-as-a-Service (FaaS) systems only provide statistical guarantees, so bursty, real-time applications cannot rely on them to deliver quality guarantee computation. We propose Rate-based Abstract Machine (RBAM) – an extension of the FaaS computation service that associates a guaranteed invocation rate to the FaaS function. This guarantee enables timely invocation allocation, enabling applications to meet real-time requirements.

We develop analytical models to study RBAM, proving its capability in bounding FaaS invocation allocation latency which is sufficient to guarantee applications' real-time deadlines. We also use the analytical model to prove the feasibility of RBAM implementation and derive a naive RBAM allocation algorithm. We use this algorithm to create an RBAM realization, named Real-time Serverless (RTS).

To demonstrate RBAM applicability, we implement a distributed real-time video analytic application with Real-time Serverless. RTS ensures both statistical and absolute application performance guarantees. These new capabilities are also robust against a wide range of workload burstiness, opening new performance engineering space with flexible deployments. Further, we use RTS as a building block to create Storm-RTS, a stream processing engine that utilizes the rate guarantee to implement application performance transparency and predictability. Such new features allow Storm-RTS to gain robust performance stability, high deployment flexibility, and can even optimize application deployment for different objectives with simple declarative policies.

Given this foundation, we will provide an efficient RBAM implementation. We plan to propose several guaranteed invocation rate allocation algorithms that can deploy any rate-guarantee with low overhead. We will also develop analytical studies and simulations to show their efficiency and then implement them inside RTS. We expect to make RTS implementation cost asymptotically small for a large number of RBAMs.

Contents

1 Thesis Statement

Current FaaS systems provide only *statistical performance SLOs*, and thus cannot meet *real-time deadlines*. We propose RBAM – a new FaaS execution model that pairs *guaranteed invocation rates* with each FaaS function deployment. RBAM allows applications to meet real-time deadlines. Furthermore, RBAM can be implemented with low *overhead* and can be generalized to other classes of applications.

Terms in italic are defined as follows.

- *Statistical Performance SLOs:* performance SLOs are defined in statistical terms. For example: "99-th of invocation latency is less than 1 second".
- *Real-time deadlines*: the maximum allowable delay a task can tolerate. The delay is measured from when the task emerges to when it completes. In the proposal, unless specifically mentioned, we consider real-time deadline as hard deadline: missing a deadline is prohibited.
- *Guaranteed invocation rate:* FaaS function is guaranteed to get new invocations up to a certain rate.
- *Overhead:* the gap between resources allocated to a FaaS function and resources consumed by its outstanding invocations.

2 Introduction

2.1 The Rise of Bursty, Real-time Applications

Technology advances such as infrastructure improvement, the increasing popularity of mobile devices as well as the growth of cloud and edge computing enable more and more IoT applications to emerge. New IoT applications are not only diverse but also experience massive-scale deployment. Examples include Amazon's 100 million intelligent assistants [37], large-scale urban and rural monitoring systems [49, 93], and even expensive large equipment [67, 103]. Many of these lead to more than 13.14 billion devices created by 2022 and the figure is projected to continue increasing in the years to come [68].

Widespread IoT offers tremendous value. One noticeable benefit is increasing productivity and efficiency. IoT-enabled automation allows taking actions at high precision and speed [1, 7] with longer operation hours and predictive maintenance [63, 9, 12]. IoT systems also offer high adaptability [8, 11], mobility [3, 6], and compliance [10, 4] to create promising solutions with improved insight at a low cost [5, 2]. Futher, these values are expected to continue flourishing along with increasing improvement of connectivity (e.g., 6G systems [121]), computation capability (e.g., emerging Edge and Cloud-Edge computing), and great leaps

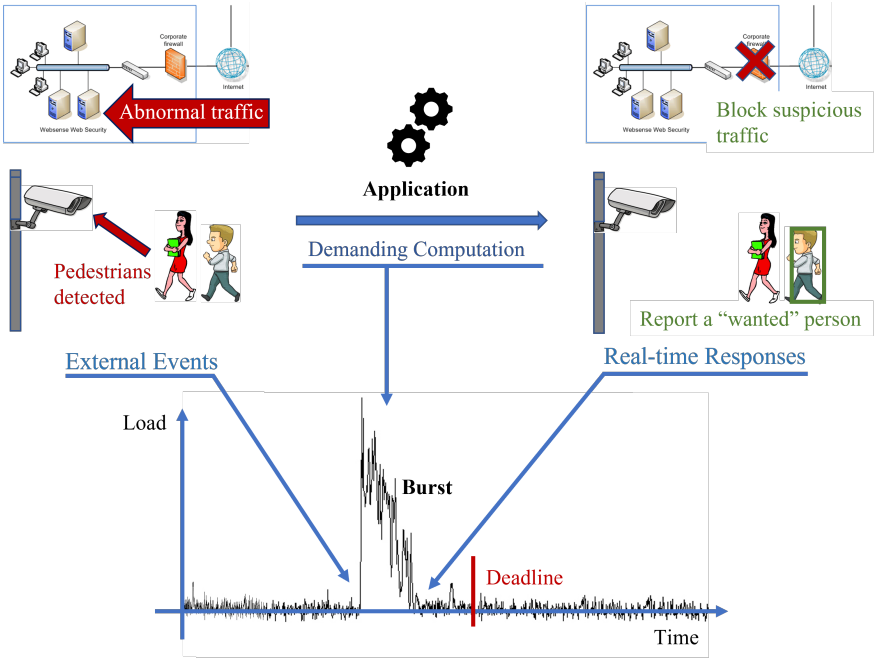


Figure 1: Real-time Bursty Application Examples: External events such as cyberattack, pedestrian appearance trigger application to start demanding computation that have to finish before a deadline. To respond in time (e.g., block suspicious traffic, detect identities, etc.) the application consumes significantly more resources than normal, causing a burst in application load.

in data science (e.g., Artificial Intelligence). This trend requires a right focus on software solutions to make IoT-enabled computation go efficiently and seamlessly.

However, observations of IoT workloads reveal many difficulties. IoT-enabled applications are driven by physical world events that are hard to predict yet require computation-intensive and time-critical reactions. For example, in Figure 1, a network administration application detects abnormal traffic so it triggers an in-depth analysis to decide whether it is a sign of a cyber attack. If so, it will take cybersecurity actions such as closing vulnerable ports and blocking suspicious traffic to protect internal systems. Another example is a crowd control application that collects video streams to detect suspicious persons. Anytime a person appears in the video, the application performs an in-depth analysis. If the analysis detects a "wanted" person, the application will file a report to the authorities. In both examples, responses must be taken in time to avoid negative impacts (e.g, letting a fugitive escape, or putting internal systems under a Denial-of-Service attack). Making such timely actions demand a lot of resources, leading to a burst in their workloads.

We use the term *bursty real-time* to refer to workloads characterized by both bursty load and real-time requirements as mentioned above. Precisely, a bursty real-time application possesses the following key characteristics:

- *Bursty*: The demand of the workload is not stable but often experiences bursts which have the following attributes.
  - *Low duty factor*: bursts are typically short and rarely happen, thus their duty factor (i.e, ratio of burst periods over time) is very low, typically less than 10%.
  - *High computation demand*: Bursts create significant higher computation demand than normal, could be 10x to 100x or even more.
- *Real-time*: The computation demand of bursts must be served within a strict deadline.

Not only in IoT, bursty real-time workload also emerges from other classes of applications. Noticeable examples include scientific data streaming, online gaming, stream processing, etc. Some of them even experience extreme computation intensity and stricter real-time deadlines. For example, in high energy physics (HEP), finding HEP events containing evidence of new physical phenomena, such as new partial or dark matter, requires Large Hadron Collider (LHC) systems to capture 40 million or even more video frames for every detected collision where each frame has to be processed within 300ns [113]. Online gaming also witnesses growing bursty traffic. Pokemon Go [115], for example, has approximately 600 million active players worldwide that could generate up to 50,000 interactive requests per second yet special events may create bursts with 50x expected load that could last for hours or even several days [113].

The computation quality of bursty, real-time applications depends on both analysis accuracy and processing speed. Failing to either make a proper decision or deliver decisions after the deadlines result in low value/quality or even system failures. Unlike analysis accuracy, the application cannot control processing speed all by itself but has to rely on resource availability (i.e., CPU, memory, etc.). When a burst arrives, computation demand increases dramatically requiring an equivalent growth in resource availability to maintain the computation pace. Failing to grow resource availability forces the application to slow down or delay some activities to fit in, thereby prolonging computation time and may miss the deadline. Rapidly allocating a high quantity of resources within a short duration for real-time deadlines is difficult. Worse, serving extreme workloads such as the LHC or Pokemon Go presented above goes beyond the capability of current general-purpose systems (e.g., public cloud). Instead, their developers have to build specialized systems [99, 100] or make an exclusive contract with infrastructure providers [35] to make these applications possible.

2.2 Existing Solutions

Given the growing emergence of real-time bursty workload as well as its increasing importance and computation intensity, designing an appropriate computation model to efficiently support their burstiness and real-time requirements is critical. This section presents existing solutions to the workload and their limitations.

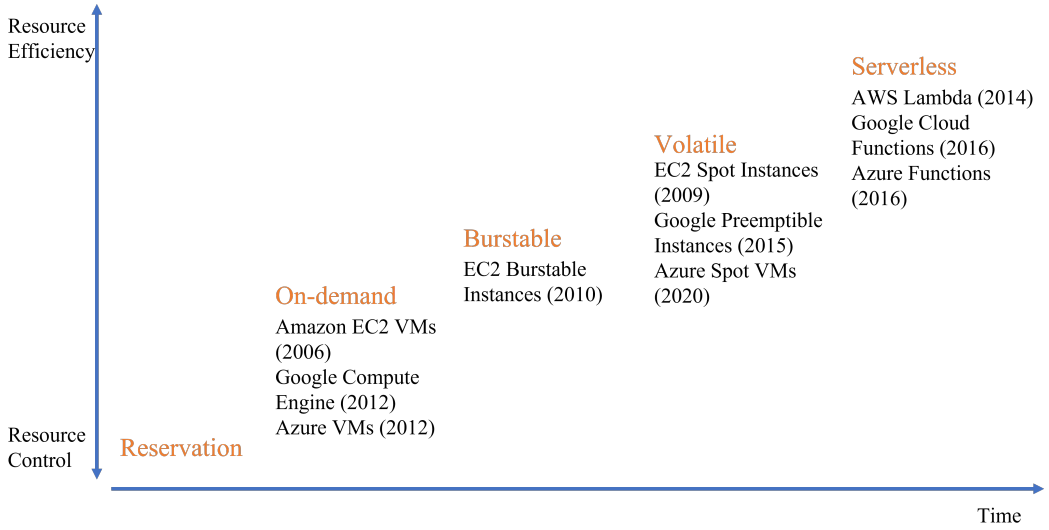


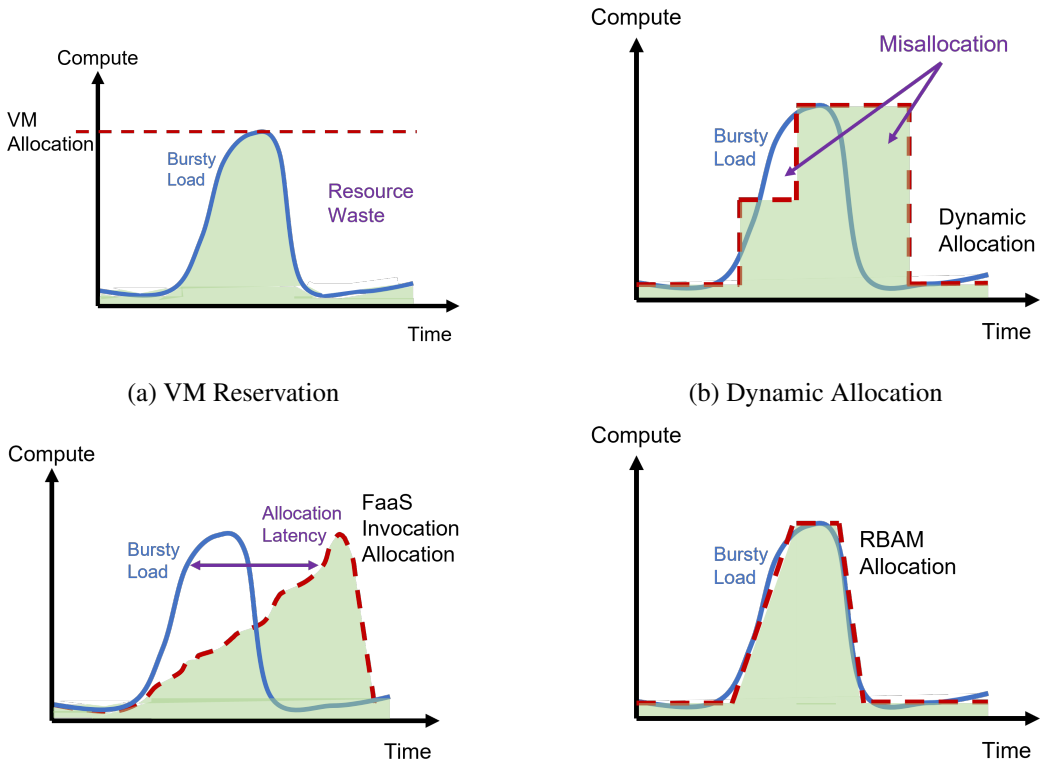
Figure 2: Timeline of existing Cloud computation models. Newer compute models offer better resource efficiency yet reduce users' control over cloud resources.

2.2.1 Traditional Cloud Computation Models

Most of today's real-time bursty applications rely on cloud data centers for computation. Cloud providers offer various types of computation models addressing different needs of applications. However, the current growth of cloud service offerings does not focus on improving application guarantee capability but reducing cost and improving cloud utilization by sacrificing users' control over cloud resources (Figure 2). Starting with VMs, cloud providers allow users to control the whole stack of software development. However, due to coarse-grained allocations and the lack of adaptation to workload dynamics, VMs incur high resource waste that increases the cost and reduces resource utilization. Newer computation models limit user control over cloud resources to offer new opportunities for waste reduction. For example, burstable instances have users keep their CPU utilization low so that they can use extra CPU at bursts at a lower cost while cloud providers can exploit the unused CPU cycles for other users, increasing utilization. Volatile instances also help increase resource utilization and cost by offering unused resources to users at high discounts with the risk of getting preempted at any time.

This trend makes bursty real-time implementation difficult as highly usable computation models offer insufficient real-time support while highly controllable services require too much deployment effort and cost. For example, burstable instances allow applications to absorb their bursts within a short duration yet capped both in terms of time and quantity by credit accumulation. Such limitations make them unusable to workloads experience bursts with long or hard to predict duration.

Due to the lack of real-time support, application developers have to use highly controllable services, such as pre-allocated VMs, in advance which gives them sufficient resource



(a) VM Reservation

(b) Dynamic Allocation

(c) Best-effort FaaS Allocation

(d) RBAM Allocation

Figure 3: Resource allocation solutions for Bursty Real-time Applications. Due to the lack of real-time support, current approaches (VM, dynamic allocation, and FaaS) either fail to meet real-time deadlines or waste too many resources. RBAM, on the other hand, allows FaaS invocations allocated at a similar rate with workload's burst enabling real-time guarantee at low resource waste.

quantity and privilege to guarantee hard real-time requirements. The pre-allocated quantity must be large enough to absorb the burst demand which sometimes cannot be done automatically but has to go through face-to-face negotiation. Further, due to the low duty factor property, VMs are left under-utilized most of the time creating a huge resource waste (Figure 3a).

2.2.2 Dynamic Resource Allocation

Many of today's solutions for bursty real-time workload workaround the resource waste problem through dynamic allocation. Instead of running the application over a fixed set of resources, application developers try to dynamically adjust allocations close to the needs. They also exploit different computation models for different situation, using one as a compliment of another for better outcomes. The dynamic allocation is an iterative control process consisting of two steps:

- *Burst Detection*: the application detects bursts through various methods, including workload monitoring [96, 95] and real-time violation detection (soft real-time) [81, 80]. Some application avoids violation by trying to predict potential burst arrivals and then proactively allocate resources in advance [33].

- *Allocation*: the application allocates additional resources to absorb the burst. The tricky part is to allocate just enough resources needed for real-time constraints. This typically results in complicated prediction (e.g., inference from historical data) [80, 95]. There are also approaches combining different cloud services, using highly flexible services, such as FaaS, to absorb the burst [96, 81].

However, all dynamic allocation approaches are constructed upon heuristic methods. Any application that has its burstiness properties or real-time constraints uncovered by the heuristic solution may end up with misallocation that leads to either real-time violation or resource waste (Figure 3b).

2.2.3 Function-as-a-Service

Function-as-a-service (FaaS), also known as Serverless, [17, 18, 32] offers promising features, such as auto-scaling and pay-as-you-go, that can resolve limitations of dynamic allocation. In FaaS, computations are carried out inside invocations triggered by the application. Each invocation has a resource configuration specifying how many resources it can utilize. Invocations do not share resources and hold them until termination. By this scheme, an application can request more resource allocations by simply executing more invocations. These properties allow FaaS resource allocation naturally to scale with computation demand. This gives a huge advantage to bursty real-time applications: when bursts arrive, the application simply invokes as many invocations as required to match the bursting demand and then releases them once computation completes. Doing so not only delivers sufficient resources for real-time guarantee but also significantly reduces the time, efforts, cost of application development and deployment.

Unfortunately, current FaaS systems allocate invocations in a best-effort manner. There is no restriction on allocation time or even on whether an allocation succeeded. This scheme adds complexity and uncertainty to application performance because they cannot time the resource allocation correctly. For the case of bursty, real-time application, the surge of load at burst urges the FaaS systems to aggressively seek a large number of additional resources for a timely response. This creates heavy pressure on the underlying resource manager systems, and without any allocation restriction, that usually results in long allocation delays or even cancellations. Consequently, as visualized in Figure 3c, the application fails to keep its computation up with the load growth and misses the deadlines.

Recent years experience efforts on mitigating this problem. However, none of the approaches above address the best-effort allocation limitation on allocation time but rather focus on improving its implementation using the following techniques

Approaches	Real-time Guarantee	Efficiency	Applicability
Reservation	Yes	Low	High
On-demand Instances	No	High	High
Burstable Instances	Limited	High	Low
Volatile Instances	No	High	Low
Regular FaaS	No	High	High
Dynamic Allocation	No	High	Low
RBAM	**Yes**	**High**	**High**

Table 1: Evaluate approaches to deploy Real-time Bursty Applications over the cloud against solution requirements. A good solution must be able to guarantee real-time requirements with high efficiency and applicability.

- *Pre-allocation*: FaaS systems pre-allocate invocations before applications need them, effectively reducing allocation latency to zero. Pre-allocation can be manually configured by the application developer [31], or issued by a load predictor typically signaled by workload traces inference [127], upstream load [57, 131, 36], etc.
- *Allocation Recycling*: FaaS systems recycle resources allocated to terminated invocations for new invocations, effectively mitigating dependency on the system state [29, 32, 18, 123, 42, 120].

Both techniques are built atop heuristic mechanisms. This means, similar to dynamic allocation, they may fail in uncovered circumstances. For example, the burst load suddenly goes too high, exceeding the pre-allocation quantity. Consequently, the best application can get is some sort of performance improvement in some circumstances which is insufficient to guarantee real-time requirements.

2.3 Problem Statement

Due to the limitations of FaaS solutions to bursty real-time workload discussed above. This thesis presents a new approach to overcome these limitations, allowing bursty real-time application to enjoy advantages of the FaaS computation with real-time guarantees. In particular, we address the following problem:

Problem Statement. *Regular FaaS allocations gives no guarantee on allocation time preventing applications from meeting their real-time requirements.*

In order to solve the problem above, we believe FaaS systems must replace their best-effort allocation with a new allocation scheme that can allow the application to bound their computation latency. Particularly, the new scheme must satisfy the following requirements:

- *Real-time Guarantee*: applications can guarantee to meet their real-time deadlines.

- *Efficiency*: the real-time guarantee capability can be implemented with low overhead.
- *Applicability*: the new FaaS allocation scheme should be usable by a wide spectrum of applications with different burstiness properties and real-time requirements, and if possible, can open new capabilities to use computation resources wisely.

While the real-time guarantee is a must-have, meeting the efficiency and applicability requirements are also important as they ensure the realization cost is not prohibitively high that prevents the new FaaS allocation from being implemented in production and be general enough to be used by any application. Table 1 examines approaches of deploying real-time bursty applications over the cloud using the requirement listed above. As we have explained, none of existing approaches satisfy all requirements.

2.4 Approach: Rate-based Abstract Machine

We propose a new FaaS system called Rate-based Abstract Machine (RBAM) replacing the best-effort allocation with a rate guarantee allocation. More precisely, the Rate-based Abstract Machine associates to each FaaS function a *guaranteed invocation rate A*. RBAM ensures that the application will have enough resources to create *at least* one invocation for the function in any interval of length $1/A$. For example, if a FaaS function has a guaranteed invocation rate of $A = 10$ invocations per second then RBAM will ensure that there will be at least one invocation available to the application in any interval of length $1/10 = 0.1$ second throughout the function lifetime.

The guaranteed invocation rate lets RBAM deliver new invocations with high predictability. Particularly, any RBAM function with a guaranteed invocation rate A lets the application know that the inter-arrival between two consecutive invocations is at most $1/A$ seconds. Thus, if the request rate is equal to or less than A, it is guaranteed that resources will be available at the same rate of demand, ensuring workload stability. The application can exploit this property for real-time guarantee. As shown in Figure 3d, with a guaranteed invocation rate equal to the demand increment rate at burst, RBAM invocations can grow at the same pace as the load, ensuring sufficient resources for timely responses.

More importantly, if the application requests for new invocations with a rate smaller or equal to A (i.e., less than one invocation request for every $1/A$ interval) then the longest time the application has to wait for the invocation to start is $1/A$. In other words, the guaranteed invocation rate allows the application to bound allocation time by $1/A$. In the example above, guaranteed invocation rate $A = 10$ invocation/sec lets all workloads with 10 or fewer invocation requests per second bound their invocation allocation latency by 0.1 second. This guarantee is extremely useful to bursty, real-time applications as they can proactively schedule the invocation requests for performance. Suppose the application has a computation deadline d and maximum execution time m (configurable through conventional FaaS API). With a guaranteed invocation rate A, invocation allocation time is bounded by $1/A$. Thus, by requesting a new invocation for this computation at any time before $d - m - 1/A$, the

application is guaranteed that the requested invocation will be available no later than m second before the deadline and will complete before d, meeting its deadlines.

