THE UNIVERSITY OF CHICAGO

RATE-BASED ABSTRACT MACHINE: AN EFFICIENT COMPUTATION MODEL FOR GUARANTEEING PERFORMANCE OF BURSTY, REAL-TIME APPLICATIONS

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY
HAI DUC NGUYEN

CHICAGO, ILLINOIS
AUGUST 2024

Dedicated to my family

Do not believe in something because it is reported. Do not believe in something because it has been practiced by generations or becomes a tradition or part of a culture. Do not believe in something because a scripture says it is so. Do not believe in something believing a god has inspired it. Do not believe in something a teacher tells you to. Do not believe in something because the authorities say it is so. Do not believe in hearsay, rumor, speculative opinion, public opinion, or mere acceptance to logic and inference alone. Help yourself, accept as completely true only that which is praised by the wise and which you test for yourself and know to be good for yourself and others.

— The Buddha, The Kalama Sutta, Anguttara Nikaya 3.65

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

This dissertation stands as evidence of the collective effort of everyone who has supported me on this journey. Without their help, this achievement would not have been possible.

First and foremost, I would like to express my deepest appreciation to my advisor, Prof. Andrew A. Chien, for his unwavering patience, motivation, and exceptional guidance throughout my academic and research endeavors. His wisdom and knowledge have significantly influenced my critical thinking, broadened my perspectives, and refined my abilities to conduct rigorous research, for which I will be forever grateful. Prof. Chien's consistent support and direction have been invaluable, and I am immensely grateful for his unwavering belief in my potential.

I extend my heartfelt gratitude to the members of my dissertation committee, Prof. Junchen Jiang, Prof. Kyle Chard, and Prof. Sanjay Krishnan, for generously dedicating their time, effort, and providing constructive feedback. Your diverse perspectives and thorough evaluations have significantly enhanced the quality of my work.

I want to extend a special thanks to my ZCCloud teammates: Chaojie (Sam) Zhang, Liuzixuan (Peter) Lin, Rajini Wijayawardana, Varsha Rao, Tristan Sharma, and Tian Yang. Additionally, I express my gratitude to the rest of the team at LSSG: Yuanwei (Kevin) Fang, Chen Zou, Andronicus Samsundar Rajasukumar, Tianshuo Su, Marziyeh Nourian, Yuqing (Ivy) Wang, Jiya Su, Tianchi (Tony) Zhang, Alexander Fell, Ruiqi (Jerry) Xu, and Rajat Khandelwal. Your support, collaboration, and stimulating discussions have significantly contributed to my work and have been a constant source of motivation and inspiration. Thank you all for being an essential part of my academic journey and for contributing to my growth as a researcher.

I am deeply grateful to my family for their unconditional love and support. Your endless encouragement has been invaluable, especially during times of challenge and difficulty. Thank you for always being there when I needed you most.

Finally, to my friends, thank you for your encouragement and for always being there to provide a much-needed balance between work and life. I am grateful for the joy and perspective you bring to my life. While I cannot name everyone individually, please know that I am sincerely thankful to each and every one of you for your support and friendship.

ABSTRACT

Cloud Function-as-a-Service (FaaS) systems offer statistical guarantees and cannot meet the deadlines of bursty, real-time applications. To address this limitation, we propose the Rate-based Abstract Machine (RBAM), a performance abstraction for FaaS functions. RBAM provides a guaranteed invocation rate for each FaaS function, enabling applications to meet real-time requirements. In this dissertation, we address the generality, efficient realizability, and applicability of the RBAM model.

First, we show that RBAM is complete as a primitive, able to realize the deadlines of any rate-monotonic real-time workload at a bounded cost. We do so by constructing an analytical framework based on rate-monotonic workloads and then proving that RBAM effectively bounds FaaS invocation latency. This ensures the applications meet their real-time deadlines. Additionally, we derive bounds on the required invocation rate and corresponding resource overhead, showing how to realize any set of guarantees.

Second, we demonstrate that the RBAM performance abstraction can be efficiently realized in today's cloud environments. To do so, we propose new rate-based invocation scheduling and resource management algorithms and show how they employ resource sharing and exploit the underlying resource allocation statistics to enforce the rate guarantees. We document their ability to achieve performance guarantees at a low resource cost and scale well to large systems. Finally, we show that they are robust across varied application dynamics and deployment environments.

Third, we demonstrate RBAM's applicability by using it to implement a stream processing engine called Storm-RTS and also a distributed real-time video analytics application. These applications illustrate RBAM's expressiveness in describing application performance guarantees, how implementation can deliver robust performance against various workload burstiness, and how its performance guarantees can enable application deployment flexibil-

ity. The latter is particularly useful as declarative policies can ensure performance stability and optimized deployment for various objectives (e.g., cost or carbon).

In summary, RBAM is a performance abstraction that extends the FaaS model to efficiently and scalably support bursty, real-time applications across various scenarios. In addition, RBAM simplifies the implementation and deployment of applications with real-time guarantees. The rate guarantee can be used as a quality of service parameter to help applications configure for specific performance needs.

CHAPTER 1

INTRODUCTION

1.1 Bursty, Real-time Applications

Bursty real-time applications have become increasingly prevalent due to the growing demand for real-time analytics, IoT data processing, and multimedia streaming services. In this section, we will formally define bursty real-time applications, using motivating examples from various domains, and discuss specific requirements for their software solutions.

1.1.1 The Rise of Bursty, Real-time Applications

We use "*bursty, real-time applications*" to refer to applications with occasional intensive demand and timely requirements. Formally, they are defined as follows.

Definition 1 (Bursty, Real-time Applications)**.** *A bursty, real-time application possesses the following characteristics:*

- ***Bursty****: Computation demand changes with occasional bursts that have*
  - *Low duty factor: bursts are typically short and rarely happen. Their duty factor (i.e., the ratio of burst periods over time) is typically less than 10%.*
  - *High variance: Bursts create significantly higher computation demand than usual, which could be 10$\times$ to 100$\times$ or even more.*
- ***Real-time****: The computation demand of bursts must be served within a strict deadline.*

Bursty real-time workloads are emerging and expected to grow across various application domains, exhibiting high diversity in burstiness, computation intensity, real-time requirements, and scale.

Internet of Things (IoT). IoT devices (e.g., sensing devices) play a crucial role in providing real-time information to offer high adaptability [10, 14], mobility [4, 7], compliance [13, 5], and cost-effective solutions [6, 3] with improved insights. Due to the intermittent nature of their working environment, these devices generate and transmit information with burstiness, requiring timely processing to trigger appropriate reactions. Consequently, IoT data processing applications are often bursty and real-time. The advancement of technology has fueled the emergence of massive-scale deployments of Internet-of-Things (IoT) applications, such as Amazon's intelligent assistants [75], large-scale monitoring systems [91, 184], and expensive large equipment [120, 212]. By 2025, it is estimated that there will be over 55.7 billion IoT devices in use. This number is expected to increase further in the coming years [156]. The trend suggests a rise in bursty, real-time IoT applications in quantity, computation intensity, and scale.

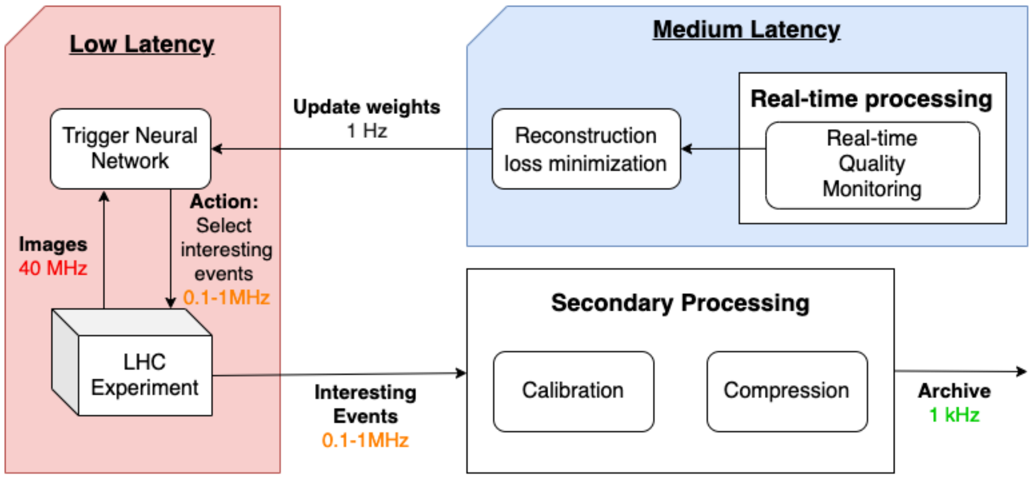


Figure 1.1: HEP Event Filtering and Analysis System Workflow

Scientific Data Streaming. High data rate instruments, such as DNA sequencing and advanced photon sources, produce high-bandwidth scientific data streams [248, 249]. These streams rely increasingly on real-time processing capabilities and machine learning workflows for filtering, analysis, and adaptive experiment control. In addition to high data rates, they require low-latency responses for real-time control. The scientific data streams are generated by sampling internal physics processes, which are highly bursty depending on the experiment process configuration. For example, in high energy physics (HEP), Figure 1.1, finding evidence of new physical phenomena, such as new partial or dark matter, requires Large Hadron Collider (LHC) systems to capture 40 million or more video frames for every detected collision from occasional experiments, each within a 300ns time constraint [234]. This bursty, real-time workload demands extreme computational intensity and tight real-time deadlines, often requiring specialized systems for handling.

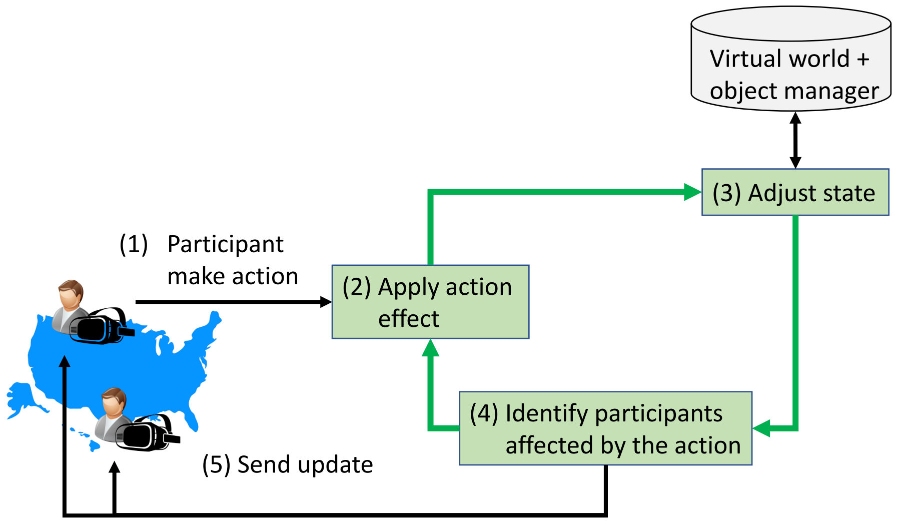

Virtual world + object manager
(3) Adjust state
(1) Participant make action
(2) Apply action effect
(4) Identify participants affected by the action
(5) Send update

Figure 1.2: Simplified architecture of a distributed VR/AR application.

Distributed Virtual/Augment Reality (VR/AR). The application allows many geographically distributed participants to join and interact in a virtual world (Figure 1.2). The application computation is driven by real-time interactions made by participants, which are highly bursty and subjected to tight deadlines (tens of milliseconds). Pokemon Go [236] is an excellent example of a massive distributed AR application. By 2020, Pokemon GO had approximately 600 million active players worldwide [292], which can generate around 4.2 billion interactions a day [234], an average rate of 50,000 requests per second! This demand is not uniform but temporally and spatially skewed, with more than 50% of participants from Pacific Asia [292]. In 2016, Pokemon GO received a never-seen-before demand increase of 50x their expected load, causing severe experience disruptions for days before it successfully upgraded in Google Cloud and became one of their biggest services since then [73].

Apart from the mentioned domains, bursty, real-time applications are also arising from online gaming [87, 215], video analysis [71, 329, 260, 97, 259, 190], and more. Thus, bursty, real-time applications play an important role in the application landscape. Proposing solutions to support them effectively is critical.

1.1.2 Challenges in Supporting Bursty, Real-time Applications

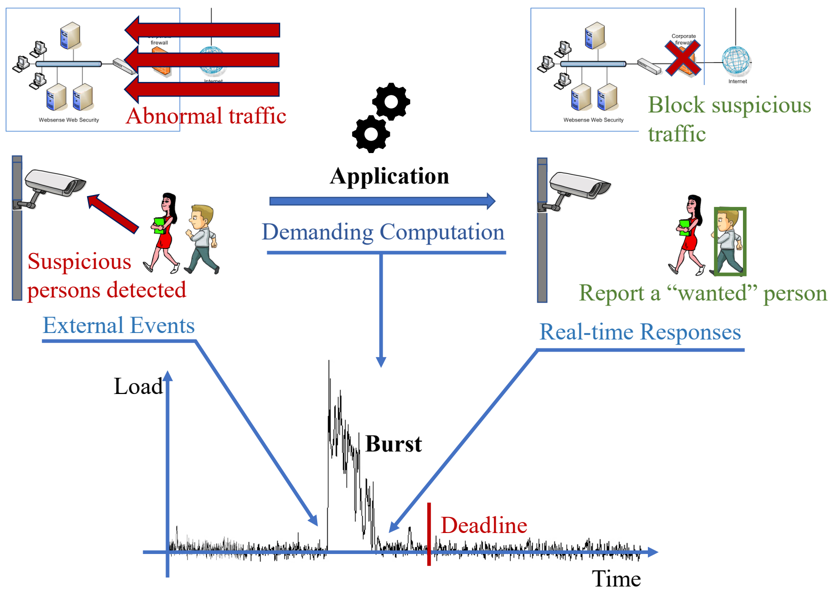


Figure 1.3: Bursty, real-time application examples: External events, such as cyberattacks and pedestrian appearance, trigger applications to start demanding computation with a strict deadline. The application consumes significantly more resources than usual (i.e., burst) to respond (e.g., block suspicious traffic and detect identities) in time.

The illustrative workloads described above reveal many difficulties in effectively supporting bursty, real-time applications. Computation burst arrivals are driven by real-world events that are hard to predict yet demand computation-intensive and immediate reactions. For instance, in Figure 1.3, a network administration application detects abnormal traffic, triggering an in-depth analysis to determine if it is a cyber-attack. If so, the application must swiftly take cybersecurity measures like blocking suspicious traffic and closing vulnerable ports to protect internal systems. Another example is a crowd control application that collects video streams to identify suspicious persons. The application conducts an in-depth analysis whenever a person appears in the video. The application must notify authorities if the analysis identifies a "wanted" person. In both cases, timely responses are crucial to prevent adverse impacts (e.g., allowing a fugitive to escape or subjecting internal systems to a Denial-of-Service attack). However, ensuring timely action necessitates significant resources, leading to sudden surges in workload demand.

The computation quality of bursty, real-time applications depends on the processing accuracy and speed. Failure to make a proper decision or deliver it before the deadline can result in poor quality outcomes or system failures. While the application can control the accuracy, it has to rely on resource availability (i.e., CPU and memory) for processing speed. When a burst arrives, computation demand increases dramatically, requiring an equivalent growth in resource availability to maintain the computation pace. If the application fails to grow resource availability in time, it has to slow down or delay some activities, prolonging computation time and may miss the deadline. Rapidly allocating a high quantity of resources within a short duration for real-time deadlines is challenging. Worse, extreme workloads such as those seen in LHC or Pokemon Go are beyond the capability of current general-purpose systems (e.g., public cloud). To make such applications possible, developers must build specialized systems [203, 204] or make an exclusive contract with infrastructure providers [73].

1.1.3 Solution Requirements

Given the growing emergence of bursty, real-time applications and their increasing importance and computation intensity, designing an appropriate solution to support their burstiness and real-time requirements efficiently is critical. In this dissertation, we focus on finding a *general solution* that has to meet the following requirements.

- *Real-time Guarantee*: the solution ensures bursty, real-time applications meet their real-time deadlines.
- *Efficiency*: the solution is implemented with low overhead (i.e., the amount of resources required in addition to the actual use is small).
- *Applicability*: the solution is usable by a broad 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. They ensure that the implementation cost is not too high and that the solution is versatile enough to be used in any application.

Many bursty, real-time applications rely on cloud resources for computation [326, 256, 234]. However, cloud providers often have to sacrifice resource control for usability and vice versa with limited performance guarantee support (See Section 2.1). As a result, meeting all solution requirements is challenging.

- For real-time guarantee, applications need complete control over their resources to minimize the impact of uncertain factors, such as VM preemption or resource contention with other colocated applications, on their performance. However, complete control complicates application deployment and operation, potentially reducing its applicability.

- For applicability, on the other hand, the application needs a service with high usability, such as Container-as-a-Service, but their resource control is limited, making real-time guarantee challenging.

- For efficiency, the applications need the capability to adjust resource allocation according to workload dynamics. However, there is no sweet spot for this along the usability versus resource control spectrum. Applications either have to spend much effort on resource adjustment (e.g., high controllable services), resulting in low applicability, or have their performance suffer from uncontrollable factors (e.g., high usability services), resulting in low real-time guarantee.

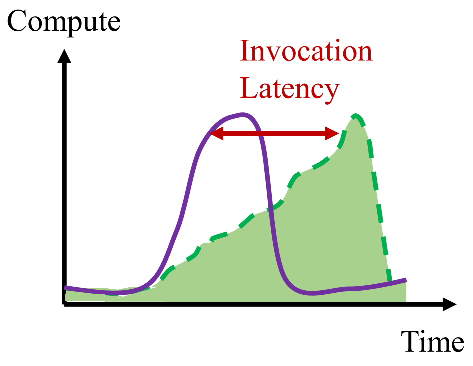
1.2 Approach

We address this challenge by introducing performance abstraction, which shifts resource control from the application space to the cloud provider. In return, the abstraction offers high-level Software-level Agreements (SLAs) to allow applications to express their performance needs, guiding the cloud provider in effectively managing resource control toward their goals. This approach simplifies application performance configuration while still enhancing cloud resource efficiency and usability.

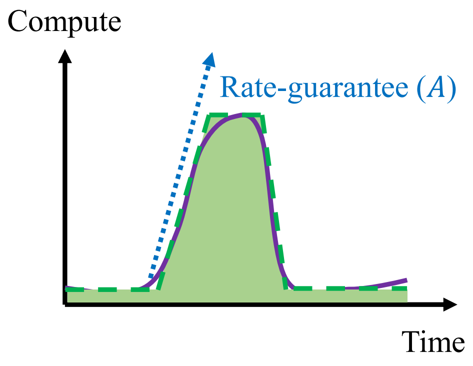
1.2.1 Function-as-a-Service

We construct the performance abstraction based on Function-as-a-service (FaaS), also known as Serverless[1] [23, 25, 68]. FaaS is one of the latest computation services offered by the cloud that aims to provide effortless application development and operation. 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

1. We use the terms *Serverless* and *FaaS* interchangeably



(a) FaaS



(b) RBAM (A = Peak Rate)

Figure 1.4: Supporting bursty, real-time applications based on the ideas of performance abstraction: (a) FaaS abstraction provides high applicability but fails to meet real-time deadlines (b) RBAM resolves the issue with guaranteed invocation rate, versatilely enabling real-time guarantees at low resource overhead.

resources and hold them until termination. By this scheme, an application can request more resource allocations by executing more invocations. These properties allow FaaS resource allocation to scale naturally with computation demand, giving an excellent efficiency potential 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 the computation completes. This delivers sufficient resources for real-time guarantee and significantly reduces the time, effort, and cost of application development and deployment, thus improving their applicability.

Unfortunately, conventional (or regular[2]) FaaS systems invoke invocations in a best-effort manner without any restriction on invocation latency. This scheme adds complexity and uncertainty to application performance because resource allocation cannot be timed 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

2. We use the terms "*regular FaaS*" and "*conventional FaaS*" interchangeably, both to refer to the current best-effort, heuristic-based implementation of FaaS systems.

allocation restriction, that usually results in long allocation delays or even cancellations. Consequently, as visualized in Figure 1.4a, the application fails to keep its computation up with the load growth and misses the deadlines.

Worse, FaaS systems take over invocation scheduling and resource management yet provide very limited support for applications to participate in or even give "hints" to the scheduling and resource management process to help them meet their performance needs. As a result, satisfying real-time requirements using FaaS is challenging, if not impossible.

1.2.2 Rate-based Abstract Machine

We address the above limitations by extending the FaaS abstraction to introduce a Rate-based Abstract Machine (RBAM), a novel performance abstraction that hides serverless invocation scheduling and resource management behind a configurable performance parameter called the "**guaranteed invocation rate**." This rate, treated as a Software-Level Agreement (SLA), is associated with a serverless function deployment to ensure a minimum function invocation starting rate defined as follows.

Definition 2 (Guaranteed Invocation Rate)**.** *Given a serverless function f_i, a guaranteed invocation rate A_i associates to f_i ensures there will be* ***at least*** *one invocation available for the function execution within* ***any*** *interval of length $\frac{1}{A_i}$.*

By the definition, a guaranteed invocation rate A_i is equivalent the following two guarantees:

- *Invocation Ramp-up Guarantee*: for any interval of length t, at least $\lfloor A_i t \rfloor$ invocations are guaranteed to get started. Thus, the function can *continuously* scale up its concurrency at a speed equal to A_i. For example, suppose a FaaS function f_i has a guaranteed invocation rate of $A_i = 10$ invocations per second. In that case, RBAM will ensure that at least 10 invocations are available for every 1 second, 100 for every 10 second, and so on.

- *Bounded Invocation Latency:* as long as the invocation request arrival rate is equal to or smaller than A_i, the invocation latency is *bounded* by $\frac{1}{A_i}$. For example, function f_i with guaranteed invocation rate $A_i = 10$ is ensured to get at least one invocation for every 0.1 seconds. Thus, as long as the inter-arrival of the function invocation requests is smaller than or equal to 0.1 second (i.e., arrival rate does not exceed ten invocation/sec), all invocation requests are guaranteed to start within 0.1 sec, effectively bound their invocation latency by $0.1 = \frac{1}{A_i}$.

By specifying the guaranteed invocation rate, the application delegates the responsibility of meeting this rate and the above guarantees to the cloud provider, freeing up time and effort for other development tasks. The guaranteed invocation rate provides a high-level, human-friendly interface to describe performance requirements. Converting application performance needs, such as minimum throughput or maximum acceptable invocation latency, into a guaranteed invocation rate is straightforward. With these features, RBAM offers a highly applicable solution for bursty, real-time applications.

Furthermore, with proper guaranteed invocation rate configuration, the developer can meet real-time deadlines cheaply. The bounded invocation latency guarantee ensures the application can always find the appropriate guaranteed invocation rate to complete invocation executions before the deadline. For example, the rate guarantee can be calculated as the inverse of the deadline minus the execution time, thus meeting real-time requirements. With the invocation ramp-up guarantee, the application can specify the rate guarantee to match the peak rate during bursts, ensuring that new invocations can be started at the same rate as the burst without requiring extra resources, thus meeting efficiency requirements (Figure 1.4b).

1.3 Thesis Statement

Current FaaS systems provide only *statistical performance Service-Level Objectives (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 italics are defined as follows.

- *Statistical Performance Service-Level Objectives (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 dissertation, unless specifically mentioned, we consider real-time deadlines as hard deadlines: 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.

1.4 Dissertation Contributions

In this thesis, we propose the Rate-based Abstract Machine (RBAM), a novel performance abstraction built on the foundation of the FaaS abstraction, to efficiently and scalably support bursty, real-time applications. RBAM simplifies application implementation and deployment. It further helps applications configure and realize various performance needs.

RBAM enables applications to meet real-time deadlines at a bounded cost. We build an analytical framework connecting rate-monotonic real-time workloads with RBAM's guaranteed invocation rate, allowing us to analyze the dynamic execution of the FaaS function under real-time constraints. Our analysis demonstrates that RBAM's guaranteed invocation rate effectively bounds the invocation latency of FaaS functions, ensuring real-time task deadlines. Additionally, we develop a method to constrain the required rate guarantee and the corresponding resource overhead to achieve any desired set of guarantees.

RBAM can be efficiently and scalably implemented on the cloud. We extend the conventional FaaS system implementation to propose a new RBAM implementation architecture. This new architecture incorporates rate-based invocation scheduling and resource management algorithms, leveraging underlying resource allocation statistics to enable cost-effective RBAM implementation. Through overallocation strategies, we further effectively utilize shared resources, allowing the robust implementation of thousands of RBAM functions on the cloud.

RBAM has broad applicability, demonstrated through implementing two different applications: distributed real-time video analytics and a stream processing engine called Storm-RTS. We systematically conduct analytical and experimental evaluations on RBAM implementations of these applications. The results show that RBAM effectively enables these applications to meet their real-time requirements across various workloads and deployment scenarios. Furthermore, RBAM can derive new forms of guarantee, providing additional capabilities such as performance transparency, predictability, and flexible deployment across multiple data centers.

The scientific contributions of the dissertation include:

- Rate-based Abstract Machine (RBAM), a new performance abstraction extending the FaaS abstraction to provide the guaranteed invocation rate. RBAM simplifies application implementation and deployment while providing the rate guarantee as a quality of service parameter, enabling applications to configure for various performance needs.

- RBAM ensures real-time deadlines for all real-time applications with a minimum rate guarantee equal to the total task release rate of the application at resource overhead never exceeding 100% compared to actual use.

- By leveraging the underlying resource allocation distribution, appropriate resource sharing, and overallocation strategies, we can reduce overhead for providing performance guarantee by 10×. This efficiency remains robust across various workload dynamics and deployment environments. Even in extreme cases, up to 100,000× worse than current practice, the resource cost required to fulfill the guarantee is less than 10× compared to practical settings.

- RBAM can be implemented scalably over the shared cloud resources. We can deploy thousands of RBAM functions over large-scale shared resources at 99.9999% guarantee availability with only 38% resource overhead.

- RBAM simplifies application implementation and greatly reduces the need for deployment reconfiguration for performance needs. The capability is applicable across a wide range of applications with diverse real-time demands and remains robust against varying workload burstiness. Even with workload characteristics varying by 25×, RBAM incurs only a 3× increase in rate requirement.

- RBAM's guaranteed invocation rate serves as a building block for creating new capabilities, including robust performance stability, high deployment flexibility, and simple performance management across distributed resources.

1.5 Dissertation Organization

The rest of the dissertation is organized as follows. In Chapter 2, we cover the background of the dissertation. We then present the research questions to be addressed in the dissertation and our approach to answer them in Chapter 3. Chapters 4, 5, and 6 present the primary content of the dissertation, including RBAM guarantee capability, how to implement it efficiently, and how to use the model for various bursty, real-time applications, respectively. We give a brief literature review of related work in Chapter 7 and summarize the dissertation with potential future research directions in Chapter 8.

CHAPTER 2
BACKGROUND

2.1 Cloud-Edge Continuum

The cloud-edge continuum is a seamless integration and coordination of computation resources and services across the cloud and edge. The cloud-edge continuum plays a crucial role in supporting bursty real-time applications, especially those with large-scale, distributed deployment [236]. By distributing computing resources closer to the edge where data is generated, the continuum reduces latency and ensures the timely processing of real-time tasks. Additionally, the continuum allows flexible resource allocation, enabling efficient utilization of cloud and edge resources based on application requirements and burstiness characteristics. In this section, we will briefly describe cloud and edge computing, including their key infrastructure characteristics (Sections 2.1.1 and 2.1.2) and challenges (Section 2.1.3).

2.1.1 Cloud Computing

Cloud Computing captures a large portion of Internet and enterprise IT computing services [138]. Cloud resources are shaped by virtualization technologies. Resources provided to users are abstracted from their physical counterparts in the form of virtual machines, containers, or serverless functions. Virtualization allows cloud providers to hide the complexity of underlying physical resources for better resource utilization, usability, and productivity [63]. Further, virtualization provides performance isolation so that cloud providers can guarantee customer resource quality.

Cloud resources often come from hyper-scale datacenters that consist of thousands to hundreds of thousands of machines offered by giant cloud providers, such as Amazon Web Service [54], Microsoft Azure [217], and Google Cloud Platform [144]. Conceptually, the cloud has an unlimited amount of computation resources covering a wide variety of hardware, including processing units (e.g., CPU, GPS, TPU), memory (Gigabytes to Terabytes of memory per machine), storage (e.g., SSD, NVM, HDD), and network bandwidth. That enables great scalability, allowing the applications to easily scale up for increasing demand and eliminating the need for costly investment in on-premises infrastructure. With highly elastic resources, computing resources can be provisioned and released on-demand, enabling applications to rapidly scale up or down based on their workload dynamics, delivering highly flexible and cost-effective computing solutions.

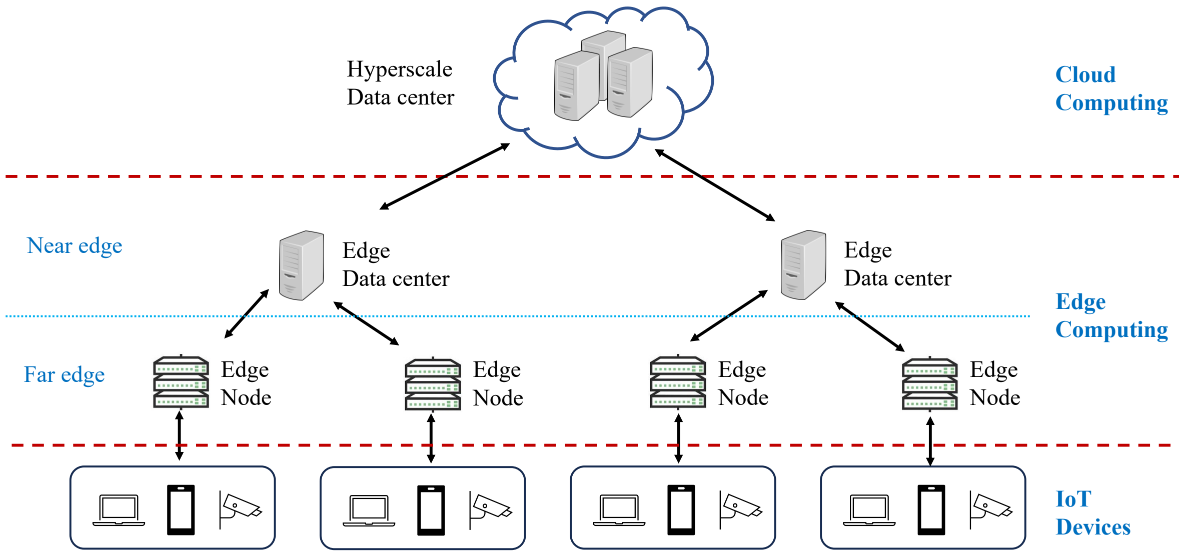


Figure 2.1: Cloud Edge Continuum

2.1.2 Edge Computing

While cloud resources are centrally organized in highly available, well-connected data centers, edge computing often comprises distributed resources that are close to the locations where they are needed. In this computing paradigm, data are processed and analyzed locally on edge devices, such as IoT devices, gateways, or edge data centers.