Cloud providers can implement RBAM efficiently. The guaranteed invocation rate lets an RBAM system know the maximum load that an RBAM deployment must service, essentially providing an upper bound on the number of resources needed to implement any RBAM. Also, with low duty factor, real-time bursty workloads typically have very low resource consumption versus the rate guarantee, RBAM systems can exploit this fact to further reduce the overhead by applying different techniques such as sharing instances among independent/anti-correlated bursty loads, buffering with dynamic load adaptation, etc.

RBAM is also highly applicable. A regular FaaS deployment is actually an RBAM deployment with a zero rate guarantee. Thus, any applications deployable with FaaS can be deployed with RBAM without any significant change. Further, the guaranteed invocation rate is a basic real-time guarantee that can be translated into other forms of performance guarantee. For example, soft real-time applications can use the rate guarantee to meet their real-time requirements or treat it as a performance engineering parameter that can be tuned to balance the real-time quality and cost (e.g., using a small rate guarantee causes more real-time violates yet reduces cost). Stream processing applications can use the rate guarantee to stabilize their throughput and gain performance transparency for flexible deployment.

2.5 Goals and Scope of the Thesis

The thesis focuses on proving RBAM is the right FaaS-based solution for the real-time guarantee problem. The proof will show RBAM meets all three requirements for a FaaS solution to the problem as mentioned in Section 2.3. This includes three parts.

First, we prove the capability of delivering real-time guarantees of RBAM's guaranteed invocation rate. This is done by systematically constructing an analytical framework that captures key features of bursty, real-time applications and then relates them to the FaaS and RBAM computation models. We use the framework to prove that RBAM's guaranteed invocation rate can bound FaaS invocation latency for the bursty workload, which is sufficient to guarantee the workload's real-time deadlines.

Second, we show that RBAM implementation is feasible, and if we properly exploit workload burstiness structures and underlying allocation statistics, we can even implement RBAM with a low overhead.

Finally, we demonstrate RBAM's applicability by using the computation model to implement different applications exhibiting different combinations of workload burstiness and real-time requirements. These applications can easily use RBAM to implement their logic. Furthermore, the guaranteed invocation rate not only guarantees their real-time requirements but also expands the design space so that the applications can use it as a building block to achieve other important properties to gain robust performance stability, high deployment flexibility, and can even be optimized for performance, cost, and Carbon footprint with simple declarative policies.

2.6 Proposal Organization

The rest of the proposal is organized as follows. Section 3 proposes research questions related to the real-time guarantee problem and RBAM. Section 4 discusses our approaches, strategies, and plan to answer these questions as well as expected contributions of the thesis. Section 5 briefly reports the current progress of the project. We give a brief discussion of related work in Section 6. Finally, we present plans for the remaining parts of the thesis in Section 7

3 Research Questions

The thesis primarily focuses on revealing if RBAM is the right solution for real-time guarantee on FaaS. The question is answered by evaluating RBAM against the FaaS requirements mentioned in Section 2.3. In particular, we first consider if RBAM enables bursty, real-time workloads to meet their computation deadlines. Then we will propose several RBAM implementation algorithms, prove their scalability in comparison with state-of-the-art regular FaaS systems. Finally, we try implementing various applications with different burstiness properties and real-time requirements using an RBAM system to evaluate RBAM's applicability. The whole process leads to the following questions.

3.1 Real-time Guarantee Support

Can RBAM guarantee real-time deadlines for applications?

- Q.1.1 Can RBAM's guaranteed invocation rate meet fixed rate guarantee?
- Q.1.2 Can RBAM's guaranteed invocation rate support a bursty guarantee with bounded resource demand slope.
- Q.1.3 Can RBAM's guaranteed invocation rate supports a resource quantity guarantee.

3.2 Efficient Implementation

Can RBAM be implemented scalably for the Cloud?

- Q.2.1 Can RBAM be implemented to support a full range of rates
- Q.2.2 Can RBAM also be implemented with low overhead?

3.3 Applicability

Can an application's real-time goals be effectively mapped onto RBAM's guaranteed invocation rates?

- Q.3.1 Can RBAM be used to implement a specific, diverse set of demanding real-time applications?

Q.3.2 Can RBAM be used to construct a other real-time guarantees that capture a broad class of applications with quality guarantee?

4 Research Plan

We propose a research plan to answer each research questions as follows. We construct an analytical model that captures key features of applications and RBAM and use this framework to deliver mathematical proofs for each real-time guarantee question (i.e., Q.1.1, Q.1.2, and Q.1.3) and then combine them to prove RBAM's real-time guarantee capability. Next, we show that any guaranteed invocation rate is implementable by various algorithms to prove RBAM's feasibility (i.e., Question Q.2.1). We then prove RBAM's efficiency (i.e., Question Q.2.2) by using these algorithms to implement an RBAM system to support millions of RBAM functions concurrently at low overhead. Finally, we use the RBAM system to implement 3 demanding real-time application classes as RBAM functions. We evaluate these applications across different configurations to demonstrate how RBAM robustly delivers real-time guarantees (Question Q.3.1) and enables other forms of performance guarantees that can be exploited for different needs (Question Q.3.2).

We present the ideas in more detail in Section 4.1, we then propose a plan with a list of specific tasks to accomplish the ideas in Section 4.2. Finally, we discuss the expected contributions after completing the plan in Section 4.3 and the corresponding outline of the thesis in Section 4.4.

4.1 Approach and Strategies

4.1.1 Real-time Guarantee Support

We construct an analytical model to connect key workload burstiness and real-time deadlines with guaranteed invocation rate and use the model to answer the research questions about the real-time guarantee capability of RBAM as follows.

- *Can RBAM's guaranteed invocation rate meet fixed rate guarantee? (Q.1.1)* We answer the question by proving that RBAM can bound FaaS invocation latency of any fixed rate workload, and this bounded invocation latency gives enough scheduling information for the applications to guarantee their real-time deadlines.

- *Can RBAM's guaranteed invocation rate support a bursty guarantee with bounded resource demand slope? (Q.1.2)* We answer the question by generalizing the results from fixed-rate workloads for bursty workloads with bounded demand, showing that RBAM's capability of bounding invocation latency still holds in this case, and so does the real-time guarantee capability.

- *Can RBAM's guaranteed invocation rate supports a resource quantity guarantee? (Q.1.3)* We answer the question by proving that any finite RBAM's guarantee invoca-

tion rate can be implemented with a bounded resource quantity, even in worst-case scenarios, independent from the workload and the underlying infrastructure. This bound guarantees application resource quantity and ensures the feasibility of any RBAM-based solution.

By proving the statements listed above, we can show that RBAM can deliver real-time guarantees at bounded resource cost for both periodic and non-periodic bursty applications with bounded computation demand – broad enough classes of workload making RBAM a workable FaaS model for practical real-time bursty applications.

4.1.2 Efficient RBAM Implementation

To know if RBAM can be implemented scalably for the Cloud, we propose several RBAM implementation algorithms based on computation services supported by the Cloud. We then evaluate them against four categories: rate-guaratee support (i.e., whether an algorithm supports any rate guarantee), resource efficiency, additional knowledge (e.g., workload patterns, underlying resources statistics, etc.), and scalability. We use the insights of the evaluation to answer RBAM research questions as follows.

- *Can RBAM be implemented to support a full range of rates? (Q.2.1)* We show that there exist RBAM algorithms that can support any rate-guarantee with bounded resource quantity. Thus, the Cloud which has conceptually unlimited resources can host a full range of rates.
- *Can RBAM also be implemented with low overhead? (Q.2.2)* We will show that the exists algorithms that can implement RBAM with small resource waste and scale up to millions of RBAM deployments. We also implement these algorithms in a practice FaaS system and use the system to realize real-time bursty applications. Based on the applications, we conduct empirical evaluations to demonstrate RBAM's real-time guarantee capability and scalability.

4.1.3 RBAM Applicability

To show the applicability of RBAM, we will use RBAM to implement three different classes of applications that represent a diverse set of demanding applications with different real-time requirements:

- *Distributed Real-time Video Analytic*: a popular application class with bursty demand driven by highly unpredictable external events. The bursty demand is soft real-time.
- *Demanding Scientific Sensor Instrument Data Processing*: extremely highly demanding applications (millions or more events per second). Although events arrive at a fixed rate, their processing deadlines are hard. Examples include applications in High Energy Physics, Advanced Photon Sources, etc.

- *Distributed Stream Processing*: a critical framework in many IoT and wide-area applications. Data are created as streams and processed on-the-fly. Stream processing applications typically manage their execution through a stream processing engine with many built-in supports aiming for stable high throughput and low latency.

We conduct both analytical and experimental studies over RBAM implementations of these applications to answer RBAM applicability research questions as follows.

- *Can RBAM be used to implement a specific, diverse set of demanding real-time applications? (Q.3.1)* Through the distributed real-time video analytic and scientific sensor instrument data processing applications, we will show that the RBAM can guarantee a wide range of real-time requirements, starting from loose soft deadlines of a few seconds down to extremely strict hard deadlines at a sub-microsecond scale. Further, this capability is robust against different workload burstiness: from periodic ones up to highly unexpected burst loads with extremely high burst rates.
- *Can RBAM be used to construct a other real-time guarantees that capture a broad class of applications with quality guarantee? (Q.3.2)* By implementing a stream processing engine with RBAM, we demonstrate that the real-time guarantee delivered by RBAM can be transformed into a real-time processing guarantee which ensures stable stream processing throughput with low latency independent from the execution environment. That new form of guarantee even opens new capabilities, such as flexible deployment across multi datacenters, that broaden the scope of RBAM usage to deliver more computation value.

4.2 Research Plan

Based on the strategy presented above, we propose the following plan with specific tasks, each aiming to address one or a part of a research question. By combining these tasks together, we will have a complete set of answers to the thesis research questions.

- *T1*. Prove RBAM can guarantee real-time deadlines of applications (*Done*).
  - T1.1 Develop an analytical framework for theoretical analysis and evaluation (Address Q.1.1, Q.1.2, and Q.1.3, *Done*).
  - T1.2 Use the analytical framework to prove that guarantee invocation rate allows application to meet real-time deadlines (Address Q.1.1, Q.1.2, and Q.1.3, *Done*).
- *T2*. Show that we can implement RBAM functions with at most 15% more resources compare to their regular FaaS counterparts. (*Partly Done*)
  - T2.1 Prove the feasibility of RBAM realization, and implement an RBAM system that allow applications to deploy RBAM functions of any rate (Address Q.2.1, *Done)*.

T2.2 Propose several scalable allocation algorithms that can reduce the overhead of deploying RBAM functions at any rate-guarantee over the cloud (Address Q.2.2, *In Progress*).

T2.3 Evaluate proposed RBAM allocation algorithms to understand their efficiency (Address Q.2.2, *In Progress*).

T2.4 Integrate proposed RBAM algorithms in a practical FaaS system (Address Q.2.2, *In Progress*).

T2.5 Conduct empirical studies to demonstrate that RBAM implementation cost is asymptotically small for a large number of RBAMs (Address Q.2.2, *In Progress*).

- *T3*. Demonstrate RBAM applicability by implementing various applications with different burtiness properties and real-time requirements *(Partly Done)*.

  T3.1 Design and implement an RBAM implementation of a distributed real-time video analytic application (Address Q.3.1, *Done*).

  T3.2 Design and conduct experiments to show that RBAM allows the application to guarantee the value/quality of their real-time video processing, also prove that the real-time guarantee delivered by RBAM is robust against application burstiness variations (Address Q.3.1, *Done*).

  T3.3 Design and implement an RBAM-based stream processing engine (Address Q.3.2, *Done*).

  T3.4 Design and conduct experiments to show that Storm-RTS deliver real-time stream processing capability for stream processing applications, also demonstrate new capabilities arisen from this guarantee (Address Q.3.2, *Done*).

  T3.5 Implement an demanding Scientific Sensor Instrument Data Processing application using RBAM functions (Address Q.3.1, *In progress*).

  T3.6 Design and conduct experiments to show that the RBAM functions enable the application to meet its real-time requirements with efficiency (Address Q.3.1, *In progress*).

- *T4*. Write the Dissertation (*In Progress)*

  T4.1 Summarize the research and write a full PhD Dissertation. (*In Progress*)

  T4.2 Dissertation Defense *(In Progress)*.

4.3 Expected Research Contributions

We expect the complete thesis would include the following research contributions:

- RBAM's guarantee invocation rate ensures real-time deadlines for bursty, real-time applications.

- We can exploit workload and cloud services statistics to implement finite guaranteed invocation rates for FaaS functions with less than 15% overhead.
- RBAM can be used to implemented applications with a wide range of workload burstiness and real-time requirements. RBAM helps these applications not only guarantee their real-time deadlines but also create new capabilities including robust performance stability, high deployment flexibility, and simple performance management across distributed resources.

4.4 Full Thesis Outline

The draft outline of the full thesis is as follows

1. Introduction
   (a) The Rise of Bursty, Real-time Applications
   (b) Limitations of Current Solutions
   (c) Approach: Rate-based Abstract Machine
   (d) Goals and Scope of Dissertation
   (e) Dissertation Contributions and Organization
2. Background
   (a) Cloud computing and Its Applications
   (b) Function-as-a-Service
3. Performance Guarantee Challenges of Bursty, Real-time Applications
   (a) The Deadline Guarantee Problem
   (b) Solution Requirements
4. Rate-based Abstract Machine
   (a) Definition
   (b) Real-time Guarantee Capability
   (c) RBAM Implementation
      i. Feasibility
      ii. Rate-guarantee Allocation Algorithms
      iii. Real-time Serverless: Scalable RBAM Implementation
5. Evaluation
   (a) Methodology

(b) Real-time Guarantee Capability

(c) RBAM Implementation Efficiency

6. RBAM Applicability

(a) Distributed Real-time Video Analytic

(b) Demanding Scientific Sensor Instrument Data Processing

(c) Stream Processing Engine

7. Related Work

(a) Supporting Bursty, Real-time Applications

(b) FaaS Performance Guarantee

(c) Distributed Real-time Video Analytic

(d) Stream Processing

(e) Demanding Scientific Sensor Instrument Data Processing

8. Summary

(a) Conclusions

(b) Future Work

5 Initial Results

This section presents our initial results collected from executing the plan proposed in Section 4.2. At the time of this writing, we have completed answering RBAM real-time guarantee research questions by showing that the computation can guarantee real-time deadlines for broad classes of real-time bursty applications. We also partly addressed the RBAM implementation questions by proving its feasibility with a naive RBAM system called Real-time Serverless (RTS). We used RTS to implement 2/3 of the proposed real-time application classes, namely distributed real-time video analytics and stream processing, partly demonstrating RBAM applicability.

5.1 Current Progress

We first summarize the above accomplishments in this section including a lists of tasks we have completed according to the research plan presented in Section 4.2, their results and how they help us answer the thesis research questions.

5.1.1 Real-time Guarantee Support

We completed proving RBAM capability of guaranteeing real-time deadlines for applications. According to the plan, we have done the following

- We constructed an analytical framework for theoretical analysis and evaluating RBAM's real-time guarantee capability. (Complete Task T1.1)
- We used the framework to deliver proofs that answers research questions regarding RBAM's real-time guarantee support (Complete Task T1.2):
  - RBAM's guaranteed invocation rate can meet fixed rate guarantee. (Answer Question Q.1.1)
  - RBAM's guarantee invocation rate can support bursty guarantee with bounded resource demand slope (Answer Question Q.1.2).
  - RBAM's guarantee invocation rate supports a resource quantity guarantee (Answer Question Q.1.3).

These results allow us to argue that RBAM can deliver real-time guarantees at bounded resource cost for both periodic and non-periodic bursty applications – broad enough classes of workload making RBAM a workable FaaS model for practical real-time bursty applications. And by that, we delivered a complete set of answers to RBAM's real-time guarantee research questions.

5.1.2 Efficient RBAM Implementation

We applied the analytical framework to design a naive RBAM implementation algorithm. We used this algorithm to prove RBAM feasibility and built an RBAM prototype named Real-time Serverless. These complete the first of RBAM implementation task in the plan (Task T2.1) that answers the first research question concerning RBAM implementation (i.e., RBAM can be implemented to support a full range of rate, Q.2.1, Section 3.2).

5.1.3 RBAM Applicability

We use Real-time Serverless to implement 2/3 demanding real-time applications in the plan, namely distributed real-time video analytics and distributed stream processing. We also carried out experiments on these RBAM-based implementations to demonstrate RBAM applicability. According to the plan in Section 4.2, we have done the following:

- We used Real-time Serverless to implement an RBAM implementation of a distributed real-time video analytic application (Complete Task T3.1).
- We designed Design and conducted experiments showing RBAM allows the application to guarantee the value/quality of their real-time video processing, also prove that

the real-time guarantee delivered by RBAM is robust against application burstiness variations (Complete Task T3.2).

- We designed and implemented an RBAM-based stream processing engine (Storm-RTS) based on Real-time Serverless (Complete Task T3.3).
- We designed and conducted experiments showing that Storm-RTS ensures real-time stream processing guarantee for stream processing applications. We also demonstrate new capabilities arisen from this guarantee that the applications can take advantage of for different deployment purposes (Complete Task T3.4).

The results help us partly answer research questions concerning RBAM applicability (Q.3.1 and Q.3.2 – Section 3.3):

- By translating RBAM's rate-guarantee into stream processing real-time guarantee, we shown that RBAM can be used to constructed other real-time guarantees that capture a broad class of application (i.e, stream processing). Thus Q.3.2 is completely answered.
- We also showed that RBAM can be used to soft real-time deadlines with diverse set of demanding workloads. Hard real-time deadlines guarantee, however, have not been addressed yet. Thus, only half of Q.3.1 is complete.

In the next subsequent sections, we will present the results in more detail. Section 5.2 present results related to real-time guarantee supports with a formal description of the analytical framework and real-time guarantee proofs. Section 5.3 proves RBAM feasibility with a naive RBAM implementation – Real-time Serverless. Section 5.4 provides RBAM applicability evidences with RBAM-based implementations of the distributed real-time video analytic and stream processing engine and evaluation results demonstrating RBAM applicability.

5.2 Real-time Guarantee Support

This subsection presents the results addressing RBAM real-time guarantee supports including an analytical framework to model workloads, real-time deadlines, and FaaS/RBAM rate guarantee and proofs using the framework to show that RBAM can guarantee real-time deadlines across broad workload classes.

5.2.1 Analytical Framework

We model the workload of an FaaS application as a sequence of events $E_1, E_2, ...$ An even E_i is identified by

- *Event emergence time* s_i: the time E_i emerges and requires the application to take an action in response (e.g., a file is uploaded, a sensor send data to a cloud server).

- *Request arrival time* (or simply arrival time) r_i: the application issues an FaaS request at r_i asking the FaaS system for a new invocation that will be used to execute some computation to generate a response to the event.

Note that the request arrival time r_i does not necessarily happen *after* the event emergence time s_i. Some applications do proactive processing that asking for resources before the actual demand arrives for efficiency. We call such cases *pre-allocation* that results in $s_i > r_i$. All events has a *deadline D* that is maximum allowable latency for the application to get the computation response to the event. For simplicity, we assume the time is continuous and do not assign it to any specific unit of time (e.g., second, millisecond) yet measure it in a general term *time unit*.

The FaaS system responds to a request at r_i by allocating a FaaS invocation to handle the computation. Once the computation completes, the result is returned to the application and the invocation is terminated. For simplicity, we assume the application uses only one FaaS function – f for all events. The whole process of allocating a new invocation for f is modeled as follows.

- *Allocation time* a_i: the time the FaaS invocation is allocated to the request issued at r_i.
- *Execution time* ε_i: the time it takes to complete the computation after the request obtains the invocation.

In practice, execution time may vary, mainly depending on the computation intensity of the request and resources allocated to the invocation handling the request. Since both factors are configurable by the application, we assume execution time is deterministic and short enough to make meeting computation deadline possible, i.e.,

$$\forall i : \varepsilon_i \leq D \tag{1}$$

Also in practice, A FaaS invocation is not delivered immediately after a request is issued but it usually takes the FaaS system a while to find available resources and initialize the new invocation before giving it to the request. We call this gap *allocation latency* δ_i calculated as follows

$$\delta_i = r_i - a_i \tag{2}$$

We also call FaaS requests waiting for an invocation *pending requests*, and define $Pend(t)$ as the number of pending request at the time t:

$$Pend(t) = |\{i : s_i \leq t < a_i\}| \tag{3}$$

Next, we characterize a single FaaS function f by its guaranteed invocation rate, defined as follows.

Definition 5.1 (Guarantee Invocation Rate)**.** A guarantee invocation rate A_f associates to a single FaaS function f. For any point in time t, if $Pend(t) > 0$ then A_f ensures that there will be *at least* one invocation allocation available to these requests within the time interval $[t, t + 1/A_f)$.

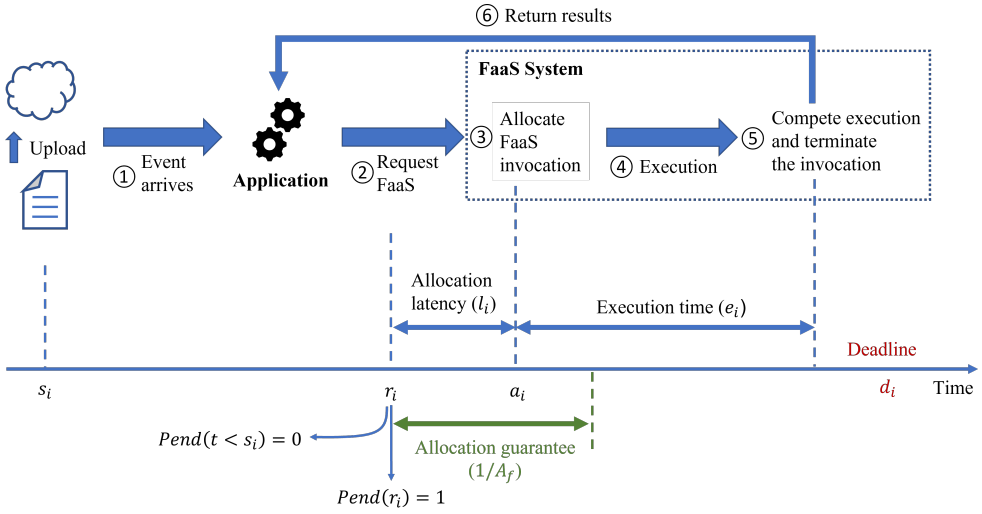


Figure 4: Visualization example of an event processing using RBAM/FaaS model with notations defined in Table 2: a file is uploaded to the cloud resulting in an event which triggers an application to send a FaaS request to an RBAM/FaaS System. The RBAM/FaaS system allocates a new invocation to handle the request computation. If the invocation has guaranteed invocation rate $A_f > 0$ then the request will to get a new invocation within $1/A_f$ since it is the only request issued until a_i. Upon invocation allocation, execution starts and finishes just before the event's real-time deadline.