Edge computing is increasingly used as a complement to cloud computing due to its remarkable advantages (Figure 2.1). By processing data locally on edge devices, edge computing bypasses the roundtrip time to send the data from and to the centralized cloud data center, enabling ultra-fast processing and analysis. This also helps save network bandwidth and improve reliability by allowing the applications to continue functioning even when connectivity to the cloud is lost. Offloading processing from the cloud data centers also improves scalability and reduces costs.

2.1.3 Challenges

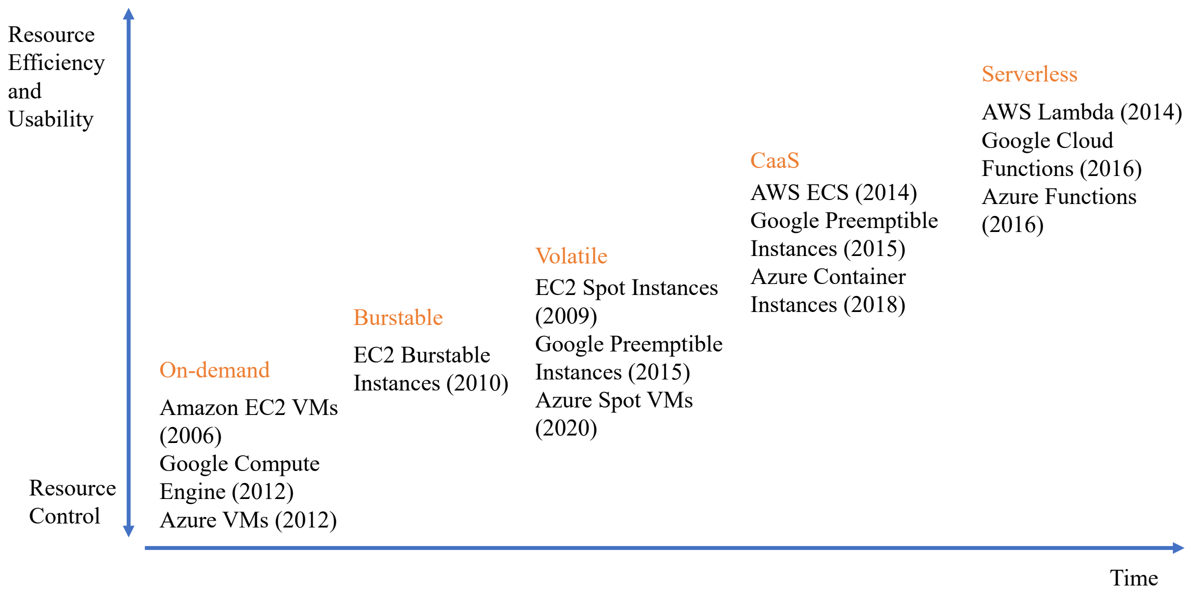

Resource Efficiency and Usability
Resource Control
Time
On-demand
Amazon EC2 VMs (2006)
Google Compute Engine (2012)
Azure VMs (2012)
Burstable
EC2 Burstable Instances (2010)
Volatile
EC2 Spot Instances (2009)
Google Preemptible Instances (2015)
Azure Spot VMs (2020)
CaaS
AWS ECS (2014)
Google Preemptible Instances (2015)
Azure Container Instances (2018)
Serverless
AWS Lambda (2014)
Google Cloud Functions (2016)
Azure Functions (2016)

Figure 2.2: Timeline of existing Cloud computation models. Newer computing models offer better resource efficiency yet reduce users' control over cloud resources.

Lack of Performance Guarantee Support. Consider Figure 2.2; in the early days, cloud and edge providers offered computation resources as virtual machines (VMs) in the form of on-demand or reserved instances that give users control of the entire software development stack, enabling them to employ sophisticated mechanisms for performance. However, VMs' coarse-grained resource allocations and the lack of adaptation to workload dynamics lead to high resource waste that drives up costs and reduces resource utilization. Further, taking over the whole software stack requires much effort for application development, deployment, and operations. Newer computation models limit user control over cloud resources to reduce waste. For example, volatile instances [57, 146] increase resource utilization and cost-effectiveness by offering unused resources to users at high discounts. However, this comes with the risk of being preempted at any time, thereby losing control of their computation lifetime. More recent services such as Container-as-a-Service (CaaS) and Serverless even take the computation scheduling and resource management out of application space and implement them inside the cloud. This not only opens more opportunities for the cloud to maximize resource utilization but also relieves the burden of application deployment and operation from the application developer, thereby improving their productivity. However, this shift also means that the application has to rely more on cloud provider resource management, which can adversely affect performance if the resource management decisions do not align well with their needs. This highlights the practical implications of the trade-off for cloud users, who must weigh the benefits of increased efficiency and reduced costs against the potential loss of control over their resources.

Distributed Resources. When application deployments span over multiple cloud and edge data centers, many issues associated with distributed resources arise, including

- *Data consistency*: Maintaining data consistency is challenging as it often comes with replication for better reliability and availability, which, in turn, complicates the processing pipeline, data synchronization, and bandwidth optimization.

- *Resource Management*: Optimizing resource allocation, computation placement, and scaling strategies in response to bursty demand to meet real-time deadlines is a significant challenge.

- *Distributed Orchestration*: Ensuring applications are deployed efficiently and communicate seamlessly over distributed resources requires careful planning and management.

Heterogeneous Resources. Data centers use different sets of hardware with different capacities and even get managed under different policies and mechanisms, making it hard to glue them together efficiently. Further, some data centers may be under the administration of private parties, adding another layer of security and privacy restrictions on resource access and usage.

Recent studies have proposed Function-as-a-Service (FaaS) as a potential solution to the problems above [230, 94, 64]. FaaS removes resource heterogeneity by abstracting resources under a uniform execution model of stateless functions. It also eliminates the burden of resource management (hence its alternative name, Serverless), providing more freedom to effectively exploit distributed resources. Further exploration of this topic will be provided in the next section.

2.2 Function-as-a-Service

2.2.1 Overview

Function-as-a-Service (FaaS) or Serverless is a resource abstraction that enables dynamic scalability with minimum effort and cost. The serverless abstraction lets applications exploit the underlying resources through *invocations*. An invocation is a discrete execution unit limited in time and resource use (e.g., timeout, CPU, memory). Applications associate invocations to their logics in the form of *stateless functions*. Each function is typically a specific task (e.g., read an image, resize an image, write an image to cloud storage, etc.) having a unique identifier (usually a URL). A function can be called (or invoked) by sending a request, along with required arguments, to this identifier (typically via HTTP POST). Call requests are automatically handled by a *serverless system* running at the cloud provider side (e.g., AWS Lambda [23], Azure Function [26], Google Cloud Function [25], etc.). For each request, the serverless system allocates computing resources to launch an invocation, which

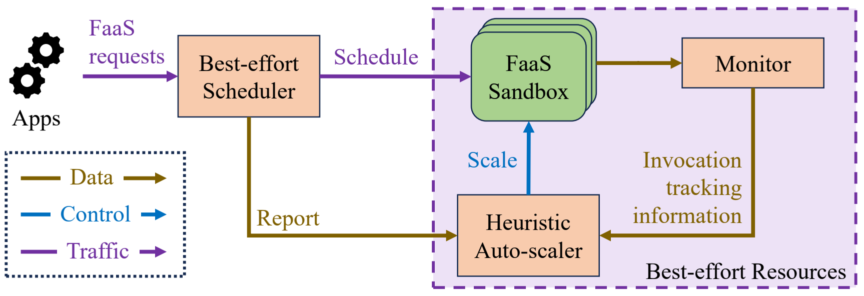


Figure 2.3: Conventional FaaS architecture

executes the function logic on the given arguments to complete the task. The serverless system is also responsible for cleaning up after the invocation terminates, keeping resource allocation minimal.

Serverless systems allow multiple invocations to be executed simultaneously and also implement an auto-scaling mechanism to automatically invoke up to thousands of function executions at the same time. Such abilities enable dynamic scalability with minimum efforts and cost, opening great opportunities for achieving cost-effective, scalable solutions [133, 51]. Given these advantages, FaaS is gaining popularity outside cloud data centers. There are efforts to realize the FaaS ideas inside edge data centers [230] and for high-performance computing workflows [289].

2.2.2 FaaS Architecture

Figure 2.3 depicts the architecture used by conventional FaaS systems [23, 68, 25, 180, 240, 20] to handle FaaS invocation requests. Usually, the system consists of two key components:

- *Best-effort resources:* provide computation resources (e.g., CPU and memory) in isolated environments called *sandboxes* to execute invocations. Sandboxes are typically implemented as Virtual Machines [44, 27], containers [115, 116, 303], or Unikernel [206, 211]. For simplicity, we assume each sandbox executes only one invocation at a time. Sandboxes are managed by an *autoscaler* that collects FaaS execution information through a *monitor* and then applies *heuristic scaling strategies* to (de)allocate sandboxes accordingly.

- *Scheduler:* dispatch invocation requests from the applications to appropriate sandboxes in a *best-effort* manner. Invocations are started immediately as long as there is an available sandbox. Otherwise, the scheduler has to queue them and report the situation to the autoscaler for more sandboxes to drain the queue quickly.

2.2.3 FaaS Limitations

With the given architecture, conventional FaaS systems take over invocation execution and resource management, enabling application developers to focus on their core activities. However, because the FaaS systems implement these functionalities in best-effort and heuristic-based manners, the developers face new challenges, including (i) limited applicability and (ii) counterproductive performance configuration.

Limited Applicability The current FaaS systems' decisions are driven by collective performance goals (e.g., minimizing the cold start [116, 44, 27] and its appearances [278]). However, the impact (and side-effect) of fulfilling these goals are different across applications. For example, many FaaS systems use lightweight sandboxing mechanisms (e.g., [115, 206]) to reduce the cold start to a sub-second level. This dramatically reduces the end-to-end latency of short invocations (last in a few hundred milliseconds) but virtually has no impact on invocations that take several minutes to complete. These invocations may prefer heavy-weight sandboxing isolation (e.g., [44]) to protect them from interference or resource contention that could slow their execution. Further, current FaaS systems achieve these goals through best-effort, heuristic-based approaches (See Section 7.2.1) and evaluate them via statistical metrics (e.g., average or 99th percentile). This means the efficiency of their invocation scheduling and resource management is uncertain and subjected to uncontrollable factors As a result, conventional FaaS systems have limited applicability. Applications with strict performance requirements (e.g., real-time [117]) or characteristics not well supported by the best-effort, heuristic FaaS implementation (e.g., bursty, highly demanding applications [234, 235]) will find it challenging to use FaaS in practice, if not impossible.

Counterproductive Performance Configuration Conventional FaaS systems provide limited support to let applications configure their FaaS deployments for performance. There is no way for the application developers to express their performance requirements and instruct the FaaS system to enforce them. Instead, they must manually tune FaaS deployment and execution parameters to achieve the desired performance. Mitigating cold-start is a well-known example in this regard. Currently, the most common and straightforward approach to avoid cold-start is sacrificing cost-effectiveness. The application developer either has the FaaS system to pre-allocate sandboxes or extend the lifetime of those used by terminated invocations and recycle them for incoming invocation requests [65, 181]. The tricky part is to find the correct number of pre-allocated sandboxes and a keep-alive period to minimize unused resources. Unfortunately, there is no magic formula to the problem. It depends on the workload dynamics, which are widely different across applications [278]. The application developer may need much effort with many rounds of "try-and-error" to capture the dynamic and find the optimal configuration. This contradicts the FaaS's very first objective: to relieve them from the burden of such concerns for productivity.

Addressing the Limitations These limitations restrict not only the deployment of real-time bursty applications using FaaS, as discussed in Section 1.2.1 but also other demanding application with strict performance requirement, such as stream processing (see Section 6.2). The main reason is the lack of support for applications to express their performance requirements in a way that can inform FaaS scheduling and resource management/autoscaling decisions. RBAM, on the other hand, allows applications to define their performance requirements in terms of a guaranteed invocation rate and use it as a quality of service (QoS) parameter to configure and control their performance. This effectively resolves these limitations, as discussed later in the next chapters.

CHAPTER 3
RESEARCH QUESTIONS AND APPROACHES

The dissertation proves that the RBAM performance abstraction is a general solution for efficiently guaranteeing real-time performance on FaaS. To meet this goal, we evaluate RBAM against the requirements outlined in Section 1.1.3. We seek to answer three research questions: (i) Can RBAM ensure real-time application deadlines? (ii) If yes, then can this be done efficiently? and (iii) Can RBAM apply to a broad class of applications?

To answer these questions, we first consider whether RBAM enables bursty, real-time workloads to meet their computation deadlines. Second, we propose an RBAM implementation architecture and associated algorithms to demonstrate its efficiency and scalability. Third, we use RBAM to implement various applications with different burstiness properties and real-time requirements to evaluate its applicability. The following sections will elaborate on these three points in more detail.

3.1 Real-time Guarantee

3.1.1 Research Questions

The first part of the dissertation evaluates the capability of the RBAM performance abstractions to support real-time guarantees. This includes showing that RBAM can support any real-time deadline. Also, is it possible to derive a corresponding rate guarantee to meet a specific deadline given its associated workload dynamics (e.g., execution time and arrival rate)? Is the required rate guarantee feasible in practice? Can we bound the rate to make it practical? We summarize these questions as follows.

Real-time Guarantee Capability. Can RBAM ensure real-time deadlines for all real-time applications?

Q.1.1 Can RBAM's guarantee invocation rate enable applications to meet their real-time deadlines?

Q.1.2 Can we show how to derive bounds on RBAM invocation rates for any set of real-time workloads?

3.1.2 Approach

We construct the answers to these questions in two steps. First, we develop an analytical framework based on rate-monotonic real-time workloads that are broad enough to cover the majority of practical real-time applications. Second, by connecting the fundamental properties of rate-monotonic real-time workloads with RBAM's guaranteed invocation rate, we answer each research question as follows.

- *Can RBAM's guarantee invocation rate enable applications to meet their real-time deadlines? (Q.1.1)* We prove that RBAM's guaranteed invocation rate can bound the start time of any rate-monotonic real-time task, ensuring that it meets its deadline. Then, we generalize this result to multiple rate-monotonic real-time tasks, proving that RBAM can meet all deadlines.

- *Can we show how to derive bounds on RBAM invocation rates for any set of real-time workloads? (Q.1.2)* We develop a method to derive RBAM's guaranteed invocation rate for specific rate-monotonic workload properties, showing that any real-time workload can meet its deadlines with a bounded RBAM guaranteed invocation rate and corresponding resource overhead.

By proving the above statements, we show that RBAM can deliver real-time guarantees at *bounded* resource cost for *any* rate-monotonic real-time applications. This makes RBAM a *practical* FaaS model for a *wide range* of real-time bursty applications, sufficiently demonstrating its real-time guarantee capability.

We present the work following this approach in Chapter 4 with the analytical framework described in Section 4.1. Next, we use the framework to answer questions Q.1.1 and Q.1.2 in Sections 4.2 and 4.3, respectively. Finally, we construct a case study based on practical real-time applications to confirm the theoretical proofs in Section 4.4.

3.2 Efficient RBAM Implementation

3.2.1 Research Questions

The second part of the dissertation proposes an RBAM implementation architecture, including both rate-based invocation scheduling and resource management algorithms. These elements are used to show that RBAM can be implemented efficiently (at a moderate overhead as in Section 1.1.3) across a wide range of deployment and workload dynamics. Further, the cloud expects to support multiple RBAM concurrently with broad rate-guarantee requirements, so the implementation must support a wide range of rate guarantees and scale up to thousands and more functions. To sum up, we need to answer the following questions.

Implementation Efficiency. Can RBAM be implemented efficiently?

Q.2.1 Can RBAM be implemented with any finite guaranteed invocation rate?

Q.2.2 Can RBAM also be implemented with low overhead and is robust across various scenarios?

Q.2.3 Can RBAM be implemented scalably?

3.2.2 Approach

To answer these questions, we take three steps. First, we perform a statistical study to understand the statistical structure of the workload dynamics and deployment environment. We use these insights to identify RBAM implementation challenges. Second, we propose ideas to address these challenges and then combine them to propose an RBAM implementation architecture and design corresponding algorithms to leverage the architecture for efficient RBAM implementation. Third, we use the implementation to answer the research questions as follows.

- *Can RBAM be implemented to support any finite guaranteed invocation rate? (Q.2.1)* We show that RBAM algorithms can support any finite rate guarantee with a 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 and is robust across various scenarios? (Q.2.2)* We design a set of experiments to examine the RBAM system against different combinations of deployment environment settings and workload dynamics, including both practical ones and extreme synthetic settings that could be far worse than in practice. We record the overhead incurred by the system in implementing RBAM in all these cases and use the results to demonstrate RBAM's cost-effectiveness.

- *Can RBAM be implemented scalably? (Q.2.3)* Similar to answering Question Q.2.3, we deploy RBAM functions at different scales with different RBAM requirements and deployment settings. We then use the recorded overhead and guarantee capability to illustrate RBAM scalability.

Chapter 5 presents the work following this approach to prove RBAM implementation efficiency. We first prove RBAM implementation feasibility to answer question Q.2.1 in Section 5.1. Next, we mention efficiency challenges based on statistical studies in Section

5.2. Sections 5.3, 5.4, 5.5, and 5.6 present ideas to address the challenges and realize the RBAM system. We use the experimental results in Section 5.7 to answer Questions Q.2.2 and Q.2.3.

3.3 RBAM Applicability

3.3.1 Research Questions

The third part of the dissertation addresses RBAM's applicability. This consists of two parts. First, we study the applicability of the RBAM performance abstraction across applications with various burstiness structures and real-time requirements. Second, we address whether RBAM can be used as a building block to realize more complicated requirements (e.g., performance transparency and predictability) and offer new capabilities for other application classes, such as flexible deployment across distributed resources. These are equivalent to answering the following research questions.

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 other real-time guarantees that capture a broad class of applications with quality guarantee?

3.3.2 Approach

We use RBAM to implement two different classes of applications that represent a diverse set of demanding applications with different real-time requirements:

- *Distributed Real-time Video Analytics*: a popular application class with bursty demand driven by highly unpredictable external events. The bursty demand can be either soft or hard real-time.

- *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 analytics, we show that the RBAM performance abstraction can guarantee a wide range of real-time requirements, from loose soft deadlines of a few seconds to stringent hard deadlines at a microsecond scale. Further, this capability is robust against different workload burstiness, from periodic ones to highly unexpected burst loads with highly demanding computation.

- *Can RBAM be used to construct 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 multiple data centers, that broaden the scope of RBAM usage to deliver more computation value.

Chapter 6 presents the work following the approach to prove RBAM applicability. The chapter consists of two sections. Section 6.1 presents the distributed video analytic implementation using RBAM and experimental results to answer Question Q.3.1. Meanwhile, Section 6.2 answers Question Q.3.2 with the case of stream processing.

CHAPTER 4

RBAM REAL-TIME GUARANTEE CAPABILITY

In this chapter, we will prove that RBAM can ensure real-time deadlines for all real-time applications. In Section 4.1, we propose an analytical framework by extending the rate-monotonic real-time workload model to the case of dynamic task execution on FaaS resources. By utilizing the widely studied, well-understood foundation of the rate-monotonic framework, we perform execution analysis of real-time tasks and reveal that current FaaS systems cannot support hard real-time applications due to the unbounded latency of regular FaaS resource provisioning.

Section 4.2 considers the effect of RBAM's invocation rate guarantees and shows that the rate guarantee can bound their invocation latency, effectively ensuring hard real-time deadlines. This is significant because it allows hard real-time guarantees to be achieved without wasting application resources. Following this, in Section 4.3, we introduce an application technique called *pre-invocation*, which reduces the required guaranteed invocation rate by trading some resource waste. Finally, we apply insights from the analytical results to illustrate how applications can use RBAM to meet hard real-time constraints through a practical case study in distributed virtual/augmented reality in Section 4.4. We summarize our findings in Section 4.5.

4.1 Analytical Framework

4.1.1 Real-time Execution Model

We construct an analytical framework that bridges real-time deadlines with FaaS serving by mapping the execution of rate-monotonic workload onto FaaS dynamic allocation. Based on this mapping, we develop theory and mathematical proofs around task execution analysis to characterize FaaS real-time support limitations and get insights into how a guaranteed invocation rate could resolve the situation.

We aim to support periodic real-time applications, such as those described in a rate-monotonic workload, but with dynamic allocation, so that minimum compute resources are used.[1] Consider a *rate-monotonic* real-time workload [198] with n periodic real-time tasks $T_1, ..., T_n$ each characterized by

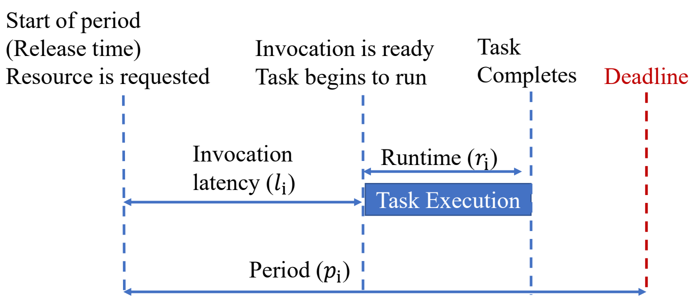
- *Period* (p_i): the task recurs every p_i time units. For simplicity, we assume tasks are released at the beginning of each period, and have to finish by the end of that period (i.e., the hard real-time deadline).
- *Runtime* or *Execution time* (r_i): the time for task to run.
- *Slack* $s_i = p_i - r_i$: the time a task does not spend on execution within a period.

We consider two ways of implementing a task T_i:

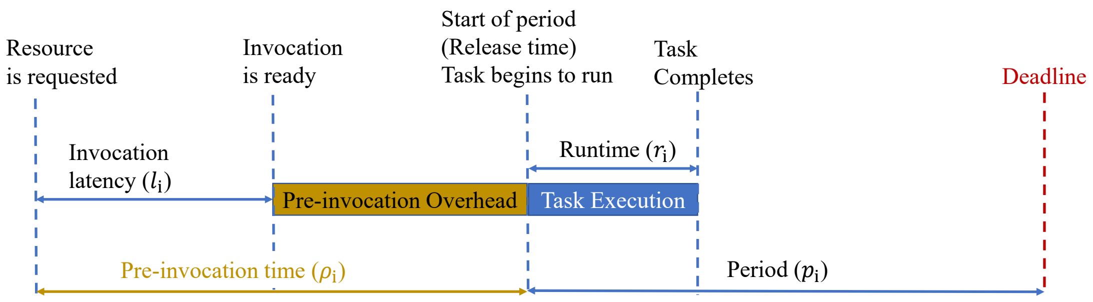
- **FaaS**: stateless functions invocations serve each task at one invocation per task release.
- **RBAM**: similar to FaaS, except that for each function, invocation rates can be guaranteed (A_i), ensuring that the number of invocations provided must be at least equal to 1 for any arbitrary period of length $\frac{1}{A_i}$.

Due to implementation overhead (e.g., initialization, resource allocation latency, etc.), both FaaS and RBAM invocations have to wait for l_i time unit(s) (*invocation latency*) after being requested to start execution. With RBAM, the guaranteed invocation rate promises at least 1 invocation for any $1/A_i$ period, thus the invocation latency is bounded by $l_i \leq 1/A_i$ as long as $A_i \geq \frac{1}{p_i}$ (i.e., the rate of task release does not exceed the guaranteed invocation rate). Based on the guaranteed invocation rate definition (See Section 1.2.2), FaaS is equivalent

1. we can also handle bursty versions of rate monotonic workloads where tasks conform to the rate monotonic structure *when they occur*, but they often don't appear in their periods.



(a) Task on FaaS invocation



(b) Task on FaaS with pre-invocation

Figure 4.1: Supporting periodic tasks dynamically on FaaS compute model.

to RBAM with a guaranteed invocation rate of zero, which means its invocation latency is unbounded.

Figure 4.1 shows two examples of timing diagrams for single periodic task execution. Normally, an invocation is requested right at the time a task is released (Figure 4.1a), so the response time for task T_i would be $l_i + r_i$. To workaround invocation latency, invocations can be requested *in advance* to make it available at the time a task is released for immediate execution. We call this technique *pre-invocation* and use ρ_i to denote pre-invocation time as shown in 4.1b. Note that pre-invocation shortens task response time at the cost of unused resources (waste) when invocation gets ready before a task release (brown bar in Figure 4.1b). We summarize the framework notations in Table 4.1.

Symbol	Time Interval	Note
p_i	Period	$p_i \geq r_i$
r_i	Runtime/Execution Time	$r_i > 0$
s_i	Slack for task $i = p_i - r_i$	
l_i	Invocation latency for task T_i	
ρ_i	Pre-invocation time for task T_i	

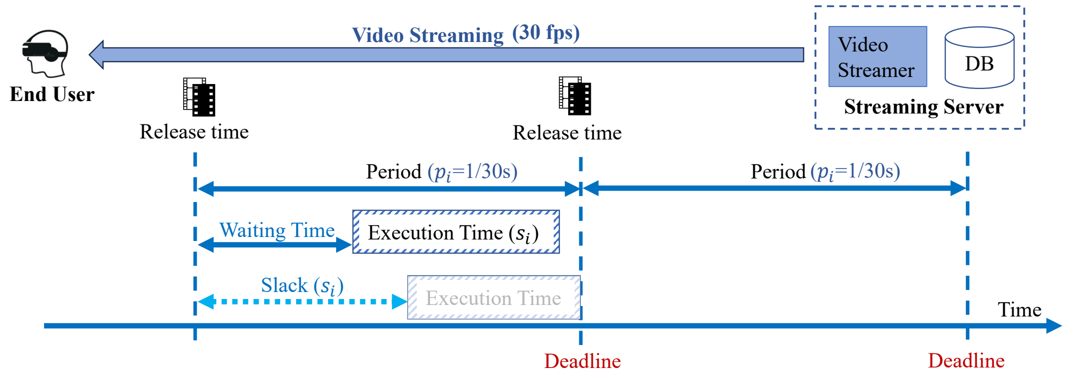
Table 4.1: Rate Monotonic Notation for each task T_i

4.1.2 Illustrative Example

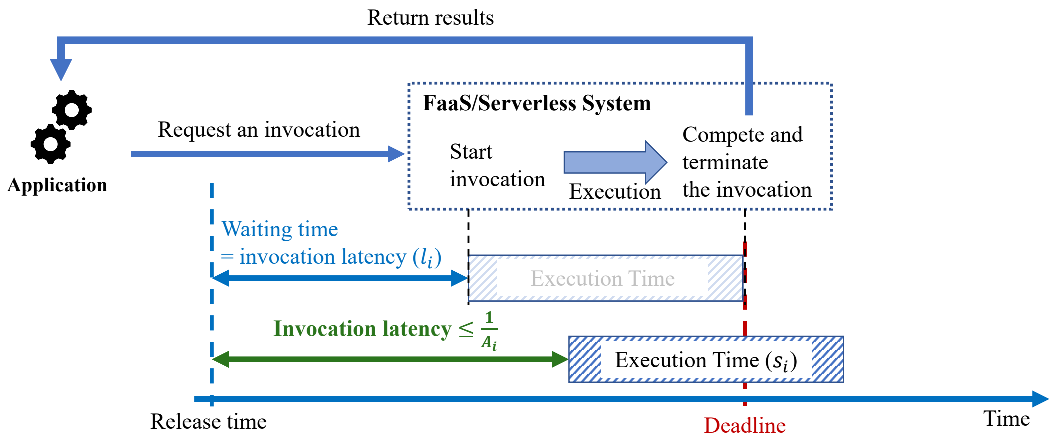
Before applying the analytical framework to explore FaaS and RBAM real-time guarantee capability, let us provide an illustrative example to explain and clarify the framework. This example will demonstrate how to model a practical bursty, real-time application as a rate-monotonic workload and analyze its real-time performance on the dynamic FaaS execution model.

Figure 4.2a illustrates a simplified implementation of a VR/AR streaming application comprising a streaming server and its end users. When an end-user selects video content from the application interface on their device, the device connects to the streaming server, requesting the selected video content. The server checks its database for the requested video content and verifies the user's authorization to access it. If the video exists and the user is authorized, the server's video streamer initiates streaming, delivering the video content to the end user.

Streaming is a sequence of video frames continuously transferred from the server to the end user. For a high-quality experience, the video streamer has to transfer the video frame at the minimum speed of 30 frames per second (fps). Each video frame transfer must be completed before the next one. The process can be modeled as a single rate-monotonic periodic real-time task T_i, with a period $r_i = \frac{1}{30}$ seconds. At the beginning of each period, one video frame is released, triggering the task execution. Suppose the task runtime time (r_i) is *at most* 10ms. The task runtime must be completed within $\frac{1}{30}$ seconds before the other video frame releases, given its slack $s_i = 1/30 - 0.01 \approx 23ms$.



(a) Modeling the VR/AR streaming as a rate-monotonic, periodic real-time task



(b) Implementing the task as a FaaS function and handling its releases as invocation executions.

Figure 4.2: Illustrative Example of the Analytical Framework: VR/AR Streaming

Applying the FaaS/RBAM computing model, we can implement the task T_i as a single FaaS function. Every time a video frame is released, the video streamer requests a new invocation to transfer this frame (Figure 4.2b). This is done by sending an invocation request to a FaaS system in the cloud. Once the request is received, the FaaS system starts an invocation to handle the task. However, the invocation execution does not start immediately but experiences a noticeable invocation latency l_i. Once the invocation starts, the frame transfer is handled by the invocation until completion. By then, the FaaS system terminates the invocation and returns the frame-transferring result to the streamer.

To meet the user experience quality, the streamer has to receive the results before the deadline. In other words, the end-to-end latency of the invocation execution—from the moment the request is sent to the moment the streamer receives the results—must be shorter than the task period. In the following sections, we will discuss the challenges associated with achieving this goal using a regular FaaS system, and demonstrate how RBAM can configure its rate guarantee based on deadlines (p_i) and other workload dynamics (s_i, r_i) to resolve the challenges.

4.1.3 Limits of Regular FaaS

Based on the rate-monotonic workload model, we can easily prove that FaaS alone is unable to guarantee that the periodic tasks will meet their deadlines as follows.