By this definition, a regular FaaS function f would have guaranteed invocation rate $A_f = 0$ while an RBAM function would have $A_f > 0$. Also note that the guaranteed invocation rate only ensures invocation availability. An invocation allocation, once available, is free to go to any request. For example, assume at $t = 0$, $Pend(t) = 10$, then A_f ensures at least one invocation to be allocated at some time between 0 and $\frac{1}{A_f}$. Suppose the invocation is allocated at $t' < \frac{1}{A_f}$, then this invocation can be assigned to any one out of the ten pending requests at $t = 0$ or even assigned to a new request arrived at $t'' \in (t, t')$, if any.

We summarize terms and notations in Table 2 and visualize them in Figure 4. In the following subsection, we will use them to prove that guaranteed invocation enables meeting real-time deadline guarantee. The proof analytically answer the research questions Q.1.1, Q.1.2, and Q.1.3 that we listed early in Section 3.1:

- **Subsection 5.2.2: Fixed Rate Real-time Guarantee** (Answering Question Q.1.1) We start with a simple case, proving RBAM can bound the allocation latency for fixed rate (i.e., periodic) workload, thereby guarantee their real-time deadlines.
- **Subsection 5.2.3. Bursty Real-time Guanratee** (Answering Question Q.1.2) We will generalized the results on Step 2 for any bursty workload with bounded resource demand.

Symbol	Name/Definition	Units
Workload Characteristics		
E_i	Events that trigger FaaS requests	None
s_i	Event emergence time	time unit
r_i	Request arrival time	time unit
D	Real-time deadline	time unit
FaaS Systems		
A_f	Guaranteed Invocation Rate	invocation per time unit
a_i	The time the request issued at s_i get an invocation	time unit
ε_i	Invocation execution time	time unit
δ_i	Allocation latency	time unit
$Pend(t)$	Number of pending invocation request at t	time unit

Table 2: Terms and Notations

- **Subsection 5.2.4. Real-time Guarantee Resource Consumption** (Answering Question Q.1.3) We will prove the guaranteed invocation rate needed to support real-time guarantee consume a bounded resource quantity, thereby making RBAM feasible in practice.

5.2.2 Fixed rate Real-time Guarantee

A workload is fixed rate if its event emergence times $s_1, s_2, ...$ meet the following condition.

Definition 5.2 (Fixed Rate/Periodic Time Series)**.** A time series $t_1, t_2, ...$ is period at a fixed rate λ_{fix} if for any half-open time interval of length $\frac{1}{\lambda_{fix}}$, there exists *exactly one i* such that t_i belongs to this interval, i.e.,

$$\forall t : \exists! i : t_i \in [t, t + \frac{1}{A_{fix}}) \tag{4}$$

Now we will prove guaranteed invocation rate bounds the allocation latency for all FaaS requests issued at a fixed rate. We then use this result to show RBAM's real-time guarantee of a fixed rate workload.

Theorem 1 (Fixed rate Allocation Latency Guarantee)**.** *If a FaaS function f requested at a fixed rate λ_{fix} is associated with a guaranteed invocation rate $A_f = \lambda_{fix}$, then all of its requests get a new invocation with allocation latency bounded by $1/A_f$ time units:*

$$\forall i : \delta_i < \frac{1}{A_f} \tag{5}$$

Proof. We will prove the theorem by induction.

- **Base case.** $i = 1$. Consider the interval $[r_1, r_1 + \frac{1}{A_f})$. The guaranteed invocation rate A_f ensures there is a new invocation allocated some time within this interval. Meanwhile, the function is requests at fixed rate $\lambda_{fix} = A_f$. Thus, there is no other requests issued until $r_1 + \frac{1}{A_f}$ so the new invocation will be given to the first request. Therefore, $a_1 < r_1 + \frac{1}{A_f} \implies \delta_1 = a_1 - r_1 < \frac{1}{A_f}$.
- **Inductive Hypothesis.** $\forall i < n : \delta_i < \frac{1}{A_f}$
- **Inductive step.** We will prove $\delta_n < \frac{1}{A_f}$ as follows. By the inductive hypothesis, for all $i < n$

$$\begin{aligned} \delta_i &< \frac{1}{A_f} \\ a_i - r_i &< \frac{1}{A_f} \\ a_i &< r_i + \frac{1}{A_f} \\ &\leq r_{n-1} + \frac{1}{A_f} \\ &= r_n \end{aligned} \tag{6}$$

  Thus, by r_n, all previous invocation requests $r_1, ..., r_{n-1}$ have already got their invocations. Further, the guaranteed invocation rate ensures an invocation allocation available within $[r_n, r_n + \frac{1}{A_f})$. Due to the fixed rate condition, this allocation must be assigned to r_n. This implies $a_n < r_n + \frac{1}{A_f} \implies \varepsilon_n < \frac{1}{A_f}$

□

Theorem 2 (Fixed rate Real-time Guarantee)**.** *If a workload whose event emergence times $s_1, ..., s_n$ follow a fixed rate λ_{fix} is processed by a FaaS function with a guaranteed invocation rate $A_f = \lambda_{fix}$, then there exists a schedule algorithm for the application to meet all real-time deadlines.*

Proof. We will prove the theorem by proposing a simple scheduling algorithm for each event and showing that this algorithm guarantee the event's read-time deadline. The algorithm is as simple as follows: For each event E_i, the application request a FaaS invocation to handle the event computation at

$$r_i = s_i + D - \varepsilon_{max} - \frac{1}{A_f} \tag{7}$$

where $\varepsilon_{max} = \max_i \varepsilon_i$ is the maximum execution of the FaaS function.

The algorithm actually constructs a request arrival series $r_1, r_2, ...$ by shifting the event emergence time series $s_1, s_2, ...$ by $T = D - \varepsilon_{max} - \frac{1}{A_f}$ time unit(s) to the right. Since T is a constant, $r_1, r_2, ...$ is also has a fixed rate λ_{fix}. Thus, by applying Theorem 1, the allocation

latency $\delta_i < \frac{1}{A_f}$ for all i. This means the computation is guaranteed to complete by

$$\begin{aligned} t_{comp} &= r_i + a_i + \varepsilon_i \\ &\leq r_i + \frac{1}{A_f} + \varepsilon_{max} \\ &= s_i + D \end{aligned} \tag{8}$$

Thus, the real-time is guaranteed. □

5.2.3 Bursty Real-time Guarantee

We generalize the results from the previous subsection by consider a broader class of workload – bursty workload. Bursty workload does not follow any request arrival pattern so their arrival rate may change over time. We characterize the bursty workload by their peak rate defined as follows.

Definition 5.3 (Invocation peak rate)**.** A FaaS function has a peak rate λ_{max} if there exists a 1-time-unit interval in which the number of invocation requests for that function is λ_{max} **and** for any other 1-time-unit intervals, the number of invocation requests for that function is smaller than or equal to λ_{max}.

Note that by this definition, $\lambda_{max} \geq 1$, Otherwise, the function has no request at all. We only consider bursty workload with bounded peak rate (i.e., $\lambda_{max} =$ constant).

Theorem 3 (Bursty Real-time Allocation Latency Guarantee)**.** *If a FaaS function f is requested at a peak rate λ_{max} is associated with a guaranteed invocation rate $A_f = \lambda_{max}$, then all of its requests get a new invocation with invocation latency bounded by* 1 *time units:*

$$\forall i : \delta_i < 1 \tag{9}$$

Proof. Let $T_1, T_2, ...$ be disjoint consecutive one-time unit intervals started from $[r_1, r_1 + 1)$ (i.e., $T_1 = [r_1, r_1 + 1), T_2 = [r_1 + 1, r_1 + 2), ...$). We will prove the theorem with three steps:

- **Step 1:** We will prove that every T_j has an instance of time with zero pending invocation requests, i.e.,

$$\forall j : \exists t \in T_j : Pend(t) = 0 \tag{10}$$

- **Step 2:** we use the result from Step 1 to prove that once a FaaS request is issued at r_i, A_f ensures the function f to witness zero invocation pending in less than 1 time unit:

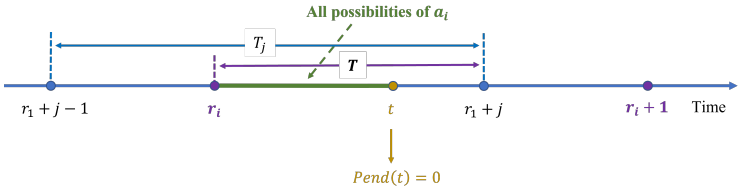
$$\forall i : \exists t \in [s_i, s_i + 1) : Pend(t) = 0 \tag{11}$$

- **Step 3:** The moment t when $Pend(t) = 0$ indicates that the number of FaaS requests (r_i's) and invocation allocations (a_i's) are equal. This means *all* request issued *before* t had already obtained an invocation (otherwise, there must be an invocation allocated before its request which is impossible). By proving there exists such $t \in [r_i, r_i + 1)$ in step 2, we show that $a_i \leq t < r_i - 1$ or $a_i - r_i < 1$ for all i and thus prove the theorem.

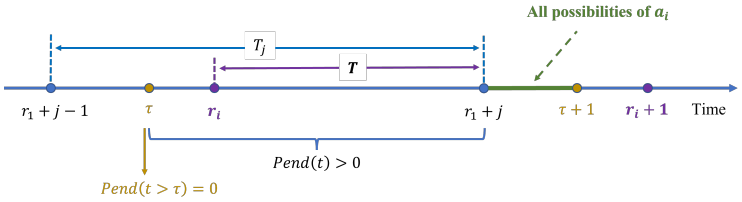
We now prove the statements in Step 1 and 2 as follows.

- **Step 1.** We will prove $\forall j : \exists t \in T_j : Pend(t) = 0$ by induction.
  - **Base case.** We will prove $\exists t \in T_1 : Pend(t) = 0$ by contradiction. Suppose $\forall t \in T_1 : Pend(t) > 0$. This means there always a pending request waiting for an invocation throughout the interval. By definition, the guaranteed invocation rate A_f ensures at least one invocation allocated within the following pairwise-disjoint intervals $[r_1, r_1 + \frac{1}{A_f}), [r_1 + \frac{1}{A_f}, r_1 + \frac{2}{A_f}), [r_1 + \frac{2}{A_f}, r_1 + \frac{3}{A_f}), ...$ There are A_f of such intervals in T_1. Thus, by the time $r_1 + 1$, at least A_f invocations are allocated. Also, by the peak rate assumption, T_j must witness at most A_f requests. This means at t' – the time when the last invocation is allocated in T_j – all requests get their invocations, and there is no new request arrival after t'. Thus, $\forall t \in [t', r_1 + 1) : Pend(t) = 0$, a contradiction.
  - **Inductive Hypothesis.** $\forall j < n : \exists t \in T_j : Pend(t) = 0$.
  - **Inductive Step.** We will prove $\exists t \in T_n : Pend(t) = 0$ as follows. Recall $T_n = [r_1 + n - 1, r_1 + n)$. If $Pend(r_1 + n - 1) = 0$ then the statement is proven. Otherwise, when $Pend(r_1 + n - 1) > 0$, let τ be the last time, starting from $r_1 + n - 1$ and going backward $t = 0$, we experience non-zero pending requests ($\forall t \in [\tau, r_1 + n - 1], Pend(t) > 0$). By inductive hypothesis, $t \in T_{n-1}$. Consider the interval $T' = [\tau, \tau + 1)$. Suppose $\forall t \in T' : Pend(t) > 0$ then, similar to the argument we made for the base case, at least A_f invocations are allocated. Due τ definition, we experience zero pending requests just before τ so all of the new invocations allocated within T' are given to requests also arrived within this interval, whose quantity are at most A_f. Thus, any time after the last invocation is allocated would witness zero request pending, a contradiction. Therefore, $\exists t \in T' : Pend(t) = 0$. Because $\tau \in T_{n-1} \implies \tau + 1 \in T_n$ so $t \in T_n$, the statement is proven. This complete the first step.

- **Step 2.** We prove $\forall i : \exists t \in [r_i, r_i + 1) : Pend(t) = 0$ as follows. Consider an individual request arrives at r_i. Let $T_j = [r_1 + j - 1, r_1 + j)$ be the interval that contains r_i. Consider the interval $T = [r_i, r_1 + j)$, there are two possibilities:
  - $\exists t \in T : Pend(t) = 0$: Because $r_i \in T_j \implies r_1 + j \leq r_i + 1$ then $t \in [r_i, r_i + 1)$ (Figure 5a).



(a) $\exists t \in T : Pend(t) = 0$



(b) $\forall t \in T : Pend(t) > 0$

Figure 5: Visualization for the Step 2 in the proof of Theorem 3

– $\forall t \in T : Pend(t) > 0$: Let τ be the last time, starting from r_i and going backward $t = 0$, we experience $Pend(t) > 0$. By Step 1, $\tau > r_i + j - 1$. Consider the interval $T' = [\tau, \tau + 1)$, Suppose $\forall t \in T' : Pend(t) > 0$ then, similar to the argument we made in Step 1, at least A_f invocations are allocated. Due τ definition, we experience zero pending requests just before τ so all of the new invocations allocated within T' are given to requests also arrived within this interval, whose quantity are at most A_f. Thus, any time after the last invocation is allocated would witness zero request pending, a contradiction. Therefore, $\exists t \in T' : Pend(t) = 0$. Because $\tau \in T_{n-1} \implies \tau + 1 \in T_n$ so $t \in T_n$ (Figure 5b).

Since we can find an $t \in [r_i, r_i + 1) : Pend(t) = 0$ in both case, the statement is proven. This together with arguments in Step 3 prove the theorem.

□

Theorem 4 (Bursty Real-time Guarantee)**.** *If a workload whose event emergence times $s_1, \ldots, s_n$is bursty with maximum rate λ_{max} is processed by a FaaS function with a guaranteed invocation rate $A_f = \lambda_{max}$, then there exists a schedule algorithm for the application to meet all real-time deadlines.*

Proof. Similar to the proof of Theorem 2, We will propose a scheduling algorithm for each event and showing that this algorithm guarantee the event's read-time deadline. The algorithm is as simple as follows: For each event E_i, the application request a FaaS invocation to handle the event computation at

$$r_i = s_i + D - \varepsilon_{max} - 1 \tag{12}$$

where $\varepsilon_{max} = \max_i \varepsilon_i$ is the maximum execution of the FaaS function.

The algorithm actually constructs a request arrival series $r_1, r_2, \ldots$ by shifting the event emergence time series $s_1, s_2, \ldots$ by $T = D - \varepsilon_{max} - 1$ time unit(s) to the right. Since T is

a constant, $r_1, r_2, ...$ is also bursty with max rate λ_{max}. Thus, by applying Theorem 3, the allocation latency $\delta_i < 1$ for all i. This means the computation is guaranteed to complete by

$$\begin{aligned} t_{comp} &= r_i + a_i + \varepsilon_i \\ &\leq r_i + 1 + \varepsilon_{max} \\ &= s_i + D \end{aligned} \tag{13}$$

Thus, the real-time is guaranteed. □

Note that the real-time guarantee capability is independent of application. There is no application assumption except the peak rate is required to deliver the bound on allocation latency. This means regardless of how application would schedule or prioritize invocations across their pending requests, allocation latency is always bounded. This adds a lot of freedom to application design in using RBAM. For example, they can let very urgent request to get new invocation before the other without sacrifice their real-time guarantee.

5.2.4 Real-time Guarantee Resource Consumption

Finally, we will prove that by using RBAM function with $A_f > 0$, the resource consumption of *guaranteed invocations* is bounded, allowing the application to implement real-time guarantee at bounded cost. Not that by *guaranteed invocations*, we mean that additional invocation are not counted as their allocation grant is unexpected, thereby adding no contribution to real-time guarantee. For example, assume the application issues two invocation requests at $t = 0$, then the guaranteed invocation rate A_f ensure *at least* on invocation with $[0, \frac{1}{A_f})$. This means in some circumstances, we may have two invocations allocated by $\frac{1}{A_f}$ for both requests but only one is counted as guaranteed invocation, the other is situational and will be excluded.

Theorem 5 (Real-time Guarantee Resource Consumption)**.** *Consider an application using an RBAM function f with guaranteed invocation rate A_f. Let*

$$E_{max} = \max_i(\varepsilon_i) \tag{14}$$

be the maximum execution time of FaaS invocation in response to the application requests, and

$$Consume(t) = |\{i : a_i \leq t < a_i + \varepsilon_i\}| \tag{15}$$

be the number of guaranteed invocation consumed by the application at time t, then

$$\forall t : Consume(t) \leq A_f \cdot E_{max} \tag{16}$$

Proof. Consider a time t, then by definition of E_{max}, all invocation consumed by the application must be allocated after $t - E_{max}$. Within this interval, $[t - E_{max}, t]$, there is at most $A_f \cdot E_{max}$ invocations are guaranteed to allocated, therefore $Consume(t) \leq A_f \cdot E_{max}$. □

Note that the Theorem does not requires any assumption from the application, so it also provide an upper bound on the invocation resources that an RBAM implementation needed to prepare to realize any rate guarantee A_f. Also, since we already assume invcation execution time is deterministic, E_{max} is finite, so is the bound. This result show that the resource consume by RBAM application is finite, and only depends on configurable parameters A_f and E_{max}. This allows the application to actually configure RBAM deployment to bound the execution cost, and RBAM implementation to bound their RBAM realization cost.

This result in combination with the real-time guarantee capability in stated in Theorem 2 and 4 show that RBAM, with guaranteed invocation rate, provides a sufficient capability to let any bursty applications with bounded rate to meet their real-time deadline at finite resource consumption. We argue this class of bursty, real-time application is general enough to represent real-time bursty application in practice as unbounded bursty applications are generally incurs unbounded resource consumption, which is impractical in current cloud systems.

5.3 RBAM Implementation

Based on the analytical results above, we addressed the first RBAM implementation question, proving that *RBAM can be implemented to support a full range of rates* (Q.2.1 – Section 3.2). The proof consists of two parts: we first propose a naive RBAM implementation algorithm that can realize any RBAM deployment of any guaranteed invocation rate (subsection 5.3.1) then we integrate the algorithm into OpenFaaS to develop an RBAM system called RTS (subsection 5.3.2).

5.3.1 Implementation Feasibility

As covered in Section 5.2.4, an RBAM function is characterized by its guaranteed invocation rate A_f and maximum execution time E_{max} requires at most $A_f \cdot E_{max}$ invocation to deliver the rate-guarantee allocation. Based on this bound, we develop a naive implementation of an RBAM function with finite A_f and E_{max} as follows

Theorem 6. *Given a FaaS function f with finite guaranteed invocation rate A_f and finite maximum execution time E_{max}. A_f is guaranteed if the Algorithm 1 is used to allocate new invocations to the application.*

Proof. We will prove by contradiction. Suppose there exists an interval of length $1/A$, $[t, t+\frac{1}{A})$ in which $Pend(t) > 0$ yet the application fails to get any new invocation by its end $t + 1/A$.

In Algorithm 1, the 4-th line ensures the gap between two consecutive invocation allocations is at least $\frac{1}{A_f}$ time unit(s). Thus, the interval $[t, t+\frac{1}{A_f}$ only experience at most one allocation attempt (Line 6), and for our assumption remains true, this attempt must fail. Since the algorithm does not add or remove invocations from the RBAM pool, the only reason for

Algorithm 1 Naive RBAM Implementation

Pre-allocate an RBAM pool of $A_f \cdot E_{max}$ warm invocations
$t_{lastAlloc} \leftarrow$ current time
while $Pend(\text{current time}) > 0$ **do**
    $T_{wait} \leftarrow \max[0, \frac{1}{A_f} - (\text{current time} - t_{lastAlloc})]$
    Wait for T_{wait} seconds
    Allocate 1 invocation
    $t_{lastAlloc} \leftarrow$ current time
end while
for each invocation returned from application **do**
    Add the invocation back to the RBAM pool
end for

the allocation failure is all of $A_f \cdot E_{max}$ preallocated invocations are already allocated to the application. Again, as the time gap between two consecutive allocations is at least $1/A$, It requires at least E_{max} seconds to give the application that many invocations. And by the time the last invocation is allocated to the application, the first allocation reaches its execution time limit E_{max} and returns to the pool. This means the number of invocations allocated to the application will never reach $A_f \cdot E_{max}$, thereby allocation must success, a contradiction. □

This implementation proves that the implementation of an RBAM with a finite guaranteed invocation rate and maximum execution time is feasible.

5.3.2 Real-time Serverless

We have built an RBAM implementation named Real-time Serverless (RTS) to demonstrate RBAM capability with practical workloads. The system architecture is depicted in Figure 6 with many components inherited from OpenFaaS [117]. We developed an *admission control* and a *predictive container management* to support rate guarantee.

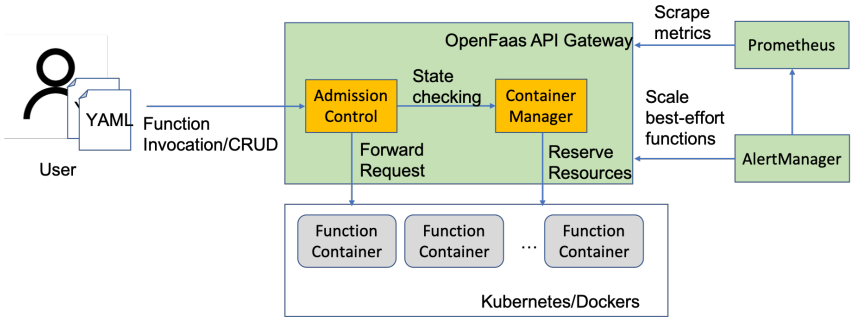


Figure 6: Real-time Serverless Implementation derived from OpenFaaS. Orange modules are added for rate-guarantee support.

Similar to OpenFaaS, RTS executes invocation inside containers (i.e., *function containers*). These containers are managed by a *predictive container manager* whose responsibility is to collect information about function execution statistics (e.g., execution time, arrival time, etc. from Prometheus) and underlying resource manager to adjust invocation allocated to each function to a right quantity that just enough to meet their rate guarantee (minimize cost). In the current implementation, the function container just follows the naive implementation algorithm and simply allocates $A \cdot E$ invocation per function.

We add admission control into the FaaS deployment and execution path to ensure rate requirements are guaranteed in an efficient manner. All FaaS deployment submission and invocation requests have to go through Admission control before getting deployed and executed. Admission control can reject deployment requests if there are insufficient resources to meet the rate guarantee. It can also reject invocation requests if the request rate exceeds the guaranteed rate, ensuring resources are shared efficiently among applications.

When the user submits an RBAM function, RTS first extracts its guaranteed invocation rate A and maximum runtime E then triggers admission control to check if there are sufficient resources for deployment. The admission control has the container manager try allocating a quantity of resources that is high enough to meet the rate guarantee. If the allocation succeeds, the container manager will hold the resources for invocation execution and admission control return deployment success. If the allocation fails, then the current resource availability is insufficient, admission control lets the resource manager roll back the allocation and rejects the deployment request.

The current implementation of RTS uses the naive RBAM allocation algorithm in the previous section to support rate guarantee in the FaaS execution path. In particular, RTS creates one waiting queue for each RBAM function and puts every invocation request received from the application into this queue. Inside the admission control, we create a timer per RBAM function, each tick for every $1/A$ second. Once a timer ticks, the admission control restarts this timer, then checks the waiting queue, if there is a waiting invocation, admission control wrap this request inside an HTTP POST request, then randomly forwards it to a free function container hosting the function invocation. Once receiving the HTTP request, the function containers unpack the request to get the invocation request and start execution. When the execution completes or reaches the execution time limit E, the function container stops the invocation, cleans up temporary data, if needed, then returns the results, if any, back to admission control. Admission control returns results back to the user and then marks the container as ready for another new invocation.

5.4 RBAM Applicability

We implement two real-time applications with Real-time Serverless to demonstrate RBAM applicability. First, we partly answer the research question Q.3.1 (i.e., Can RBAM be used to implement a specific, diverse set of demanding real-time applications?) by presenting how distributed real-time video analytic applications use RTS to guarantee their real-time

requirements. Further, the applications can also use the guaranteed invocation rate to manage their computation quality across different cost and workload configurations with high robustness. Next, we demonstrate how RBAM can be generalized for a different form of performance guarantee (Question Q.3.2) by translating stream processing models into RBAM deployments to construct *real-time processing guarantee*. This guarantee not only enables stable stream processing with high latency independent of resource configurations but also enables flexible application reconfiguration for different deployment purposes.

5.4.1 Distributed Real-time Video Analytic

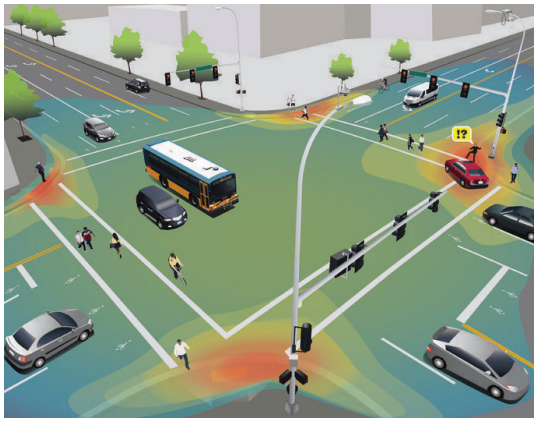


Figure 7: Traffic Monitoring: An example of distributed real-time video analytic applications

Distributed Real-time Video Analytics are popular representatives of bursty real-time application. The application collects video recorded by multiple cameras installed over a specific region to extract useful information or to take an action when an event happens, both usually lead to a bursty demand that needs a real-time response. One example of such application is traffic monitoring which continuously analyzes video streams of traffic recorded by cameras installed at intersection (Figure 7). The application is interested in unusual events such as car crashes or a pedestrian falls down. etc. Once such event happens, it triggers an in-depth analysis to understand the situation for making proper decisions (e.g., call police, broadcast warning signal, etc.). The in-depth analysis uses a sophisticated model on video streams of high resolution, and eventually creates significant high resource demand. Further, base the event is crucial, the analysis needs to be processed as fast as possible. Prolonged computation latency decrease application value/quality (e.g. delay to call an emergency medical service after an accident may lead to severe consequences).

We use FaaS to implement video processing functionalities. Every time an in-depth analysis is triggered, distributed cameras will send high resolution video frames to the cloud, each request an FaaS invocation to process the video frame content. Since in-depth analyses only required by some unexpected event, their emergence generates a burst to the FaaS system with a fixed invocation rate equal to the video frame-per-second. Consider an

in-depth processing burst with n video frame $E_1, ..., E_n$ ordered by arrival time. Since the frame processing does not have a hard deadline, we evaluate the video frame processing of a frame E_i with a new metric: *Value* V_{req}, defined as

$$V_{req}(i) = V_{max} e^{-\frac{\delta_i}{\tau}} \tag{17}$$

where

- V_{max} is maximum value the frame processing can achieve. For simplicity, $V_{max} = 1$.
- δ_i is FaaS allocation latency.
- τ presents the application's need for fast video frame processing. An application with high τ has a stricter deadline and requires a higher speed of invocation allocation for request processing.

Clearly, the value decays as allocation latency increase. The speed of the decay depends on τ. The higher τ, the bigger $V_{req}(i)$ drops per time unit. Figure 8 shows how value decreases as latency increases. Here, τ is chosen to make the value drop by half per minute.

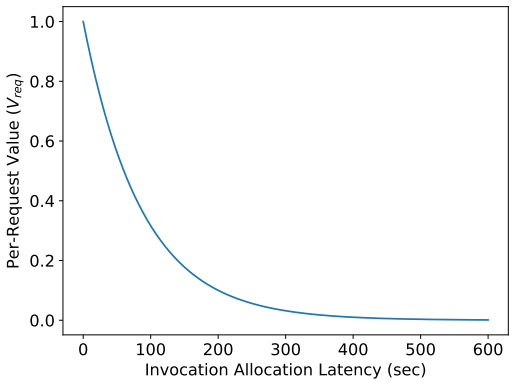


Figure 8: Value decay as invocation allocation latency increases.

$V_{req}(i)$ are aggregated to compute *burst value*:

$$BurstValue = \sum_{i<D} V_{req}(i) = V_{max} \sum_{i<D} e^{-\frac{l(i)}{\tau}} \tag{18}$$

We use *BurstValue* to evaluate the efficiency of FaaS implementations of video analytic applications.

We first compare RBAM versus regular FaaS implementations of the distributed real-time video analytic application via simulation with synthetic workloads. The workloads are sequences of in-depth analysis (i.e., bursts) generated with with following burstiness configuration:

- *Burst arrival rate* follows a Poisson process with average rate $\lambda = 0.3/hour$.

- *Burst duration D* (e.g., number of video frame per burst) varies according to Gaussian distribution with mean $D = 2$ min.
- *Invocation rate H*: fixed 30 invocation per second (equivalent to a 30 fps video stream).
- *Invocation execution time*: every frame processing takes $e = 5$ seconds.
- $\tau = 2,607$ – value decays 1/2 per minute

We vary FaaS invocation rate A from 0 (regular FaaS) to H (rate-guaranteed equal invocation rate) and use Algorithm 1 to simulate rate-guarantee allocation. Thus, assuming video frames get new invocations in FIFO fashion, the allocation latency of a video frame E_i is

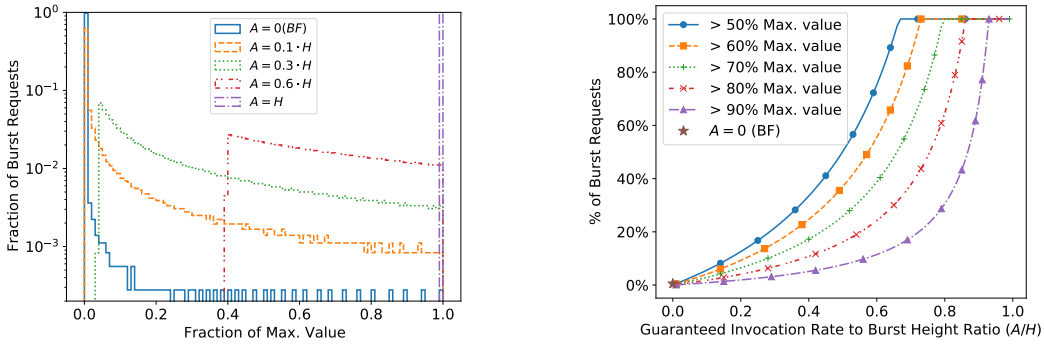
$$\delta_i = i(\frac{1}{A} - 1) \tag{19}$$

In the case of regular FaaS, $A = 0$ (i.e., best effort) which gives the application no guarantee to obtain additional invocations rather than the ones allocated at the normal state (i.e., no in-depth analysis). Since the burst load is significantly higher than the normal load, we assume 1 invocation is sufficient for the normal load, but during a burst, this quantity is not enough. A request arriving at i has to wait for all of its preceding ones ($i \cdot H$ in total) to complete, this causes processing latency:

$$\delta_i = i(H \cdot e - 1) \tag{20}$$

Real-time Processing Efficiency. Figure 9a shows the distribution of per-frame guaranteed value using FaaS at different guaranteed invocation rate A. For the baseline, at $A = 0$, the lack of rate guarantee forces the application to rely on invocation allocated at normal state for value guarantee so only a tiny fraction of the maximum value is delivered (the request values mostly at left). With $A > 0$, as the guaranteed allocation rate, A, is increased, the application can service a burst by allocating invocations more and more rapidly. Even a small A significantly increases the number of frames achieving close to the maximum value (orange), and further increases in A (green, red) improve the situation dramatically. For example, with $A = 0.6H$, the application can ensure that all frames exceed 40% of the maximum value. And as A increases towards H, a growing frame value guarantee can be achieved, reaching 100%. This illustrates that using the guaranteed invocation rate helps applications improve computation value/quality.

Another way to think about application quality is to ask what fraction of requests achieve a particular fraction of maximum value. We plot this metric versus the guaranteed invocation rate in Figure 9b. To achieve 50% of the maximum value for even half of the frames, the application needs to use the FaaS function with A equal to half of the burst request rate H. To achieve 50% of the maximum value for 100% of the frames, an A of $0.67 \cdot H$ is needed. At the high end, to achieve 90% of the maximum value, $0.85 \cdot H$ is required for 50% of the frames, and 0.9 for 100% of the requests. At $A = H$, the application can deliver



(a) Distribution of per-frame value

(b) Fraction of frames achieving a specific fraction of maximum value

Figure 9: Achievable per-frame guaranteed value with various guaranteed allocation rate

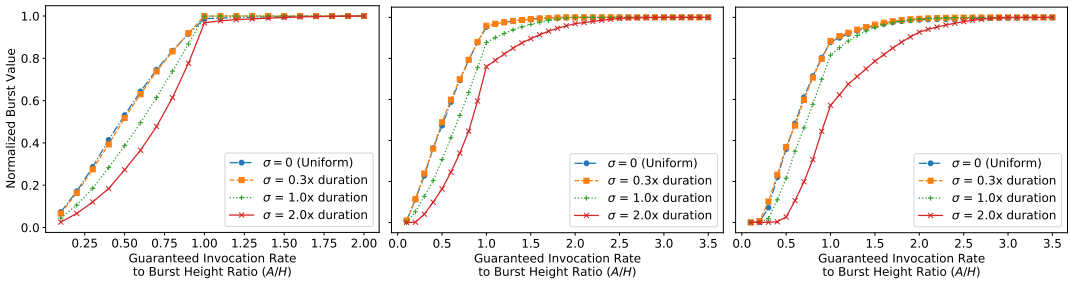
100% of the maximum value. This illustrates that RBAM enables bursty applications to provide a guarantee of high quality. The results are essential because they suggest that with a proper choice of A, applications are able to meet *any* quality target. Such capability unlocks rational designs that open more space for applications to operate and exploit resources more efficiently.

In summary, our analytical model shows that adding RBAM enables bursty, real-time applications to guarantee high computation value. In fact, the results show that the guaranteed invocation rate is the critical enabler of high value. By configuring FaaS guaranteed invocation rate A to match burst demand H, the application is guaranteed to meet its computation deadline with zero value degradation. Furthermore, even at $A < H$, RBAM can still guarantee a fraction of application proportional to A/H ratio. This enables rational design for quality where the application can utilize the guaranteed invocation rate as a quality control parameter to balance quality with other factors such as cost.

Robustness against Burst Duration Variability we vary burst duration variability by generating workloads at different duration standard deviations (σ). Figure 10 shows the achievable guaranteed burst value at different guaranteed invocation rates normalized by the maximum burst value in three duty factor scenarios: high ($DF = 0.25$), medium ($DF = 0.1$), and low ($DF = 0.01$) where DF is burst duty factor, defined as

$$DF = H \cdot D \tag{21}$$

At low duty factor, changing σ does not cause many effects on guaranteed burst value (Figure 10a). However, as the duty factor increases, value deterioration becomes more severe and high variability bursts will experience a worse impact. At duty factor $DF = 0.1$, burst with $\sigma = 1$x mean duration suffers 15% burst value loss at $A = H$ (Figure 10b), and if duty factor jumps to $DF = 0.25$, the loss increases to 19%. If σ is doubled to 2x duration then the loss is 2.2x to around 42% (Figure 10c). However, value reduction can be solved by simply



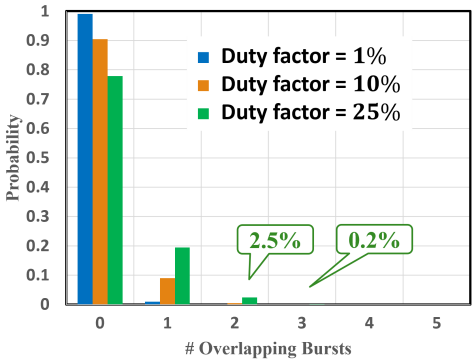
(a) $DF = 0.01$ (b) $DF = 0.1$ (c) $DF = 0.25$

Figure 10: Burst value vs. Guaranteed Allocation Rate (A) with varied burst duration standard deviation (σ)

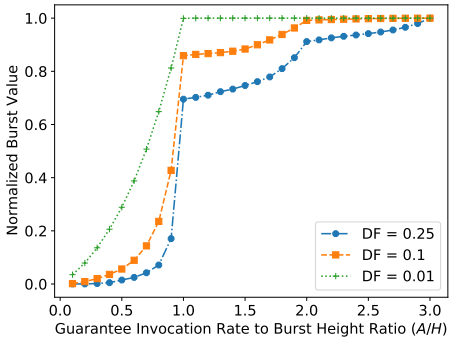
increasing the guaranteed invocation rate. For example, $A = 2 \cdot H$ will let the application achieve 100% burst value from bursts with $\sigma = 1$x duration. Even for highly variable bursts of $\sigma = 2$x duration, a guaranteed invocation rate of $3 \cdot H$ is sufficient. Also note that burst variability only takes effect at high duty factor, but in response, only a small allocation rate increase is needed to saturate the impact. Even at $\sigma = 2$x duration, rising duty factor from 0.01 to 0.25 (25x demand increase) only requires 3x guarantee invocation rate increment (i.e., 3x resource commitment increase). This confirms the robustness of RBAM against burst variability.

Robustness against Burst Interference In practice, the application may need to process data from multiple sources (e.g., video analytics receive video frames from multiple cameras). This creates a chance for *burst interference* – multiple bursts happen concurrently that dramatically increase demand for new FaaS invocations. We simulate this phenomenon by varying the burst duty factor. At high duty factor, bursts arrive more frequently and last longer, increasing the probability of two or more bursts occur concurrently.

Figure 11a shows the probability of having one, two, and more bursts at a time under different duty factors. There are two important observations from the figure. First, the high duty factor increases the chance of burst interference as we explained above. For example, at duty factor $DF = 1\%$, the chance to have a two-burst interference is only 0.005% while $DF = 25\%$ makes the chance go up to 2.5%. Second, varying the duty factor is also equivalent to varying burst demand. Thus, we can consider increasing the duty factor from 0.1 to 0.25 as an increase in the workload demand by 25x. However, due to the burstiness structure, this does not increase the bursty load by the same amount: at $DF = 0.1$, The occurrence chance of more than three bursts at a time is extremely low, less than 0.00001%. Increase demand by 25x, at $DF = 0.25$, more than six bursts at a time has the same chance of occurrence. From the resource allocation point of view, this means at the same risk level, we need only 3x more resources to handle 25x more loads. RBAM users can exploit this property to configure the rate efficiently.



(a) Probability of number of burst overlapping



(b) Burst value vs. guaranteed invocation rate.

Figure 11: Burst interference probability and achievable guaranteed burst value at different duty factor (varying λ)

Figure 11b show guaranteed burst value at various duty factors. Note that at burst interference, demand is doubled, tripled, or more depending on the number of bursts involves. This means to saturate double burst interference, the application needs to allocate resources 2x faster, for triple burst interference, 3x allocation rate is required, and so on. Thus, in the figure, the breaks of curves at $A = H$ and $A = 2H$ indicate the value reduction effect of burst interference. However, due to low interference probability, the impact is manageable: 14% of the burst value for $DF = 0.1$, and 30% of the burst value for $DF = 0.25$. Further, achieving 100% of burst value in the face of a 25x duty factor increase only requires a 3-fold increase in guarantee invocation rate. Therefore, RBAM is robust against burst interference.

Concurrent Bursty Real-time Applications One can think of high duty factors as a single application with many events or as a combination of multiple independent applications with much lower duty factors sharing a single RBAM function. Thinking of the latter, we explore how higher guaranteed allocation rates can increase burst value toward the potential maximum.

We examine the potential for multiple applications to share a single RBAM function efficiently. Consider 10 applications, each accounting for $DF = 0.01$ summing to $DF = 0.1$ and 25 applications, each accounting for $DF = 0.01$ summing to $DF = 0.25$, and so on as shown in Figure 12. For low burst value (< 0.5) there is little difference in the required A. For moderate values, the difference grows but at a deeply sublinear rate. For example, for the value of 80% potential maximum value, an increase from 1 to 25 applications require a 2x increase in A, resulting in only 2x more cloud resource commitment.

The curves cover the guaranteed invocation rate needed for a wide range of duty factors from 0.01 to 0.25 but they are very close to each other indicating that only a small increment of allocation rate is sufficient to deal with a significant increment of burst demand. At 90% max guaranteed value, the multiple is even smaller. requiring a 1.6x guaranteed invocation

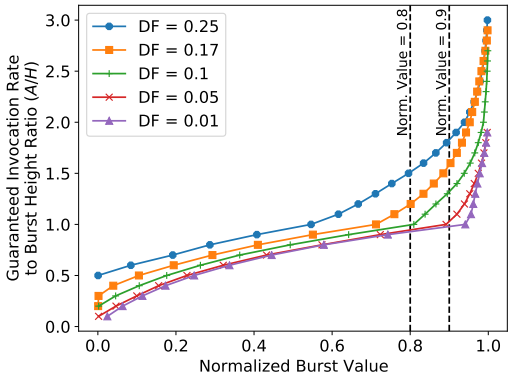


Figure 12: Allocation rate needed to achieve burst value fraction (at varied duty factors).

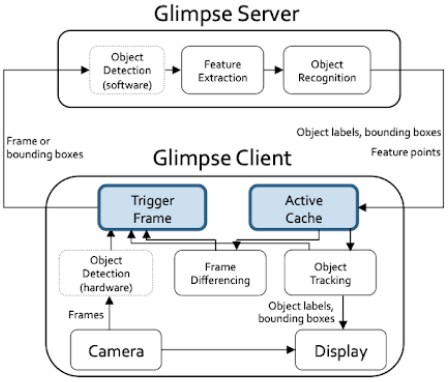


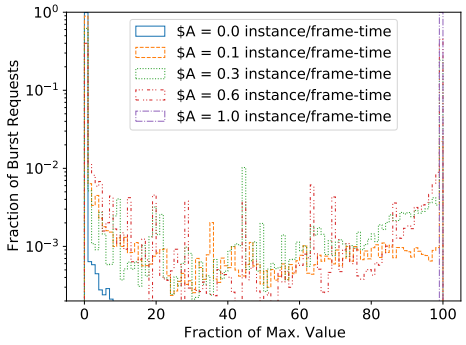
Figure 13: Glimpse System Architecture (from [51]).

rate increase for a 25x increase in the number of applications. These results suggest that RBAM scales well – supporting a growing number of bursty, real-time applications at high quality with a slowly growing number of resources. Our results show this growth is deeply sublinear, suggesting that RBAM may be best implemented as a shared cloud service (not privately by a single application) and that doing so may be quite cost-attractive for cloud providers.

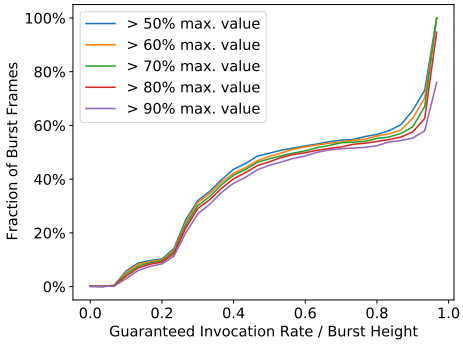
Case Study: Traffic Intersection Monitoring We model a Glimpse-like pipeline with a client and server (the cloud) that processes video frames considered interesting by the shallow processing at the client [51] (see Figure 13). Glimpse uploads frames to the server for object detection. Our empirical measurements characterized the server-side frame processing cost at 20x for object detection, but for richer analytics, this ratio could be much higher. We use a rush-hour traffic video captured from a traffic camera in Southampton, NY at the intersection of County Rd. 39A and North Sea Rd. and available from [136]. The video is 30 frames per second at a resolution of 1920 $\times$ 1080 color pixels per frame.