Theorem 1. *Regular FaaS functions cannot guarantee that a single periodic task in a rate-monotonic workload will meet its deadline.*

Proof. Given a periodic task T_i, with invocation latency l_i and runtime r_i, the time to complete the task can be written as $l_i + r_i$ which, for the task to meet its deadline must be

$$l_i + r_i \leq p_i \implies l_i \leq p_i - r_i \tag{4.1}$$

Let $l_i = \tau$ for a FaaS invocation and $Prob(\tau > x)$ be the probability that τ is greater than x. Because the FaaS invocations are best effort, $0 < \tau < \infty$, and for any given $p_i - r_i$, the

$$Prob(\tau > p_i - r_i) > 0 \tag{4.2}$$

This means that

$$Prob(l_i + r_i > p_i) > 0 \tag{4.3}$$

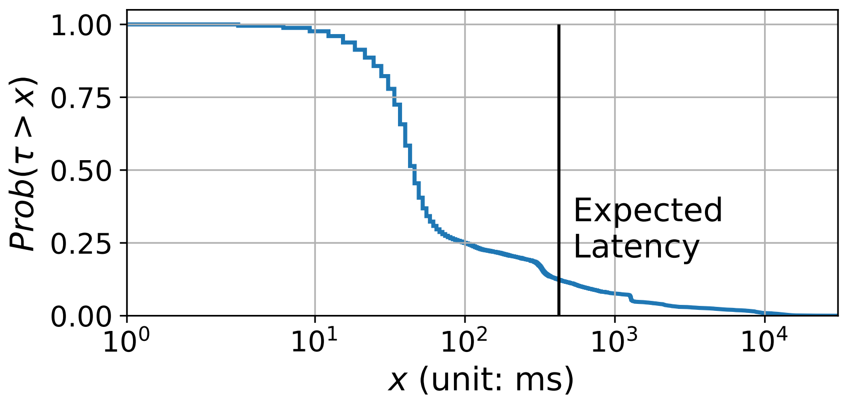


Figure 4.3: Google Cloud Functions invocation latency distribution [36]

that is the chance that the task misses its deadline is greater than zero – its real-time performance is not guaranteed. □

Figure 4.3 shows $Prob(\tau > x)$ estimated from invocation latency statistics of Google Cloud Functions [145, 36], a commercial FaaS system. This is a long-tailed distribution. Invocation latency can exceed 30 seconds, more than 10-100x longer than the expected invocation latency [278]. This suggests that the unbounded invocation latency is a practical issue, not purely theoretical. Also, the latency is widely distributed, leading to significant delays. This makes realizing real-time applications on top of regular FaaS very challenging, if not impossible.

FaaS is insufficient for even a single task, so we can easily show that it is unable to support the entire rate-monotonic workload of multiple periodic tasks.

Theorem 2. *Regular FaaS functions cannot guarantee that a set of periodic tasks in a rate-monotonic workload will meet their deadlines.*

Proof. Choose an arbitrary task T_i from the multiple periodic tasks. Theorem 1 shows that FaaS cannot guarantee T_i will meet its deadline. Therefore, FaaS cannot guarantee that all of the tasks in the rate monotonic workload meet their deadlines. □

Takeaway. (Limits of Regular FaaS) Regular FaaS has unbounded invocation latency and thus fails to meet real-time deadlines.

4.2 RBAM Enables Real-time Performance

Unbounded invocation latency is the primary limitation that prevents regular FaaS from meeting real-time deadlines. In contrast, RBAM can bound this latency with its guaranteed invocation rate. We first prove that this bound can guarantee tasks meet their deadlines as long as the tasks have non-zero slack. Later, we will show how this requirement can be relaxed by using pre-invocation.

Theorem 3. *RBAM can guarantee* <u>*one*</u> *periodic task* T_i *meets its deadline, if it has slack of* $s_i = (p_i - r_i) > 0$.

Proof. For T_i to meet its deadline, we must have

$$l_i + r_i \leq p_i \tag{4.4}$$

or

$$l_i \leq (p_i - r_i) \tag{4.5}$$

RBAM provides a guarantee of invocation rate, such that there is at least one invocation in every period of length $1/A_i$, where A_i is the guaranteed invocation rate chosen for RBAM. This means that as long as $A_i \geq \frac{1}{p_i}$, l_i can be bounded as follows:

$$l_i \leq \frac{1}{A_i} \tag{4.6}$$

So, to meet the deadline, we must ensure that

$$\frac{1}{A_i} \leq (p_i - r_i) \tag{4.7}$$

Which we can assure for any slack $s_i = (p_i - r_i) > 0$, by picking a sufficiently large $A_i \geq \frac{1}{p_i - r_i} \geq \frac{1}{p_i}$ and gives

$$l_i \leq \frac{1}{A_i} \leq (p_i - r_i) \tag{4.8}$$

which is true because $s_i > 0$. So the deadline is met. □

Applying the result to the illustrative example in Section 4.1.2: we only meet the deadline if the end-to-end latency of frame transferring is shorter than the task period ($p_i = 1/30s$). Thus, if the FaaS function implementing the task is associated with an RBAM guaranteed invocation rate A_i, by Equation 4.6, we need to configure

$$A_i \geq \frac{1}{l_i} = \frac{1}{p_i - r_i} = \frac{1}{s_i} \approx \frac{1}{0.023} \approx 44 \text{ invocation/sec} \tag{4.9}$$

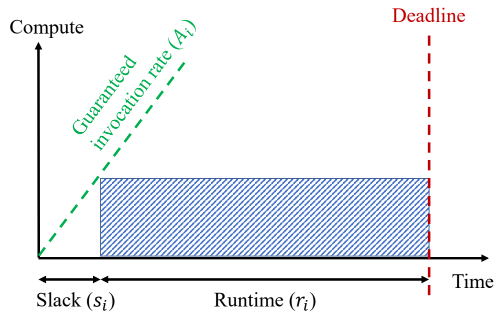
Recall that the slack $s_i = p_i - r_i = 1/30 - 10 \approx 23ms$. Let choose $A_i = 50 > 44$ invocation/sec, then we effectively bound the latency l_i by $\frac{1}{50} = 20ms$. Thus, every time a video frame releases, its end-to-end latency is $l_i + r_i \leq 20 + 10 = 30ms < 1/30$ seconds, ensuring the real-time performance for the streaming task.

Now, let us generalize Theorem 3, considering a workload with multiple periodic tasks $T_1, ..., T_n$.

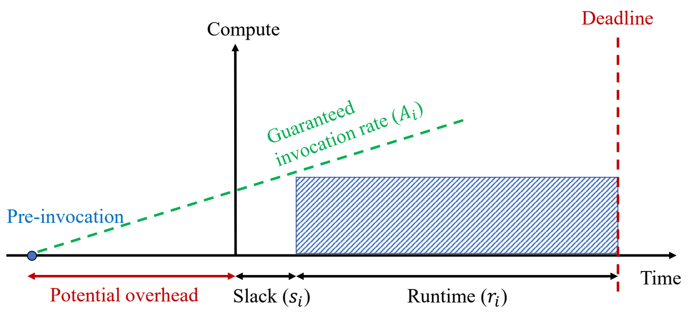
Theorem 4. *RBAM can guarantee that a set of n periodic tasks $T_1, ..., T_n$ meet their deadlines, if each has slack of $(p_i - r_i) = s_i > 0$.*

Proof. Consider a task T_i, because it is served by a dedicated function, by Theorem 3, the invocation rate of

$$A_i = \frac{1}{p_i - r_i} \tag{4.10}$$



(a) Invoke at task release



(b) Invoke before task release (Pre-invocation)

Figure 4.4: Tasks with short slack require high invocation rate guarantees which can be reduced by pre-invocation at a waste in resources.

is sufficient for T_i to guarantee meeting its deadline. We assume each single task T_i uses a dedicated function with a finite guaranteed invocation rate A_i. From the application point of view, there is no invocation contention between the tasks. So, we can repeat the argument for each of the other tasks, then the theorem is proved. $\square$

Takeaway. (Real-time Guarantee) RBAM's rate guarantee ensures that invocation latencies are bounded, allowing rate-monotonic real-time workloads to meet their deadlines with positive slack.

4.3 Efficient Real-time Guarantee on RBAM

Theorems presented in Section 4.2 prove RBAM's capability of ensuring real-time deadlines of rate monotonic applications. The one restriction was that the rate-monotonic tasks have non-zero slacks. Further, as in Theorem 3, we can see that the required guaranteed invocation rate can be high for small slack. For example, if a task has a period of 15 seconds and a slack of only 1 second, the required guaranteed invocation rate would be 1/second, or 15× higher than the task rate. While in many realistic settings, many applications have non-zero, or better yet, a large slack for each task, the properties proved in Theorem 4 can be sufficient.

However, to go further, in this section, we will relax this restriction, using pre-invocation, and further show that pre-invocation can dramatically reduce the rate requirements.

Pre-invocation arises from the notion that RBAM depends exclusively on the dynamic acquisition of resources from the underlying resource management system (as does FaaS). This means the delay in acquiring such resources is critical in delivering real-time guarantees. That connection is illustrated in Figure 4.4a for a single rate-monotonic task. The required guaranteed invocation rate is determined by the slack and is much greater than $\frac{1}{p_i}$ – though intuitively, that rate matches the average needs of the rate monotonic task.

Pre-invocation allocates resources early, anticipating the arrival of a task, as shown in Figure 4.4b. Because the resources are acquired before they are needed, pre-invocation wastes resources. However, as we will see, it can significantly reduce the invocation rate guarantee requirement.

4.3.1 Efficient Real-time Guarantee for Single Task

First, we explore pre-invocation for a single task.

Theorem 5. *With a finite pre-invocation of ρ_i, an application can use RBAM to guarantee deadlines of <u>one</u> periodic task T_i with any zero or positive slack (i.e., $s_i = p_i - r_i \geq 0$). This loosens the requirement of Theorem 3.*

Proof. Assume the task T_i employs pre-invocation of

$$\rho_i = r_i \tag{4.11}$$

Then by serving T_i with RBAM of rate

$$A_i = \frac{1}{p_i} \tag{4.12}$$

Its invocation latency is bounded as

$$l_i \leq \frac{1}{A_i} = p_i \tag{4.13}$$

There are two possible cases

- $l_i \leq \rho_i$ meaning there is an invocation available at the time the task releases, so it requires only r_i to complete and

  $$r_i \leq p_i \tag{4.14}$$

  so the deadline is met.

- $\rho_i < l_i \leq p_i$ meaning the task has to wait for its invocation, so it waits $(l_i - \rho_i)$, and completes in

  $$(l_i - \rho_i) + r_i \tag{4.15}$$

  which is equal to l_i, and $l_i \leq p_i$, so the deadline is met.

Thus, pre-invocation with RBAM can guarantee a single periodic task meeting its deadline, even if the task has no slack. □

With Theorem 5, RBAM is sufficient for *any* single periodic task to guarantee its deadlines. Now, let us consider the cost of achieving real-time guarantees. The cost has two components: pre-invocation (wasted computation or overhead) and guaranteed invocation rate (higher rate guarantee requires more implementation effort and hence more costly, See Section 5). With our model, we study the interplay between these costs. First, given Theorem 5 and a finite but very small pre-invocation, we will show that real-time guarantees can be met with essentially no pre-invocation overhead and a guaranteed invocation rate of $A_i \geq \frac{1}{p_i}$ as follows.

Theorem 6. *A single periodic task T_i requires at least an invocation rate*

$$A_i \geq \frac{1}{p_i} \tag{4.16}$$

to meet its deadline.

Proof. We will prove by contradiction. That is, assuming that T_i is guaranteed to meet its deadlines at invocation rate of $A'_i < \frac{1}{p_i}$. Now, let

$$\epsilon = \frac{1}{p_i} - A'_i > 0 \tag{4.17}$$

Consider an interval of length $m = \frac{1}{\epsilon}$. Let I_i be the number of invocations needed to be completed by T_i within this interval, then

$$I_i \geq \lfloor \frac{m}{p_i} \rfloor \tag{4.18}$$

while the number of invocations we are guaranteed to have at the rate A'_i is

$$N_i = \lfloor m \cdot A'_i \rfloor \tag{4.19}$$

A necessary condition to guarantee that T_i does not miss any deadline over m is

$$N_i \geq I_i \tag{4.20}$$

However, because

$$(\frac{m}{p_i}) - (m \cdot A'_i) = m(\frac{1}{p_i} - A'_i) = \frac{1}{\epsilon}\epsilon = 1$$

Then

$$I_i - N_i \geq \lfloor \frac{m}{p_i} \rfloor - \lfloor m \cdot A'_i \rfloor = 1 \tag{4.21}$$

This means that $I_i > N_i$ so T_i is unable to guarantee its deadlines, contradicting the hypothesis. Thus, the invocation rate must be at least $\frac{1}{p_i}$. This proves the theorem. $\square$

More generally, let's derive a precise expression for the required pre-invocation, given a sufficient guaranteed invocation rate:

Theorem 7. *Given task guaranteed invocation rate of $A_i \geq \frac{1}{p_i}$, we can ensure task T_i meeting its deadline with a pre-invocation time of*

$$\rho_i \geq \frac{1}{A_i} - (p_i - r_i) \tag{4.22}$$

Proof. Let us consider two possible cases

- *Case 1.* $\rho_i \geq \frac{1}{A_i}$, then

  $$l_i \leq \frac{1}{A_i} \leq \rho_i \tag{4.23}$$

  then there is always an available invocation before the task releases, meaning it only requires r_i to complete and

  $$r_i \leq p_i \tag{4.24}$$

  so the deadline is met.

- *Case 2.* $\frac{1}{A_i} - (p_i - r_i) \leq \rho_i < \frac{1}{A_i}$ then l_i is no longer bounded by ρ_i. The invocation may arrive before the task releases, then it falls back to Case 1, where T_i meets the deadline. Otherwise, T_i has to wait for invocation after releasing, so it would take the

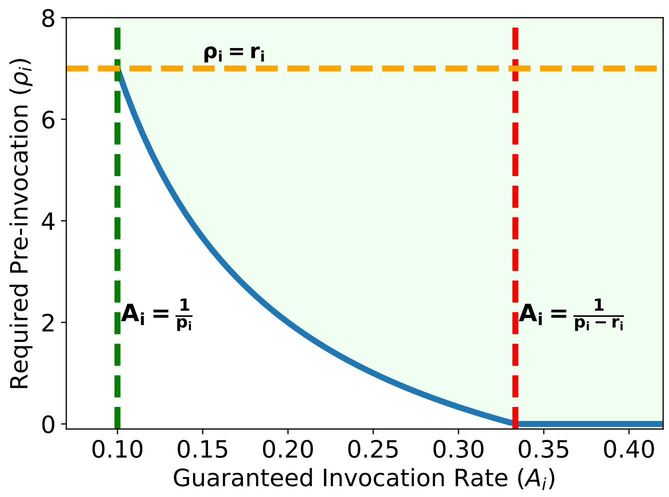


Figure 4.5: Required pre-invocation to guarantee real-time deadlines of a single task $T_i(p_i = 10, r_i = 7)$ varying guaranteed invocation rate A_i. The colored area shows (rate-guarantee, pre-invocation) combinations that ensure the task's real-time deadlines.

task RT_i time unit(s) to complete, where RT_i is determined as

$$\begin{aligned} RT_i &= l_i - \rho_i + r_i \\ &\leq \frac{1}{A_i} - \rho_i + r_i \\ &\leq \frac{1}{A_i} - [\frac{1}{A_i} - (p_i - r_i)] + r_i \\ &= p_i \end{aligned}$$

Thus, T_i also meets its deadline.

Therefore, $\rho_i \geq \frac{1}{A_i} - (p_i - r_i)$ ensures the task to guarantee meeting its deadline. □

Theorem 7 shows that given a sufficient guaranteed invocation rate, we can choose the minimum pre-invocation needed to enable a single task to ensure its real-time deadlines. This is the most efficient (least resource waste), given a sufficient guaranteed invocation rate.

The pre-invocation and guaranteed invocation rate relationship is shown in Figure 4.5. Realizing the lower invocation rate bound $A_i = \frac{1}{p_i}$ (green line) requires a pre-invocation of

$p_i = r_i$ (orange line). As A_i increases, the invocation latency bound gets tighter, then the required pre-invocation decreases proportionally. Finally, at $A_i = \frac{1}{p_i - r_i}$ (red line), the slack is large enough so no pre-invocation is needed.

Takeaway. (Efficient Real-time Guarantee for Single Task) Through pre-allocation, RBAM ensures real-time deadlines of a single task with a rate guarantee equal to the task release rate, while keeping overhead bounded by the task execution time.

4.3.2 Efficient Real-time Guarantee for Multiple Tasks

Now, let us generalize the results above to the case of multiple periodic tasks.

Theorem 8. *Pre-invocation enables RBAM to guarantee* <u>*many*</u> *periodic tasks* $T_1, ..., T_n$ *meeting their deadlines without the positive slack requirement (i.e.,* $s_i = p_i - r_i \geq 0$*).*

Proof. Given a task T_i, let us request an invocation for each of its releases in ρ_i time unit ahead, where

$$\rho_i = r_i \tag{4.25}$$

By Theorem 7, this pre-invocation enables T_i to achieve its real-time guarantee with a finite invocation rate

$$A_i = \frac{1}{p_i} \tag{4.26}$$

Applying the same argument for other tasks then all the tasks are guaranteed to meet their deadlines. This proved the theorem. □

Similar to Theorem 5, Theorem 8 extends Theorem 4's scope to tasks with no slack, which finally, proves that RBAM is sufficient for *any* combination of multiple periodic tasks to achieve real-time guarantee.

Next, we generalize theorems on the bound of the cost to multiple tasks. However, representing the cost by a set of guaranteed invocation rates used by each task is complicated, hard to analyze, and make comparisons. Hence, we consider the invocation cost as the invocation rate for the application as a whole. In particular, given multiple tasks $T_1, ..., T_n$ served with invocation rates of $A_1, ..., A_n$ then the invocation cost for these tasks would be the sum of the number of invocations created at rates $A_1, ..., A_n$ per time unit.

Theorem 9. *An RBAM system can meet the deadlines for rate-monotonic workload with periodic tasks $T_1, ..., T_n$ given a guaranteed invocation defined as*

$$A_{total} \geq \sum_{i=1}^{n} A_i = \sum_{i=1}^{n} \frac{1}{p_i} \tag{4.27}$$

Proof. Consider the time interval of length M:

$$M = \prod_{j=1}^{n} p_j \tag{4.28}$$

Clearly, M is a common multiple of $p_1, ..., p_n$ so if we are able to ensure $T_1, ..., T_n$ to meet their deadline within this interval, they are guaranteed to meet deadlines in any interval. Consider a task T_i, let N_i be the number of its releases within the interval, then

$$N_i = \frac{M}{p_i} = \frac{\prod_{j=1}^{n} p_j}{p_i} = \prod_{j \neq i} p_j \tag{4.29}$$

Thus, the total number of task releases is

$$N_{total} = \sum_{i=1}^{n} N_i = \sum_{i=1}^{n} \prod_{j \neq i} p_j \tag{4.30}$$

Clearly, there would be N_{total} invocation requests within the interval so in order to ensure that no task misses its deadline, the invocation rate must be fast enough to make at least N_{total} invocations available. Therefore, the lower bound for the shared guaranteed invocation rate is

$$A_{total} \geq \frac{N_{total}}{M} = \sum_{i=1}^{n} \frac{\prod_{j \neq i} p_j}{M} = \sum_{i=1}^{n} \frac{1}{p_i} = \sum_{i=1}^{n} A_i$$

Thus, the theorem is proved. □

Note that $A_{total} = \sum_{i=1}^{n} A_i$ is just a lower bound for invocation rate, stating how *fast* the FaaS system should deliver their invocations, not *when* should they deliver invocations. In fact, given A_{total}, the FaaS system can even decompose it back to $A_1, ..., A_n$ where $\sum_{i=1}^{n} A_i = A_{total}$ to serve $T_1, ..., T_n$ individually.

Theorem 10. *Invocations delivered at rate A_{total} can be partitioned to form n different invocation rates $A_1, ..., A_n$ where*

$$A_{total} = \sum_{i=1}^{n} A_i$$

Proof. Consider an arbitrary interval of length T, the number of invocations guaranteed to be available within T at rate A_i is

$$N_i = \lfloor T \cdot A_i \rfloor$$

while the number of invocations delivered by A_{total} is

$$N_{total} = \lfloor T \cdot A_{total} \rfloor = \lfloor T \cdot \sum_{i=1}^{n} A_i \rfloor \geq \sum_{i=1}^{n} \lfloor T \cdot A_i \rfloor = \sum_{i=1}^{n} N_i$$

Thus, at any interval, invocations given at rate A_{total} is always greater than or equal to the total number of invocations needed by $A_i, ..., A_n$. Therefore, by temporally shifting invocations created at rate A_{total} within the interval, we can form n guaranteed invocations. This proves the theorem. □

The theorem states that different real-time deadlines can be ensured *simultaneously* with a single guaranteed invocation rate. Combined with Theorem 9, we provide important results that allow us to ensure hard real-time deadlines with great flexibility that can be essential to deal with different practical scenarios. For example, we can multiplex different tasks into a single RBAM function to simplify function management given a large number of periodic tasks. Or if the rate guarantee of a function is too high, making its deployment impracticable, we can decompose it into identical ones with smaller rates. Finally, we use these theorems to realize the lower pre-invocation bound for A_{total}.

Theorem 11. *The lower bound from Theorem 9 for guaranteed invocation rate of*

$$A_{total} = \sum_{i=1}^{n} A_i = \sum_{i=1}^{n} \frac{1}{p_i} \tag{4.31}$$

can be achieved with total pre-invocation overhead (wasted compute) of

$$\rho_{total} = \sum_{i=1}^{n} \rho_i = \sum_{i=1}^{n} r_i \tag{4.32}$$

Proof. By applying Theorem 10, we can decompose A_{total} into

$$A_i = \frac{1}{p_i} \tag{4.33}$$

By using A_i to serve T_i, Theorem 7 indicates that a pre-invocation of

$$\rho_i = r_i \tag{4.34}$$

is needed to achieve its real-time guarantee. Applying the argument for other tasks, then at the invocation rate of

$$A_{total} = \sum_{i=1}^{n} A_i = \sum_{i=1}^{n} \frac{1}{p_i} \tag{4.35}$$

the real-time guarantees are met only with the total pre-invocation of

$$\rho = \sum_{i=1}^{n} \rho_i = \sum_{i=1}^{n} r_i \tag{4.36}$$

This makes the pre-invocation overhead (wasted compute) bounded by $\sum_{i=1}^{n} r_i$. Thus, the theorem is proven. $\square$

Theorem 11 generalizes the conclusions of the Theorem 7, showing that pre-invocation required to ensure no task misses its deadlines will never exceed the real computation cost. In other words, a pre-invocation overhead of 100% is sufficient to reduce the guaranteed invocation rate requirements to their minimum, $A_{total} = \sum_{i=1}^{N} \frac{1}{p_i}$.

Takeaway. (Efficient Real-time Guarantee for Multiple Tasks) Through pre-allocation, RBAM ensures deadlines for multiple tasks, with a rate guarantee equal to their total release rate, while ensuring that the overhead never exceeds their actual resource usage.

4.4 Demonstration

In this section, we will demonstrate the implications of the above theoretical results by using RBAM to serve a distributed real-time virtual/augmented reality (VR/AR) application. We aim to demonstrate how the application design and deployment can be enhanced by utilizing our theoretical findings, leading to improved user experience, simplified deployment, and streamlined management.

4.4.1 Methodology

4.4.1.1 VR/AR Application and Workload

Distributed VR/AR has gained significant attention in recent years due to its innovative user experience. Quality of experience (QoE), measured by the smoothness of user interaction, is one of its critical requirements. Maintaining high QoE requires timely processing of many tasks, such as rendering and synchronizing multi-player actions. We will show that by

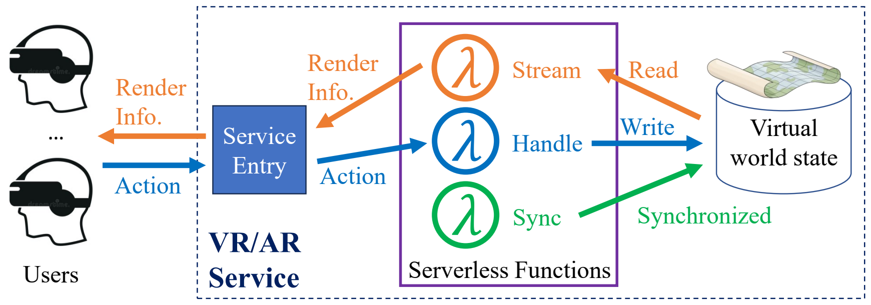


Figure 4.6: The VR/AR service. Requests come from various users creating corresponding serverless function invocations.

specifying these tasks as hard real-time and enforcing their deadlines using RBAM, we can unlock new, game-changing capabilities to these applications. These capabilities allow them to control QoE and simplify their management in ever-changing workloads and deployment environments.

We extend the illustrative example in Section 4.1.2 to model a more complicated VR/AR application implemented as a cloud service that creates a virtual world serving as a common place for hundreds or even thousands of users to interact with each other simultaneously (Figure 4.6). The virtual world state, including users' locations, appearances, and movements, etc., is maintained in global storage and is continuously updated to match users' actions (e.g., talk, move, make a purchase, etc.). We select three representative time-sensitive tasks that are frequently executed by VR/AR applications and model them as a rate-monotonic workload with parameters listed in Table 4.2:

- *Stream*: reads the virtual world state, generates render information, and then sends it to the user's end devices for constructing the world from their point of view. Since 30 fps is a standard for video streaming, we set the task period to 30ms and 15ms execution time.

- *Handle*: triggers when a user takes actions that require an immediate reaction from the application (e.g., talk, pick up an item, etc.). Based on in-game analysis [222],

Task	**Period**	**Runtime**	**Slack**
Stream (T_1)	$p_1 = 30$	$r_1 = 15$	$s_1 = 15$ (50%)
Handle (T_2)	$p_2 = 50$	$r_2 = 25$	$s_2 = 25$ (50%)
Sync (T_3)	$p_3 = 90$	$r_3 = 50$	$s_3 = 40$ (44%)

Table 4.2: Rate monotonic tasks collected from VR/AR applications (milliseconds).

there can be up to 17 clicks per second in aggressive gaming situations, so we set the task period to 50ms with 25ms runtime.

- *Sync*: Synchronizes the virtual world's state, ensuring its consistency across users. We assume the application synchronizes the state once for every 3 video frames, so the task period is 90ms.

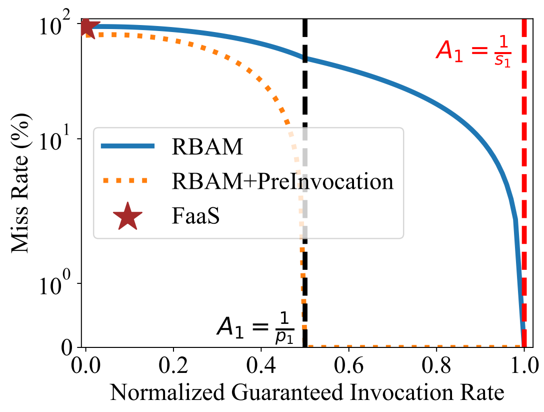
4.4.1.2 Approaches

We compare these task executions on regular FaaS, RBAM, and RBAM with pre-invocation (RBAM+PreInvocation) via simulation. Every time a task is released, it needs an invocation for execution. In the base case, the task requests a new invocation at the beginning of its period. If pre-invocation is enabled, then the task makes invocation requests earlier than the beginning of its periods. If the task requests a regular FaaS invocation, the invocation latency is simulated based on Google Cloud Functions cold start latency statistics [36] (Figure 4.3). For RBAM invocations, we collect the invocation latency statistics from deploying and executing functions with equivalent rate guarantees on a Real-time Serverless – an RBAM prototype (See Section 6.1 and [235]).

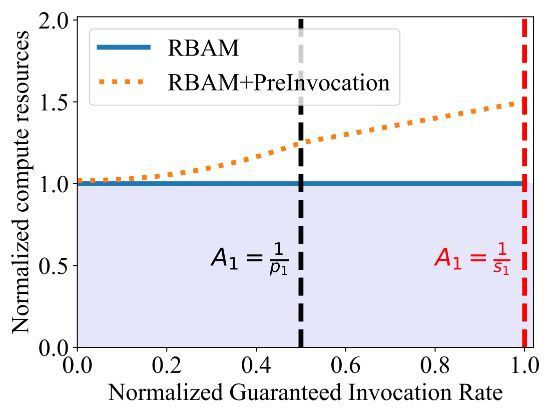
Once a task gets an invocation, it starts the execution with a constant runtime (r_i, Table 4.2). If an execution completes *after* the beginning of the next period, we count it as *missing the deadline*. We report the percentage of executions missing the deadlines (i.e., *miss rate*) as a QoE metric. The *compute resource* (or resource usage), calculated by aggregating the invocation lifetime, including the pre-invocation overhead, is a cost metric.

4.4.2 Experimental Results

4.4.2.1 Validating Theoretical Results



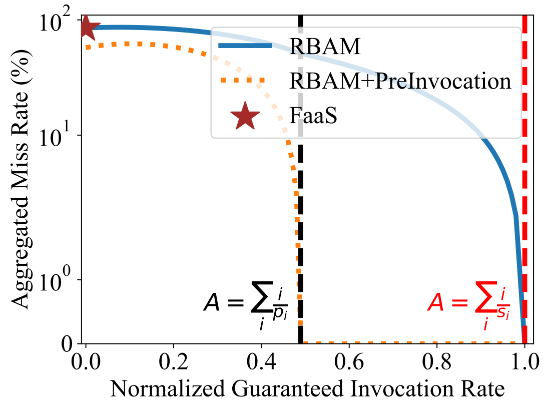
(a) Miss rate vs rate-guarantee (A_1, normalized to $\frac{1}{s_1}$). Pre-invocation: $\rho_1 = p_1 - s_1$



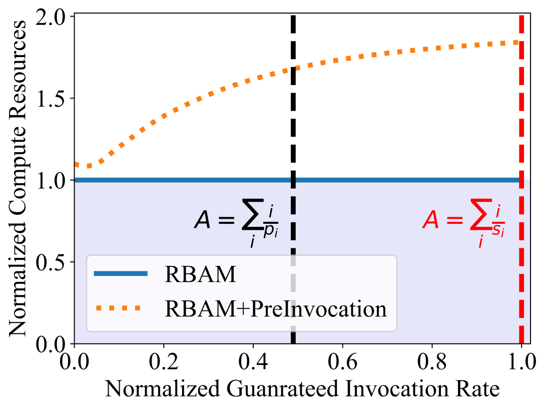
(b) Compute resources normalized to useful ones (colored area) vs. rate-guarantee (A_1, normalized to $\frac{1}{s_1}$)

Figure 4.7: Implementing event streaming (i.e., T_1 – *Stream*) using RBAM ensuring the task's real-time deadlines at bounded resource cost.

Single Task. In Figure 4.7a, we plot the percentage of missed deadlines for a single task "*Stream*" (T_1) implemented by different approaches mentioned above. The regular FaaS leaves invocation latency unbounded so many invocations fail to start within the task's slack – 15ms, leading to more than 85% miss rate (the red star at the top-left corner). RBAM enables bounding the latency through the rate-guarantee A_1 so the higher the rate, the tighter the bound and thus, the lower the miss rate. Once $A_1 = \frac{1}{s_1} = 66.67$ invocations per second, the task is guaranteed to meet its deadlines, validating Theorem 3. With pre-invocation, the task uses pre-invocation of $\rho_1 = r_1 = 15ms$ ahead, equal to the upper bound for efficiency. Doing so adds extra time waiting for the new invocation, so T_1 sees lower deadline misses than RBAM alone (the orange line vs. the blue line). Pre-invocation eliminates the deadline misses at a much lower guaranteed invocation rate, $33.33 = 1/p_1$ invocations per second, confirming Theorem 6. By pre-invocation, however, applications have to hold invocations longer than usual if they arrive before the task is released. This increases resource usage as shown in Figure 4.7b. Further, as the invocation rate increases, invocation arrives faster and thus, creates more overhead yet it never exceeds 100% of the total runtime, as shown in Theorem 7.