	Burst Duration (frames)				Burst Height			
	Mean	StdDev	Min	Max	Mean	StdDev	Min	Max
Night	116	186	30	2,445	21	3	20	80
Day	120	216	30	2,323	20	3	20	80
Rush hours	917	1293	30	7,464	48	23	20	200
Overall	197	503	30	7,464	24	11	20	200

Table 3: Burst Statistics for Traffic Video



(a) Value Distribution vs. A_{RTS}

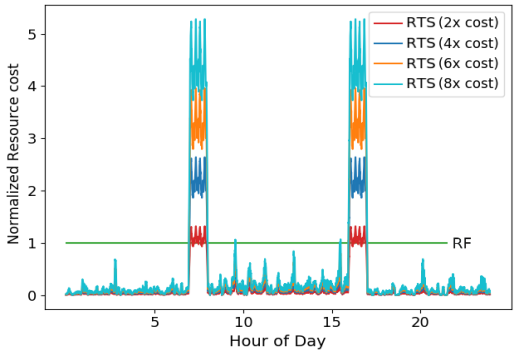


(b) Value Distribution vs. Guaranteed Invocation Rate

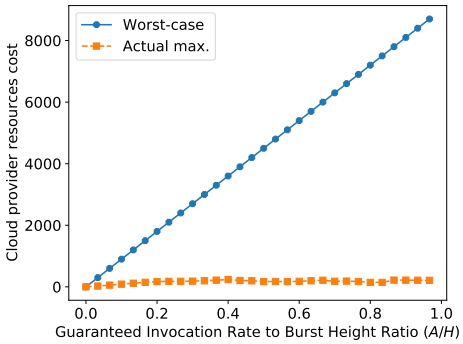
Figure 14: Per-Frame Value achieved by Application varying Guaranteed Invocation Rate

Because we are interested in analyzing complex behavior such as erratic driving, reckless walking, or traffic incidents, we use an efficient model [84] with ResNet trained on KITTI dataset[76]) to process the video and annotate it with object appearance and departure intervals. These object intervals are combined and collectively create the bursts (see Table 3). To scale up to a full 24-hour from our short, rush-hour clip, we replicate it to create two 60-minute segments (morning and afternoon rush hour). We scale the time base by 20x while holding object interval duration constant, creating an 8-hour segment of lower traffic (daytime). Finally, we scale the time base by 40x while holding object interval duration constant, creating a 14-hour segment of the lowest traffic (nighttime). The total number of bursts is 2,311 and the duty factor is 0.175 for the 24-hour period. The number of objects present in each frame multiplying the server-side frame computation cost ratio (20x) defines the burst height.

Quality Guarantee We first explore the basic characteristics of the traces, as shown in Table 3. The burst durations are much shorter than those explored in Section 5.2 with an average burst duration of 7 seconds (210 frame-times), as shown in Table 3. Moreover, both the burst duration and height are highly variable within each part of the day.



(a) Traffic Monitoring application cost for varied RTS cost scenarios (5-minute sliding average)



(b) Cloud provider resource cost for guaranteed allocation rate

Figure 15: RTS can be efficient in terms of both application cost and cloud provider resource cost

Figure 14a shows the value distribution of the video trace; while similar qualitatively to Figures 9a the real burst trace is much noisier. The traffic analysis quality benefits from increasing A, are shown in Figure 14b. Three curved sections are visible and correspond to the three different operating points – rush hour, daytime, and nighttime. At A as little as $0.25 \cdot H$, all of the nighttime value is captured. At A of 0.9, the daytime value is captured. In Figure 14b, we see flat curves and very little separation by a fraction of the per-request value. This reflects a difficult workload for increases in A to improve application quality. To achieve full quality on the intense activity during rush hour (10 objects in frame) requires $A = H$.

Results from Figure 14 confirm that using real-time serverless guarantees the traffic monitoring application value. And by increasing A, the application can improve the guaranteed quality, although compared to the synthetic data, quality increase much slower due to the extremely high duty factor during rush hours. Furthermore, at $A = H$, real-time serverless enables the application to achieve the maximum target value. This confirms the robustness of RTS against realistic workloads.

Rate Guarantee Realization Cost Figure 15a shows how the burst load varies over the 24-hour period. To illuminate how the RTS system responds with time, we overlay the application cost of both an RTS implementation at various cost ratios (k). The baseline is Reservation FaaS (RF) which pre-allocates just enough invocations in advance to get 100% value. The benefits of dynamic management are clear. Considering the full 24-hour day, the RTS approach is 8.3x less expensive for $k = 2$; 4x less expensive for$k = 4$, and 2x less expensive for $k = 8x$. In short, the RTS resources are 16x more valuable than traditional UI resources.

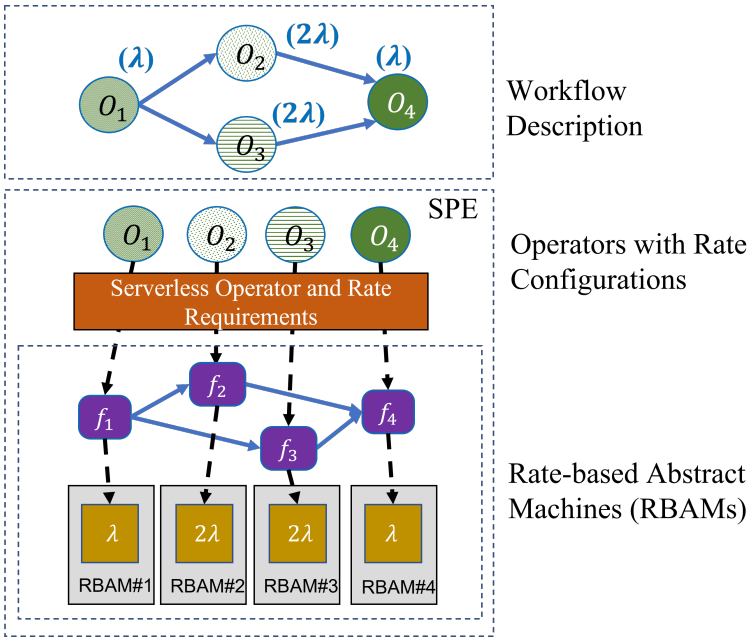


Figure 16: The Rate-based Abstract Machine (RBAM) approach to stream-processing: Operators are wrapped by FaaS functions, providing invocation-level dynamic resource management. One RBAM for each operator ensures its required invocation rate.

As discussed in Section 5.3.1, the guaranteed allocation rate can be achieved at cloud provider resource cost linear in $A \cdot E$. However, we find that for this application, the effective cloud provider resource requirement is much lower as shown in Figure 15b. This is because the typical invocation execution is much smaller than E, so many invocations return much sooner than the worst case. Consequently, resource cost is reduced 70-fold to only 123 invocations. We expect savings such as this to occur in many cases but will vary in magnitude.

In summary, real-time serverless allows applications to allocate resources on burst occurrence and thus, autoscale the cost to computation demand. Furthermore, short actual instance runtime per invocation allows the cloud to quickly reuse RTS instances for multiple burst frames to scale actual allocation to resource commitment, thereby improving cloud resource management flexibility. These capabilities make real-time serverless cost attractive to not only applications but also cloud providers.

5.4.2 Stream Processing

Since any FaaS function is an RBAM function with a guaranteed invocation rate $A = 0$, all FaaS-based applications can be converted to RBAM-based without nontrivial modification. Also, the guaranteed invocation rate can be reconfigured by the applications without the need of exposing any of their internal information and structure. Thus, at the very least, RBAM can be used to support any application FaaS does. Furthermore, the rate guarantee it offers can open more capabilities for the application. We demonstrate this by using RBAM to support

stream processing applications – an application class that is typically used for handling an endless amount of data that is continuously generated from multiple sources. The data are organized as separate tuples, each gets processed by going through a DAG of processing units called *operator*. Each operator is responsible for a small part of the processing. Once an operator finishes processing a tuple, it will pass the output to its downstream in the DAG for further processing and so on until reaching final results at a sink operator.

Current stream processing applications rely on the worker execution model which dispatches application computation components, i.e., *operators*, onto worker abstraction. Depending on design choice and infrastructure support, worker abstraction is typically realized as threads, processes, or containers. Worker performance is tied to the resource configuration of worker realization, which highly varied under changes in underlying systems (e.g., hardware, OS scheduler, etc.) and execution environment factors (e.g., collocated applications, hardware failures, etc.). This makes application performance transparency and predictability extremely difficult, if not possible, to achieve.

We can use RBAM to resolve these problems. Figure 16 shows a stream processing application running over RBAM instead of the worker execution model. Each operator is mapped into an RBAM function and their connections are encoded as RBAM call chains with tuples passed as invocation arguments. For example, in Figure 16, each operator O_1, O_2, O_3, O_4 becomes a separate FaaS function f_1, f_2, f_3, and f_4, respectively. When a tuple arrives, f_1 is triggered and processes the tuple. After completion, it requests new invocations of f_2 and f_3, passing its output along with these requests. f_2 and f_3's new invocations unpack the request, treat the output as their input then continue the tuple processing, and so on.

By this approach, the guaranteed invocation rate associated with an RBAM function specifies the processing rate of the operator mapped to the function. This makes performance predictability possible to stream processing applications as they can tell whether the load can be supported given a specific deployment by comparing its guaranteed rate configurations and operator input rates. Further, if the load exceeds the rate guarantee configurations, the application can easily recover performance stability by reconfiguring the rate guarantee configuration to match the new load. As rate guarantee configurations are RBAM-specific configurations, independent of the underlying resource configuration, the application can reuse the configuration for any deployment environment yet still get the same result. That resolves the performance transparency challenge.

Storm-RTS. To demonstrate such new capability, we implement a new stream processing engine called Storm-RTS. Storm-RTS use RBAM as a core execution model to host applications' operators and leverage Apache Storm API to deliver state-of-the-art stream processing development and management supports. Storm-RTS is designed to offer the following functionalities:

- *Workflow performance stability*: achieve desired throughput and latency across distributed configurations, reconfiguration (migration) and varied competitive loads

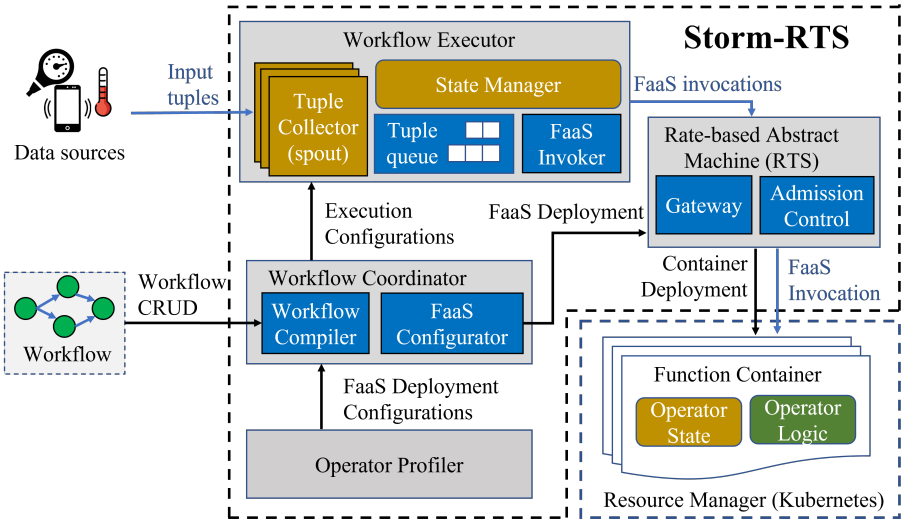


Figure 17: Storm-RTS Architecture: Operator Profiler, Workflow Coordinator, Executor, and Rate-based Abstract Machine

- *Modular (local) resource management*: can partition workflow across multiple sites or datacenters. Individual site resource managers can independently decide if a workflow can placed and meet its performance requirements
- *Compatibility*: support Storm workflows and features with similar efficiency and modest change.

The key elements of Storm-RTS are shown in Figure 17. At the high level, Storm-RTS has four main components, each is responsible for one of the functionalities listed above.

- *Workflow Coordinator* responsible for enforcing performance stability. This component translates workflow operators received from developers into FaaS functions and associated them with appropriate configurations allowing the workflow to sustain the desired load. `Workflow coordinator` also automatically reconfigures FaaS configurations to protect workflow performance from disruptions such as competitive loads, workflow reconfiguration, migration, etc.
- *Operator Profiler* responsible for resource requirement predictability. The component run workflow operators offline to profile their computing and memory requirements. This information is used to configure FaaS functions' resource requirements, ensuring their invocations always have sufficient resources to execute their associate operators.
- *Rate-based Abstract Machine (RBAM)* responsible for enabling modular resource management. FaaS functions created by `workflow coordinator` are deployed separately inside RBAM allocations. Each RBAM allocation is a resource contract between Storm-RTS and the underlying resource manager (e.g., Kubernetes). Once established,

the resource manager guarantees sufficient resources given to RBAM allocations to enable them to execute at configured rate regardless of other competitive loads.

- *Workflow Executor* responsible for executing workflows and compatibility supports. `Workflow Executor` collects tuples from data sources and then triggers corresponding FaaS invocations to start workflow execution. `Workflow Executor` also monitors and orchestrates workflow executing through reading `Operator State` deployed on every function container. `Workflow Executor` reuses Storm's modules to format and process monitoring data ensuring Storm-RTS offers similar data processing supports as Storm.

Storm-RTS Implementation

Workflow Coordinator Workflow developers submit workflow descriptions directly to the `workflow coordinator`'s `compiler`. The description includes workflow topology and rate configuration. Rate configuration consists of a *desired rate* λ that developers expect the workflow to handle and per-*operator parallelism* μ_i representing the ratio of each operator's expected input rate and λ.

The `workflow compiler` extracts operators' logic from workflow topology and then encapsulates each of them inside a FaaS function. The operator starts processing tuples by invoking the corresponding FaaS function with the tuples provided as invocation arguments. The whole processing happens inside the invocation of its wrapper function and finishes once the invocation terminates.

Each FaaS function is configured by `FaaS Configurator` to determine its (i) rate guarantee, (ii) per-invocation resource requirement (mainly CPU and memory), and (iii) maximum invocation runtime (i.e., timeout). In particular, `FaaS configuration` lets `Workflow Profiler` execute operator logic offline to determine invocation resource requirement and maximum execution time. Meanwhile, the function rate guarantee A_i is configured to the number of invocations expected to invoke per second if tuples are generated at the desired rate λ:

$$A_i = \frac{\lambda \cdot \mu_i}{b} \tag{22}$$

where b is the number of tuples to be processed in one invocation (e.g., batch size) also provided by the `workflow profiler`. This A_i guarantees at least one invocation available for the operator wrapped by the FaaS function to process all incoming tuples sent at any rate less than or equal to λ, thereby satisfying the performance stability requirement.

For deployment, the `FaaS Configurator` sends FaaS functions and their configurations to RBAM to check if the underlying resource manager can support their guarantee. If it can then the workflow coordinator triggers the workflow executor to begin execution. Throughout the execution, `FaaS Configuration` keeps it informed by RBAM and underlying resource management changes. If any of the changes would affect workflow performance, it will reconfigure FaaS deployment to ensure performance stability.

Operator Profiler The profiler characterizes operator execution properties by running operators offline with tuples sampled from historical input stream data. The running environment is configured to be identical to the environment targeted to execute workflow operators. Profiling is triggered by the FaaS coordinator, when a new workflow is submitted or when workflow reconfiguration is required, to collect the following information

- *Per-invocation resource (CPU and memory) requirement and maximum runtime*: the profiler executes operators on a variety of CPU and memory capacities starting with excess resources, and gradually reduces the allocation until observing a 20% execution time increase. The function wraps this operator will have its resource and runtime configuration equal to the point where the execution time starts to increase.
- *Batch size*: the profiler estimates a minimum batch size, designed to ensure reasonable efficiency by comparing the tuple processing for the operator natively, and then with the FaaS wrapper. An efficiency of 70% was deemed acceptable and used to determine the batch size for this operator.

Workflow Executor For each successful deployment, the `workflow executor` creates a set of `tuple collectors` realizing the workflow source operators ("spout" in Storm terminologies) to continuously collect new tuples from data sources. The `tuple collector` batches tuples to amortize the FaaS invocation cost. Thus, new tuples are put into `queues` based on their destination. When enough tuples are available, a `FaaS invoker` retrieves a batch from the queue and requests a new FaaS invocation for the appropriate workflow operator, passing the batched tuples as an argument. Each invocation processes one batch. After completion, to pass on output tuples, the invocation calls the wrapper functions for the operators downstream, passing output tuples in batch as an argument. This allows the downstream function, in response, to extract the tuples, perform the operator computation, and call its downstream operator wrappers as needed, and so on. This mechanism forms tuple processing as serverless function chains which are self-synchronized that do not require the dedicated messaging systems as in worker-based SPEs (e.g., Storm [16] relies on Netty [22] for inter-node messaging).

Also at completion, invocations update `operator state` maintained inside their containers. These states are periodically synchronized with the `state manager` providing information for consistency, progress tracking, monitoring and recovery. Storm-RTS reuse Storm's modules to implement such functionalities. Thus, workflows executing over Storm-RTS receive supports equivalent to Storm.

Extending Storm-RTS to Cloud-Cloud and Cloud-Edge Distributing workflow execution across multiple cloud data centers or cloud and edge can produce important benefits in performance (latency, throughput, and cost) and flexibility. Opportunities include exploiting available edge resources to reduce cost, latency, upload bandwidth, or even carbon foot-

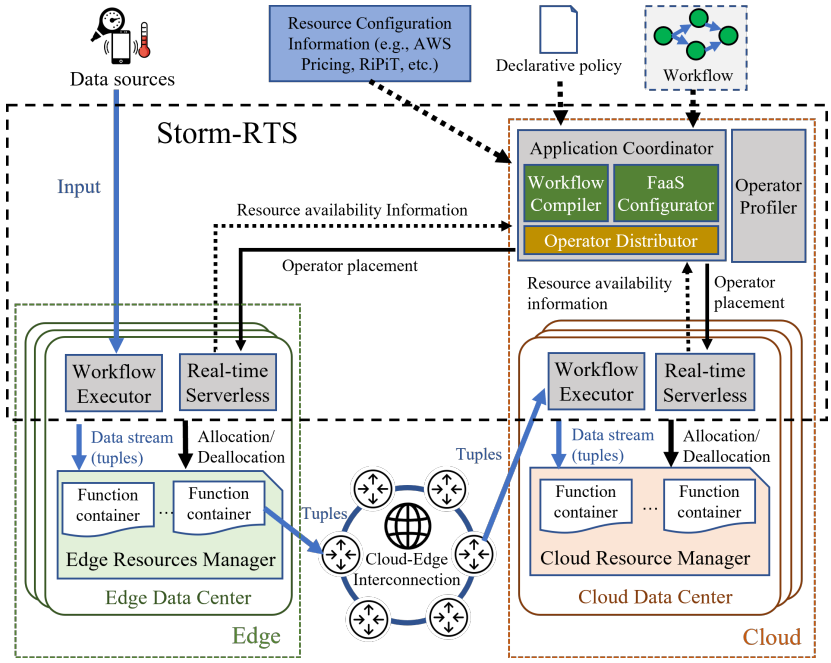


Figure 18: Storm-RTS for Edge: the application coordinator manages multi-site deployment. As before, each site has admission control and performance monitoring that implements the local RBAM guarantees.

print (selecting the lowest carbon resources). However, such opportunities come with many challenges because edge resources are both heterogeneous and can vary in availability.

Storm-RTS supports workflow flexibility across multiple clouds and the edge. First, Storm-RTS utilizes underlying resources via serverless abstraction so as long as the cloud or edge data center supports FaaS, the resource can be accessed through Storm-RTS. Resource heterogeneity can be masked via FaaS, and performance assured via operator profiling and admission control. Second, edge resources with varying availability may require workflow reconfiguration. By leveraging the rate-based abstract machine, Storm-RTS ensures that such reconfiguration will not affect workflow performance, enabling applications to optimize their deployments for cost, carbon, or other criteria.

Using Storm-RTS, as shown in Figure 18, stream-processing applications can be deployed over multiple clouds and even cloud-edge configurations – with minimal application change or retuning. Each cloud or edge data center runs Storm-RTS as in Figure 17, but now the `workflow coordinator` is promoted to `application coordinator` and now orchestrates FaaS deployments across the datacenters (via the Storm-RTS runtimes in each location). Apart from original components, the `application coordinator` adds an `operator distributor` that places the FaaS encapsulated operators across data centers, implementing the desired application policy.

Common policies include keeping operators close to data sources (often at the edge). If multiple data centers can host an operator, the coordinator implements the application's

`deployment policy`, which picks application configurations from amongst the candidates. The performance stability ensured by the Storm-RTS model across configurations is the key to enabling simple declarative specifications of policy.

For example, if edge resources are zero-cost, when available, a policy that simply minimizes total deployment cost would push operators to the edge when it is idle, and pull them back to the cloud when it is not. If there is variable pricing at the edge – time of use rates - then the edge might be favored at some times and not others. If sustainability is the objective, then minimizing carbon emissions (due to brown power use), the application coordinator might push operators to the edge when solar panels create plentiful green power, but back to the cloud datacenter, when the solar panels stop generating sufficient green power. Storm-RTS implements policies by collecting and assessing two sources of information:

- *Resource configuration:* (e.g., resource cost from cloud provider pricing pages, Carbon intensity information from RiPiT [23], etc.) to give insights of cloud/edge resource properties for efficiency exploitation.
- *Resource availability*: collected from resource managers of data centers. The application coordinator also communicates with admission controllers (RTS systems) to determine if an operator placement (and desired rate) is feasible at any particular site.

The tuple movement between Storm-RTS operators is implemented at FaaS invocations, so no other middleware beyond HTTP and TCP/IP is required for communication between systems.

Evaluation To validate if Storm-RTS design and implementation meet these requirements as well as demonstrate new capabilities enabled by RBAM, we design and carry out sets of experimental evaluations, comparing Storm-RTS with three state-of-the-art worker-based stream processing engines: Apache Storm, EdgeWise, and Dhalion. Stream processing engines are fed by workloads collected from RIoTBench under different load shapes, resource configurations, competitive loads, deployment policies, and migration scenarios to answer the following questions:

- *Efficiency*: Does Storm-RTS achieve comparable performance versus worker-based processing engines?
- *Performance Stability*: Does Storm-RTS provide stable performance corresponding to application rate guarantee configurations regardless of execution environment changes?
- *Flexible Reconfiguration*: Does Storm-RTS let applications flexibly reconfigure themselves in response to the dynamic changes of the running environment to achieve high-level objectives (e.g., minimize cost, minimize Carbon footprint)?

We use *RIoTBench* benchmark suite [126], designed specifically for evaluating SPE implementations. The benchmark performs analysis over a real-world smart cities dataset

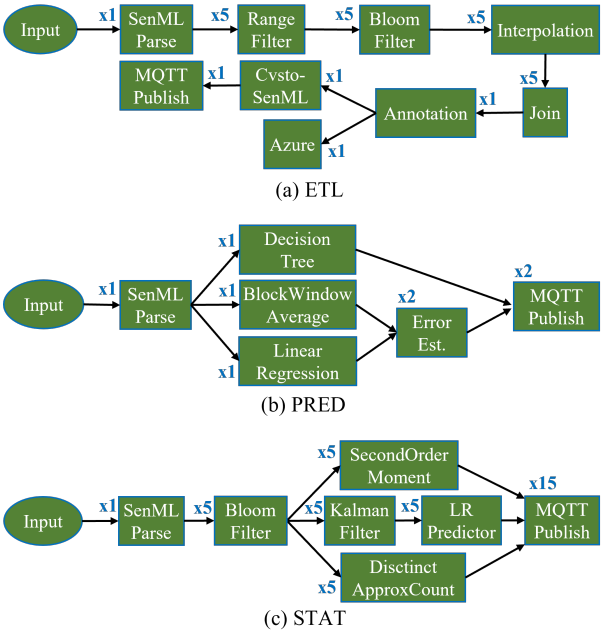


Figure 19: RIoTBench workflows.

[43]. We select 3 workflows, that capture the behavior of common stream processing settings, namely PRED (make predictions on streamed data), ETL (perform data extraction, transformation, and load), and STATS (apply statistical summarization) and are illustrated in Figure 19. The green boxes represent their operators while the number associated with each box represents operator parallelism configurations.

We compare Storm-RTS with four different stream processing engines

- *Storm* [16] a popular worker-based SPE. Workers are implemented as threads in a Java Virtual Machine. The workflow developer configures worker allocation through parallelism configuration. Worker allocation and mapping are fixed throughout the workflow lifetime.

- *EdgeWise* [74] similar to Storm, except workers are shared among operators so the SPE can assign operators to workers prioritizing those with more tuples queued to improve efficiency.

- *Dhalion* [73] a worker-based SPE supporting supports dynamic scaling. The SPE will allocate additional resources whenever workflow throughput does not match the input rate. It also frees unused resources if the workflow is over-provisioned.

- *Storm-Serverless* implements the Storm API on FaaS. Storm-Serverless' architecture is similar Storm-RTS with RBAM is replaced with OpenFaaS [117], thereby operators have no rate-guarantee.

All SPEs handle data streams using Storm 1.1.1 interface requiring essentially no modifications to the workflow source code. The one exception is that for Storm-RTS, the workflow

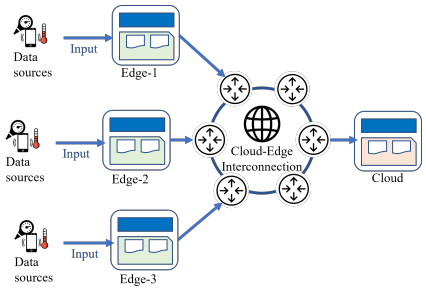


Figure 20: A Cloud-edge resource configuration

must declare the desired tuple rate-guarantee (single number) which is used to determine the underlying RBAM resource provisioning.

Experiments are conducted over three configurations

- *Cloud VM*: SPEs are installed over virtual machines provided by the public cloud vendors, including Amazon EC2 (m5zn instances), Microsoft Azure (Das_v4 instances), and Google Cloud (e2-standard instances) to evaluate SPE performance over realistic settings where they typically run over virtual, oversubscribed environment inside data centers.

- *Bare Metal*: SPEs are installed on a dedicated physical machine with no resource sharing. The machine is an Intel Xeon Gold 6138 (80 cores) with 512 GB of memory. SPEs are deployed with cgroups for resource control. We also used bare metal options from Azure (Dasv4-Type1) and Amazon (m5zn.metal).

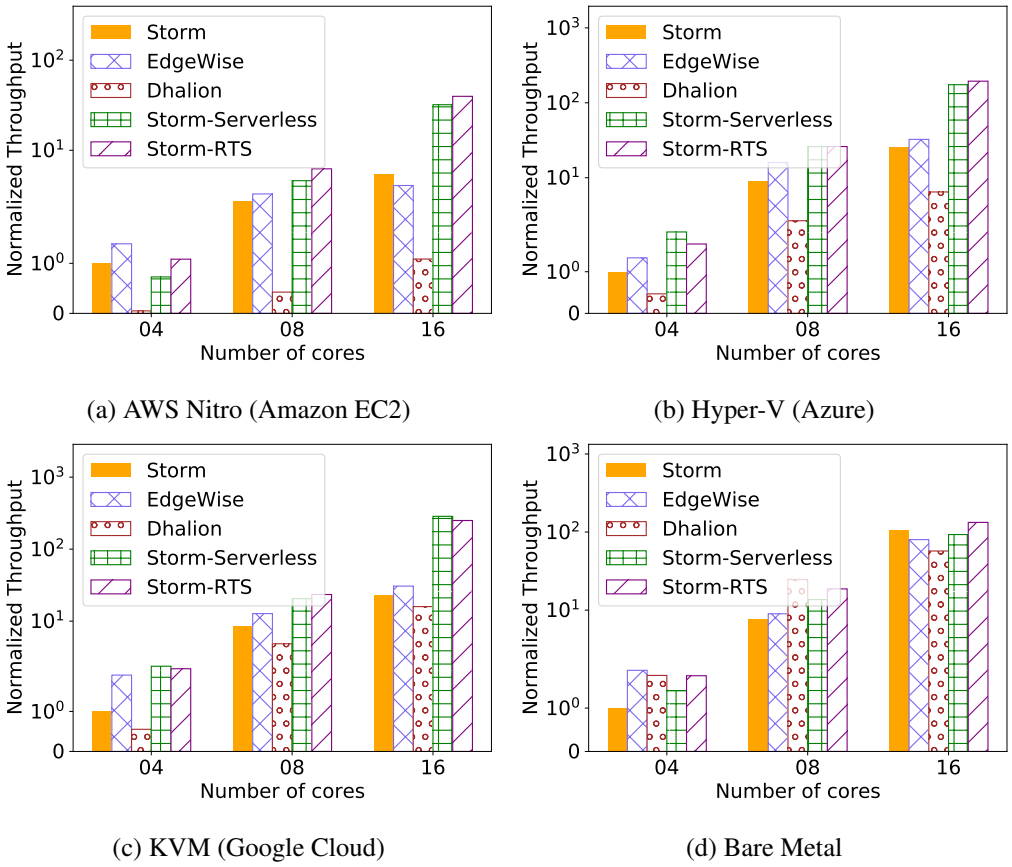
- *Cloud-Edge* SPEs are installed across Cloud and Edge data centers as in Figure 20; the data centers are emulated using Chameleon Cloud [14]. We use 4 clusters of machines. One is *cloud*: an unlimited number of machines, each with 92 cores and 192GB memory. The other three: *edge1*, *edge2*, *edge3* represent edge clusters, each has 4 virtual machines equipped with 12 cores and 48GB memory. To emulate the realistic cloud-edge interconnection, we refer to Amazon Cloud Infrastructure's network performance [30] to configure the connection bandwidth and latency. Intercloud latency modeled constant 5.5 milliseconds and 100Gbps. Cloud-edge networking is also assumed to be 100Gbps with latency randomized with Gaussian distribution with 5.5ms mean and 2ms variance.

SPEs are evaluated through the following metrics:

- *Throughput*: (tuples/second) The number of tuples received per second at the workflow's sink operators.

- *Latency* end-to-end latency of a given tuple starting when the tuple arrived at source tasks and finishing when the tuple totally consumed by the sink tasks.

- *Resource utilization*: CPU-Utilization (100% per core).
- *Cost* = CPU utilization * cost-factor (location). The cost-factor is a dimensionless relative measure of resource cost, and depends on the resource location. For the cloud-edge configurations shown in Figure 20, we set the cost of edge1, edge2, and edge3 equal to 25%, 50%, and 75% respectively relative to the cloud's 100%.



(a) AWS Nitro (Amazon EC2)

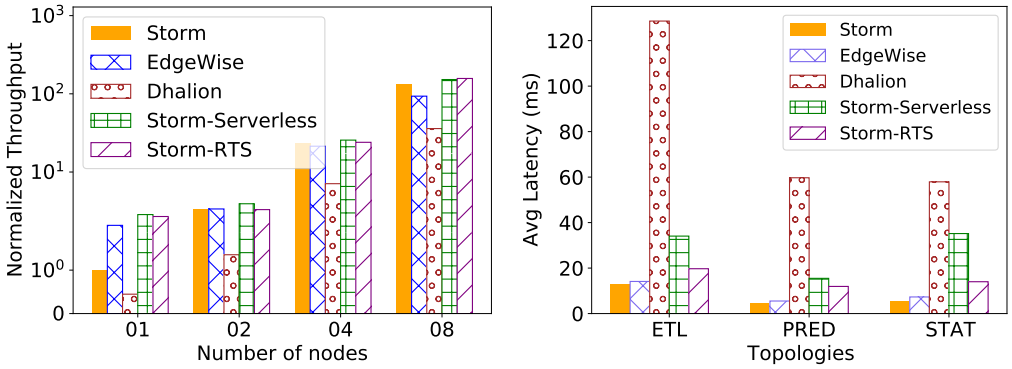
(b) Hyper-V (Azure)

(c) KVM (Google Cloud)

(d) Bare Metal

Figure 21: Maximum throughput of RIoTBench workflows on a single machine (varying from 4 to 16 cores). Geometric mean of workflows' throughput, each is normalized by Storm throughput on a 4-core machine.

Resource Efficiency We evaluate the resource efficiency of the four SPE systems, using a single workflow, and varying the tuple input rate. Each of the three RIoTBench workflows is deployed over a single machine with fixed CPU (4, 8, and 16 cores) managed by the all five SPE systems – Storm, EdgeWise, Dhalion, Storm-Serverless, and Storm-RTS. The workflows are fed tuples at a constant rate, and we gradually increase the rate until the system saturates. When tuple processing latency increases sharply and the throughput (tuple output rate) fails to match the tuple input rate, the system is considered saturated. We report the throughput just before this point, calling it the maximum throughput. All four SPEs have workflows' parallelism configurations shown in Figure 19.



(a) Throughput

(b) Average Latency

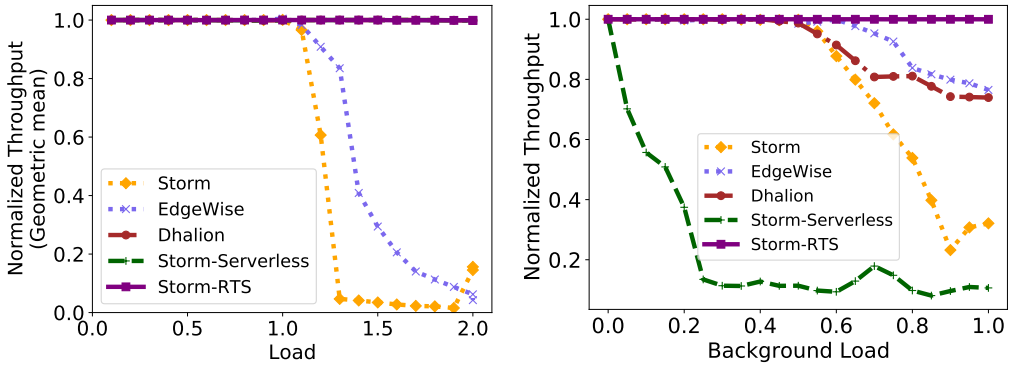
Figure 22: Resource efficiency results on Azure. Storm-RTS achieves comparable throughput and latency versus worker-based SPE across different workflows and the number of nodes.

We report the geometric mean of the normalized throughputs of three RIoTbench workflows performance for each of the four SPEs on 4 different machine configurations in Figure 21. Note the log scales. The performance of Storm, Edgewise and Dhalion scale poorly, falling slightly behind at 8 cores and badly behind at 16 cores. Because of their dynamic allocation, both Storm-RTS and Storm-Serverless scale well with system capacity, with workflow maximum throughput increases almost linearly with the number of cores. These results are consistent across all of the cloud VMs and also the bare metal configuration. These results show that the FaaS-based SPEs can achieve equal or superior resource efficiency.

Next we consider a more realistic configuration: the distributed environment of the public cloud. SPEs are deployed over multiple 4-core virtual machines, the most popular machine type in today's cloud. For various numbers of machines, Figure 22a reports the geometric mean normalized throughput for each SPE, running on Azure. The other resource configurations (two clouds, one bare metal) are both omitted, because their results are the same as we have presented for Azure. In this case, all four SPEs have comparable performance, scaling well to 8 nodes. Also as before, both Storm-Serverless and Storm-RTS scale well, increasing throughput with the number of machines. This result confirms their resource efficiency, compared to worker-based SPEs, in a distributed computing setting.

Figure 22b shows the average per-tuple end-to-end latency of RIoTBench workflows at the steady state when the load is at around 70% of available capacity for all SPEs in Azure (we also omit other configurations due to similarity). Storm-RTS and Storm-Serverless keep the latency staying below 20ms, just slightly higher than Storm and EdgeWise while significantly better than Dhalion. The results demonstrate that besides throughput, Storm-RTS is also efficiently equivalent to other worker-based SPEs in terms of processing speed.

Performance Stability Serverless-based SPEs allocate resources on-demand, enabling workflow performance to scale dynamically without configuration tuning. To illustrate this,



(a) Dynamic Scaling

(b) Performance Isolation

Figure 23: Storm-RTS can flexibly reconfigure for various workloads and protect workflow performance from collocated applications while other SPEs fail to do so. (Results from Azure VMs only, other configurations are omitted due to similarity).

we run each RIoTBench workflow, one at a time, at varying tuple input rates. The system has ample resources, but we keep the workflow parallelism and rate configurations fixed as the tuple rate is increased.

The results are presented in Figure 23a. The x-axis values are the tuple input rate normalized by the saturation rate (maximum throughput) of the Storm system. The y-axis values are the geometric mean of workflow throughputs normalized by input rate. A perfect system would produce a flat line across the top – full performance with no saturation.

Our results show that all five SPE systems scale well up to Storm's saturation rate (normalized to 1.0). Beyond the tuple rate, among worker-based SPEs, only Dhalion with dynamic scaling support can handle the load. Storm and EdgeWise static worker allocations are both overwhelmed, causing their throughput to drop. At a saturation ratio of 1.5, both of their throughputs are below 20% of the input rate, and at 2.0, their throughput drops further approaching 0%. In contrast, Storm-Serverless and Storm-RTS perform dynamic allocation, using the underlying serverless dynamic allocation to acquire more resources and support higher tuple processing rates. As a result, their performance is not limited by workflow configuration, and continues to match the growing tuple for all workflows well beyond 1.0x and even 2.0x the Storm saturation rate.

The results above reveal the configuration inflexibility of the worker-based model. Any changes in workflow and input tuple rate require worker configuration adjustment, either manual or automatic, to achieve desired performance. On the other hand, serverless-based SPEs do not require any parameter tuning to meet performance goals. This eases deployment effort.

We consider the case of multiple workflows competing for shared resources. This is a common occurrence in production settings and can lead to performance interference. To evaluate performance isolation in this situation, we run each of the RIoTBench workflows

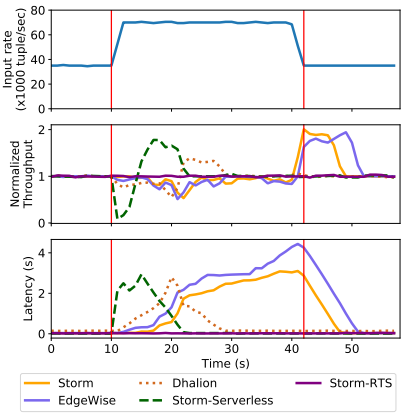


Figure 24: Storm-RTS guarantees the performance of bursty workloads while other SPEs fail to do so.

with one that competes for resources – SCAN. The SCAN workflow is a single-bolt workflow that performs expensive arithmetic operators on input tuples, so it competes for CPU cycles with the foreground RIotBench workflows.

In Figure 23b, we report the geometric mean of the throughputs for the RIoTBench workflows normalized by their saturation input rate. The x-axis values are normalized background load (SCAN), with 1.0 indicating the ability to consume 100% of the CPU capacity.

All worker-based SPEs – EdgeWise, Storm and Dhalion – fail to provide performance isolation, showing a performance decrease after the background load exceeds 50%. Storm-Serverless is even worse, dropping from the introduction of very small levels of resource competition. This is because Storm-Serverless depends on a traditional best-effort serverless system. The decrease is severed, and nearly 100% loss of throughput with about 30% competitive load. In contrast, Storm-RTS provides good performance isolation all the way up to 100% competitive load. The RBAM allocations used by Storm-RTS enforces rate-guarantees with strong resource isolation. As a result, workflows deployed by Storm-RTS always maintain the desired throughput regardless of background load. This demonstrates the capability of delivering performance predictability of RBAM SPEs as we discussed in Section **??**.

We consider bursty workflows whose input rate varies over time. Workflow developers can configure Storm-RTS to handle such workflows by setting the desired input rate equal to the peak input rate when the load bursts. For demonstration, we deploy a PRED workflow that operates at around 35 thousand tuples/sec on Azure VMs. However, after the 10-th second, the input rate is doubled and lasts for around 30 seconds (see the first graph of Figure 24. We execute this load with different SPEs. The workflow's throughput and latency are shown in the second and third graphs of Figure 24, respectively.

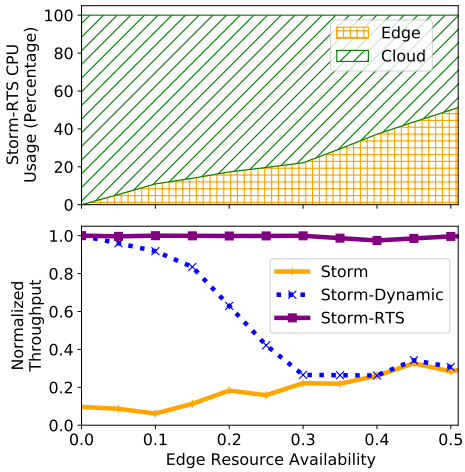
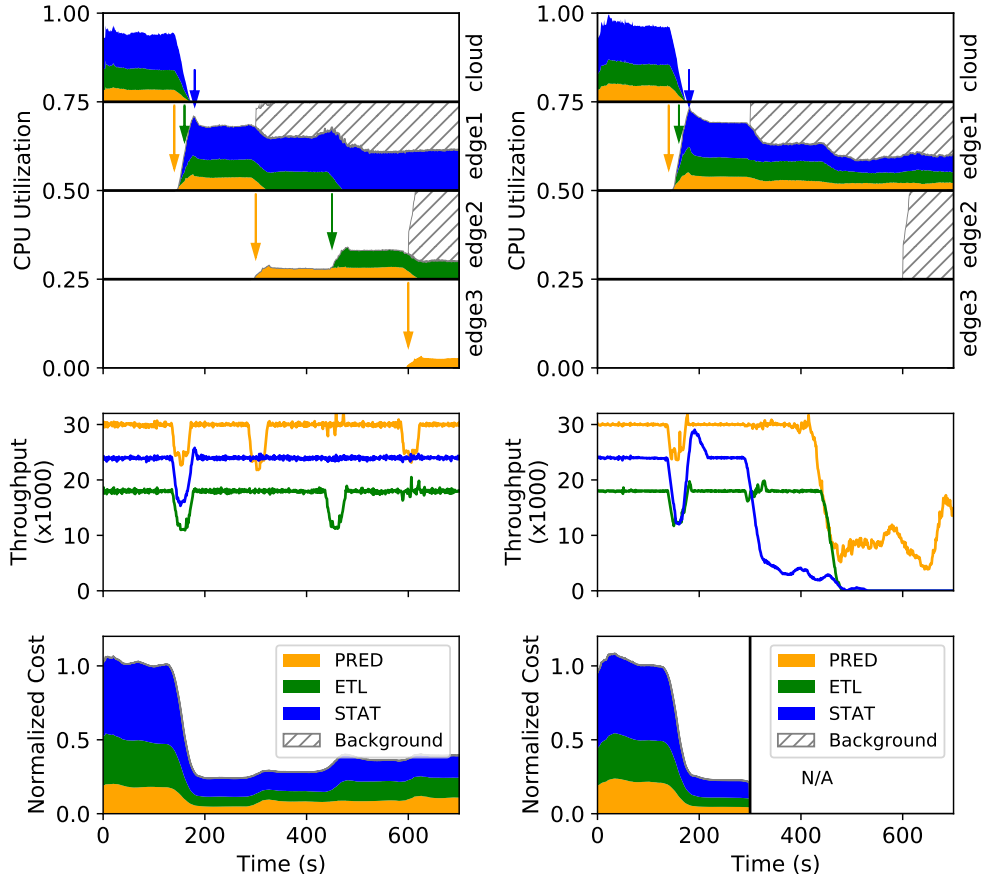


Figure 25: Storm-RTS, Storm-Dynamic, and Storm across cloud and edge. Storm-RTS has well-defined performance guarantees, enabling it to respond to resource availability changes and maintain desired throughput.

Storm and EdgeWise have their resource allocated statically, only sufficient to handle the normal load. When the burst arrives, they are unable to process the additional tuples and have to let them wait until the burst end. Subsequently, both see significant high processing latency with a noticeable drop in throughput. Dhalion and Storm-Serverless support dynamic allocation so they can scale up during the burst. However, it takes time for both to detect the burst, wait for the underlying resource to deliver new resources, and initialize them. Thus, from the beginning of the burst, their performance significantly degrades for 10-20s, around 35 to 65% of the burst period. Storm-RTS, on the other hand, has desired rate set to the burst peak (70 thousand tuples/sec) and it maintains the desired throughput and latency throughout the burst period.

Flexible Reconfiguration across Cloud and Edge Most SPEs are designed for cloud deployment, but increasingly there are opportunities for stream-processing at the edge in combination with the cloud. However, edge resources are limited, and when there is competition for resources, they may be unavailable. To maintain stable performance for stream processing workflows, ideally an SPE would be able to manage response across the cloud and edge when resource availability changes.

We evaluate Storm-RTS running a single workflow (either ETL, PRED, and STAT) across cloud and edge. We vary edge resource availability from 50% (half of the workflow can deploy at the edge) to 0% (i.e., no edge resources available), see Figure 25. In the first graph, when the availability of edge resources decreases, Storm-RTS shifts operators from edge to the cloud. This confirms Storm-RTS' ability to reconfigure workflow deployment in response to resource availability change.



(a) Storm-RTS (b) Storm-Serverless

Figure 26: Storm-RTS shifts workflows across edge datacenters, while maintaining stable performance. The flexibility enabled by Storm-RTS allows simple optimization of cost.