(a) Miss rate vs rate-guarantee (A, normalized to $\sum_i \frac{1}{s_i}$). Pre-invocation: $\rho_i = p_i - s_i$



(b) Compute resources normalized to useful computation (colored area) vs. Rate-guaranteed (A) normalized to $\sum_i \frac{1}{s_i}$

Figure 4.8: Serving combinations of multi-tasks. RBAM ensures real-time guarantee for any task combination with a finite rate-guarantee at a resource cost bounded by 2× of useful computation.

Multiple Tasks. Next, we consider deploying all tasks in Table 4.2 simultaneously. The theorem 10 indicates that we do not need to deploy each task with a separate RBAM function. Instead, only one RBAM function with a rate-guarantee equal to the total per-task rate requirement is sufficient. Figure 4.8a confirms this implication as at a guaranteed invocation rate of $A = \sum_i \frac{1}{s_i} \approx 132$ invocation/sec, a single RBAM function reduces all three tasks' aggregated miss rate to zero (blue curve) – ensuring all real-time deadlines are met. Further, Theorem 11 implies that we can even ensure real-time deadlines with an even lower rate-guarantee through pre-invocation. Every time a task T_i releases, we pre-invoke a new invocation r_i seconds ahead. As a result, resources become available for the task sooner, reducing the miss rate (the orange curve). And as proved in Theorem 11, $A = \sum_i \frac{1}{p_i} \approx 64$

invocations per second is already enough for guaranteeing real-time deadlines. Also similar to the case of a single task, pre-invocation incurs high resource use, but by Theorem 11, the overhead never exceeds 100% of useful resources (light blue area) as shown in Figure 4.8b.

Takeaway. (Experimental Validation) The experimental results validate our theoretical conclusions, confirming that RBAM's guaranteed invocation rate ensures deadlines for any rate-monotonic real-time workload.

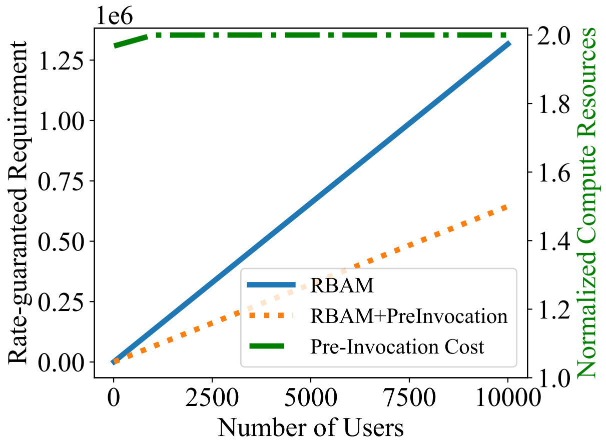


Figure 4.9: Rate-guarantee and resources requirement (normalized to useful computation) varying number of active users.

4.4.2.2 Distributed Deployment

Finally, we consider the distributed deployment of the application where there can be thousands of users simultaneously interacting at a time. Yet the number of active users may vary widely, especially during special events, such as launch time, anniversary, etc., applications expected to obtain a burst load of 10x or more active users than the average [234]. In traditional deployments, this requires careful preparation to make just enough room for the burst to ensure the desired QoE at a reasonable cost.

With RBAM, the solution is much simpler. All the application has to do is simply reconfigure the guaranteed invocation rate to match the task release rate at burst, as demonstrated in the previous experiments. Figure 4.9 shows the guaranteed invocation rate required for a single RBAM function to meet the real-time deadlines of tasks released by different numbers of users, ranging from 0 to more than 8,000. Since guaranteed invocation rates are combinable, the required rate-guarantee increases linearly with the number of active users demonstrating good scalability. Further, with pre-invocation, we can reduce the rate by half at the additional resource uses of at most 100% of the available resources.

It's worth noting that the rate-guarantee is not only combinable but can also be decomposed into smaller ones if needed. For instance, to serve 8,000 users, the required rate-guarantee is over 1 million invocations per second, which may exceed the current capability of FaaS technologies. The application can decompose the FaaS function into others with lower rate-guarantees and distribute them across different cloud regions (e.g., 1000 functions with 1000 invocation/sec, each serving users from a specific area across the globe). This approach not only works around current technology limitations but also leverages distributed resources deployment (e.g., cloud+edge) to achieve better load balancing and cost efficiency.

Takeaway. (Distributed Deployment) RBAM's guaranteed invocation rates can be combined or decomposed without losing real-time guarantee capability, making application deployment simpler and more flexible.

4.5 Summary

In this chapter, we constructed an analytical framework based on the rate-monotonic workload to analyze the real-time performance of real-time applications on the FaaS dynamic execution model. Our theoretical analysis shows that regular FaaS has unbounded invocation latency, preventing real-time applications from guaranteeing their real-time deadlines.

On the other hand, RBAM's guaranteed invocation rate helps the applications bound their invocation latency, ensuring *any* real-time deadline for *any* workload. This is done with a finite rate-guarantee configuration, as minimal as the task release rate, with an overhead bounded by the actual resource usage of the application.

We validate the theoretical results via an experimental evaluation based on practical distributed VR/AR application deployment. The results confirmed our theoretical findings and highlighted the great flexibility of RBAM's rate guarantee configuration. This flexibility provides a variety of deployment and rate configurations for applications to select for their real-time performance and deployment objectives.

To conclude, let us revisit the research questions posed in Section 3.1 and use our analysis results to answer them as follows.

Real-time Guarantee Capability. Can RBAM ensure real-time deadlines for all real-time applications?

Q.1.1 Can RBAM's guarantee invocation rate enable applications to meet their real-time deadlines?

- **Answer: Yes**, we theoretically prove it using Theorem 8 and have it confirmed by experimental results in Section 4.4.2

Q.1.2 Can we show how to derive bounds on RBAM invocation rates for any set of real-time workloads?

- **Answer: Yes**, by Theorem 9, we prove that we can guarantee the real-time deadlines of any real-time application with a minimum rate-guarantee equal to the total release rate of the application. And by Theorem 11, we further prove that the guarantee can be achieved at an overhead bounded by the application resource usage. Both theorems and their application also validated and demonstrated by experiments in Section 4.4.2

With these answers, we complete illustrating the RBAM real-time guarantee capability.

CHAPTER 5

EFFICIENT RBAM IMPLEMENTATION

In this chapter, we will show that RBAM can be implemented efficiently. First, we prove that RBAM implementation is feasible by showing that we can realize any *finite* guaranteed invocation rate with a bounded quantity of resources (Section 5.1). Next, in Section 5.2, we present a statistical study on FaaS workloads and sandbox allocation latency distribution, showing that efficiently implementing RBAM is challenging but not impossible. In Sections 5.3 5.4, 5.5, and 5.6, we show how three new techniques: overallocation, multiple tries, and resource sharing, can be integrated into a brand-new RBAM architecture. This architecture achieves scalable, efficient RBAM deployment. Section 5.7 systematically evaluates the RBAM system to demonstrate its efficiency, robustness, and scalability.

5.1 Implementation Feasibility

Figure 5.1 shows a performance abstraction implementation based on the conventional FaaS architecture. The performance abstraction allows applications to declare their desired performance through predefined SLAs (e.g., guaranteed invocation rates in RBAM). These SLAs serve as input parameters for resource management to allocate sandboxes from cloud resources. The scheduler maps invocation requests to these sandboxes, ensuring SLA-compliant

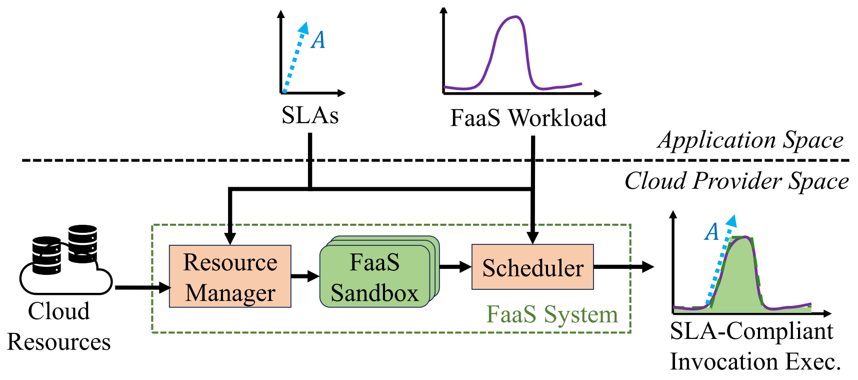


Figure 5.1: FaaS Performance Abstraction Implementation

execution. For the case of RBAM, the performance SLA is per-FaaS guaranteed invocation rate. We will prove that it is feasible to enforce the FaaS resource management and scheduling to generate invocation executions that comply *any* finite guaranteed invocation rate. This is because the execution time of FaaS invocations is limited by a timeout requirement. Even if a function ramps up very quickly and runs for a long time, its concurrency will stop growing when invocations start reaching the timeout and get terminated, limiting the function concurrency.

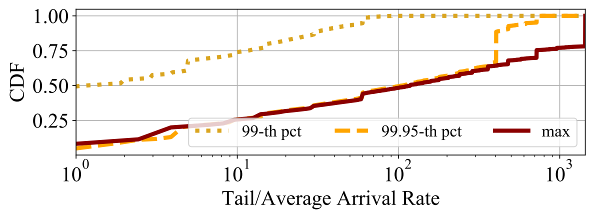
In the case of RBAM, the guaranteed invocation rate is the minimum ramp-up rate. However, the cloud service provider is only responsible for ramping up function invocations to meet this guaranteed rate. Although it would be beneficial if the cloud could ramp up faster, it is not necessary. Thus, the cloud providers can throttle its ramp-up to the guaranteed rate, allowing them to bound the RBAM implementation cost, as shown in the following theorem.

Theorem 12. *Given a FaaS function f_i associated with a finite guaranteed invocation rate A_i and maximum execution time E_i. Let $c_i(t)$ be the number of invocations guaranteed by A_i (i.e., excessive invocations that arrive faster than the rate-guarantee are not counted) that are still under execution at a time t, then*

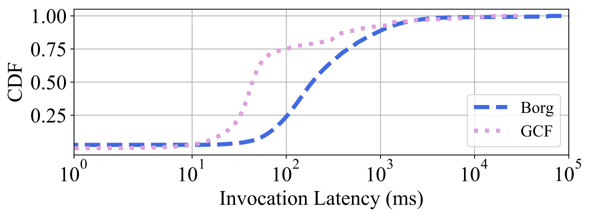
$$\forall t : c_i(t) \leq \lceil A_i E_i \rceil \tag{5.1}$$

Proof. Consider an arbitrary time t. By the definition of E_i, all invocation started before $t - E_i$ must be completed by t. Thus, $c_i(t)$ is bounded by the number of invocations guaranteed by A_i that started between $t - E_i$ and t. By definition of A_i

$$\begin{aligned} c_i(t) &\leq \lceil A_i[t - (t - E_i)] \rceil \\ &= \lceil A_i E_i \rceil \end{aligned}$$



(a) "Tail" arrival rates vs. average rate [279]



(b) Real-world invocation latency distributions [36, 304]

Figure 5.2: Implementing RBAM rate-guarantee is challenging: (a) static pre-allocation is costly while (b) naive dynamic allocation is inefficient due to the unbounded, widely distributed sandbox allocation latency.

This proves the theorem. □

By Theorem 12, the cloud provider can implement any function f_i with finite guaranteed invocation rate A_i and maximum execution time E_i with a straightforward **pre-allocation** strategy: First, it allocates resources that are sufficient to handle $\lceil A_i E_i \rceil$ concurrent invocations to f_i for its whole lifetime. Second, whenever the function receives an invocation request, throttle the request execution rate to A_i. In that way, the cloud always ensures sufficient resources to meet the function's rate guarantee requirements, thereby proving the feasibility of RBAM implementation.

Takeaway. (RBAM Implementation Feasibility) Any RBAM function with finite guarantee invocation rate A_i and maximum execution time E_i can be implemented with the resource cost bounded by $\lceil A_i E_i \rceil$.

5.2 Efficient Implementation Challenges

Because the performance abstraction is implemented in the cloud, it must also meet the cloud's operational goals. This dissertation focuses on minimizing resource costs (or overhead), aiming to align sandbox allocations closely with actual usage. Since resource allocation and scheduling decisions depend on FaaS workload and cloud resources, achieving this goal requires a thorough understanding of their characteristics.

High Workload Variability Recent studies reveal that FaaS workload is highly variable [278, 346]. For example, Figure 5.2a shows the ratio of the tail arrival rates to the average arrival rate of Azure Functions [278]. Most functions (75%) have their peak (i.e., "max") rate at least 10x higher than the average. One-third of these even witnessed a peak rate exceeding 1000x. The ratio between the average and other tail latency is less extreme but still very high. At the 99.95th percentile, more than 25% of FaaS functions have at least a 400x ratio. This number reduces significantly for the 99th percentile but is still as high as 10x.

Complicated Cloud Shared Environment Cloud resources are distributed across data centers with high heterogeneity and are shared across multi-tenants with various behaviors. Consequently, allocating sandboxes from these resources has highly variable and unpredictable outcomes.

- **Long-tail, Unbounded Allocation Latency.** Figure 5.2b illustrates the allocation latency distribution of a *single sandbox* estimated from the Google Borg VM allocation latency [304] and Google Cloud Functions cold start [36]. Both distributions exhibit a heavy tail, with the 99.95th percentile latency reaching about 30s, 50x longer than the median. Worse, allocation requests are subjected to cancellation [323, 304], making the latency practically unbounded.

- **Allocation Correlation.** The shared environment makes cloud events highly correlated. For example, cloud management systems such as Borg [304, 313] and Kubernetes [41] tend to provision sandboxes through shareable resource reservations. Sandboxes within the same reservation experience similar allocation latencies. Thus, if a bad cloud event happens (e.g., slow reservation creation), it can spread across multiple allocations, exacerbating its negative impact.

FaaS systems have to effectively manage the impact of workload variability and cloud resource uncertainty to produce SLA-compliance invocation executions. Current mainstream solutions address these issues by allocating more resources than needed [235, 65], using extra resources to absorb unforeseen spikes in workload demand, and hiding the latency of sandbox allocations affected by unexpected bad cloud events (see Section 7.2). The amount of extra resources depends on the cloud's desired SLA reliability, which is driven by a guarantee availability defined below.

Definition 3 (Guarantee Availability)**.** *A Performance SLA is enforced with a* ***guarantee availability*** X $(0 \leq X \leq 1)$ *if there is at least an* X *probability for each invocation request to comply with the SLA.*

For example, conventional FaaS systems follow a guarantee availability of 99.95% [31, 42, 29], meaning up to 99.95% of invocation requests are guaranteed to execute successfully.

The pre-allocation approach proposed in Section 5.1 assumes invocation requests always arrive at a rate equal to the guaranteed rate and allocate resources for this rate in advance at deployment time. In practice, however, the actual invocation arrival rate varies widely, as we have shown previously in Figure 5.2a. An application that wants to use RBAM to ensure the performance of their bursty periods will need to set A equal to their maximum arrival rate, incurring the required sandbox allocation up to three orders of magnitude higher than the average use, which is far from acceptable in practice. FaaS systems can reduce costs by relaxing guarantee availability, yet the return is not worth the trade-off. For example, they can support the SLAs up to the 99.95-th percentile arrival rate, willing to sacrifice 0.05% guarantee availability, matching the current cloud availability SLA but costs at least 100x for half of the functions (Figure 5.2a).

Another alternative is dynamic (de)allocation. The FaaS system dynamically adjusts the sandboxes allocated to a FaaS function according to the instantaneous arrival rate. The approach, however, depends on resource allocation latency, which is practically unbounded and widely distributed. Figure 5.2b shows the sandbox allocation latency distribution estimated from the Google Borg VM allocation latency [304] and Google Cloud Functions cold start [36]. Both distributions have a heavy tail where the 99.95-th percentile latency is more than 100s, which is around 600x compared to the median. Thus, the FaaS system must allocate resources for the invocation 100s in advance to ensure 99.95% of the invocations meet the rate guarantee. In contrast, the invocation execution time is typically less than one second [278], meaning the resource cost is potentially 100x compared to actual use, as expensive as the pre-allocation. Worse, the allocation requests are subjected to cancellation [323, 304], making the latency potentially unbounded and impossible for a FaaS system to always ensure rate guarantee with dynamic allocation.

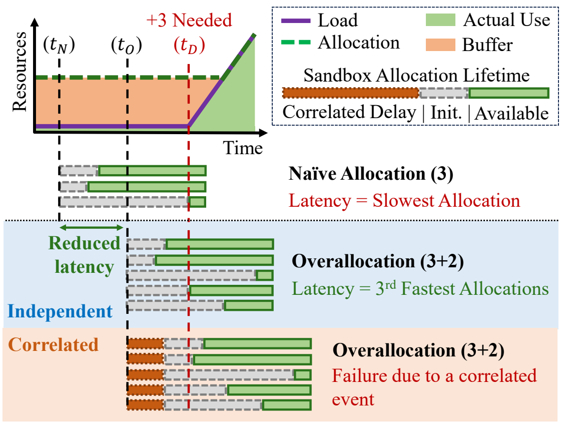
The main issue of both pre-allocation and naive dynamic allocation is that they use extra sandbox resources to handle workload variability (e.g., unforeseen spikes) and cloud resource uncertainty (e.g., slow allocations affected by unexpected bad cloud events). However, because both workload variability and cloud uncertainty are widespread, the required overhead to make the approaches practical is huge. Even if we relax the guarantee availability to $X = 99\%$, which is already 20x less strict than the current commercial cloud standard – 99.95%, both approaches incur an unacceptable cost of 10x to 30x higher than the actual use. To resolve the problem, we propose an efficient guarantee-cost management solution that allows the FaaS system to implement RBAM guaranteed invocation rates with extremely high availability (i.e., 99.95% and beyond) at reasonable resource cost. The solution includes a new FaaS architecture (Section 5.4) that enables our new rate-based scheduling (Section 5.5) and resource management (Section 5.6) algorithms to efficiently exploit the

RBAM configurations and sandbox allocation statistics to enforce any set of deployed rate guarantees.

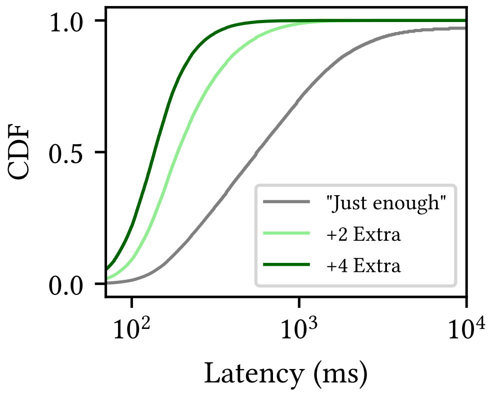
Takeaway. (RBAM Implementation Challenges) Efficiently implementing RBAM is challenging. Current approaches are too expensive, even when availability trade-offs are made to reduce costs.

5.3 RBAM Implementation Ideas

The main challenge in RBAM implementation is the huge number of sandboxes to be prepared in advance due to high workload variability and long-tail sandbox allocation latency distribution. Our approach is to reduce the latency and variability of allocation statistically by using three strategies: (i) overallocation, (ii) multiple tries, and (iii) sandbox sharing.



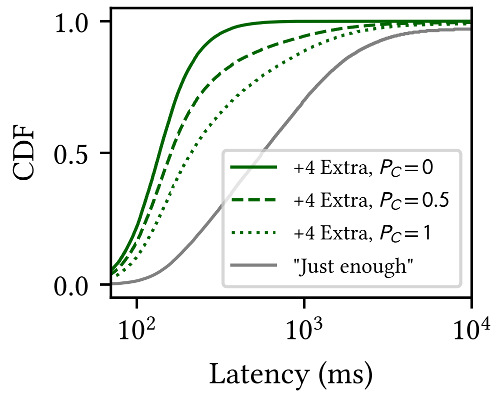
(a) Overallocation Example



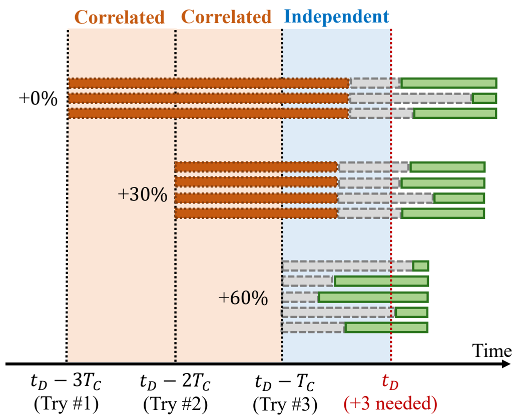
(b) Overallocation vs. Naive Allocation

Figure 5.3: Overallocation statistically shortens allocation latency

Over-allocation. We shorten the long-tail allocation latency statistically by proactively sending more sandbox allocation requests than needed. In Figure 5.3a, for example, suppose that we want three more sandboxes at t_D (red line). If we naively allocate exactly 3 sandboxes, the overall latency is determined by the slowest one. But if we allocate two extra sandboxes (five in total), the latency is actually the third fastest one, which is much shorter than the naive one. Figure 5.3b shows the latency distribution of the two approaches. The overallocation (light green) latency is much shorter than the naive one (gray curve). Adding more allocations (e.g., ten, dark green line) reduces the latency further. With shorter allocation latency, we can allocate sandboxes at a closer time to when they are needed, wasting fewer resources.



(a) The benefit of overallocation diminishes with the emergence of correlation.

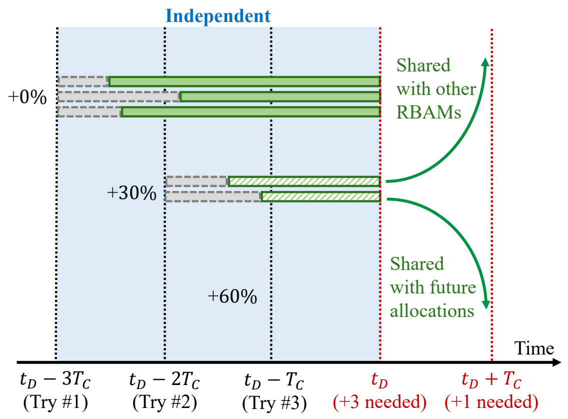


(b) Multiple Tries Example

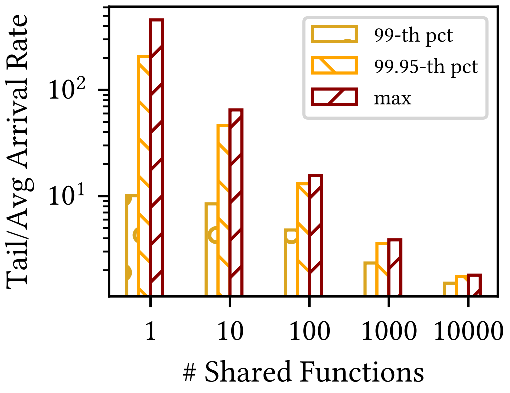
Figure 5.4: Try to overallocation multiple times to reduce the impact of badly correlated events.

Multiple Tries. Correlation between allocation requests hurts overallocation effectiveness. As in the bottom of Figure 5.3a, we assume all five allocations are correlated to a bad cloud event (e.g., slow reservation generation), prolonging all of them (red bars) and eliminating the benefit of overallocation. Fortunately, these incidents last for a short period of time [281, 264]. To work around this, we avoid allocating all sandboxes simultaneously but spread them across different points in time. For example, suppose we need 3 sandboxes at a future time t_D. Instead of allocating five sandboxes only once, we can do it thrice: five sandboxes at $t_D - T_C$, four at $t - 2T_C$, and three at $t - 3T_C$ (Figure 5.4b) where T_C is the minimum distance between two allocations that make them independent. Note that the further from

t_D, the more time we get to wait for allocations, so we need fewer extra allocations to achieve the same result (+60%, +30%, and 0% at $t_D - T_C$, $t_D - 2T_C$ and $t_D - 3T_C$, respectively). In this way, we temporally spread the risk of failing to allocate sandboxes in time. Thus, only one successful try is sufficient to enforce defined SLAs, making the implementation robust against the complicated cloud environment.



(a) Sandbox Sharing Example



(b) Sharing sandboxes reduces workload variability

Figure 5.5: Sharing sandboxes across functions greatly reduces extra resources incurred by multiple tries of overallocation and workload variability

Sandbox Sharing. Overallocation and multiple tries potentially leave many unused extra sandboxes. For example, in Figure 5.4b, we have to allocate 12 sandboxes while only needing 3, leaving 9 sandboxes unused, 3x overhead. To reduce this cost, we share unused sandboxes across (i) time and (ii) functions. Specifically, we consider extra sandboxes of past tries as discounts to reduce the number of required allocations in future tries. For example, in Figure 5.5a, just before the second try, we already have two sandbox allocations from the first try completed. Thus, we only need one more, then at the overallocation factor of 30%, we just need to allocate two more sandboxes instead of four as in Figure 5.4b. Similarly, on the third try, the two additional sandboxes and the leftover from the first try were completed, so we have a surplus of two sandboxes, meeting the requirement at t_D. Thus, no allocation is required for the third try. We further share the one surplus sandbox with the future time, $t_D + T_C$, so no more allocation is needed in the future. We also share sandboxes across functions. This is possible by enabling re-purposing a warm sandbox of a function for another [192, 153, 325] and using zygote sandboxes [238, 191] to create shareable isolation environments across functions. Doing so creates more chances to exploit surplus sandboxes and helps reduce workload variability. In Figure 5.5b, we randomly combine Azure functions into groups of 1, 10, 100, 1k, and 10k functions. Then, we calculate the aggregate arrival rates of each group and report the ratio of tail arrival rates over the average one in the figure. Since bursty requests are unlikely to appear concurrently among functions. The ratio decreases significantly as we group more functions. This indicates that grouping functions together greatly reduces their aggregate variability, making invocation demand less fluctuating. As a result, serving them requires fewer dynamic sandbox allocations. Consequently, fewer extra sandboxes are needed, reducing overhead.

Takeaway. (RBAM Implementation Ideas) We can implement RBAM efficiently by statistically shortening allocation latency with multiple tries of overallocation and reducing workload variability with resource sharing.

5.4 RBAM System Architecture

The new FaaS Architecture for efficient RBAM implementation (i.e., RBAM system) is shown in Figure 5.6. The architecture inherits all existing components in the conventional FaaS system (See Figure 2.3 in Section 2.2.2) and adds three extensions:

- *Rate-based Scheduler (RS)*: find the *right time and place* to execute invocations, based on RBAM configurations.

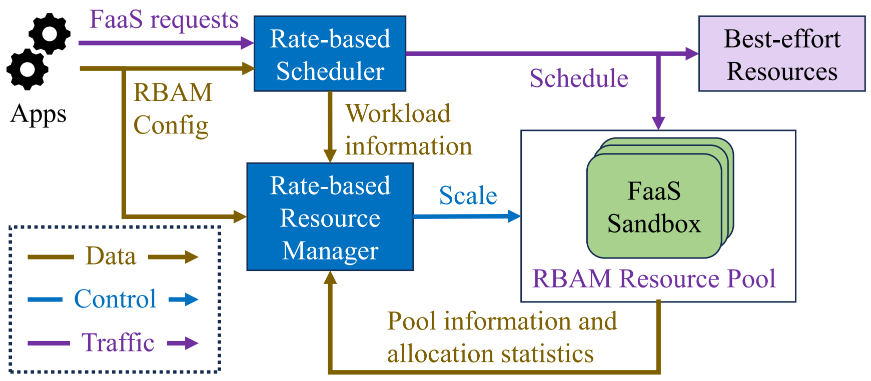


Figure 5.6: RBAM System Architecture: Invocation executions are driven rate-based resource manager and scheduler.

Symbol	Name/Definition	Unit	Notes
f_i	RBAM function	N/A	
A_i	Rate-guarantee	invocation/second	
E_i	Maximum execution time	second	
X	Guarantee Availability Target	N/A	$0 \leq X \leq 1$

Table 5.1: RBAM Configurations Notations

- *RBAM Resource Pool*: contain shareable sandboxes used for enforcing guaranteed invocation rates.
- *Rate-based Resource Manager (RRM)*: manage the RBAM resource pool through scaling up and down decisions driven by RBAM configurations (defined below).

The critical difference between the RBAM system and its conventional counterpart (presented in Section 2.2.2) is the addition of the RBAM resource pool managed based on RBAM configurations provided by the applications, making the system decisions align with its applications' desired performance.

RBAM Configurations. The RBAM configurations consist of performance-related parameters of all serverless functions $f_1, ...f_N$ deployed by the FaaS system (See Table 5.1). Each function, f_i, has two parameters

- E_i: Maximum invocation execution time (i.e., timeout).

- A_i: Guaranteed invocation rate.

RBAM configurations also have a *guarantee availability target* X (defined in Definition 3) given by the cloud provider, indicating the minimum probability of meeting *all* rate-guarantees $A_1, ..., A_N$ for every invocation request. The guarantee availability target X allows the cloud provider to balance their resource cost (by choosing small X) and service-level agreement (SLA) with their customers (by choosing large X). We summarize the notations in Table 5.1

Rate-guarantee Enforcement. The main objective of the proposed RBAM architecture is to meet *all* rate-guarantee requirements $A_1, ..., A_N$ with *any* given availability target X at *low* resource cost. This is accomplished by enforcing:

- The rate-based scheduler to start at least one invocation, if any, in every $\frac{1}{A_i}$ interval for all f_i.
- The Rate-based resource manager to ensure the sandboxes in the RBAM resource pool are sufficient to *immediately* start any invocation once it is scheduled in the pool.

Once the two components fulfill their tasks, any function f_i with rate-guarantee A_i is ensured to get at least one invocation to start immediately within any $\frac{1}{A_i}$ interval, thereby meeting its guarantee requirement. We will discuss in more detail how the scheduler and resource manager accomplish their task in the next two sections.

5.5 Efficient RBAM Scheduling

Algorithm 1 presents a rate-based invocation scheduling algorithm that we use to implement the rate-based scheduler in the RBAM architecture (See Figure 5.6).

First, let us describe how the algorithm ensures the invocation scheduling rate required by the rate guarantees $A_1, ..., A_N$. For each function f_i, the scheduler keeps track of the

Algorithm 1 RBAM Scheduling

```
 1: for each $f_i$ do
 2:     $last[f_i] \leftarrow -\infty$
 3: end for
 4: while true do
 5:     for each $f_i$ do
 6:         $r \leftarrow getInvocationRequest()$
 7:         if $r \neq$ null then
 8:             $t \leftarrow arrivalTime(r)$
 9:             if $B(t) > 0$ and $t - last[f_i] \geq \frac{1}{A_i}$ then
10:                 $RBAMSharedPool.exec(r)$
11:                 $last[f_i] \leftarrow currentTime()$
12:             else
13:                 $BestEffortPool.exec(r)$
14:             end if
15:         end if
16:     end for
17: end while
```

start time of the last invocation scheduled in the RBAM pool ($last[f_i]$). Every time an invocation request arrives (line #7), if its arrival time t is at least $\frac{1}{A_i}$ seconds behind $last[f_i]$ (i.e., $t - last[f_i] \geq \frac{1}{A_i}$, Line #9), the request will be scheduled immediately in the RBAM resource pool (Line #10). The mechanism ensures that as long as the invocation request rate is less than or equal to A_i, they will be scheduled at the rate guarantee A_i, and if the resource manager does its job well, they will start execution at the same rate, meet the rate-guarantee requirement.

If, however, the invocation request rate is higher than A_i or the resource manager fails to prepare sandboxes in time (i,e., the number of free sandboxes in the RBAM pool $B(t) = 0$ at request arrival time t, Line #9) then the scheduler will fail over to the best-effort mode, forwarding all invocation requests to the best-effort resources (Line #13), for two reasons. First, this limits the schedule rate on the RBAM pool by $\sum_{i=1}^{N} A_i$, making it easier to manage the pool while still enabling the excessive requests to execute at a standard FaaS quality. Second, redirecting requests out of the RBAM pool when it fails to offer sandboxes in time avoids creating a backlog in the pool that could exaggerate the situation and prolong the pool recovery.

Takeaway. (RBAM Scheduling) The rate-based scheduler uses RBAM resources to handle requests up to the rate guarantee, ensuring minimum rate guarantee requirements. Excessive requests are served using best-effort resources, achieving a quality equivalent to the current cloud standard.

5.6 Efficient RBAM Resource Management

Next, we will explain how the Rate-based Resource Manager (RRM) ensures immediate invocation execution of scheduled requests in the RBAM pool given a guarantee availability X. We begin with the high-level implementation ideas presented in Section 5.3. Following this, we formally derive a smart dynamic algorithm and provide theoretical proof of its rate guarantee capability and efficiency analysis (Sections 5.6.1 and 5.6.2).

5.6.1 Sandbox Allocation Modelling

Because sandbox allocation statistics are crucial, let us first model the sandbox allocation and then use it to realize and analyze RRM implementation later in Section 5.6.2. Specifically, when the RRM allocates sandboxes, their latency is governed by the following factors.

- *Single sandbox allocation latency* is a random variable ($\mathcal{P}$) so $Pr[\mathcal{P} \leq \Delta t]$ is the probability of successfully completing a *single* sandbox allocation in Δt seconds. (assuming the allocation is *independent* from other).
- *Allocation Correlation*: sandbox allocations may have their latency correlated if they share a common path (e.g., get resources from the same physical machine) during their allocation processes. We assume the common path sharing is temporary and lasts for

at most T_C seconds (e.g., two allocations are correlated only if they are at most T_C seconds apart). We call T_C *correlation period* and model allocation correlation by dividing the time into fixed slots of length T_C, each having a *correlation probability* P_C. If a sandbox is allocated in a correlated slot, its latency is at least equal to the first sandbox allocation latency of that period. Otherwise, its latency will follow $\mathcal{P}$.

The RRM, however, does not need to know these factors or their actual values. It just needs to know their *impact* on the sandbox allocation latency to make proper decisions. This is done by monitoring an empirical distribution $P^c_{\Delta t}(a)$ representing the probability of obtaining **at least** c sandboxes in Δt second(s) after sending a allocation requests. For example, $P^3_1(5) = 90\%$ means allocating 5 sandboxes will give us at least 3 in 1 second with a 90% probability. We assume $P^c_t(a)$ is *monotonic* with the following characteristics

- $P^c_{\Delta t}(a)$ is a non-decreasing function of a (i.e., More allocation increases chances of obtaining desired sandboxes).

$$\forall a_1 \leq a_2 \leq M : P^c_{\Delta t}(a_1) \leq P^c_{\Delta t}(a_2) \tag{5.2}$$

- $P^c_{\Delta t}(a)$ is a non-increasing function of c (i.e., overallocation is less effective if we desire more without allocating more).

$$\forall c_1 \leq c_2 \leq a \leq M : P^{c_1}_{\Delta t}(a) \geq P^{c_2}_{\Delta t}(a) \tag{5.3}$$

- $P^c_{\Delta t}(a)$ is a non-decreasing function of Δt (i.e., the sooner allocating sandboxes, the better chance of timely delivery).

$$\forall \Delta t_1 \leq \Delta t_2 : P^c_{\Delta t_1}(a) \leq P^c_{\Delta t_2}(a) \tag{5.4}$$

Symbol	**Name/Definition**	**Notes**
$\mathcal{P}$	Single sandbox allocation latency	
T_C	Length of correlation Period	Unit: sec
P_C	Correlation probability	$0 \leq P_C \leq 1$
M	Allocation limit in a single time slot of length T_C	
$P^c_{\Delta t}(a)$	Probability of getting c/a sandboxes in Δt sec	

Table 5.2: Sandbox Allocation Notations

where M is the maximum allocation requests the underlying sandbox allocator can handle in each T_C interval without losing the above monotonic properties. All notations are summarized in Table 5.2.

5.6.2 Smart Dynamic Allocation Algorithm

The RRM aims to allocate just enough sandbox resources to ensure the guaranteed invocation rate up to a guarantee availability target X. In other words, this is equivalent to solving the following optimization problem:

$$\begin{aligned} \forall t: \quad & \min B(t) \\ \text{s.t.} \quad & Pr[u(t) \geq 0] \geq X \end{aligned} \tag{5.5}$$

where

- $u(t)$ is the difference between the RBAM pool capacity and the sandboxes required by FaaS functions to meet their guaranteed invocation rate at a time t. Thus, $u(t) \geq 0$ indicates that the RBAM pool has sufficient sandboxes to meet the demand of guaranteed invocation rates at time t. This means $Pr[u(t) \geq 0]$ indicates the probability for *all* invocation requests arrived at t get executed immediately to comply with their rate-guarantee so enforcing $Pr[u(t) \geq 0] \geq X$ for all t will ensure provide rate guarantees with a guarantee availability X, satisfying RRM's objective.

- $B(t) = \max[u(t), 0]$ is the number of free sandboxes at a time t, which is also the resource overhead.

To achieve this goal, we propose a smart dynamic allocation algorithm (Algorithm 2) that employs three techniques: overallocation, multiple tries, and sandbox sharing, as discussed in Section 5.3.

Algorithm 2 Smart Dynamic Allocation (Simplified)

```
1:  $L \leftarrow numTriesEst(X)$
2:  Prealloc($\sum_{i=1}^{N} \lceil A_i \min(E_i, T_C L) \rceil$)
3:  for every $T_C$ seconds do
4:      $t_k \leftarrow currentTime()$
5:      $a \leftarrow scaleUpEst(t_k, L, X)$
6:      if $a > 0$ then
7:          Alloc(a)
8:      else if $t > \Delta t_L$ then
9:          $d \leftarrow scaleDownEst(t_k)$
10:         if $d > 0$ then
11:             Dealloc(d)
12:         end if
13:     end if
14: end for
```

The algorithm includes three parts:

- **Initialization.** (Lines #1 and #2) RRM estimates the number of multiple tries L for the multiple tries strategy (explained later). Multi-try scaling takes at least LT_C seconds to be fully effective. Thus, the algorithm pre-allocates the RBAM pool with $\sum_{i=1}^{N} \lceil A_i \min(E_i, \Delta t_L) \rceil$) sandboxes (Line #2) fully covering scheduled requests in the first LT_C seconds, giving enough time for the dynamic scaling to warm up and maintain the guarantee for the rest of the time.

- **Scaling up.** (Lines #5 to #7) The RRM scales up the RBAM pool when it needs additional sandboxes to *enforce guaranteed invocation rates with availability* X over the time (i.e., $\forall t : Pr[u(t) \geq 0] \geq X$). This is done by periodically trying to scale the RBAM pool in every T_C seconds. The T_C-second gap between tries ensures allocation requests sent by scaling decisions at different times are independent of each other, avoiding the allocation correlation effect. For each scaling decision at a time t_k, the RRM relies on RBAM configurations and the underlying sandbox allocation latency statistics to estimate the number of sandbox allocations needed in the next LT_C seconds, subject to the guarantee availability objective (e.g., $\forall t_k \leq t \leq t_k + LT_C : Pr[u(t) \geq 0] \geq X$, Line #5, the *scaleUpEst* function described later in Section 5.6.2.1). Then, the RRM sends corresponding sandbox allocation requests (Line #7) to the underlying cloud allocator.

- **Scaling down.** (Lines #8 to #11) If no additional sandboxes are needed (Line #8), then the RBAM pool must already have sufficient sandboxes to meet the guarantee availability in the next LT_C seconds. The RRM then checks for the surplus sandboxes (Line #9, the *scaleDownEst* function described later in Section 5.6.2.2). If they are more than enough (Line #10), it will deallocate them to minimize the overhead (Line #11).

In the rest of this section, we will present how the algorithm estimates the number of sandboxes to be allocated (i.e., the implementation of the $scaleUpEst(t_k)$ function in Line #5) and (de)allocated (i.e., the $scaleDownEst(t_k)$ function in Line #9) in more detail.

5.6.2.1 Scaling Up

The main objective of scaling up is to ensure there are always sufficient sandboxes in the RBAM pool to meet the rate guarantees up to a guarantee availability of X. In other words,

$$\forall t : Pr[u(t) \geq 0] \geq X \tag{5.6}$$

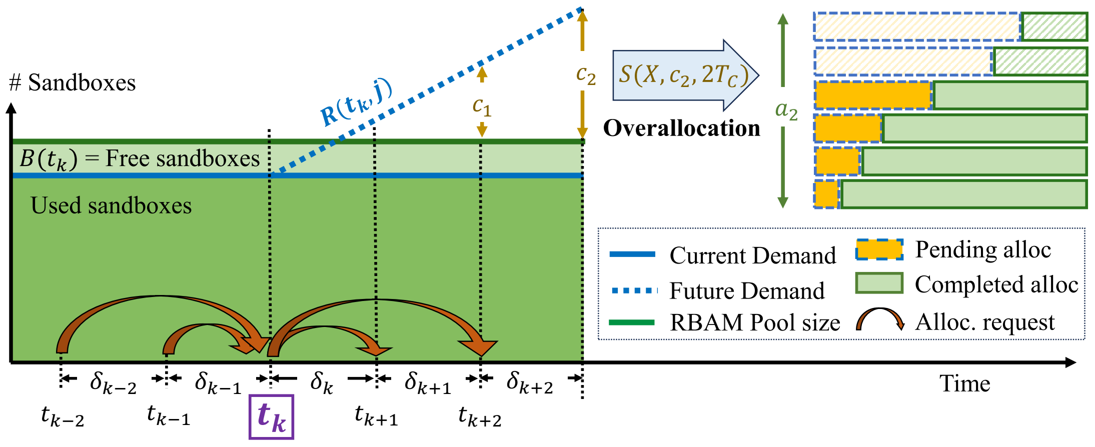


Figure 5.7: Overallocation and Multiple Tries ($L = 2$) Visualization

To achieve this objective, RRM divides the time into fixed slots $\delta_0, \delta_1, ...$ of length T_C and periodically makes scaling decisions at each slot start time $t_k = kT_C$. This ensures separate scaling decisions are not affected by the same correlation factor and thus have independent latency.

Recall that L is the number of multiple tries. At a scaling time t_k (See Figure 5.7), the RRM presumes past decisions at $t_{k-1}, ..., t_{k-L}$ already ensured scaling objective of the current slot (i.e., the Equation 5.6 holds $\forall t' \in \delta_k$), so it just needs to allocate sandboxes for future slots: $\delta_{k+1}, ..., \delta_{k+L}$, ensuring for the Equation 5.6 holds $\forall t' \in \delta_{k+j}$ where $1 \leq j \leq L$.

For each future time slot, δ_{k+j}, RRM first estimates additional sandboxes needed to meet the guaranteed invocation rate requirements until the time slot ends (c_j). Since the scheduler limits the schedule rate for each function f_i to A_i, the maximum sandboxes consumed by the function from t_k to δ_{k+j} is $\lceil A_i T_C (j+1) \rceil$ capped by $\lceil A_i E_i \rceil$ because once an invocation completes, the RRM can reuse its sandbox for the new ones. Thus, the additional sandboxes required by the scheduler in δ_{k+j} is

$$c_j = R(t_k, j) = \sum_{i=1}^{N} \lceil A_i \min(T_C(j+1), E_i) \rceil - B(t_k) \tag{5.7}$$

recall $B(t_k) = \max[u(t_k), 0]$ is number of free sandboxes at t_k. Note that this is a part of the sandbox sharing strategy as $B(t)$ contains both required and extra sandboxes allocated successfully by t. The higher the $B(t)$, the fewer additional sandboxes are required; thus, fewer allocation requests. To meet the objective in δ_{k+j}, RRM has to ensure these additional c_j sandboxes are allocated successfully before t_{k+j}, with probability X. Given the sandbox allocation latency statistics $P^c_{\Delta t}(a)$, RRM needs to allocate a_j sandboxes such that

$$P^{c_j}_{t_{k+j}-t_k}(a_j) = P^{c_j}_{jT_C}(a_j) \geq X \tag{5.8}$$

This implies

$$a_j \geq (P^{c_j}_{jT_C})^{-1}(X) \tag{5.9}$$

where $(P^{c_j}_{jT_C})^{-1}(X)$ is the inverse function of $P^{c_j}_{jT_C}(a_j)$. In special cases where $c_j \leq 0$, we let $(P^{c_j}_{jT_C})^{-1}(X) = 0$. The function indicates the number of allocation requests needed to get at least c_j sandboxes in jT_C seconds. Thus, by allocating $a_j = (P^{c_j}_{jT_C})^{-1}(X)$ sandboxes, the RRM ensures to get the required sandboxes, c_j, by t_{k+j} with probability X, satisfying the scaling up objective in Equation 5.6. However, due to the allocation limitation M, we cannot allocate more than M sandboxes within a time slot. This means $P^c_{\Delta t}(a)$ is only defined for $a \leq M$ and its inverse function, $(P^c_{\Delta t})^{-1}(X)$ may not exist for all X. We need to adjust a_j as follows.

$$a_j = S(X, c_j, jT_C) = \begin{cases} (P^{c_j}_{jT_C})^{-1}(X) & \text{if } (P^{c_j}_{jT_C})^{-1}(X) \text{ exists} \\ M & \text{otherwise} \end{cases} \tag{5.10}$$

The probability of obtaining c_j sandboxes at t_{k+j} becomes

$$p_j = P_{jT_C}^{c_j}(a_j) = \begin{cases} X & \text{if } (P_{jT_C}^{c_j})^{-1}(X) \text{ exists} \\ P_{jT_C}^{c_j}(M) & \text{otherwise} \end{cases} \tag{5.11}$$

Note that p_j is the probability of obtaining the required sandboxes in *one try*, and this try alone is not sufficient if $(P_{\Delta t}^{k_j})^{-1}(X)$ does not exist. Thus, we need multiple tries, but how many (i.e., L) are sufficient? We need a large L to ensure the guarantee even in the worst case where there are no free sandboxes (i.e., $B(t) = 0$) and the sandbox allocation is too slow that the RRM has to allocate $a_j = M$ sandboxes every time it scales up. In such case, by Equation 5.7, $c_j = \sum_{i=1}^{N} \lceil A_i \min(T_C(j+1), E_i) \rceil$ so by Equation 5.11, the worst case probability of a single try is

$$p_j^w = P_{jT_C}^{\sum_{i=1}^{N} \lceil A_i \min(jT_C, E_i) \rceil}(M) \tag{5.12}$$

Since scaling tries are independent, we can consider them as a variant of the Bernoulli trials with different success probabilities $p_1^w, ..., p_L^w$. We want *at least* one try to succeed in meeting the availability guarantee. Then L becomes

$$L = numTriesEst(X) = \min \{ l : 1 - \prod_{j=1}^{l} (1 - p_j^w) \geq X \} \tag{5.13}$$

Algorithm 3 Scaling Up Estimation

```
1: function SCALEUPEST($t_k, L, X$)
2:     for $j \leftarrow 1$ to $L$ do
3:         $c_j \leftarrow R(t_k, j)$                    ▷ Sandbox Est. (Equ. 5.7)
4:         $a_j \leftarrow S(X, c_j, jT_C)$              ▷ Overallocation (Equ. 5.10)
5:     end for
6:     return $\max(M, \sum_{i=1}^{L} a_j)$              ▷ Multiple tries
7: end function
```

We summarize the whole scaling-up process described above in the *ScaleUpEst* function implementation as shown in the Algorithm 3. Given a scaling time t_k, the function calculates c_j from Equation 5.7 (Line #3) to determine the required number of additional sandboxes to meet all rate guarantees in each future time slots δ_{k+j} $(1 \leq j \leq L)$. Then, it estimates the number of sandboxes to be allocated, a_j (from Equation 5.10, Line #4), so that the pool will get c_j ones in time with a probability at least equal to X, satisfying the guarantee availability target X. Finally, the number of required allocations, a_j's, are combined and will be used by the smart dynamic allocation algorithm to scale up the pool (Line #5 and #7 in Algorithm 2).

5.6.2.2 Scaling Down

RRM employs overallocation and multiple tries strategies for scaling up, incurring a lot of potential resource waste. We try to minimize the waste by keeping the extra buffered resources just enough for sharing by bounding them with a conservative deallocation strategy. Specifically, at a given scaling time t_k, the RRM will compute

$$d = \mathit{scaleDownEst}(t_k) = -\max_{0 \leq j \leq L} R(t_k, j) \tag{5.14}$$

Recall from equation 5.7, $R(t_k, j)$ is the number of sandboxes to be added at t_k to ensure the guarantee availability in δ_{k+j}. If $d > 0$, then the current buffer size $B(t)$ is large enough to support the rate-guarantee *without* allocating more sandboxes until δ_{k+L}. Further, d, in this case, is an unnecessary waste, and the RRM will deallocate them.

5.6.2.3 The Complete Algorithm

We integrate the implementation of *scaleUpEst* (Algorithm 3), *scaleDownEst* (Equation 5.14) and *numTriesEst* (Equation 5.13) into the Algorithm 2 to create a complete version

Algorithm 4 Smart Dynamic Allocation (Completed)

```
1: $L \leftarrow \min\{l : 1 - \prod_{j=1}^{l}(1 - p_j^w) \geq X\}$    ▷ Equ. 5.13
2: Prealloc($\sum_{i=1}^{N}\lceil A_i \min(E_i, T_C L)\rceil$)
3: for every $T_C$ seconds do
4:     $t_k \leftarrow currentTime()$
5:     for $j \leftarrow 1$ to $L$ do
6:         $c_j \leftarrow R(t_k, j)$    ▷ Sandbox Est. (Equ. 5.7)
7:         $a_j \leftarrow S(X, c_j, jT_C)$    ▷ Overallocation (Equ. 5.10)
8:     end for
9:     $a \leftarrow \max(M, \sum_{i=1}^{L} a_j)$    ▷ Multiple tries
10:    if $a > 0$ then
11:        Alloc(a)    ▷ Scale-up
12:    else if $t > \Delta t_L$ then
13:        $d \leftarrow \max_{0 \leq j \leq L} R(t_k, j)$    ▷ Detect waste (Equ. 5.14)
14:        if $d < 0$ then
15:            Dealloc(-d)    ▷ Scale-down
16:        end if
17:    end if
18: end for
```

of the smart dynamic allocation algorithm in Algorithm 4. We will use the algorithm to demonstrate that we can enforce RBAM guarantee with low cost in the rest of the chapter.

5.6.2.4 Guarantee Availability and Efficiency Analysis

Next, we will theoretically prove that the Algorithm 4 always allocates sufficient sandboxes to meet guaranteed invocation rates at any given guarantee availability target X. Then, we will derive a bound on the algorithm's resource cost, analytically showing that the algorithm is cost-effective and robust.

Theorem 13 (Rate-guarantee Availability)**.** *For all $X < 1$, the Algorithm 4 ensures*

$$\forall t : Pr[u(t) \geq 0] \geq X \tag{5.15}$$

Proof. We prove the theorem by showing that the algorithm ensures the guarantee availability X for all time slot δ_k. That is:

$$\forall k : \forall t \in \delta_k : Pr[u(t) \geq 0] \geq X \tag{5.16}$$

Consider δ_k, there are two possible cases:

- $k \leq L$: the preallocation $\sum_{i=1}^{N} \lceil A_i \min(E_i, T_C L) \rceil$ is sufficient to ensure all rate-guarantees in the first LT_C seconds, which also cover δ_k. Thus

$$\forall t \leq LT_C : \quad Pr[u(t) \geq 0] = 1 \geq X \tag{5.17}$$

- $k > L$. Let t_d be the last time the RRM deallocates sandboxes before δ_k.
  - If $\Delta k = k - d \leq L$ then by the deallocation condition of the algorithm at Line #14, $R(t_d, \Delta k) \leq 0$, meaning the guarantee availability have been already ensured in δ_k without additional sandboxes. Further, as t_d is the last deallocation, the pool size will not decrease by δ_k. These guarantee $\forall t \in \delta_k : Pr[u(t) \geq 0] \geq X$.
  - Otherwise, resources at t_k will be prepared by all allocations made at $t_{k-1}, ..., t_{k-L}$. Let p_j be the probability the allocation at t_{k-j} gives the pool sufficient sandboxes for the time slot δ_k. Since these allocations are independent, we can consider them as Bernoulli trails with success probability p_j. Thus, if at least one of the trails succeeds, then $u(t) \geq 0$:

$$Pr[u(t) \geq 0] = 1 - \prod_{j=1}^{L} (1 - p_j)$$

Since p_j is determined by allocations at t_{k-j}, which are at least $S(X, c_j, jT_C)$ sandboxes (Line #7, and #9). By the Equation 5.11:

$$p_j \geq P_{jT_C}^{c_j}(a_j) = \begin{cases} X & \text{if } (P_{jT_C}^{c_j})^{-1}(X) \text{ exists} \\ P_{jT_C}^{c_j}(M) & \text{otherwise} \end{cases}$$

If $\exists j^* \in \{1, ..., L\} : p_{j^*} = X$, then

$$\begin{aligned} Pr[u(t) \geq 0] &\geq 1 - \prod_{j=1}^{L}(1 - p_j) \\ &\geq 1 - (1 - p_{j^*}) \\ &= X \end{aligned}$$

Otherwise, $\forall j : p_j = P_{jT_C}^{c_j}(M) \geq P_{jT_C}^{\sum_{i=1}^{N}\lceil A_i \min(T_C(j+1), E_i)\rceil}(M)$ (Equ. 5.3), thus

$$\begin{aligned} Pr[u(t) \geq 0] &\geq 1 - \prod_{j=1}^{L}(1 - p_j) \\ &\geq 1 - \prod_{j=1}^{L}(1 - P_{jT_C}^{\sum_{i=1}^{N}\lceil A_i \min(T_C(j+1), E_i)\rceil}(M)) \\ &\geq X \qquad \text{(Equ. 5.13)} \end{aligned}$$

Thus, all possibilities of δ_k imply $\forall t \in \delta_k : Pr[u(t) \geq 0] \geq X$. This completes the proof. $\square$

The algorithm ensures $\forall t_k : \max_{0 \leq j \leq L} R(t_k, j) \geq 0$, By Equation 5.7, this implies

$$B(t) \leq \sum_{i=1}^{N}\lceil A_i \min(T_C(L+1), E_i)\rceil \tag{5.18}$$

Recall that $B(t)$ is the number of free sandboxes. This is the upper bound of the algorithm's extra resource cost. The bound depends on RBAM configurations (A_i and E_i) and deploy-

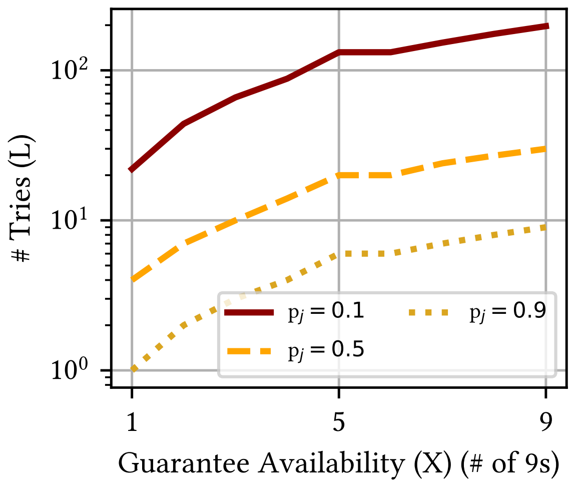


Figure 5.8: Smart Dynamic Allocation Algorithm is cost-effective at high guarantee availability targets

ment environment (T_C). The only controllable factor is L which is adjustable by configuring the guarantee availability X using Equ. 5.13. Note that by that equation, X (the right-hand side) grows exponentially as L increases (the left-hand side). This means high X is achievable with just a few allocation tries, as visualized in Figure 5.8. The figure considers a simplified version of the model where we assume p_j are identical. The required L grows exponentially slowly compared to X. For example, with $p_j = 0.5$, increasing X from two 9s to eight 9s (6 orders of magnitude better) requires only 4x more tries, demonstrating excellent cost-effectiveness of the algorithm at a high availability target. We will elaborate more on the experimental evaluation in the next section.

Takeaway. (RBAM Resource Management) The Rate-based Resource Manager allocates extra sandboxes multiple times to statistically reduce allocation latency and implementation overhead. Additionally, sandboxes are shared across functions to reduce overhead further and improve guarantee availability.

5.7 Evaluation

5.7.1 Methodology and Experimental Setup

This section will evaluate the efficiency of the RBAM implementation by simulating the scheduler and RRM under various settings. The objective is to answer the following questions:

1. Can RBAM be implemented efficiently?
2. Is the efficiency robust across different workloads and deployment scenarios?
3. Can RBAM be implemented scalably?

5.7.1.1 Metrics

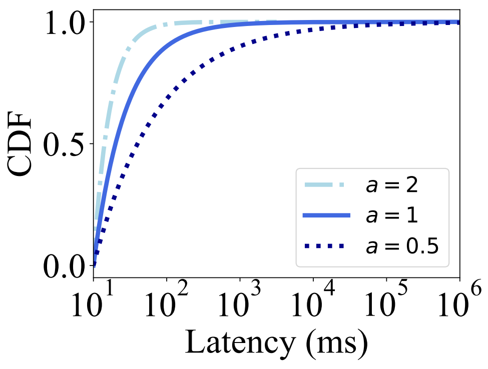
We evaluate RBAM implementation efficiency through its guarantee availability and cost-effectiveness measured through the following metrics:

- *Guarantee Availability*: The fraction of requests the RBAM pool successfully executes scheduled invocations.
- *Overhead*: the ratio of resources allocated to unused and pending sandboxes to the resources allocated to used sandboxes.

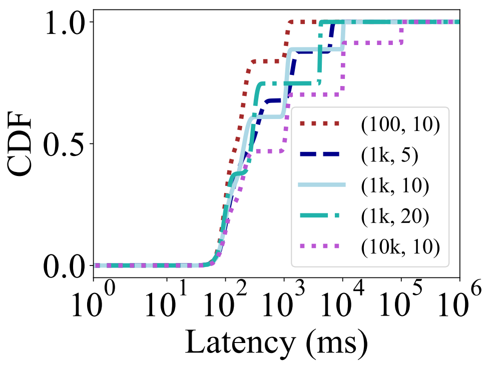
5.7.1.2 Sandbox Allocation Latency

We consider both practical and synthetic single sandbox latency distributions ($\mathcal{P}$).

- *Practical Distributions* includes *Google Cloud Functions* cold start statistics [36] and Google Borg instance schedule latency [304] (See Figure 5.2b).



(a) Pareto



(b) Mutimodal

Figure 5.9: Synthetic latency distributions with configurable parameters (Pareto: a, Mutimodal: (height, distance)).

- *Synthetic Distributions* two distribution families have a similar unbounded, long-tail structure to practical ones.
  - *Pareto*: long-tail, monotonic decreasing distribution whose shape is configurable via a real parameter a (Figure 5.9a).
  - *MultiModal*: long-tail, non-monotonic decreasing distributions that have multiple decreasing peaks with configurable height (magnitude of the peaks) and distance (gap length between peaks) as shown in Figure 5.9b. The peaks are formed by multiple Gaussian distributions $\mathcal{N}(\mu_1, \sigma_1)$, $\mathcal{N}(\mu_2, \sigma_2)$,... with the probability density function (PDF) defined as follows.

$$f_M(x) = S \sum_{i=1}^{n} f_i(x) \tag{5.19}$$

where $f_i(x)$ is the PDF of $\mathcal{N}(\mu_i, \sigma_i)$, s_i is a weight defining how much each Gaussian component contributes to the distribution, control by a *height parameter* h:

$$s_i(x) = \frac{1}{h \cdot \exp(\frac{x}{h})} \tag{5.20}$$

and $S = \sum_{i=1}^{n} s_i F_i(X > 0)$ is a scaling factor which is calculated as the sum of the Gaussian's cumulative density functions (CDF) to ensure $f_M(x)$ is a proper density function. Each component distribution, $\mathcal{N}(\mu_i, \sigma_i)$, has the mean μ_i and standard deviation σ controlled by a *distance parameter* d defined as follows.

$$\mu_i = \mu_0 + d^i \qquad \text{and} \qquad \sigma_i = \quad 2^i$$

We use various multimodal distributions shown in Figure 5.9b with $n = 5$ by varying the height and distance parameters.

5.7.1.3 RBAM Resource Management Algorithms.

We consider four resource management algorithms

- *Smart Dynamic Scaling*: our proposed resource management algorithm as presented in Algorithm 4 in Section 5.6.2.
- *Overalloc Dynamic Scaling*: Smart Dynamic Scaling without multiple tries (i.e., Algorithm 4 with $L = 1$).
- *Naive Dynamic Scaling*: Smart Dynamic Scaling without multiple tries and overallocation (i.e., Algorithm 4 with $L = 1$ and replace Line #7 of Algorithm 4 by $a_j = c_j$).
- *Pre-allocation* (i.e., the *baseline* approach) Maintain $\lceil A_i E_i \rceil$ sandboxes for each function in the RBAM pool over its lifetime.

5.7.1.4 Workloads

We use 2 sets of workloads (Figure 5.10).

- *Bursty (Production traces)*: FaaS arrival rate and execution time selected from the Azure traces [279]. We configure the guaranteed invocation rate equal to each function's peak rate and execution time equal to their longest one to validate the system guarantee enforcement capability.