Next, we evaluate Storm-RTS's ability to maintain workflow performance under reconfiguration. We use Storm as a baseline. However, Storm does not respond to resource availability, statically spreading workers across available nodes in a round-robin manner. This produces a deployment with half of the workflow in the edge, and another half in the cloud. To highlight the other capabilities that Storm lacks, we enhance it by modifying the scheduler to be able to shift workers, calling it *Storm-Dynamic*. With Storm-Dynamic, when operators on the edge see performance degradation of 25%, Storm-Dynamic will rebalances workers across cloud and edge as shown in the second graph in Figure 25. The geometric mean of the three workflows' throughput normalized by their desired throughputs shows that Storm is unable to detect performance degradation or reconfigure workflow deployment to restore performance (the roles of application coordinator and real-time serverless in Figure 18). Consequently, its normalized throughput remains under 0.3, far below the desired throughput.

Storm-Dynamic addresses Storm's limitations with its dynamic worker shifting and achieves higher throughput. However, the throughput gets worse, as more edge resources available. Workers in the cloud perform differently from those in the edge. Thus, shifting a

worker cause performance changes in operators, which then cascade through the workflow. In contrast, Storm-RTS sustains the normalized throughput close to 1 through all cases, stably maintaining the desired throughput of the three workflows. Storm-RTS has workflow reconfiguration driven by a specific understanding of performance needs – RBAM rate guarantee. When the rate guarantee is violated due to edge resource availability change, the Real-time Serverless will notify the `application coordinator` which precisely reconfigures the workflow to restore the guarantee.

Performance guarantee and isolation are a powerful combination that can dramatically simplify application management for other goals, such as resource cost. Consider a simple declarative goal – to minimize cost of resources used by a stream processing workflows at any point in time. We will call this policy MinCost. With our Storm-RTS SPE, shown in Figure 18, the policy can be reduced placing operators in the data center with the lowest cost. If the data center is full while there are still operators to schedule, then those operators will be placed in the data center with the next lowest cost factor and so on.

Consider a resource environment with four datacenters (cloud, edge1, edge2, edge3), where cost(edge1) < cost(edge2) < cost (edge3) < cost(cloud) as shown in Figure 20. On this testbed, we conduct an experiment showing how Storm-RTS allows workflows to operate stably at optimal cost. We deploy 3 workflows, STAT, ETL, and PRED.

The first graph of Figure 26a shows events that happen during the experiment and decisions made by Storm-RTS in response to minimize the cost. At $t = 0$, the three workflows are deployed in the cloud because no edge data centers are available. At $t = 150$, three edge data centers become available. The MinCost policy dictates a move to the cheapest data center, edge1, so the `operator distributor` shifts the operators for all three workflows to edge1. Then, at $t = 300$, the SCAN workflow starts at data center edge1, consuming CPU resources. The edge1 becomes oversubscribed, so the local Real-time Serverless (RTS) notifies the `application coordinator` that edge1 do not have sufficient resources to host all four workflows. This causes the `application coordinator` to have the `operator distributor` move PRED, the smallest workflow, to maintain adequate performance. To minimize resource cost, it picks data center edge2. At $t = 450$, the SCAN load increases again, causing the edge1 RTS system to again notify the `application coordinator`, leading to a move of the ETL workflow to edge2. And when SCAN expands to edge2 at $t = 600$, its resource consumption there causes the RTS system on edge2 to notify the `application coordinator` that it cannot maintain its guarantees. So, in response, SPE moves PRED to edge3 ensuring resource sufficiency for all workflows.

Through these many workflow reconfigurations, Storm-RTS maintains their performance, ensuring all three workflows stably achieve the desired throughput as shown in the second graph of Figure 26a. And, as the application coordinator always moves workflows to the data centers with the lowest cost available, the total cost is minimized (the last graph in Figure 26a). To understand the importance of Storm-RTS in implementing such declarative policy, consider the same scenario with Storm-Serverless (Figure 26b). Since Storm-Serverless

allocates resources in best-effort manner, can neither detect a shortfall nor choose a suitable destination for a migration (has enough resources available). This results in poor workflow performance in these changing resource environments.

5.5 Unanswered Questions and Remaining Tasks

While RBAM's real-time guarantee is fully proven, its implementation efficiency and applicability still need more results to be fully addressed. In particular

- **RBAM Efficient Implementation**: we still have not known if *RBAM can be implemented within 15% overhead* compared to their regular FaaS. Answering the question requires a proposal of algorithms to deploy RBAM at a reduced cost, an RBAM implementation that can realize these algorithms at scale, and evidence to show they can work well with different system configurations and workloads.
- **RBAM Applicability**: We still have no evidence if RBAM can support hard real-time requirements and extreme workloads of hundreds of thousands of requests per second and more. To answer the question, we need to demonstrate RBAM versus demanding applications such as scientific sensor instrument data processing.

According to the plan, answering these remaining parts of the research questions is corresponding to completing the following tasks

- Efficient RBAM Implementation
  - *T2.2*. Propose several scalable allocation algorithms that can reduce the overhead of deploying RBAM functions at any rate-guarantee over the cloud.
  - *T2.3*. Evaluate proposed RBAM allocation algorithms to understand their efficiency.
  - *T2.4*. Integrate proposed RBAM algorithms in a practical FaaS system.
  - *T2.5*. Conduct empirical studies to demonstrate that RBAM implementation cost is asymptotically small for a large number of RBAMs.
- RBAM Applicability
  - *T3.5*. Implement an demanding Scientific Sensor Instrument Data Processing application using RBAM functions.
  - *T3.6*. Design and conduct experiments to show that the RBAM functions enable the application to meet its real-time requirements with efficiency.

We propose the schedule of completing these tasks later in Section 7.

6 Related Work

6.1 FaaS Efficiency Improvement

FaaS or Serverless Computing is flourishing with tons of implementations that have been developed and contributed by both industry and the open-source community [17, 18, 19, 92, 117, 15]. However, Serverless is still at its early stage. Recent studies [140, 147, 57] reveals critical performance limitations in virtually every serverless implementation preventing them from supporting many potential applications [113].

High invocation overhead is arguably the most noticeable Serverless performance limitation. The problem is gradually handled through many overhead minimization efforts across the whole invocation stack. These include heavily optimizing sandboxing mechanism [24, 20, 101, 130, 26] and enabling invocation resource sharing [116] to host more invocations. Some also take sandbox snapshots to minimize cold start overhead [64]. Apart from sandboxing optimization, there are efforts on workload prediction to help avoid cold start [123], optimizing invocation routing [87] to reduce network latency, etc. However, there is still much effort needed to reduce the overhead to the sub-millisecond scale. Batching is one common workaround used by Storm-RTS, and other solutions [27, 79, 128].

High performance variability is a critical issue for serverless performance. Unfortunately, there is a lack of improvement over this space. Open source serverless implementations still rely on reactive autoscaling for invocation resource allocation [117, 15, 92]. This mechanism is also employed inside commercial serverless, according to black-box inspections [147, 140], although they do provide other supports such as provisioned concurrency [17] to help stabilize performance when load surge or workflow description [19, 17, 18] to help schedule function chains. All of the mentioned approaches, however, are best-effort and unable to protect serverless invocations from performance instability when the load or Cloud's internal resource structure suddenly changes. Consequently, much of QoS improvement emerge from application side [57, 27, 79, 139]. Storm-RTS utilizing Real-time Serverless [114], the first effort on guarantee serverless performance through rate-guarantee interface to achieve both performance transparency and modularity.

6.2 Supporting Bursty, Real-time Applications

6.2.1 Handling Workload Burstiness

Scalable internet services deal with bursty loads managing yield and latency by dropping requests [40]. For example, the WeChat microservice system has an elaborate system for load shedding that orders drops to minimize wasted work. However, all of these approaches drop requests in order to maintain service quality for accepted requests. In contrast, because we assume the cloud has sufficient resources to service our bursty, real-time applications, we take the approach of guaranteeing allocation rate to maintain quality for all of the received requests.

Most of today solutions for bursty real-time workload workaround the cost problem through dynamic allocation. Instead of running the application over a fixed set of resources, application developers try to dynamically adjust resources allocated to the application to resource needs. The dynamic allocation is an iterative control loop consisting of two steps: burst detection and allocation adjustment. To detect bursts, some approaches monitor workload changes [96, 95], while other rely on real-time violations [81, 80]. For some applications, violation is prohibited so they take a more proactive approach: predict potential burst arrival and allocate resources in advance [33]. Determine quantity of each dynamic allocation is a tricky part as applications only want to just enough resources for cost optimization. Some approaches deploy complicated analysis (e.g., using historical data) to make just-enough allocations [80, 95]. There also approach combine different cloud resources, using high flexibility service, such as FaaS, to absorb the burst [96, 81]. However, all of dynamic allocation approaches are constructed upon heuristic methods limiting their applicability. Any application that have its burstiness properties or real-time constraint uncovered by the heuristic solution may end up with mis-allocation that leads to either real-time violation or resource waste.

6.2.2 Satisfying Real-time Requirements

Many research efforts have been spent on task runtime prediction for efficient scheduling and resource allocation. Some techniques are developed for repeating jobs [54, 61, 86] while others use the job structure and characterize input to construct predicting models [72, 119]. JamaisVu [134] and 3Sigma [118] extracts tasks' features from execution history, uses them for constructing runtime distributions and applies a tournament prediction to estimate task runtime. From runtime prediction, many scheduling such as backfilling [132, 148] or packing [138, 135] can be applied to minimize real-time constraint violation while still maximizing resource utilization.

There are also other approaches in the literature. For example, [48] uses the earliest-deadline-first scheduling policy and explicitly handling the transient of dynamic real-time workload to reduce deadline misses. [38] combines three techniques: reservation, semi-partition scheduling, and period transformation with task-placement heuristics to achieve near-optimal hard real-time scheduling. Satisfying real-time constraints under power limitation is also a very active research area [50, 151].

All of the work mentioned above, however, deal with real-time constraints in a best-effort manner. They do not explicitly guarantee the deadline misses but try to minimize the misses as much as possible. In contrast, through guaranteed allocation rate, real-time serverless enables applications to plan to achieve real-time guarantee and guarantee computation quality.

In summary, to the best of our knowledge, none of the recent studies tackles the problem of cost-effectively handling bursty demands in a timely fashion. They either consider real-time constraints or bursty demand but not both as real-time serverless did.

6.3 Stream Processing

Our Storm-RTS work focuses on providing stable SPE performance across heterogeneous resources with varying availability. The key to our approach is the higher-level resource abstraction, the rate-based abstract machine. This section will discuss how our approach is distinguished from other studies in terms of ensuring workflow performance stability, exploiting FaaS abstraction, and multi-site deployments.

6.3.1 Stable Performance

For worker-based SPEs, stable performance is strongly tied to resolving their limitations in performance transparency and isolation. In terms of performance transparency, worker-based SPEs try to decouple workflow performance from the underlying system implementation by carefully considering workflow topology and the underlying system details for every operator scheduling decision. Many SPEs dynamically map operators to workers via employing heuristic scheduling strategies based on performance profiling [98, 41, 52, 104, 47] and/or workflow characterization, including operator dependencies [52, 105], queue size [74] and query context [145, 137]. In distribution settings, SPEs place workers among computing nodes in traffic-aware [108, 144, 69] or topology-aware [108, 142, 107] fashion ensuring tuple transmission is supported by the underlying network. On low-end systems, e.g., Edge, resource heterogeneity and scarcity is quite common, great efforts on workload partitioning [97, 111, 91, 141, 70] and task placement [45, 88, 91, 58, 110, 46, 55, 28] are needed.

To resolve performance isolation challenges, worker-based SPEs ensure workflow performance by leveraging control mechanisms, which are typically full loops of two steps: interference detection and interference resolution. In interference detection, the SPEs identify interference through monitoring stream traffic [94, 41] and workflow throughput [71]. Some approaches even use the monitor data for predicting potential resource contention [71, 82, 83], proactively prevent interference beforehand. Detected interferences are resolved with heuristic algorithms, which either dynamically readjust resource sharing among competitive workflows [106, 71, 82, 83] or migrate them to another set of computing nodes [41].

All of the proposed approaches, however, are heuristic. They configure SPEs to behave properly with popular workflow and resource configurations. Thus, any significant changes in workflow dynamics or underlying resource configurations will lead to misbehavior and SPEs fail to maintain stable performance. These issues are well addressed by Storm-RTS: by leveraging FaaS, Storm-RTS can scale well to workflow dynamics, and through rate-guarantee enforcement, Storm-RTS provide strong isolation from resource configurations.

6.3.2 Stream Processing and FaaS

Leveraging FaaS for dynamic scalability has been proposed in many SPEs [53, 122, 112, 13, 34, 21]. However, these SPEs only outsource the processing logics to FaaS. Other parts of

operators, such as transmission and synchronization, are implemented through the worker abstraction inheriting worker-based performance limitations. Storm-RTS uses FaaS as a higher-level abstraction wrapping whole operators inside FaaS deployments. This removes worker abstraction from SPE implementation, eliminating its performance limitation legacies.

Storm-RTS relies on RBAM for performance stability and isolation. The key idea of RBAM is to ensure serverless performance with rate-guarantee. This is different from regular serverless systems, which are best-effort [117, 15, 92, 19, 17, 18]. When these systems fail to acquire needed resources, the performance of SPE relies on them degrades. Recent years witnessed many attempts on minimizing the chance of these failures, including optimizing invocation resource consumption [66, 109], proactive pre-allocation, and reusing terminated invocations [124, 75, 77]. There are also active studies on intelligent resource sharing and function placement to improve resource efficiency and avoid interference [129, 65, 62, 102, 149]. All of the mentioned approaches, however, do not provide performance stability. When some factors, such as invocation request interarrival or resource availability, change, they may become ineffective leading to performance degradation. In contrast, by enforcing rate-guarantee with resource reservation and admission control, Storm-RTS ensure sufficient resources for serverless invocations to meet their rate guarantee, which not only achieves workflows' desired throughput but also provides strong protection from the surrounding environment.

6.3.3 Stream Processing across Multiple Sites

With distributed data sources and growing numbers of edge-based applications, stream processing across multiple sites is of growing interest. Several approaches have been proposed (e.g., [39, 133, 44, 25]). Most of them adopt the worker abstraction or use worker-based SPEs as a building block. Worker abstraction limitations combine with new challenges that arise from distribution posing many problems that require many efforts to address. These include reliability [150, 78, 85, 143, 152], communication latency and overhead [89, 146], and managing limited, heterogeneous resource pools [59, 90], balancing task placement and parallelism [60, 56, 125]. Storm-RTS simplifies SPE design and well addresses many problems above. For example, by leveraging RBAM, Storm-RTS can stabilize workflow performance across limited, variable, and heterogeneous resousrce pools. Storm-RTS's ability to implement declarative goals enables its users to optimize their deployment for latency (i.e., prioritize data centers with fast connections), reliability (i.e., automatic migration at power shortage), and more.

7 Remaining Work of Thesis

By the time this thesis proposal is written, we have completely prove the capabiilty of guaranteeing real-time deadline of applications. Further, we prove that RBAM realization is feasible by proposing an naive RBAM implement and proved its correctness. We also

Tasks	Expected Time
Scalable RBAM implementation	
Propose several allocation algorithms that can minimize the overhead of deploying RBAM functions at any rate-guarantee.	2 months
Evaluate proposed RBAM allocation algorithms to understand their efficiency.	1 months
Integrate proposed RBAM algorithms in a practical FaaS system.	1 month
Conduct empirical studies to demonstrate that RBAM implementation cost is asymptotically small for a large number of RBAMs.	1 month
RBAM applicability	
Implement an demanding Scientific Sensor Instrument Data Processing application using RBAM functions.	1 month
Design and conduct experiments to show that the RBAM functions enable the application to meet its real-time requirements with efficiency.	1 month
Complete the Dissertation	
Summarize the research and write a full PhD Dissertation.	2 months

Table 4: Future Tasks and Timeline

demonstrate the applicability of RBAM is supprting two poplar classes of bursty, real-time applications, namely Distributed Real-time Video Analytic and Real-time Stream Processing. To complete the thesis, we are expected to complete studies shown in Table 4, each are presented with expected time scheduled.

References

[1] Binsentry. `https://www.binsentry.com/` Visited January 27, 2023.

[2] Biofeeder Helps Shrimp Farmers to Automate Feeding Schedules. `https://www.digi.com/customer-stories/biofeeder-helps-shrimp-farmers-to-automate-feeding` Visited January 27, 2023.

[3] Citymesh. `https://citymesh.com/` Visited January 27, 2023.

[4] Digi Helps SEPTA Comply With Federal Mandate For "Positive Train Control" (PTC). `https://www.digi.com/customer-stories/digi-helps-septa-comply-with-federal-mandate` Visited January 27, 2023.

[5] Digi WDS Helps Evoqua Deliver an Internet-Connected Water-Monitoring Solution for Commercial Applications. https://www.digi.com/customer-stories/evoqua-creates-digital-water-management-solution Visited January 27, 2023.

[6] FirstNet. https://www.firstnet.com/ Visited January 27, 2023.

[7] How G2ControlsNW Built an Automated System for Turning on Frost Fans for Agricultural Applications . https://www.digi.com/resources/project-gallery/use-digi-connect-sensor-and-drm-for-agriculture Visited January 27, 2023.

[8] Infinitum Delivers Innovative HVAC Monitoring Solution Featuring Smaller Footprint, Predictive Maintenance. https://www.digi.com/customer-stories/infinitum-delivers-reduced-footprint-motor Visited January 27, 2023.

[9] Otis. https://www.otis.com/ Visited January 27, 2023.

[10] SmartSense. https://www.smartsense.co/ Visited January 27, 2023.

[11] Valmont Brings Green Strategies to Agriculture and Infrastructure Sectors. https://www.digi.com/customer-stories/valmont-green-tech-agriculture-infrastructure Visited January 27, 2023.

[12] Wake. https://www.wakeinc.com/ Visited January 27, 2023.

[13] Apache Flink. https://flink.apache.org, 2014.

[14] Chameleon Cloud. https://www.chameleoncloud.org/, Jul 2015.

[15] Openwhisk. https://openwhisk.apache.org/, 2016.

[16] Apache Storm. https://storm.apache.org, May 2017.

[17] AWS Lambda. https://aws.amazon.com/lambda/, 2017.

[18] Google Cloud Function. https://cloud.google.com/functions, 2017.

[19] Microsoft Azure Function. https://azure.microsoft.com/en-us/services/functions/, 2017.

[20] gvisor. https://gvisor.dev/, 2018.

[21] Amazon Kinesis Data Streams. https://aws.amazon.com/kinesis/data-streams/, 2019.

[22] netty. https://netty.io/, 2022.

[23] Right Place, Right Time (RiPiT) Carbon Emissions Service. https://http://ripit.uchicago.edu//, May 2022.

[24] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 419–434, Santa Clara, CA, Feb. 2020. USENIX Association.

[25] 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.

[26] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt. {SAND}: Towards high-performance serverless computing. In *2018 Usenix Annual Technical Conference USENIX ATC 18)*, pages 923–935, 2018.

[27] A. Ali, R. Pinciroli, F. Yan, and E. Smirni. Batch: machine learning inference serving on serverless platforms with adaptive batching. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2020.

[28] G. Amarasinghe, M. D. De Assuncao, A. Harwood, and S. Karunasekera. A data stream processing optimisation framework for edge computing applications. In *2018 IEEE 21st International Symposium on Real-Time Distributed Computing (ISORC)*, pages 91–98. IEEE, 2018.

[29] Amazon Web Services Elastic Compute Cloud. https://aws.amazon.com/ec2/, 2007.

[30] Amazon. Amazon Cloud Infrastructure. https://aws.amazon.com/about-aws/global-infrastructure/, Feb 2021.

[31] AWS Lambda Reserved Concurrency. https://docs.aws.amazon.com/lambda/latest/operatorguide/reserved-concurrency.html.

[32] Azure Function. https://azure.microsoft.com/en-us/services/functions/.

[33] A. F. Baarzi, T. Zhu, and B. Urgaonkar. Burscale: Using burstable instances for cost-effective autoscaling in the public cloud. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 126–138, 2019.

[34] P. A. Bernstein, T. Porter, R. Potharaju, A. Z. Tomsic, S. Venkataraman, and W. Wu. Serverless event-stream processing over virtual actors. In *CIDR*, 2019.

[35] B. Beyer, N. R. Murphy, D. K. Rensin, K. Kawahara, and S. Thorne. *The site reliability workbook: practical ways to implement SRE*. " O'Reilly Media, Inc.", 2018.

[36] V. M. Bhasi, J. R. Gunasekaran, P. Thinakaran, C. S. Mishra, M. T. Kandemir, and C. Das. Kraken: Adaptive container provisioning for deploying dynamic dags in serverless platforms. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 153–167, 2021.

[37] D. Bohn. Amazon Says 100 Million Alaxa Devices Have Been Sold — What's Next?, January 2019. `https://www.theverge.com/`.

[38] B. B. Brandenburg and M. Gül. Global Scheduling not Required: Simple, Near-optimal Multiprocessor Real-time Scheduling With Semi-partitioned Reservations. In *2016 IEEE Real-Time Systems Symposium (RTSS)*, pages 99–110. IEEE, 2016.

[39] M. Branson, F. Douglis, B. Fawcett, Z. Liu, A. Riabov, and F. Ye. Clasp: Collaborating, autonomous stream processing systems. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 348–367. Springer, 2007.

[40] E. A. Brewer. Towards Robust Distributed Systems. In *PODC*, volume 7, 2000.

[41] T. Buddhika, R. Stern, K. Lindburg, K. Ericson, and S. Pallickara. Online scheduling and interference alleviation for low-latency, high-throughput processing of data streams. *IEEE Transactions on Parallel and Distributed Systems*, 28(12):3553–3569, 2017.

[42] J. Cadden, T. Unger, Y. Awad, H. Dong, O. Krieger, and J. Appavoo. Seuss: skip redundant paths to make serverless fast. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–15, 2020.

[43] D. Canvas. Sense your city: Data art challenge. `http://datacanvas.org/sense-your-city/`, Jun 2022.

[44] V. Cardellini, V. Grassi, F. Lo Presti, and M. Nardelli. Distributed qos-aware scheduling in storm. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, pages 344–347, 2015.

[45] V. Cardellini, V. Grassi, F. Lo Presti, and M. Nardelli. Optimal operator placement for distributed stream processing applications. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, pages 69–80, 2016.