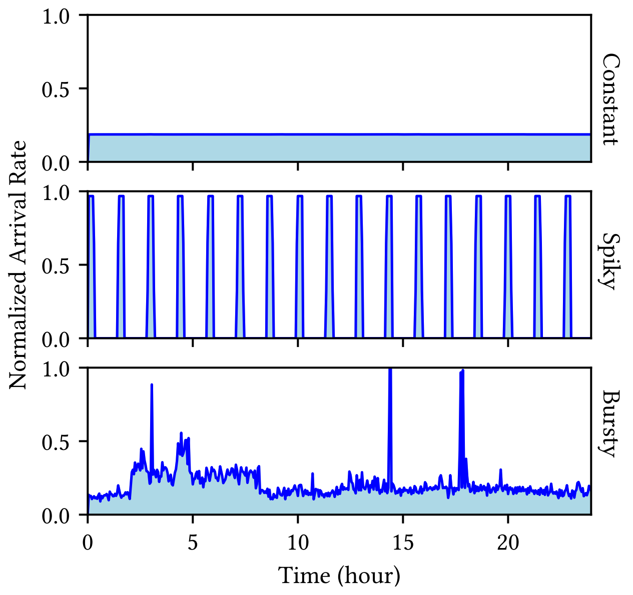


Figure 5.10: Synthetic (*constants* and *spiky*) and production (*bursty*) Workloads

- *Synthetic*: We modify the arrival of the 10000 FaaS functions in the production traces to create two additional synthetic workloads: (i) *constant* with a fixed arrival rate to capture stable workloads and (ii) *spiky* with periodic load spikes that experience invocation requests at a rate of the rate-guarantee to capture the extreme case of workload surges. Both workloads have an average invocation arrival rate equal to *bursty*.

5.7.2 Experimental Results

We use the experimental evaluation results to prove that RBAM can be implemented efficiently. This consists of three parts. First, in Section 5.7.2.1, we will show the RBAM implementation efficiency in practical settings. Second, we extend the study to consider more extreme scenarios using synthetic data to show that the implementation efficiency is robust against various workloads and deployment settings (Section 5.7.2.2). Third, we demonstrate the system's scalability by examining its efficiency under different scales, show-

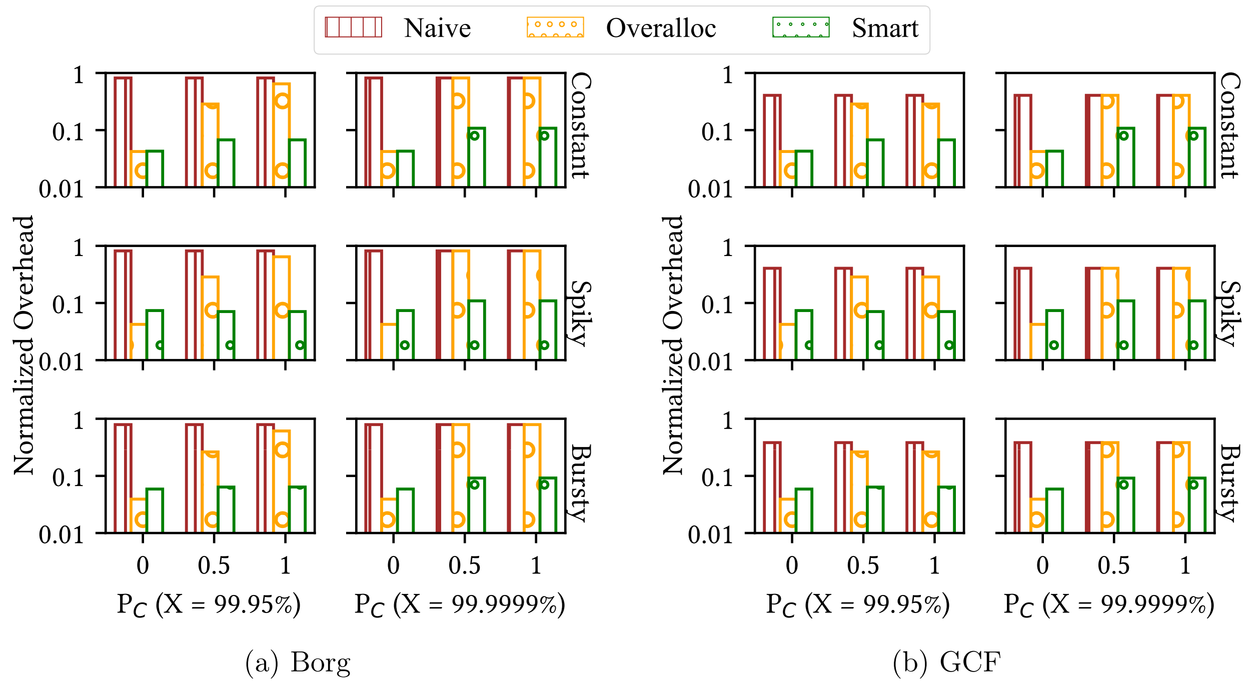


Figure 5.11: RBAM Implementation Efficiency

ing that we can implement thousands of RBAM functions efficiently under high correlation in Section 5.7.2.3.

5.7.2.1 RBAM Implementation Efficiency

We evaluate RBAM implementation efficiency under realistic settings. We construct workloads by selecting 1000 random functions from the Azure traces. The new RBAM system (Figure 5.6) serves workloads with the RBAM resource pool managed by one of the four resource management approaches mentioned in Section 5.7.1.3. Sandboxes are allocated with latency following production traces (Figure 5.2b). We examine different correlation possibilities (P_C), from 0 (independent) to 1 (always correlated), with correlation period $T_C = 1$ second (as a majority of allocations takes less than 1 sec).

Figure 5.11 shows the overhead required by the dynamic scaling normalized by the baseline overhead (i.e., *Pre-allocation*) when the allocation latency follows the *Borg* (5.11a) and *GCF* (5.11b) distributions. *Naive* scaling is inefficient as its overhead is close to pre-

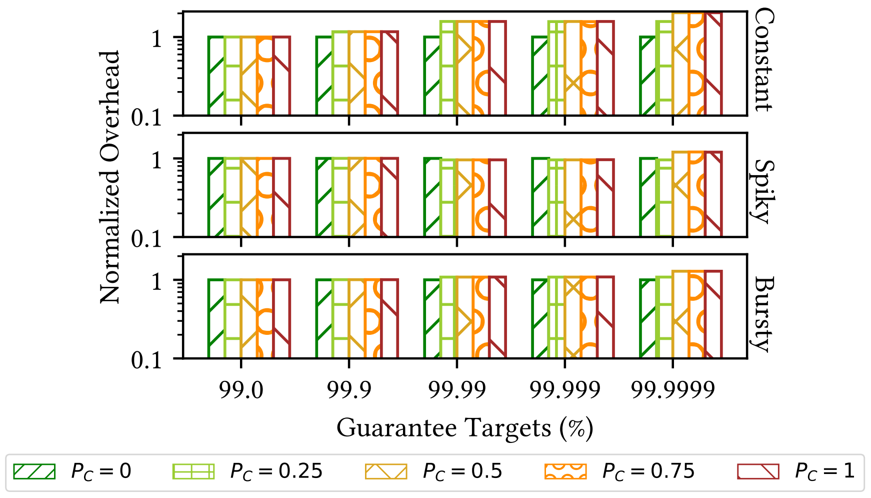


Figure 5.12: RBAM implementation is robust against various guarantee availability targets (X)

allocation in all settings (brown columns). *Overalloc*, on the other hand, reduces the overhead significantly (orange columns). At current cloud standard availability (i.e., 99.95%), employing overallocation alone reduces the overhead by at least 14x compared to pre-allocation. However, overallocation alone becomes ineffective when aiming for higher availability (i.e., 99.9999%), especially with correlation. Even at $P_C = 0.5$, Overalloc is as bad as the *Naive*. *Smart* scaling, in contrast, employs multiple tries to mitigate the correlation effect and thus consistently meets availability targets with the overhead 10× smaller than pre-allocation in all scenarios.

Takeaway. (Efficiency) Overallocation and Multiple tries enable efficient RBAM implementation, achieving high guarantee availability even under high correlation.

5.7.2.2 Robustness

We investigate RBAM implementation robustness by extending the efficiency evaluation across a wide range of different settings using synthetic configurations. We start with a *base setting* that is close to the practice settings used in the previous experiment. Then, we try

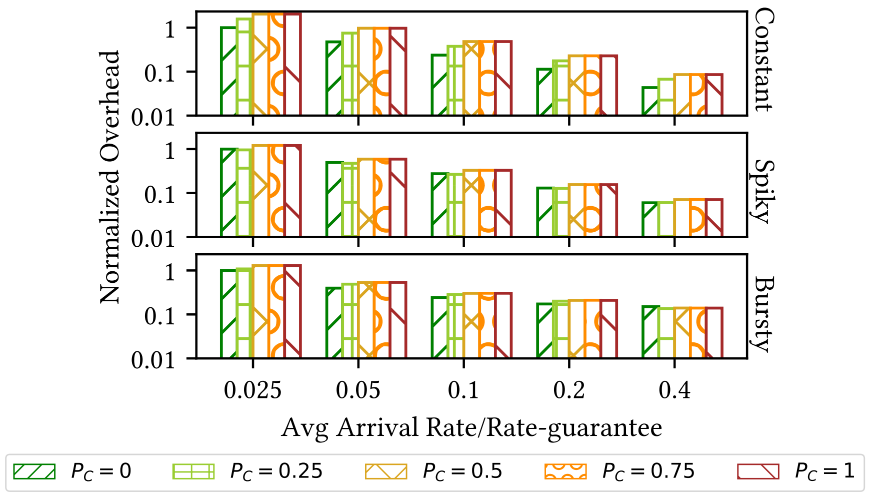


Figure 5.13: RBAM implementation is robust against various workload arrival rates.

to push the Smart algorithm to work on more extreme deployment scenarios with higher workload variability and more complicated cloud configurations. We report the corresponding overhead while adjusting related parameters to evaluate how well the algorithm handles difficult settings.

The *base setting* is chosen as follows: guarantee availability target $X = 0.999999$, correlation period $T_C = 1s$, with correlation probability $P_C = 0.5$, and allocation latency follows the Pareto distribution with $\alpha = 1$ and $M = 2\times$ of the total rate-guarantee requirement. In the rest of this section, we first represent the impact of the workload arrival rate and guarantee availability target. Then, we evaluate the impact of the cloud environment by varying the allocation latency, allocation limit, correlation period T_C, and probability P_C.

Impact of Guarantee Availability Targets Figure 5.12 shows the overhead needed to achieve different guarantee availability targets (X) normalized to the case of $X = 99\%$ with no correlation (e.g., the first green column). From 99% to 99.9999% availability, the risk of failure rate guarantee is reduced by five orders of magnitude while the overhead increases by less than 10x. Thus, RBAM can be implemented with high availability and a diminishingly increasing overhead. This sublinear relationship proves the robustness of the

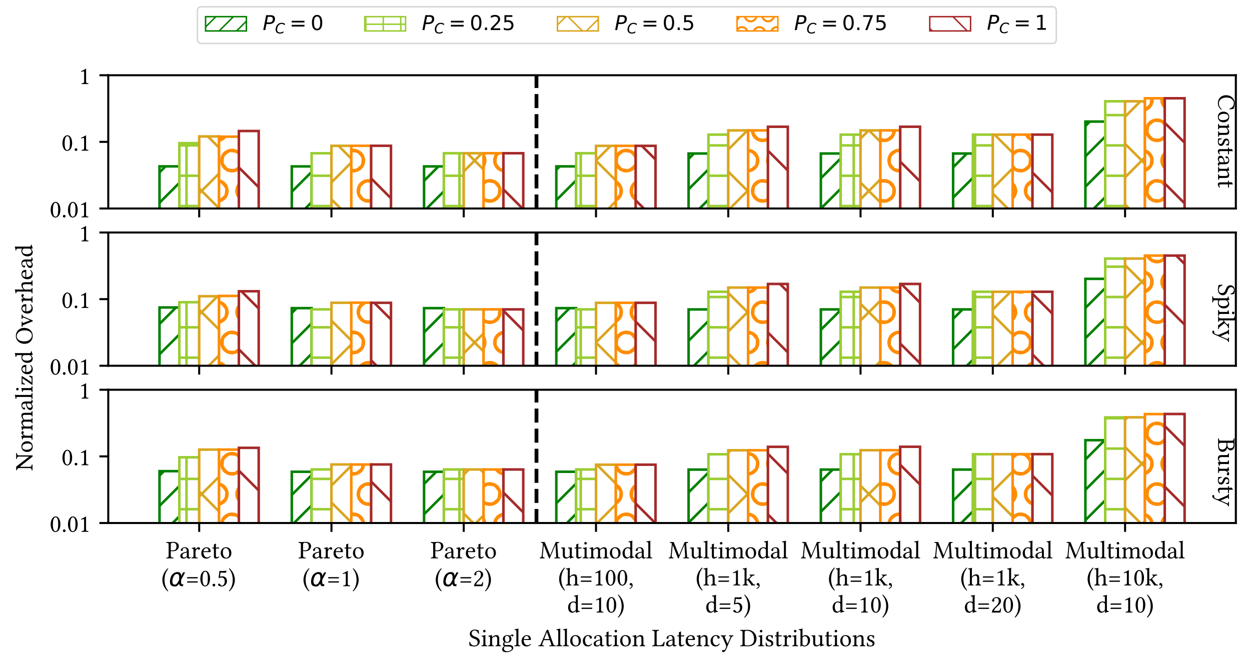


Figure 5.14: RBAM implementation is robust against various single sandbox latency distributions ($\mathcal{P}$).

implementation against availability targets, allowing RBAM to become a *reliable* computing model for applications with *strict* real-time and availability requirements.

Impact of Workload Arrival Rate Next, we adjust the average arrival rate of the three workloads from 0.025 to 0.4 of the rate guarantee and report the required overhead to meet the 99.9999% guarantee availability in Figure 5.13. From Section 5.6.2, we showed that the RBAM pool's number of free sandboxes necessary to maintain the rate guarantee remains constant regardless of workload dynamics (Equation 5.18). This means that no matter how users change the request arrivals, the buffer size remains unchanged. As a result, the recorded overhead decreases as we increase the arrival rate, as confirmed in the figure. These results underscore the reliability of RBAM in offering consistent performance, independent of workload dynamics, at a reasonable resource cost.

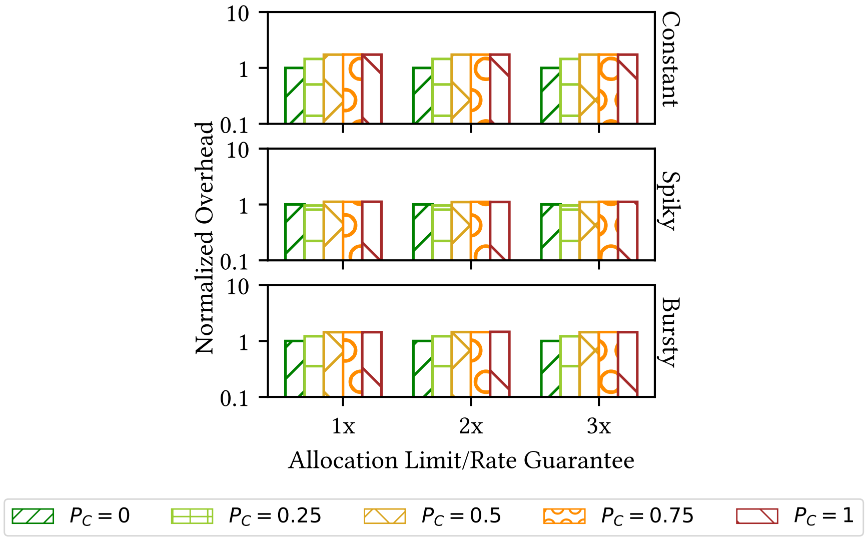


Figure 5.15: RBAM implementation is robust against various allocation limits (M).

Takeaway. (Robustness #1) RBAM implementation is robust against workload variability and availability targets.

Impact of Single Sandbox Allocation Latency Distribution In Figure 5.14, the RBAM deployment overhead is normalized by the pre-allocation overhead when we vary the sandbox allocation latency distribution. We consider eight distributions: three Pareto variants (See Figure 5.9a) and five multimodal variants (varying height and distance, see Figure 5.9b). Smart scaling is consistently more efficient than pre-allocation. Even in the most extreme case (*Multimodal (h=10k, d=10)*), it is still 2.3x more cost-effective. The algorithm is also robust against the shape of distributions. For example, in the case of Pareto variants, changing the α from 2 to 0.5 significantly reshapes the distribution (e.g., from 99% of request complete in less than 100ms to only 70%), but Smart could handle them both with less than 2.3x resource difference

Also, note that some of these distributions have very long latency. For example, the Pareto with $\alpha = 0.5$ has a statistical average latency that is 100,000× longer than practice

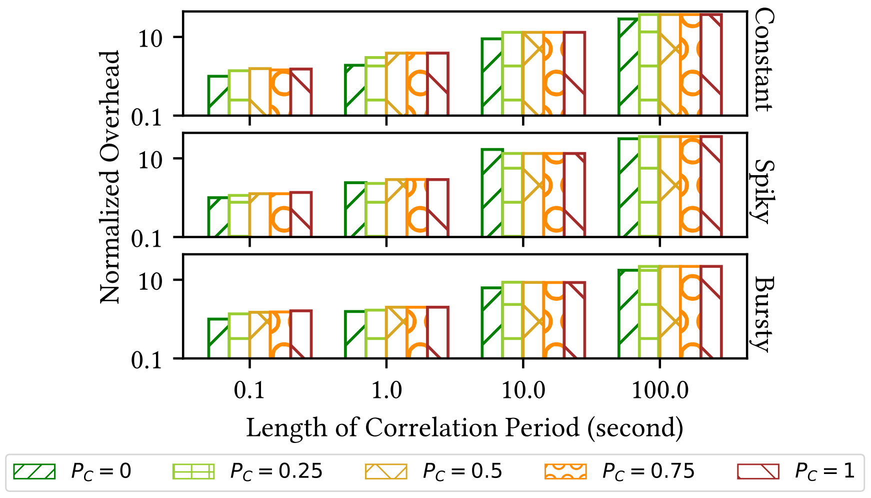
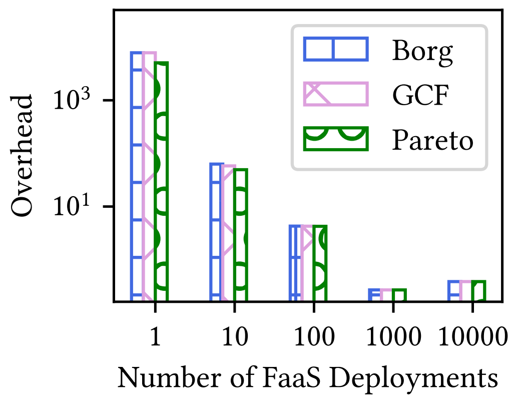


Figure 5.16: RBAM implementation is robust against various lengths of correlation period (T_C)

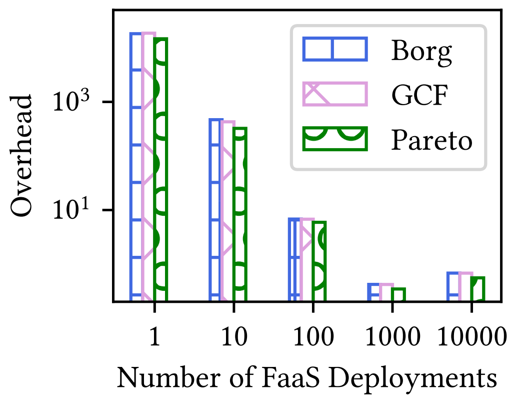
ones. Even under these unrealistic settings, the RBAM system can keep overhead low and robust against correlation. Specifically, starting from $P_C = 0$ (no correlation) to $P_C = 1$ (correlation always appears), the algorithm only needs less than 2.5$\times$ more resources to meet the required availability target across all latency distributions.

Impact of Allocation Limit Next, we vary the allocation limit M from 1$\times$ to 3$\times$ the total rate-guarantee. Figure 5.15 reveals that the overhead in these cases is almost identical. This is because sandboxes are usually allocated at a rate much lower than M (the average is 3.4% of the peak), and even if it does, the high allocation rate does not last long because the scheduler can reuse old resources for the new load. Consequently, the system is barely affected by changing the parameter.

Impact of Length of Correlation Period Figure 5.16 reveals a sublinear relationship between the overhead of ensuring 99.9999% guarantee availability and the length of the correlation period (T_C). Increasing the length from 100ms to 100s is equivalent to switching to an underlying system with correlations that last three orders of magnitude longer, but this requires less than 10x more resources. This is because the RBAM system uses a longer gap



(a) $P_C = 0$



(b) $P_C = 1$

Figure 5.17: Resource Sharing Efficiency

between scaling decisions at longer correlated periods, giving us more chances to get needed sandboxes within a time slot. This effectively reduces the number of multiple tries (L) needed to satisfy guaranteed availability, making handling long, bad correlation events less costly. This result and those in Figure 5.14 strengthen the robustness of RBAM implementation efficiency against underlying resource allocations, allowing RBAM functions to be deployable over a *wide* range of environments with *insignificant* cost changes.

Impact of Correlation Probability Across all experiment results shown from Figure 5.12 to Figure 5.16, we also vary correlation probability P_C from 0 (no correlation) to 1 (allocations always correlated) yet there are consistently slight changes in the corresponding overhead (most are less than $2\times$ for $P_C = 0$ to 1), demonstrating that the implementation is highly robust against correlation occurrence frequency. This result further strengthens the conclusion of RBAM implementation robustness against allocation correlation and unexpected cloud events, as we have already shown in Figures 5.16 and 5.15.

Takeaway. (Robustness #2) RBAM implementation is robust against a wide range of cloud environment settings, including many extreme cases that are very far from practice.

5.7.2.3 Resource Sharing Efficiency

We evaluate resource sharing efficiency by varying the number of functions to be supported by the RBAM system in the same setting as in Section 5.7.2.1 and report the overhead in Figure 5.17.

The RBAM system significantly reduces the overhead when adding new function deployments. Specifically, in Figure 5.17a, increasing the pool size from 1 to 10k functions reduces the overhead by four orders of magnitude, indicating at least a linear reduction rate. This demonstrates the advantage of resource sharing, where extra sandbox allocations can be shared among different function invocations to mitigate the impact of workload variability and long allocations. They also get reserved to avoid extra future allocations and further mitigate the side effects of overallocation and multiple tries. The overhead reduction is saturated at 38% when the system serves up to 10k FaaS deployments.

Furthermore, RBAM implementation is robust and can tolerate a high degree of correlation. Figure 5.17b shows that even when all allocations are correlated (i.e., we change P_C from zero to one), the overhead reduction pattern remains consistent, with a slight increase in overhead by at most 40%.

Takeaway. (Resource Sharing Efficiency) RBAM implementation is highly scalable and cost-effective in realistic settings.

5.8 Summary

In this chapter, we demonstrate that implementing an RBAM's guaranteed invocation rate is feasible with a resource cost that is bounded by the rate guarantee and the function's maximum execution time. However, reducing this cost below the upper limit is challenging due to the high variability of FaaS workloads and the complexity and uncertainty of the cloud environment.

To address these challenges, we propose three performance abstraction techniques: over-allocation, multiple tries, and resource sharing. These methods enable effective handling of workload variability and cloud uncertainty with low overhead. Additionally, we introduce a new RBAM implementation architecture to integrate these techniques into the FaaS system, forming new scheduling and resource management algorithms to deliver performance guarantees efficiently, robustly, and scalably.

We theoretically prove the new system ensures RBAM performance guarantee with bounded cost that is robust against various workloads and deployment settings. We validate our findings through extensive experiments, including both practical cloud settings and synthetic scenarios that cover a wide range of challenging settings. We demonstrate that RBAM can be implemented effectively with 500× higher availability than the current cloud standard while reducing the resource cost by 10× compared to the implementation upper bound. The efficiency is consistent across various settings. Even in extreme cases, which are 100,000 times worse than practical settings, the new RBAM system requires less than 10 times more resources to meet the desired guarantees. RBAM implementation also scales well. We can implement up to 10k practical FaaS deployment with 99.9999% availability but cost less than 40% extra resources.

To summarize, let us revisit the research questions related to RBAM implementation in Section 3.2 and answer them as follows

Implementation Efficiency. Can RBAM be implemented efficiently?

Q.2.1 Can RBAM be implemented to support any finite guaranteed invocation rate?

- **Answer: Yes**, by Theorem 12, we showed that *any* RBAM function f_i with *finite* guaranteed invocation rate A_i and execution time bounded by E_i can be implemented with resource cost *bounded* by $\lceil A_i E_i \rceil$.

Q.2.2 Can RBAM also be implemented with low overhead across various scenarios?

- **Answer: Yes** In Section 5.7.2, we systematically evaluate the new FaaS system using both realistic and synthetic settings to show that we can implement RBAM with 500× higher availability than the current cloud standard but cost 10× less than the implementation upper-bound. The efficiency is consistent across various settings. Even in extreme cases that are up to 100,000× worse than the current practice, the resource cost required to fulfill the guarantee is less than 10× compared to the practical setting.

Q.2.3 Can RBAM be implemented scalably?

- **Answer: Yes** Also in Section 5.7.2, we show that our RBAM system can support up to 10k practical FaaS function deployments with 99.9999% availability but costs only 38% extra resources.

With these answers, we complete showing the RBAM can be implemented efficiently.

CHAPTER 6

RBAM APPLICABILITY

In this chapter, we will demonstrate RBAM applicability using the RBAM compute model to implement two bursty, real-time applications with RBAM: distributed video analytics (Section 6.1) and distributed real-time stream processing (Section 6.2). In each section, we first briefly introduce each application and show why implementing them using current solutions is challenging (Sections 6.1.1 and 6.2.1). Next, we show how the RBAM compute model helps address these challenges and deliver corresponding RBAM implementations of the applications in Sections 6.1.2 and 6.2.2. Finally, we systematically evaluate them with different deployment and workload configurations to demonstrate RBAM applicability in Sections 6.1.3 and 6.2.3.

6.1 Distributed Real-time Video Analytics

Distributed Real-time Video Analytics are popular representatives of bursty real-time applications [71, 329]. The application collects video recorded by multiple cameras installed over a specific region to extract useful information or to take action when an event happens [260, 97, 259, 190]. Both usually lead to a bursty demand that needs a real-time response.

One example is traffic monitoring, which continuously analyzes video streams of traffic recorded by cameras installed at intersections (Figure 6.1). The application is interested in unusual events such as car crashes, pedestrian falls, etc. Once such an event happens, it triggers an in-depth analysis to understand the situation and make proper decisions (e.g., call the police, broadcast a warning signal, etc.). The in-depth analysis uses a sophisticated Machine Learning (ML) model on high-resolution video streams, eventually creating significant resource demand. Further, the analysis must be processed quickly because the event is

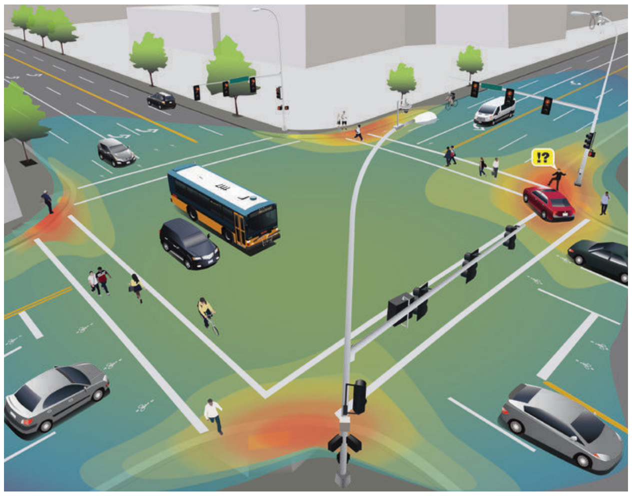


Figure 6.1: Traffic Monitoring: An example of distributed real-time video analytics

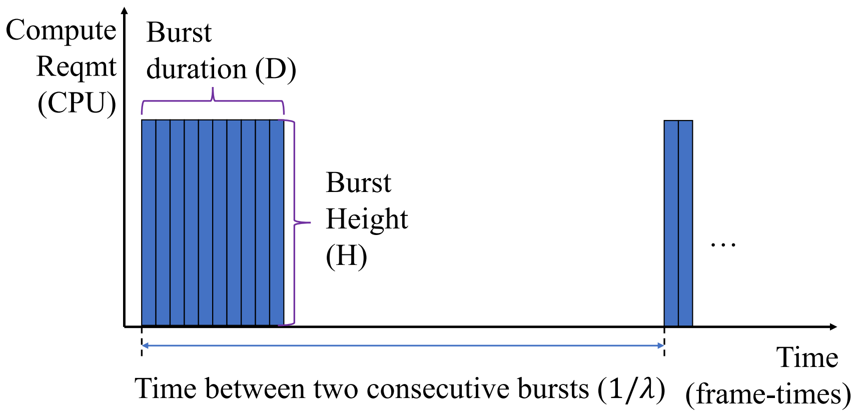


Figure 6.2: Video Analytics Burstiness Modeling

crucial. Prolonged computation latency decreases application value/quality (e.g., a delay in calling an emergency medical service after an accident may lead to severe consequences).

6.1.1 Challenges

We consider the in-depth analysis a single computation task, **Video Analytics (VA)**, and implement it considering two common approaches: pre-allocation and FaaS. But before digging into the implementation, let us first formally model VA's burstiness and real-time requirements, which will help construct its implementations and evaluation.

6.1.1.1 Analytical Model

Since in-depth analyses are only required by some unexpected events, when triggered, they will have distributed cameras generate high-resolution video frames with a fixed release rate equal to the video frame per second (fps). Assuming bursts appear with average frequency λ. Consider an in-depth processing burst with D video frames $F_1, ..., F_D$ ordered by arrival time. For simplicity, we use *frame-time*, the interarrival between two video frames, as a time unit instead of a second. We assume video frames take identical computation demand H to complete within a frame-time. Since in-depth analysis is a computation-intensive task, we use the CPU as the unit for computation demand. We visualize these parameters in Figure 6.2.

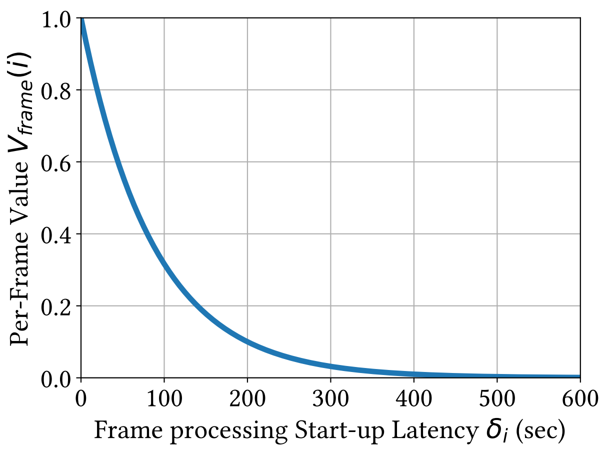


Figure 6.3: Value decay as frame processing startup latency increases.

To capture the VA real-time requirement, we use *value* or *quality* to represent the satisfaction of VA on resource availability in helping it deal with bursts. In particular, VA needs fast computation, so there is a loss of value (or quality) when frame processing is delayed. We model this as the value decays with frame processing delay. Specifically, we use an exponential decay with *frame processing startup latency* with τ as the critical time constant for VA. That is, given a single burst frame F_i, delayed by δ_i frame-time, its value $V_{frame}(i)$

Symbol	*Name*	*Definition*	*Unit*
Burstiness Properties			
λ	Burst rate	Burst average emergence frequency	burst/frame-time
F_i	Burst frame	Frame/burst indexed by arrival order	
D	Burst duration	Number of video frame per burst	frame/burst
H	Burst height	burst frame computation demand	compute/frame
Real-time Requirement Parameters			
V_{max}	Max. Value	Maximum value per frame	
δ_i	Startup Delay	Time a frame F_i waits for processing	frame-time
τ	Value delay	Characteristic time period	
$V_{frame}(i)$	Actual value	Actual value for a frame F_i	
$BurstValue$	Burst value	Aggregated frame value per burst	
Deployment Parameters			
P	Pre-allocation	static allocation resources	CPU
A	Rate guarantee	CPU guaranteed allocation/frame-time	CPU/frame-time

Table 6.1: Video Analytics Notations

is defined as

$$V_{frame}(i) = V_{max}e^{-\frac{\delta_i}{\tau}} \tag{6.1}$$

where V_{max} is the maximum value the frame processing can achieve. For simplicity, $V_{max} = 1$. The value decays as frame processing startup latency increases. The speed of the decay depends on τ. The higher τ, the bigger $V_{frame}(i)$ drops per time unit. In other words, VA with high τ has a stricter deadline and requires a higher frame processing speed. Figure 6.3 shows how value decreases as latency increases. Here, τ is chosen to drop the value by half per minute. $V_{frame}(i)$ are aggregated to compute *burst value*:

$$BurstValue = \sum_{i=1}^{D} V_{frame}(i) = V_{max}\sum_{i=1}^{D} e^{-\frac{l(i)}{\tau}} \tag{6.2}$$

We use $BurstValue$ to evaluate the efficiency of VA implementations. Notations are summarized in Table 6.1.

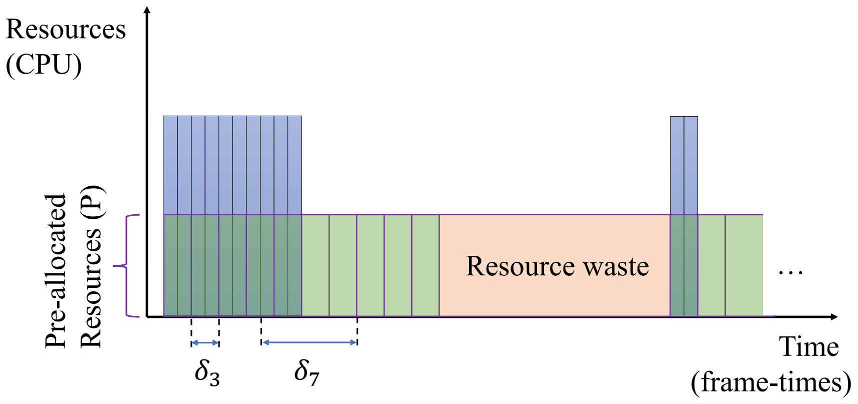


Figure 6.4: Deploying VA with pre-allocation

6.1.1.2 Pre-allocation

Pre-allocation is the most simple yet common approach. The application developer estimates the maximum resources needed when the in-depth analysis is triggered and allocates that amount statically at deployment time.

Based on the analytical model, to maximize $BurstValue$, we need the video processing delay $\delta_i = 0$ for all i. This is achieved if the pre-allocation is big enough that allows the deep analysis on a single frame to complete *just before* the next frame releases. Based on the burstiness model, let P be the amount of pre-allocation resources. Assume that VA processes frames in FIFO fashion, then the processing startup latency for frame F_i can be expressed as

$$\delta_i = \frac{i \cdot H}{P} - i \tag{6.3}$$

Clearly, the later F_i arrives (i.e., i is large), the longer it has to wait to be processed (i.e., δ_i gets larger). We visualize this effect in Figure 6.4 where $P = 0.5H$. We can see that F_7 arrives four frame-time after F_3 and have its latency 3.5x longer ($\delta_7 = 7$ frame-time while $\delta_2 = 2$ frame-time). This happens because the computation allocation P is fixed, but during burst periods, computation demand keeps increasing as more burst frames arrive. This means that once a frame processing is blocked by its preceded ones, such blocking

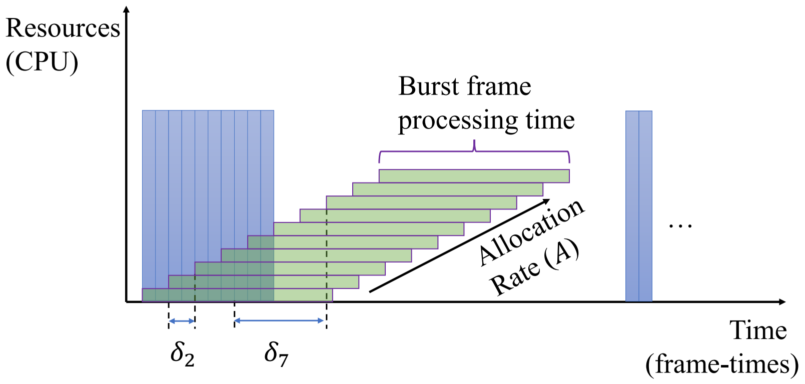


Figure 6.5: Deploying VA with FaaS and RBAM

will cascade to its succeeding ones. This makes the processing latency accumulate and get worse as the burst lasts longer. However, we can eliminate this effect if we allocate sufficient resources to completely absorb each burst frame before the next release. This can be done by having $\delta_i = 0$ for all i, which, by equation 6.3, gives us

$$P = H \tag{6.4}$$

However, recall that burst frames do not always release. During non-burst periods, which last for $1 - \lambda \cdot D$ fraction of the time, the computation demand is zero, and the H pre-allocation is wasted (Figure 6.4). That is the pre-allocation cost:

$$Cost_{prealloc} = H(1 - \lambda \cdot D) \tag{6.5}$$

6.1.1.3 FaaS

In the FaaS implementation, we realize the VA as a single FaaS function deployed by a best-effort FaaS system in the cloud. When the in-depth analysis is triggered, distributed cameras send high-resolution video frames to the cloud, each requesting a FaaS invocation to process the video frame content (Figure 6.5). Since burst frames are processed by on-demand FaaS invocations, and these invocations are terminated right after completion, the cost incurred by FaaS is the actual resource use of VA. Thus,

$$Cost_{FaaS} = 0 \tag{6.6}$$

Meanwhile, the startup latency is determined by how fast the FaaS system starts new invocations in response to their releases. Let A be the number of invocations started per frame-time. We assume each invocation is allocated with 1 CPU; thus, A is also the resource allocation rate, and each frame needs H frame time to complete once it gets started. Given A, the frame processing startup latency, δ_i, becomes

$$\delta_i = i \cdot \min[0, \frac{1}{A} - 1] \tag{6.7}$$

Different from pre-allocation, the latency δ_i is not accumulated because, in the FaaS model, each invocation is treated independently. To maximize $BurstValue$, we need $A = 1$; that is, the invocation rate must be equal to the frame release rate (i.e., the frame-per-second). However, because FaaS starts invocations in a best-effort manner, the invocation rate is not guaranteed. Thus, in the worst case, $A = 0$ during burst period, and by Equation 6.7, $\delta_i = \infty$ for all i resulting in $BurstValue = 0$.

Takeaway. (VA Deployment Challenges) Current approaches are insufficient for VA deployment. They are either costly (pre-allocation) or fail to guarantee real-time performance (FaaS).

6.1.2 RBAM-based Implementation

6.1.2.1 Video Analytics on RBAM

Now, let us use RBAM to resolve the challenges left by pre-allocation and FaaS. By definition, RBAM is FaaS with a guaranteed invocation rate. Thus, by the analytical modeling, we can deploy the VA on RBAM similarly to FaaS. The only difference is the rate guarantee configuration. With RBAM, the invocation rate A is guaranteed, and from Equation 6.7, by configuring the rate-guarantee $A = 1$, we ensure invocation will start at the same rate at video frame releases, ensuring zero frame processing startup latency, and thus, maximizing the $BurstValue$. Further, because RBAM invocations are used on burst demand, it has a similar cost to FaaS:

$$Cost_{RBAM} = 0 \tag{6.8}$$

With these results, we can see that RBAM can ensure real-time performance at pay-as-you-cost and successfully address all limitations of the current approach. This makes RBAM an appropriate solution for VA deployment.

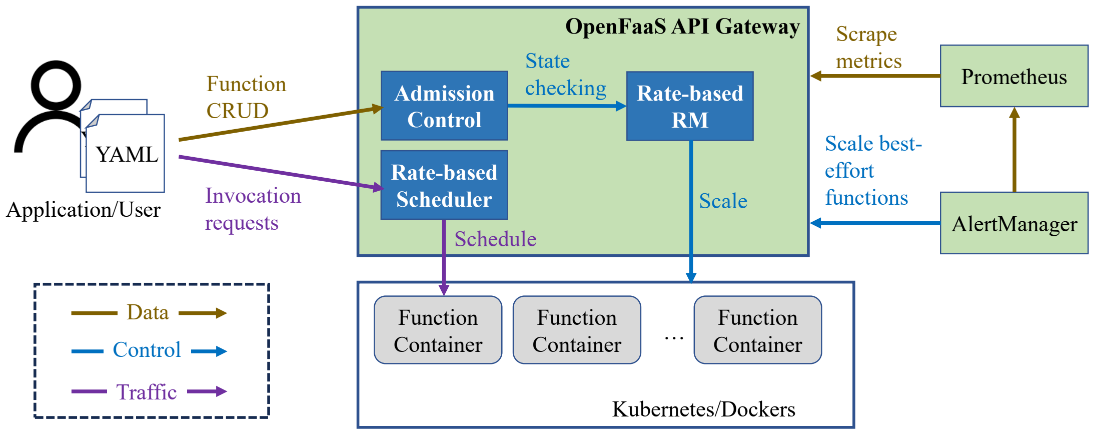


Figure 6.6: Real-time Serverless Implementation derived from OpenFaaS. Blue modules are added for rate-guarantee support.

6.1.2.2 Real-time Serverless

To demonstrate RBAM capabilities with practical workloads, we have built an RBAM prototype named Real-time Serverless (RTS). RTS is implemented based on OpenFaaS – an open-source FaaS system. The system design is depicted in Figure 6.6 following the proposed RBAM architecture in Figure 5.6 (see Section 5.4) with many components inherited from OpenFaaS [240].

Besides the rate-based scheduler and resource manager in the original RBAM architecture, we also developed an *admission control* to support FaaS function deployment and modification. All FaaS deployment submissions have to go through admission control before being deployed and executed. The admission control can reject deployment requests if there are insufficient resources to realize the rate guarantee. When the user submits an RBAM function f_i, RTS first extracts its guaranteed invocation rate A_i and maximum runtime E_i, then triggers admission control to check if there are sufficient resources for deployment. The admission control has the rate-based resource manager try allocating a quantity of resources high enough to meet the rate guarantee. If the allocation succeeds, the rate-based resource manager will hold the resources for the function f_i, and the admission control returns deployment success. If the invocation fails, the admission control lets the rate-based resource manager roll back the allocation and rejects the deployment request.

Takeaway. (VA on RBAM) Implementing VA tasks with RBAM functions allows us to maximize application value at minimum resource costs.

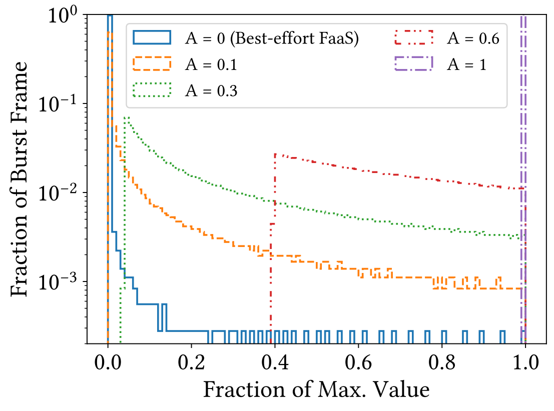
6.1.3 Evaluation

6.1.3.1 Methodology

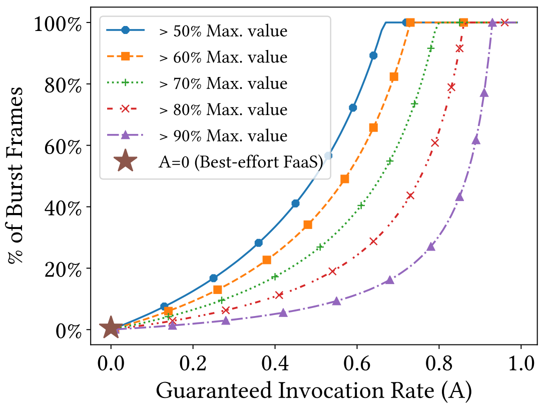
Workloads We evaluate VA deployments using synthetic workloads generated with the following burstiness configurations

- *Burst arrival rate* follows a Poisson process with average rate $\lambda = 0.3/hour$.
- *Burst duration* D (e.g., number of video frames per burst) varies according to Gaussian distribution with mean $D = 2$ min.
- *Computation demand* H: 30 CPUs
- $\tau = 2,607$ – value decays 1/2 per minute

Approaches We evaluate RBAM deployments with various guaranteed invocation rates A. For the special case $A = 0$, RBAM is equivalent to best-effort FaaS, which gives the application no guarantee to obtain additional invocations rather than those 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 this quantity is not enough during a burst.



(a) Distribution of per-frame value



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

Figure 6.7: Per-frame guaranteed value at various guaranteed invocation rates (A)

6.1.3.2 Real-time Processing Efficiency

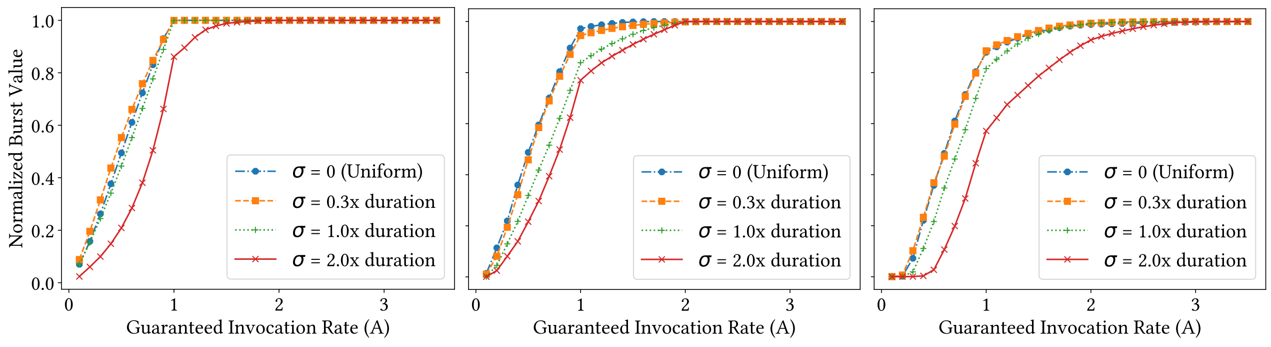
Figure 6.7a shows the distribution of per-frame guaranteed value using RBAM at different guaranteed invocation rates A. For the baseline, at $A = 0$ (i.e., equivalent to best-effort

FaaS), 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 invocation 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 $A = 0.6$, the application can ensure that all frames exceed 40% of the maximum value. As A increases towards 1 (i.e., the video frame release rate), a growing frame value guarantee can be achieved, reaching 100%. This illustrates that 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 6.7b. 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 frame release rate. To achieve 50% of the maximum value for 100% of the frames, an $A = 0.67$ is needed. At the high end, to achieve 90% of the maximum value, $A = 0.85$ is required for 50% of the frames, and 0.9 for 100% of the requests. At $A = 1$, the application can deliver 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 can meet *any* quality target. Such capability unlocks rational designs that open more space for applications to operate and exploit resources more efficiently.

The results show that the guaranteed invocation rate is the critical enabler of high value. By configuring RBAM guaranteed invocation rate A to match burst demand, the application is guaranteed to meet its computation deadline with zero value degradation. Furthermore, even at $A < 1$, RBAM can still guarantee a fraction of application proportional to A. 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.

Takeaway. (Real-time Processing Efficiency) RBAM enables bursty, real-time applications to configure for high computation value.



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

Figure 6.8: Burst value vs. Guaranteed Invocation Rate (A) with varied burst duration standard deviation (σ)

6.1.3.3 Robustness

Next, we evaluate RBAM against workload burstiness by varying burstiness properties, one by one. The goal is to demonstrate its applicability, showing that RBAM can be configured for a wide range of workload burstiness, making it a good candidate for various applications and workloads. Further, we will show that when reconfiguration is required, the changes are insignificant compared to the changes in workload burstiness, making RBAM a stable, reliable option for bursty, real-time applications.

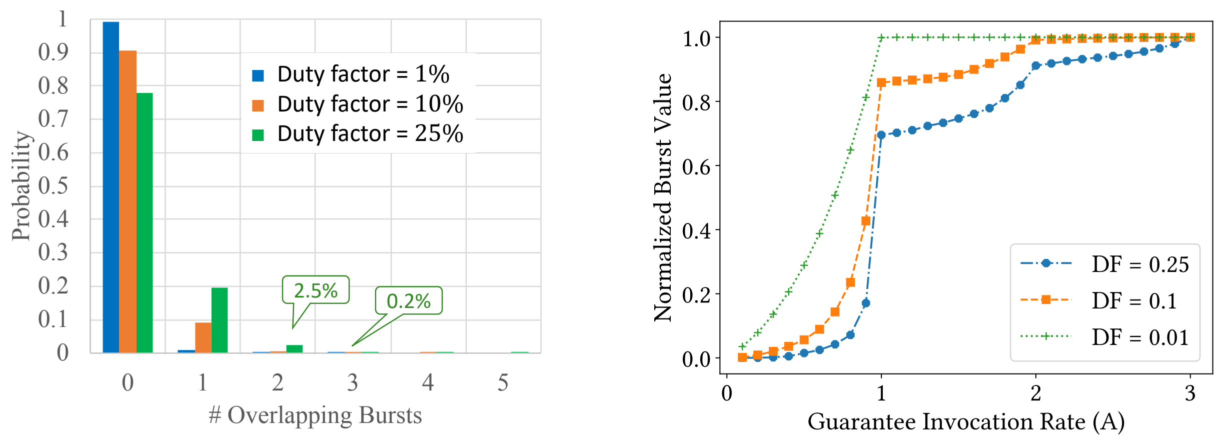
Robustness against Burst Duration Variability we vary burst duration (D) by generating workloads at different duration standard deviations (σ). Figure 6.8 shows the achievable guaranteed burst value at different guaranteed invocation rates normalized by the maximum burst value in three duty factor scenarios: high ($DF = 0.25$), medium ($DF = 0.1$), and low ($DF = 0.01$) where DF is burst duty factor, defined as

$$DF = \lambda \cdot D \tag{6.9}$$

At low duty factor, changing σ does not cause many effects on guaranteed burst value (Figure 6.8a). 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 $A = 1$ (Figure 6.8b), and if duty factor jumps to $DF = 0.25$, the loss increases to 19%. If σ is doubled to 2x duration, the loss is 2.2x to around 42% (Figure 6.8c). However, value reduction can be solved by simply increasing the guaranteed invocation rate. For example, $A = 2$ invocation/frame-time 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 $A = 3$ invocation/frame-time is sufficient. Also, note that burst variability only takes effect at high duty factor, but in response, only a small invocation 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 a 3x guarantee invocation rate increment (i.e., 3x resource commitment increase, See Section 5.6.2). 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 occurring concurrently.

Figure 6.9a shows the probability of having one, two, and more bursts simultaneously under different duty factors. There are two important observations from the figure. First,



(a) Probability of number of burst overlapping (b) Burst value vs. guaranteed invocation rate.

Figure 6.9: Burst interference probability and achievable guaranteed burst value at different duty factor (varying λ)

the high duty factor increases the chance of burst interference, as we explained above. For example, at duty factor $DF = 1\%$, the probability of having a two-burst interference is only 0.005% while $DF = 25\%$ increases the chance 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 $DF = 0.01$, 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 have 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.

Figure 6.9b shows guaranteed burst value at various duty factors. Note that at burst interference, demand is doubled, tripled, or more depending on the number of bursts involved. This means to saturate double burst interference, the application needs to allocate resources 2x faster, for triple burst interference, 3x invocation rate is required, and so on. Thus, in the figure, the breaks of curves at $A = 1$ and $A = 2$ 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 invocation rates can increase burst value toward the potential maximum.

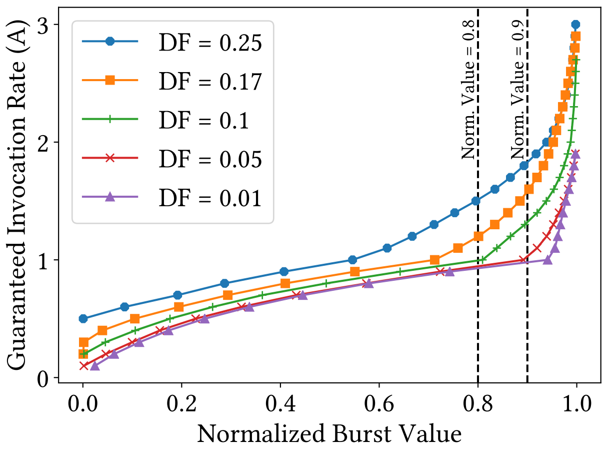


Figure 6.10: Invocation rate needed to achieve burst value fraction (at varied duty factors).

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 6.10. 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 requires a 2x increase in A, resulting in only 2x more cloud resource commitment.

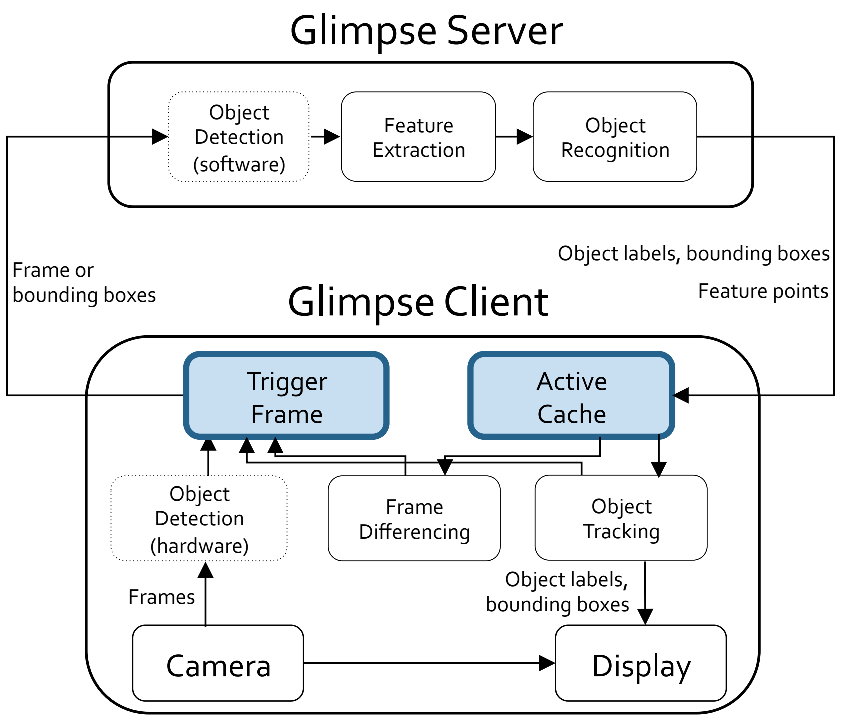


Figure 6.11: Glimpse System Architecture (from [97]).

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 invocation 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 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.

Takeaway. (Robustness) RBAM deployment is robust against burstiness properties and requires sublinear cost in response to burstiness changes.

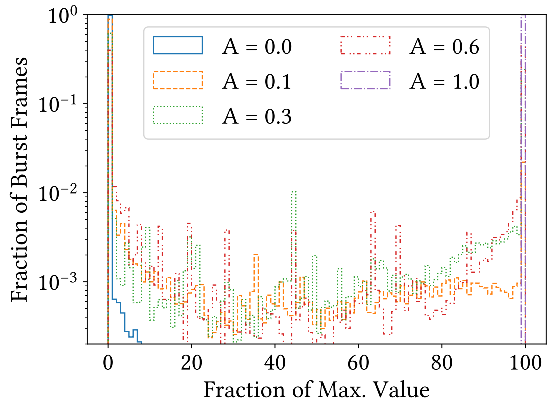
	Burst Duration (frames)				Burst Height			
	Mean	StdDev	Min	Max	Mean	StdDev	Min	Max
Night	116	186	30	2,445	21	3	20	80
Day	120	216	30	2,323	20	3	20	80
Rush hours	917	1293	30	7,464	48	23	20	200
Overall	197	503	30	7,464	24	11	20	200

Table 6.2: Burst Statistics for Traffic Video

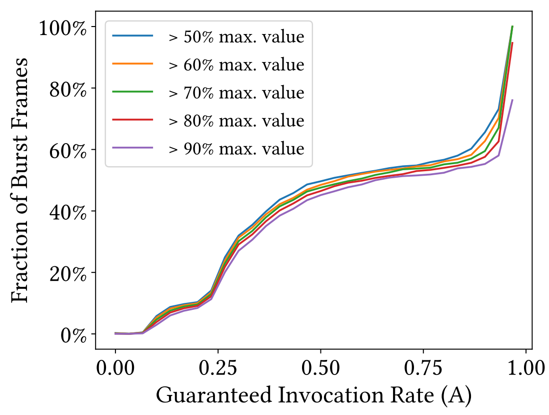
6.1.3.4 Case Study: Traffic Intersection Monitoring

Next, let us demonstrate the capability of RBAM mentioned above through a realistic application using Real-time Serverless. 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 [97] (see Figure 6.11). 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 this ratio could be much higher for richer analytics. We use a rush-hour traffic video captured by a traffic camera in Southampton, NY, at the intersection of County Rd. 39A and North Sea Rd. and available from [310]. The video is 30 frames per second at a resolution of 1920 $\times$ 1080 color pixels per frame.

Because we are interested in analyzing complex behavior such as erratic driving, reckless walking, or traffic incidents, we use an efficient model [160] with ResNet trained on KITTI dataset[139]) 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 6.2). 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 the 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



(a) Per-frame value Distribution



(b) Value Distribution vs. Guaranteed Invocation Rate

Figure 6.12: Per-Frame Value achieved by Application varying Guaranteed Invocation Rate

of objects present in each frame multiplied by 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 6.2. The burst durations are much shorter than those explored in the previous experiments, with an average burst duration of 7 seconds (210 frame-times). Moreover, both the burst duration and height are highly variable within each part of the day.

Figure 6.12 shows the value distribution of the video trace; while similar qualitatively to Figures 6.7a the real burst trace is much noisier. The traffic analysis quality benefits from increasing A are shown in Figure 6.12b. 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, all the nighttime value is captured. At A of 0.9, the daytime value is captured. Figure 6.12b shows 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 $A = 1$.

Results from Figure 6.12 confirm that RBAM guarantees the traffic monitoring application value. By increasing A, the application can improve the guaranteed quality, although

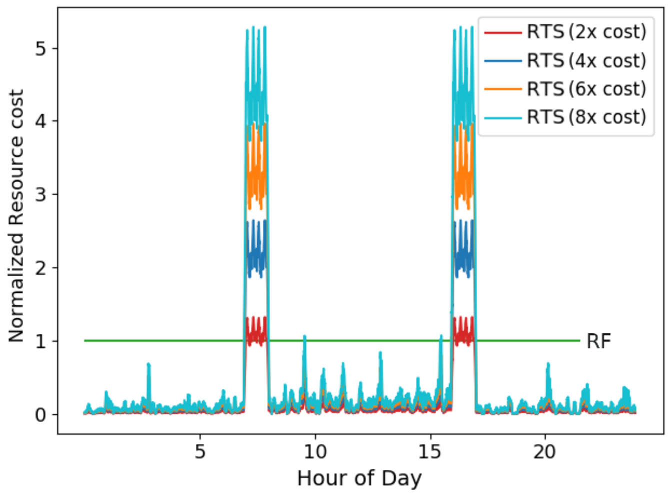


Figure 6.13: Traffic Monitoring application cost for varied RTS cost scenarios (5-minute sliding average)

compared to the synthetic data, quality increases much slower due to the extremely high duty factor during rush hours. Furthermore, at $A = 1$, 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 6.13 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 pre-allocation.

Takeaway. **(Case Study)** The case study confirms the RBAM real-time guarantee capability and cost-effectiveness.

6.2 Stream Processing

In recent years, the use of stream processing models (e.g., [339, 17, 22]) has increased [38]. This rise in popularity is driven by the need to analyze and act upon the ever-increasing data in real-time, ensuring competitiveness in today's fast-paced digital landscape [182, 135]. The stream processing model enables performing real-time analytical tasks efficiently and scalably. The model treats input streams as flows of separate *tuples* and organizes applications as Directed Acyclic Graphs (DAGs), called *workflow*, consisting of *operators* placed on distributed computing nodes. Immediately after creation, tuples are taken through the workflow and processed by their operators in an on-the-fly fashion, delivering analytical results with low latency. Also, each operator can run multiple copies concurrently to exploit the hardware parallelism capability, easing high-throughput computation [112, 324]. The resulting capabilities make stream processing an essential paradigm for data analysis at scales from smart homes [50] to large-scale industries [257].

Thus, many Stream Processing Engines (SPEs) have been proposed to automate workflow description, deployments, and operation efficiently. Many are pure, general SPEs and act as a building block for larger data analysis systems [22, 17, 269]. Meanwhile, others are customized for specific infrastructures [98], applications [305], or workloads [237, 307, 214].

6.2.1 Challenges

Modern SPEs deploy stream processing workflows by mapping operators onto *workers* – a computation abstraction provided by the underlying resource manager for efficient hardware exploitation (Figure 6.14). Popular choices of worker abstraction are threads, processes, and containers. With all computation handled by operators, operator-to-worker mapping is crucial to workflow performance. To deal with varied operator complexity, SPE assigns to each operator a *parallelism configuration*, essentially the number of the operator's copies that can execute concurrently. SPE allocates a corresponding number of workers, each to

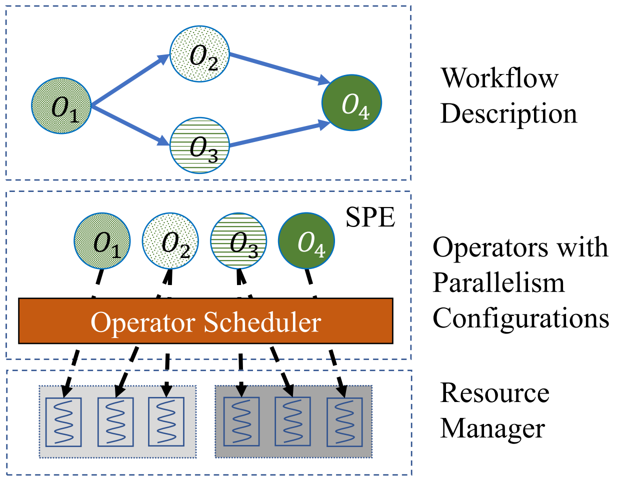


Figure 6.14: Worker-based SPE Architecture. Operators are mapped onto workers across multiple machines. The parallelism configuration specifies high-cost operators mapped onto multiple workers for efficiency.