[46] V. Cardellini, V. Grassi, F. Lo Presti, and M. Nardelli. Optimal operator replication and placement for distributed stream processing systems. *ACM SIGMETRICS Performance Evaluation Review*, 44(4):11–22, 2017.

[47] V. Cardellini, V. Grassi, F. L. Presti, and M. Nardelli. On qos-aware scheduling of data stream applications over fog computing infrastructures. In *2015 IEEE Symposium on Computers and Communication (ISCC)*, pages 271–276. IEEE, 2015.

[48] D. Casini, A. Biondi, and G. C. Buttazzo. Handling Transients of Dynamic Real-Time Workload Under EDF Scheduling. *IEEE Transactions on Computers*, 2018.

[49] C. E. Catlett, P. H. Beckman, R. Sankaran, and K. K. Galvin. Array of Things: a Scientific Research Instrument in the Public Way: Platform Design and Early Lessons Learned. In *Proceedings of the 2nd International Workshop on Science of Smart City Operations and Platforms Engineering*, pages 26–33. ACM, 2017.

[50] H. Chen, X. Zhu, H. Guo, J. Zhu, X. Qin, and J. Wu. Towards Energy-efficient Scheduling for Real-time Tasks under Uncertain Cloud Computing Environment. *Journal of Systems and Software*, 99:20–35, 2015.

[51] T. Y.-H. Chen, L. Ravindranath, S. Deng, P. Bahl, and H. Balakrishnan. Glimpse: Continuous, real-time object recognition on mobile devices. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*, pages 155–168. ACM, 2015.

[52] D. Cheng, Y. Chen, X. Zhou, D. Gmach, and D. Milojicic. Adaptive scheduling of parallel jobs in spark streaming. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*, pages 1–9. IEEE, 2017.

[53] Y. Cheng and Z. Zhou. Autonomous resource scheduling for real-time and stream processing. In *2018 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*, pages 1181–1184. IEEE, 2018.

[54] C. Curino, D. E. Difallah, C. Douglas, S. Krishnan, R. Ramakrishnan, and S. Rao. Reservation-based Scheduling: If You're Late don't Blame us! In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–14. ACM, 2014.

[55] A. da Silva Veith, M. D. de Assuncao, and L. Lefevre. Latency-aware placement of data stream analytics on edge computing. In *International conference on service-oriented computing*, pages 215–229. Springer, 2018.

[56] A. Dasilvaveith, M. D. de Assuncao, and L. Lefevre. Latency-aware strategies for deploying data stream processing applications on large cloud-edge infrastructure. *IEEE Transactions on Cloud Computing*, 2021.

[57] N. Daw, U. Bellur, and P. Kulkarni. Xanadu: Mitigating cascading cold starts in serverless function chain deployments. In *Proceedings of the 21st International Middleware Conference*, pages 356–370, 2020.

[58] M. D. de Assuncao, A. da Silva Veith, and R. Buyya. Distributed data stream processing and edge computing: A survey on resource elasticity and future directions. *Journal of Network and Computer Applications*, 103:1–17, 2018.

[59] F. R. de Souza, M. D. de Assunçao, E. Caron, and A. da Silva Veith. An optimal model for optimizing the placement and parallelism of data stream processing applications on cloud-edge computing. In *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 59–66. IEEE, 2020.

[60] F. R. de Souza, A. D. S. Veith, M. D. de Assunçao, and E. Caron. Scalable joint optimization of placement and parallelism of data stream processing applications on cloud-edge infrastructure. In *International Conference on Service-Oriented Computing*, pages 149–164. Springer, 2020.

[61] P. Delgado, F. Dinu, A.-M. Kermarrec, and W. Zwaenepoel. Hawk: Hybrid Datacenter Scheduling. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 499–510, 2015.

[62] C. Denninnart and M. A. Salehi. Harnessing the potential of function-reuse in multimedia cloud systems. *IEEE Transactions on Parallel and Distributed Systems*, 33(3):617–629, 2021.

[63] Digi. Digi Remote Manager. `https://www.digi.com/products/iot-software-services/digi-remote-manager` Visited January 27, 2023.

[64] D. Du, T. Yu, Y. Xia, B. Zang, G. Yan, C. Qin, Q. Wu, and H. Chen. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 467–481, 2020.

[65] V. Dukic, R. Bruno, A. Singla, and G. Alonso. Photons: Lambdas on a diet. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 45–59, 2020.

[66] S. Eismann, L. Bui, J. Grohmann, C. Abad, N. Herbst, and S. Kounev. Sizeless: Predicting the optimal size of serverless functions. In *Proceedings of the 22nd International Middleware Conference*, pages 248–259, 2021.

[67] G. Electric. Everything You Need to Know about the Industrial Internet of Things. `https://www.ge.com/digital/blog/everything-you-need-know-about-industrial-internet-things`. Downloaded January 2019.

[68] Ericsson. Internet of Things Forecast - Ericsson Mobility Report. `https://www.ericsson.com/en/reports-and-papers/mobility-report/reports` Visited January 26, 2023.

[69] L. Eskandari, J. Mair, Z. Huang, and D. Eyers. T3-scheduler: A topology and traffic aware two-level scheduler for stream processing systems in a heterogeneous cluster. *Future Generation Computer Systems*, 89:617–632, 2018.

[70] J. Fang, R. Zhang, T. Z. Fu, Z. Zhang, A. Zhou, and J. Zhu. Parallel stream processing against workload skewness and variance. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, pages 15–26, 2017.

[71] M. R. H. Farahabady, A. Y. Zomaya, and Z. Tari. Qos-and contention-aware resource provisioning in a stream processing engine. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 137–146. IEEE, 2017.

[72] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: Guaranteed Job Latency in Data Parallel Clusters. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 99–112. ACM, 2012.

[73] A. Floratou, A. Agrawal, B. Graham, S. Rao, and K. Ramasamy. Dhalion: Self-regulating stream processing in heron. *Proc. VLDB Endow.*, 10(12):1825–1836, aug 2017.

[74] X. Fu, T. Ghaffar, J. C. Davis, and D. Lee. Edgewise: A better stream processing engine for the edge. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 929–946, Renton, WA, July 2019. USENIX Association.

[75] A. Fuerst and P. Sharma. Faascache: keeping serverless computing alive with greedy-dual caching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 386–400, 2021.

[76] A. Geiger, P. Lenz, C. Stiller, and R. Urtasun. Vision meets Robotics: The KITTI Dataset. *International Journal of Robotics Research (IJRR)*, 2013.

[77] A. U. Gias and G. Casale. Cocoa: Cold start aware capacity planning for function-as-a-service platforms. In *2020 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 1–8. IEEE, 2020.

[78] M. Gorawski and P. Marks. Towards reliability and fault-tolerance of distributed stream processing system. In *2nd International Conference on Dependability of Computer Systems (DepCoS-RELCOMEX'07)*, pages 246–253. IEEE, 2007.

[79] J. R. Gunasekaran, P. Thinakaran, N. Chidambaram, M. T. Kandemir, and C. R. Das. Fifer: Tackling underutilization in the serverless era. *arXiv preprint arXiv:2008.12819*, 2020.

[80] J. R. Gunasekaran, P. Thinakaran, M. T. Kandemir, B. Urgaonkar, G. Kesidis, and C. Das. Spock: Exploiting serverless functions for slo and cost aware resource procurement in public cloud. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 199–208. IEEE, 2019.

[81] A. Harlap, A. Chung, A. Tumanov, G. R. Ganger, and P. B. Gibbons. Tributary: Spot-dancing for Elastic Services with Latency SLOs. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 1–14, 2018.

[82] M. R. HoseinyFarahabady, A. Jannesari, J. Taheri, W. Bao, A. Y. Zomaya, and Z. Tari. Q-flink: A qos-aware controller for apache flink. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pages 629–638. IEEE, 2020.

[83] M. R. HoseinyFarahabady, J. Taheri, A. Y. Zomaya, and Z. Tari. Qspark: Distributed execution of batch & streaming analytics in spark platform. In *2021 IEEE 20th International Symposium on Network Computing and Applications (NCA)*, pages 1–8. IEEE, 2021.

[84] J. Huang, V. Rathod, C. Sun, M. Zhu, A. Korattikara, A. Fathi, I. Fischer, Z. Wojna, Y. Song, S. Guadarrama, et al. Speed/Accuracy Trade-offs for Modern Convolutional Object Detectors. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7310–7311, 2017.

[85] J.-H. Hwang, U. Cetintemel, and S. Zdonik. Fast and reliable stream processing over wide area networks. In *2007 IEEE 23rd International Conference on Data Engineering Workshop*, pages 604–613. IEEE, 2007.

[86] V. Jalaparti, P. Bodik, I. Menache, S. Rao, K. Makarychev, and M. Caesar. Network-aware Scheduling for Data-parallel Jobs: Plan When You Can. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 407–420. ACM, 2015.

[87] Z. Jia and E. Witchel. Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 152–166, 2021.

[88] J. Jiang, Z. Zhang, B. Cui, Y. Tong, and N. Xu. Stromax: Partitioning-based scheduler for real-time stream processing system. In *International Conference on Database Systems for Advanced Applications*, pages 269–288. Springer, 2017.

[89] A. Jonathan, A. Chandra, and J. Weissman. Wasp: wide-area adaptive stream processing. In *Proceedings of the 21st International Middleware Conference*, pages 221–235, 2020.

[90] E. Kalyvianaki, M. Fiscato, T. Salonidis, and P. Pietzuch. Themis: Fairness in federated stream processing under overload. In *Proceedings of the 2016 International Conference on Management of Data*, pages 541–553, 2016.

[91] N. R. Katsipoulakis, A. Labrinidis, and P. K. Chrysanthis. A holistic view of stream partitioning costs. *Proceedings of the VLDB Endowment*, 10(11):1286–1297, 2017.

[92] Knative. `https://cloud.google.com/knative/`.

[93] C. Krintz and R. Wolski. Smart Farm Overview. `https://www.cs.ucsb.edu/~ckrintz/SmartFarm17.pdf`, 2018. NSF Funded Smart Farm Project using Smart Sensors.

[94] J. Li, C. Pu, Y. Chen, D. Gmach, and D. Milojicic. Enabling elastic stream processing in shared clusters. In *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*, pages 108–115. IEEE, 2016.

[95] W. Lin, J. Z. Wang, C. Liang, and D. Qi. A threshold-based dynamic resource allocation scheme for cloud computing. *Procedia Engineering*, 23:695–703, 2011.

[96] F. Liu, K. Keahey, P. Riteau, and J. Weissman. Dynamically Negotiating Capacity between On-demand and Batch Clusters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, page 38. IEEE Press, 2018.

[97] P. Liu, D. Da Silva, and L. Hu. Dart: A scalable and adaptive edge stream processing engine. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 239–252, 2021.

[98] X. Liu and R. Buyya. D-storm: Dynamic resource-efficient scheduling of stream processing applications. In *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*, pages 485–492. IEEE, 2017.

[99] Z. Liu, A. Ali, P. Kenesei, A. Miceli, H. Sharma, N. Schwarz, D. Trujillo, H. Yoo, R. Coffee, N. Layad, et al. Bridging data center ai systems with edge computing for actionable information retrieval. In *2021 3rd Annual Workshop on Extreme-scale Experiment-in-the-Loop Computing (XLOOP)*, pages 15–23. IEEE, 2021.

[100] Z. Liu, H. Sharma, J.-S. Park, P. Kenesei, A. Miceli, J. Almer, R. Kettimuthu, and I. Foster. Braggnn: fast x-ray bragg peak analysis using deep learning. *IUCrJ*, 9(1):104–113, 2022.

[101] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. Unikernels: Library operating systems for the cloud. *ACM SIGARCH Computer Architecture News*, 41(1):461–472, 2013.

[102] A. Mampage, S. Karunasekera, and R. Buyya. Deadline-aware dynamic resource management in serverless computing environments. In *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 483–492. IEEE, 2021.

[103] B. Marr. IoT And Big Data At Caterpillar: How Predictive Maintenance Saves Millions Of Dollars. *Forbes*, February 2017. `https://www.forbes.com/`.

[104] Y. Mei, L. Cheng, V. Talwar, M. Y. Levin, G. Jacques-Silva, N. Simha, A. Banerjee, B. Smith, T. Williamson, S. Yilmaz, W. Chen, and C. Jerry. Turbine: Facebook's service management platform for stream processing. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1591–1602. IEEE, 2020.

[105] H. Miao, H. Park, M. Jeon, G. Pekhimenko, K. S. McKinley, and F. X. Lin. Streambox: Modern stream processing on a multicore machine. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 617–629, 2017.

[106] Y. Morisawa, M. Suzuki, and T. Kitahara. Resource efficient stream processing platform with {Latency-Aware} scheduling algorithms. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*, 2020.

[107] H. Moussa, I.-L. Yen, and F. Bastani. Service management in the edge cloud for stream processing of iot data. In *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*, pages 91–98. IEEE, 2020.

[108] A. Muhammad, M. Aleem, and M. A. Islam. Top-storm: A topology-based resource-aware scheduler for stream processing engine. *Cluster Computing*, 24(1):417–431, 2021.

[109] D. Mvondo, M. Bacou, K. Nguetchouang, L. Ngale, S. Pouget, J. Kouam, R. Lachaize, J. Hwang, T. Wood, D. Hagimont, et al. Ofc: an opportunistic caching system for faas platforms. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 228–244, 2021.

[110] M. Nardelli, V. Cardellini, V. Grassi, and F. L. Presti. Efficient operator placement for distributed data stream processing applications. *IEEE Transactions on Parallel and Distributed Systems*, 30(8):1753–1767, 2019.

[111] M. A. U. Nasir, G. D. F. Morales, N. Kourtellis, and M. Serafini. When two choices are not enough: Balancing at scale in distributed stream processing. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, pages 589–600. IEEE, 2016.

[112] S. Nastic, T. Rausch, O. Scekic, S. Dustdar, M. Gusev, B. Koteska, M. Kostoska, B. Jakimovski, S. Ristov, and R. Prodan. A serverless real-time data analytics platform for edge computing. *IEEE Internet Computing*, 21(4):64–71, 2017.

[113] H. D. Nguyen, Z. Yang, and A. A. Chien. Motivating high performance serverless workloads. In *Proceedings of the 1st Workshop on High Performance Serverless Computing*, pages 25–32, 2020.

[114] H. D. Nguyen, C. Zhang, Z. Xiao, and A. A. Chien. Real-time serverless: Enabling application performance guarantees. In *Proceedings of the 5th International Workshop on Serverless Computing*, WOSC '19, page 1–6, New York, NY, USA, 2019. Association for Computing Machinery.

[115] Niantic. Pokemon GO. `https://pokemongolive.com` Visited January 27, 2023.

[116] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. {SOCK}: Rapid task provisioning with serverless-optimized containers. In *2018 USENIX Annual Technical Conference USENIX ATC 18)*, pages 57–70, 2018.

[117] OpenFaaS. `https://docs.openfaas.com`.

[118] J. W. Park, A. Tumanov, A. Jiang, M. A. Kozuch, and G. R. Ganger. 3Sigma: Distribution-based Cluster Scheduling for Runtime Uncertainty. In *Proceedings of the Thirteenth EuroSys Conference*, page 2. ACM, 2018.

[119] PerfOrator. `https://www.microsoft.com/en-us/research/project/perforator-2/`.

[120] R. B. Roy, T. Patel, and D. Tiwari. Icebreaker: Warming serverless functions better with heterogeneity. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 753–767, 2022.

[121] W. Saad, M. Bennis, and M. Chen. A vision of 6g wireless systems: Applications, trends, technologies, and open research problems. *IEEE Network*, 34(3):134–142, 2020.

[122] A. W. Service. Serverless Streaming Architectures and Best Practices. `https://d1.awsstatic.com/whitepapers/Serverless_Streaming_Architecture_Best_Practices.pdf`, June 2018.

[123] M. Shahrad, R. Fonseca, Í. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 205–218, 2020.

[124] M. Shahrad, R. Fonseca, I. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini. Serverless in the wild: Characterizing

and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 205–218. USENIX Association, July 2020.

[125] S. K. Sharma and X. Wang. Live data analytics with collaborative edge and cloud processing in wireless iot networks. *IEEE Access*, 5:4621–4635, 2017.

[126] A. Shukla, S. Chaturvedi, and Y. Simmhan. Riotbench: An real-time iot benchmark for distributed stream processing systems. *Concurrency and Computation: Practice and Experience*, 29(21):e4257, 2017.

[127] A. Singhvi, A. Balasubramanian, K. Houck, M. D. Shaikh, S. Venkataraman, and A. Akella. Atoll: A scalable low-latency serverless platform. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 138–152, New York, NY, USA, 2021. Association for Computing Machinery.

[128] T. J. Skluzacek, R. Chard, R. Wong, Z. Li, Y. N. Babuji, L. Ward, B. Blaiszik, K. Chard, and I. Foster. Serverless workflows for indexing large scientific data. In *Proceedings of the 5th International Workshop on Serverless Computing*, pages 43–48, 2019.

[129] A. Suresh, G. Somashekar, A. Varadarajan, V. R. Kakarla, H. Upadhyay, and A. Gandhi. Ensure: Efficient scheduling and autonomous resource management in serverless environments. In *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, pages 1–10. IEEE, 2020.

[130] B. Tan, H. Liu, J. Rao, X. Liao, H. Jin, and Y. Zhang. Towards lightweight serverless computing via unikernel as a function. In *2020 IEEE/ACM 28th International Symposium on Quality of Service (IWQoS)*, pages 1–10. IEEE, 2020.

[131] A. Tariq, A. Pahl, S. Nimmagadda, E. Rozner, and S. Lanka. Sequoia: Enabling quality-of-service in serverless computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 311–327, 2020.

[132] D. Tsafrir, Y. Etsion, and D. G. Feitelson. Backfilling using System-generated Predictions rather than User Runtime Estimates. *IEEE Transactions on Parallel and Distributed Systems*, 18(6):789–803, 2007.

[133] R. Tudoran, A. Costan, O. Nano, I. Santos, H. Soncu, and G. Antoniu. Jetstream: Enabling high throughput live event streaming on multi-site clouds. *Future Generation Computer Systems*, 54:274–291, 2016.

[134] A. Tumanov, A. Jiang, J. W. Park, M. A. Kozuch, and G. R. Ganger. JamaisVu: Robust Scheduling with Auto- Estimated Job Runtimes., 2016.

[135] A. Tumanov, T. Zhu, J. W. Park, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger. TetriSched: Global Rescheduling with Adaptive Plan-ahead in Dynamic Heterogeneous Clusters. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 35. ACM, 2016.

[136] Twin Forks Pest Control. Southampton Traffic Cam. `hhttps://www.youtube.com/watch?v=rpbkCUbWVio`, 2019.

[137] S. Venkataraman, A. Panda, K. Ousterhout, M. Armbrust, A. Ghodsi, M. J. Franklin, B. Recht, and I. Stoica. Drizzle: Fast and adaptable stream processing at scale. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 374–389, 2017.

[138] A. Verma, M. Korupolu, and J. Wilkes. Evaluating Job Packing in Warehouse-scale Computing. In *2014 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 48–56. IEEE, 2014.

[139] B. Wang, A. Ali-Eldin, and P. Shenoy. Lass: Running latency sensitive serverless computations at the edge. In *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*, pages 239–251, 2020.

[140] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift. Peeking behind the Curtains of Serverless Platforms. In *2018 {USENIX} Annual Technical Conference (USENIX ATC 18)*, pages 133–146, 2018.

[141] X. Wang, Z. Zhou, P. Han, T. Meng, G. Sun, and J. Zhai. Edge-stream: a stream processing approach for distributed applications on a hierarchical edge-computing system. In *2020 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 14–27. IEEE, 2020.

[142] X. Wei, X. Wei, and H. Li. Topology-aware task allocation for online distributed stream processing applications with latency constraints. *Physica A: Statistical Mechanics and its Applications*, 534:122024, 2019.

[143] X. Wei, Y. Zhuang, H. Li, and Z. Liu. Reliable stream data processing for elastic distributed stream processing systems. *Cluster Computing*, 23(2):555–574, 2020.

[144] J. Xu, Z. Chen, J. Tang, and S. Su. T-storm: Traffic-aware online scheduling in storm. In *2014 IEEE 34th International Conference on Distributed Computing Systems*, pages 535–544. IEEE, 2014.

[145] L. Xu, S. Venkataraman, I. Gupta, L. Mai, and R. Potharaju. Move fast and meet deadlines: Fine-grained real-time stream processing with cameo. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 389–405, 2021.

[146] F. Yin, X. Li, X. Li, and Y. Li. Task scheduling for streaming applications in a cloud-edge system. In G. Wang, J. Feng, M. Z. A. Bhuiyan, and R. Lu, editors, *Security, Privacy, and Anonymity in Computation, Communication, and Storage*, pages 105–114, Cham, 2019. Springer International Publishing.

[147] T. Yu, Q. Liu, D. Du, Y. Xia, B. Zang, Z. Lu, P. Yang, C. Qin, and H. Chen. Characterizing serverless platforms with serverlessbench. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 30–44, 2020.

[148] Y. Zhang, H. Franke, J. Moreira, and A. Sivasubramaniam. An Integrated Approach to Parallel Scheduling using Gang-scheduling, Backfilling, and Migration. *IEEE Transactions on Parallel and Distributed Systems*, 14(3):236–247, 2003.

[149] Y. Zhang, Í. Goiri, G. I. Chaudhry, R. Fonseca, S. Elnikety, C. Delimitrou, and R. Bianchini. Faster and cheaper serverless computing on harvested resources. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 724–739, 2021.

[150] Z. Zhang, Y. Gu, F. Ye, H. Yang, M. Kim, H. Lei, and Z. Liu. A hybrid approach to high availability in stream processing systems. In *2010 IEEE 30th International Conference on Distributed Computing Systems*, pages 138–148. IEEE, 2010.

[151] Z. Zhou, J. Abawajy, M. Chowdhury, Z. Hu, K. Li, H. Cheng, A. A. Alelaiwi, and F. Li. Minimizing SLA Violation and Power Consumption in Cloud Data Centers using Adaptive Energy-aware Algorithms. *Future Generation Computer Systems*, 86:836–850, 2018.

[152] Y. Zhuang, X. Wei, H. Li, M. Hou, and Y. Wang. Reducing fault-tolerant overhead for distributed stream processing with approximate backup. In *2020 29th International Conference on Computer Communications and Networks (ICCCN)*, pages 1–9. IEEE, 2020.