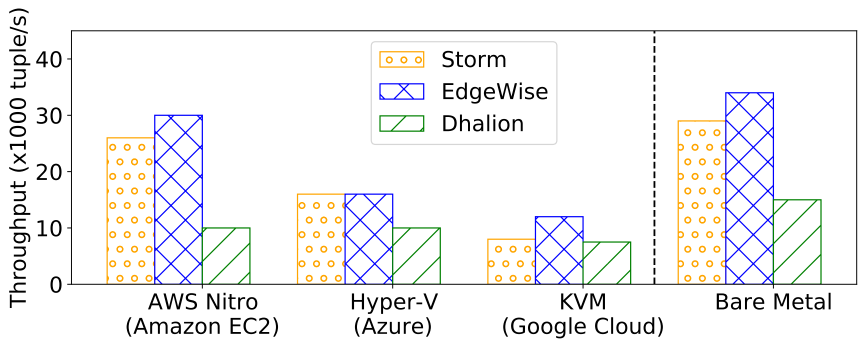


Figure 6.15: Worker-based SPEs have poor performance transparency as throughput greatly varies across different resource configurations.

run an operator copy, and distributes them across its cloud resources. For example, in Figure 6.14, O_2 and O_3 are compute-intensive operators, so they have their parallelism set to two, resulting in two copies and getting two workers, while O_1 and O_3 only have one. This configuration creates six operators, which need an allocation of six workers distributed over two machines. One hosts O_1 and O_2, and another hosts O_3 and O_4.

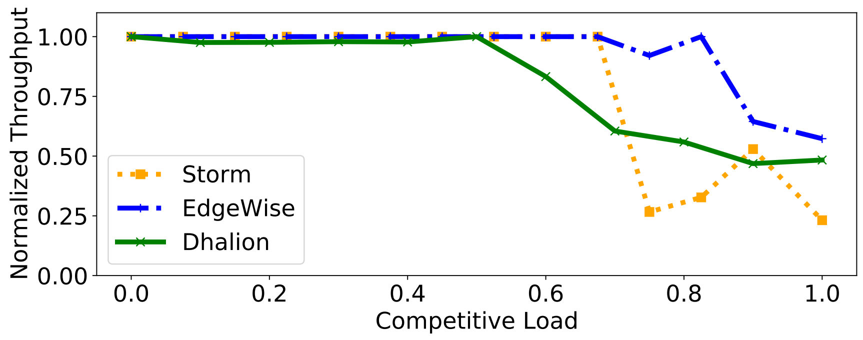


Figure 6.16: Worker-based SPEs have poor predictability as throughput is interfered with a competitive load.

6.2.1.1 Performance Transparency Challenges

Worker-based SPEs tie workflow performance to underlying worker resource configuration. Because these details are not part of the application abstraction, performance is not transparent. Figure 6.15 shows the maximum throughput of executing an ETL workflow on a 4-core machine deployed by three worker-based SPEs: Storm [22], EdgeWise [136], and Dhalion [131] (see Section 6.2.3). We try four machine configurations (Section 6.2.3): one is bare metal while the others are VMs provisioned by different hypervisors while sticking with only one parallelism configuration (Figure 6.20a). The throughput is extremely sensitive to resource configurations, with performance varying as much as 3-fold. For example, Storm running on KVM gets only 27% of bare metal throughput.

The results illustrate that workflow performance is not transparent yet strongly depends on the hardware resource configurations to which workers have access. Hence, no one-size-fits-all workflow configuration can be used for every deployment. Instead, the SPE has to understand the underlying resource configurations and reconfigure workflow accordingly to maintain good performance.

6.2.1.2 Performance Predictability Challenges

Most distributed systems operate in shared environments (e.g., the cloud and edge). This means that workers are often collocated with other applications, and the resources allocated to them can vary significantly depending on these applications' behaviors. This variability, coupled with the transparency challenge, further complicates the performance predictability for SPE applications. In Figure 6.16, we demonstrate this by plotting the throughput of an ETL workflow normalized by its input rate when collocated with an aggressive competitive load on a single Azure VM. As we gradually increase the computation demand of the competitive load, we observe a significant drop in ETL's throughput after the competitive load exceeds 70%.

The results illustrate that workflow performance is tied to its collocated applications. As a result, workflow performance is hard to predict. One deployment that works well may become ineffective when some surrounding applications change. Unfortunately, these changes are typically out of SPEs' control, making performance predictability challenging.

6.2.1.3 Implications for Applications

Application developers compensate for poor **performance transparency and predictability** by over-provisioning, wasting resources. A better approach is to repeatedly reconfigure workflow for any significant change to the environment or application until performance meets the desired level [269, 112, 131]. However, this only works if the SPE reacts properly to the change. Failing to select an appropriate configuration would result in multiple rounds of reconfiguration, causing performance instability or over-provisioning.

Another implication of poor performance transparency and predictability is difficulty in changing workflow configuration (e.g., migrating workers from one machine to another). Such application reconfiguration can be desirable to manage cost, adjust to load dynamically, or move to other resources in response to outage, preemption, or perhaps power cost. For these

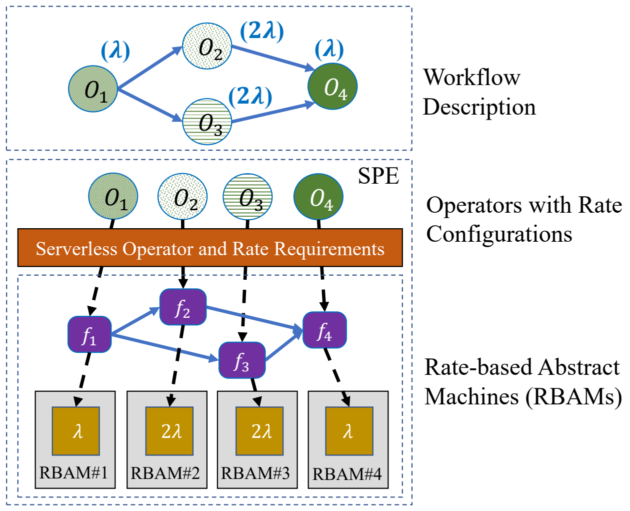


Figure 6.17: The RBAM approach to stream-processing: Operators are wrapped by FaaS functions, providing invocation-level dynamic resource management. One RBAM for each FaaS function ensures its required tuple processing rate.

reasons, most SPEs do not even support multi-site execution. For example, the design of a workflow deployment that spans two data centers or datacenter and the edge is a bespoke, manual activity [214, 237].

More directly, the above challenges make deploying a workflow over multiple data centers tricky; many manual efforts are required for each configuration. Worse, flexible reconfiguration across cloud and edge – a signature challenge for many applications – is difficult. In the edge's dramatically more complex environment of heterogeneous resources and networks, manual configuration and tuning may be impossible.

Takeaway. (Challenges) Worker-based SPEs rely on the underlying worker implementation for performance, which limits the performance transparency and predictability of stream processing applications, making it difficult for them to configure for performance.

	Worker Model	**RBAM**
Service Model	Continuous	Discrete (invocation)
Allocation	Static	Dynamic
Guarantee	None	Rate (invocations/sec)

Table 6.3: Worker-model and RBAM Comparison

6.2.2 RBAM-based Implementations

6.2.2.1 Stream Processing on RBAM

We resolve the performance challenges by replacing the worker model with FaaS invocations, as shown in Figure 6.17. Operators are implemented as FaaS functions, and the topology is encoded as FaaS invocation chains, with tuples passed as function arguments. For example, operators O_1, O_2, O_3, O_4 becomes separate FaaS functions f_1, f_2, f_3, and f_4, respectively. The SPE handles each arriving tuple by first invoking f_1. After completion, f_1 triggers f_2 and f_3 with the output embedded inside their invocation requests. These invocations extract f_1's output from the requests, process it, and then pass their results downstream until reaching the sink operator.

Regular FaaS performance is best-effort. Invocation allocation may fail or get delayed, degrading workflow performance. To workaround, we let application developers configure per-operator rate requirements, specifying the expected processing rate of these operators after deployment. Once a workflow is submitted, rate requirements are tied to their corresponding FaaS functions; each provisioned in a Rate-based Abstraction Machine (RBAM). As in Figure 6.17, f_1's rate requirement is λ, equal to the input rate of its operator, O_1. Rate configurations are equivalent to worker-based parallelism configurations yet are easier to determine by measuring input rate and can be guaranteed through RBAM, enabling a simpler, straightforward way to specify, configure, and evaluate workflow performance.

More precisely, The RBAM abstraction departs from workers in many important ways (Table 6.3):

- *Invocation-level Resource Management*: In contrast to worker abstraction's *continuous* resource access, RBAM lets applications access resources through invocations, a *discrete* notion in time and resources. This model naturally matches the stream processing workload, which is also determined by discrete tuple arrivals.

- *Dynamic Allocation*: RBAM scales invocation allocation dynamically to tuple arrival and automatically releases them after finishing processing. This scheme supports both dynamic scaling and low resource waste. In comparison, worker allocation is rather static, as one worker often represents a fixed set of resources.

- *Guaranteed Rate*: while worker allocation offers no guarantee, RBAM allocation supports a guaranteed invocation rate that enables robust, simple QoS reasoning.

By setting each RBAM rate guarantee to match the operator processing rate, the SPE guarantees the availability of resources to process tuples at the arrival rate, maintaining the desired performance. This rate configuration is independent of any underlying resource configuration, so the SPE application has full performance transparency.

RBAMs also support performance predictability: RBAM allocations ensure their operators perform well against any load whose input rate is smaller or equal to the rate guarantee. Consequently, a workflow constructed from these operators also has a performance guaranteed up to a specific input rate. This performance predictability enables simple tuple rate comparisons and negotiation with the underlying RBAM systems to determine if a new configuration is feasible. This framework enables distributed SPE configuration management for stable performance possible.

6.2.2.2 Storm-RTS: SPE for Distributed Stream Processing

We propose Storm-RTS, a new distributed SPE that translates workflow description into RBAM to achieve performance transparency and predictability. This enables it to flexibly spread stream-processing applications over multiple machines across multi-datacenters from Cloud to Edge. We describe the design of Storm-RTS to demonstrate this new capability.

Design Requirements Storm-RTS is derived from Storm and reuses its workflow models to offer essential features of a modern SPE. However, mapping Storm's workflow model to RBAM abstraction is not straightforward. First, many essential FaaS configurations, such as time limit, cannot be inferred directly from Storm workflow configurations. Second, FaaS functions are highly modular and stateless. Storm, like other worker-based SPEs, collocates workers for efficiency and maintains operators' state for various functionalities, such as consistency and fault recovery. Naively replacing Storm's workflow executor with FaaS invocations would reduce efficiency and leave some features infeasible (e.g., stateful operators). We work around these issues by meeting the following requirements.

- *Workflow performance stability*: achieve desired throughput and latency across distributed configurations, reconfiguration (migration), and varied competitive loads.
- *Predictable Resource Requirements*: operators and workflows characterized for their resource requirements.
- *Modular resource management*: can partition workflow across multiple sites/data centers. Individual site resource managers can independently decide if a workflow can be placed and meet its performance requirements.
- *Compatibility*: support Storm workflows and features with similar efficiency and modest change.

Storm-RTS Architecture The key elements of Storm-RTS are shown in Figure 6.18. At the high level, Storm-RTS has four main components, each responsible for one of the design requirements listed above.

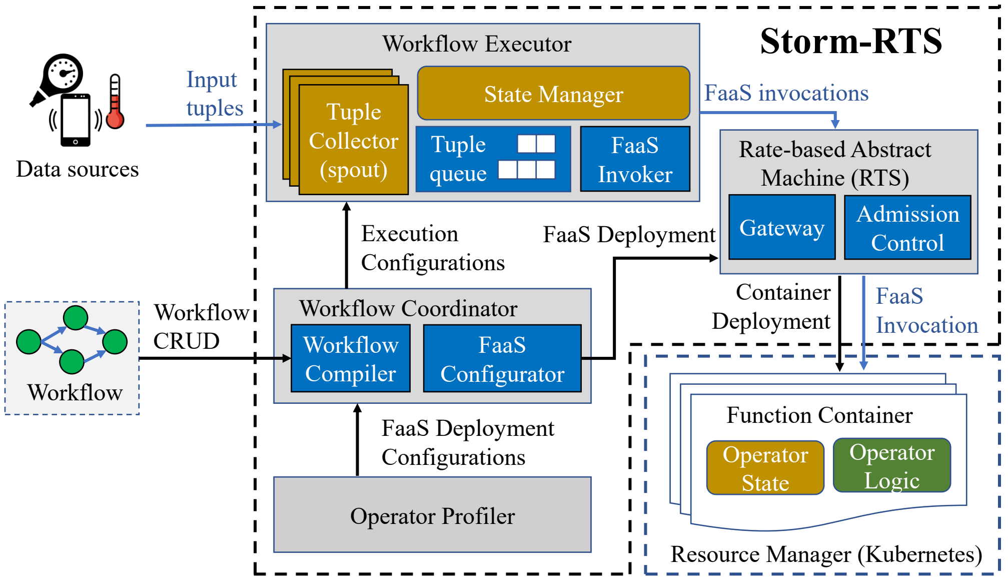


Figure 6.18: Storm-RTS Architecture: Operator Profiler, Workflow Coordinator, Executor, and Rate-based Abstract Machine

- **Workflow Coordinator**: responsible for enforcing performance stability. It translates workflow operators received from developers into FaaS functions and associates them with appropriate configurations, allowing the workflow to sustain the desired load. It is also responsible for protecting workflow performance from disruptions such as competitive loads, workflow reconfiguration, migration, etc.

- **Operator Profiler**: responsible for resource requirement predictability. The component runs workflow operators offline to profile their computing and memory requirements. This information configures 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 the `workflow coordinator` are deployed separately inside RBAM allocations. Once established, each RBAM allocation ensures new invocations are executed at the configured rate independent of each other, underlying resource configuration, and other competitive loads.

- **Workflow Executor**: responsible for executing workflows and compatibility supports. It collects tuples from data sources and then triggers corresponding FaaS invocations to start workflow execution. The `workflow executor` also reuses Storm's monitor and orchestration modules to offer similar data processing support as Storm.

Next, let us describe how these components are glued to perform workflow deployment, execution, and state management.

Workflow Deployment Workflow developers submit workflow descriptions directly to the `workflow 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 rate scales* μ_i representing the ratio of each operator's expected input rate and the desired rate. The `workflow compiler` extracts operators' logic from workflow topology and then encapsulates them inside FaaS functions.

Each FaaS function f_i has the `Operator Profile` determine its (i) per-invocation resource requirement s_i (mainly CPU and memory), (ii) time limit E_i (i.e., timeout), and (iii) batch size b_i (i.e., number of tuples processed per invocation). This is done 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.

- *Per-invocation resource (CPU and memory) requirement (s_i) and time limit (E_i)*: the profiler executes operators starting with excess resource allocation and gradually reduces the allocation until observing a 20% execution time increase. This last allocation configuration and the corresponding execution time are used to configure the FaaS wrapping the operator.

- *Batch size (b_i)*: Since invocation overhead is typically much higher than tuple processing latency (a couple of milliseconds vs. <1ms), Storm-RTS batches multiple tuples in one invocation to amortize the overhead. However, this prolongs per-tuple processing latency. To mitigate this effect, the profiler compares naive operator execution versus FaaS, varying the batch size, and considers the batch size leading to an efficiency of 70% is acceptable and used to determine the batch size for this operator.

With the information, the `workflow coordinator` configures per-function rate guarantee A_i to the number of invocations expected to invoke per second if tuples are generated at the desired rate λ:

$$A_i = \frac{\lambda \cdot \mu_i}{b_i} \tag{6.10}$$

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. After determining the above information for all FaaS functions, the `FaaS Configurator` sends these functions and their configurations to RBAM to check whether the underlying resource manager can support their guarantee and wrap FaaS functions inside RBAMs with corresponding rate guarantees. If the process completes successfully, the desired rate is guaranteed so the `workflow executor` is triggered to begin execution, no further reconfiguration/profiling is needed.

RBAM Deployments We leverage the the RBAM prototype – Real-time Serverless (RTS) – introduced in Section 6.1.2 to deploy every FaaS function deployments requested by the `workflow coordinator`. For each FaaS deployment f_i, the RTS collects its guaranteed invocation rate A_i and time limit E_i calculated above and creates corresponding RBAM deployment over the Kubernetes resource management.

Worfklow Execution and State Management For each successful workflow 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. New tuples are put into `tuple queues` by destination until their number is sufficient to form a batch. A `FaaS invoker` retrieves a batch from the queue and requests a new FaaS invocation for the appropriate workflow operator, passing 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 FaaS function chains which are self-synchronized and do not need any dedicated messaging systems as in worker-based SPEs (e.g., Storm [22] relies on Netty [35] for inter-node messaging).

Storm-RTS provides equivalent state management and functionalities to Storm, including stateful supports, exactly-once processing, out-of-order events, etc. Since serverless invocations are stateless, we have to modify RTS to embed an in-memory store called `operator state` inside each FaaS container to maintain operator state (e.g., join keys), consistency, progress tracking, monitoring, and recovery. This information is updated every time a tuple completes processing and periodically synchronized with a centric `state manager`. We reuse Storm modules to implement both `operator states` and the `state manager`, ensuring the state information is handled properly and tuples are sent to FaaS containers in a correct order and meet users' desired semantics.

6.2.2.3 Multi-site Deployment with Storm-RTS

Distributing workflow execution across multiple sites (e.g., cloud-cloud and cloud-edge) is challenging because distributed resources are heterogeneous and can vary in availability. The Storm-RTS design brings new capabilities to address both issues. First, Storm-RTS accesses underlying resources through FaaS abstraction, so as long as underlying systems support FaaS, Storm-RTS can mask heterogeneity via FaaS and assure performance via op-

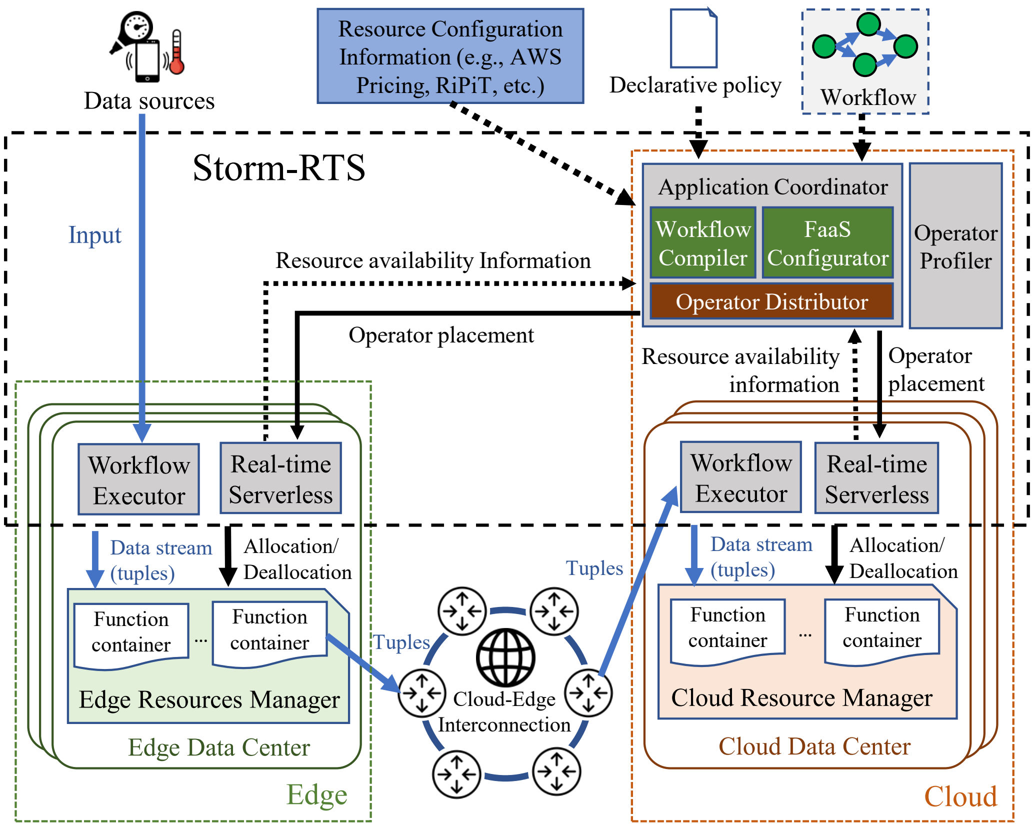


Figure 6.19: Storm-RTS for Multi-site Deployment: the application coordinator manages multi-site deployment. As before, each site has admission control and performance monitoring that implements the local RBAM guarantees.

erator profiling and RTS guarantee enforcement. Second, resources with varying availability may require workflow reconfiguration. By leveraging the RBAM, Storm-RTS ensures that such reconfiguration will not affect workflow performance, enabling applications to optimize their deployments for cost, carbon, or other criteria. To illustrate this capability, we extend Storm-RTS architecture as shown in Figure 6.19. Each cloud or edge data center runs Storm-RTS as in Figure 6.18, but now the `workflow coordinator` is promoted to `application coordinator` to orchestrate FaaS deployments across the data centers. Apart from the original components, the `application coordinator` adds an `operator distributor` that places the FaaS-encapsulated operators across data centers, implementing the desired application policy.

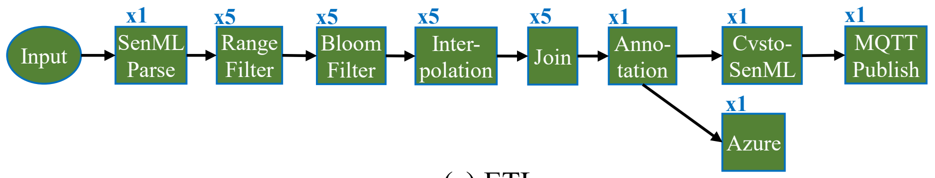
Typical policies include keeping operators close to data sources (often at the edge). If multiple data centers can host an operator, the coordinator implements the application's `deployment policy`, which picks application configurations from among the candidates. For example, if edge resources are zero-cost, when available, a policy that 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 sustainability is the objective, then the `application coordinator` might push operators to the edge when solar panels create plentiful green power but back to the cloud data center 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 pricing, Carbon intensity information [37], etc.) to give insights into resource properties for efficient exploitation.
- *Resource availability*: collected from resource managers in data centers. The `application coordinator` also communicates with RTS systems to determine if an operator placement is feasible at any particular site.

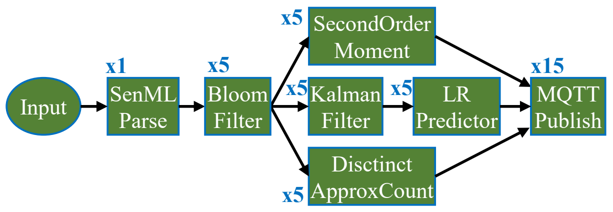
Takeaway. **(RBAM Implementation)** Replacing the worker model with FaaS invocations backed by RBAM's rate guarantees makes the SPE performance transparent and predictable. This also enables flexible stream processing across cloud and edge environments.

6.2.3 Evaluation

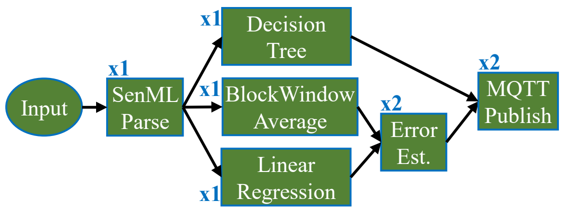
In this section, we will evaluate Storm-RTS against state-of-the-art worker-based SPEs and a FaaS-based SPE to illustrate how the RBAM's rate guarantee helps stream processing applications resolve performance transparency and predictability challenges. This enables workflow deployment over heterogeneous and distributed resources, unlocking myriad appli-



(a) ETL



(b) STAT



(c) PRED

Figure 6.20: RIoTBench workflows. Operators are shown as green boxes with numbers representing parallelism configurations.

cation flexibilities and opportunities for optimized management and simplifying distributed stream processing.

6.2.3.1 Methodology

Workloads We use the *RIoTBench* benchmark suite [286], designed specifically for evaluating SPE implementations. We select three workflows (Figure 6.20) capturing typical stream processing activities over a real-world smart cities dataset [81]: PRED (make predictions on streamed data), ETL (perform data extraction, transformation, and load), and STATS (apply statistical summarization). Their parallelism configurations are selected based on the number of tuples each operator has to process per one input tuple.

Stream-processing Engines (SPEs) We compare five SPEs, representative implementations of workflow deployment approaches discussed in Section 6.2.1 and 6.2.2.

- *Storm* [22]: Evaluation baseline. Workers are implemented as threads in a Java Virtual Machine. Worker allocation and mapping are static.

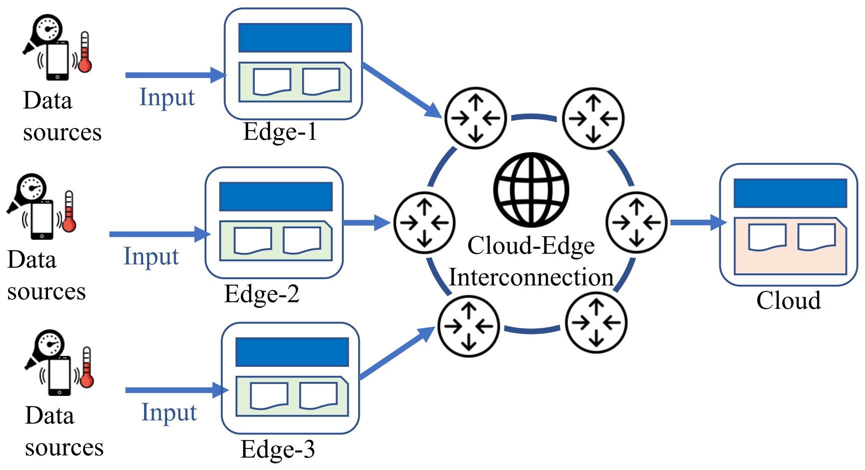


Figure 6.21: A Cloud-edge resource configuration

- *EdgeWise* [136]: a Storm variation that replaces static worker mapping with a dynamic one prioritizing operators experiencing long input queues for higher efficiency.
- *Dhalion* [131] a worker-based SPE with heuristic dynamic scaling. The SPE allocates more resources if workflow throughput fails to match the input rate and frees unused resources if the workflow is over-provisioned.
- *Storm-Serverless* implements the Storm API on FaaS. Its implementation is identical to Storm-RTS, except the RBAM is replaced with OpenFaaS [240]. Thereby, operators have no rate guarantee.
- *Storm-RTS* implements the Storm API with RBAM as described in Section 5.

In the following experiments, unless stated otherwise, worker-based SPEs use parallelism configurations shown in Figure 6.20. Storm-RTS also sets operator scale factors μ_i identical to these parallelism configurations and desired rate λ equal to the workflow input rate.

Hardware/Resource Configurations Experiments are conducted over three configurations

- *Cloud VM*: workflows are hosted by virtual machines in public clouds, including Amazon EC2 (m5zn instances), Microsoft Azure (Dasv4 instances), and Google Cloud (e2-standard instances) to evaluate SPE performance over realistic settings where they typically run over a virtual, oversubscribed environment inside data centers.

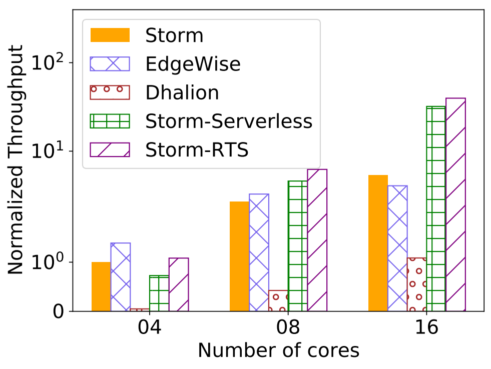
- *Bare Metal*: for raw performance measurement (no sharing). The machine has 1 Intel Xeon Gold 6138 (80 cores), 512GB RAM, and uses *cgroup* for resource control.

- *Cloud-Edge* We use Chameleon Cloud [19] to emulate the cloud-edge setting. We create four clusters (Figure 6.21) where the *cloud* emulates the cloud side with an unlimited number of machines, each having 92 cores and 192GB of memory. *edge1*, *edge2*, and *edge3* represent edge data centers. Each has 4 VMs (12 cores and 48GB memory). We configure the network based on Amazon Cloud Infrastructure's network performance [55]. All connections have 100Gbps bandwidth. Intercloud connections have 5.5ms latency while Cloud-Edge latency is randomized with Gaussian distribution with 5.5ms mean and 2ms standard deviation.

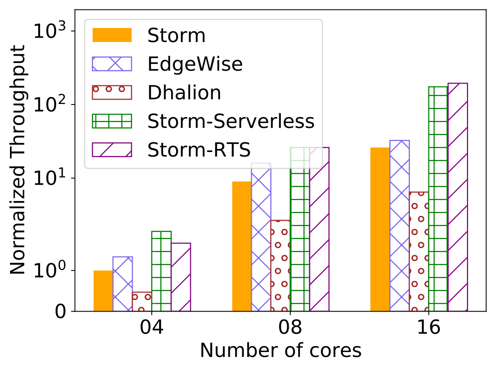
Metrics We evaluate SPEs based on *throughput* (measured at sink operators), end-to-end processing *latency*, CPU utilization (100% per core), and *cost*, measured as CPU utilization * cost-factor. The cost-factor is a dimensionless relative measure of resource cost, reflecting resource location.

6.2.3.2 Resource Efficiency

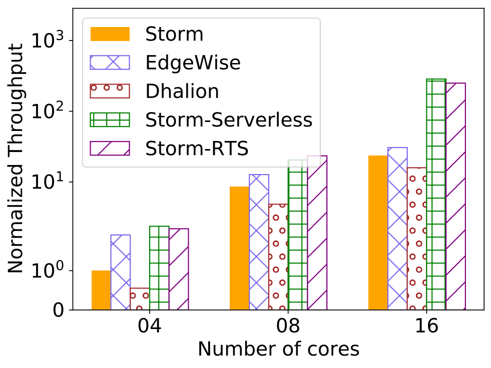
Single Machine We deploy RIoTBench workflows separately over a single machine with fixed CPUs (4, 8, and 16 cores). The workflows are fed tuples at a constant rate, and we gradually increase the rate until saturation (i.e., the tuple processing latency rises sharply, and the throughput fails to match the tuple input rate). We report the throughput just before this point, calling it the maximum throughput. We plot the geometric mean of the



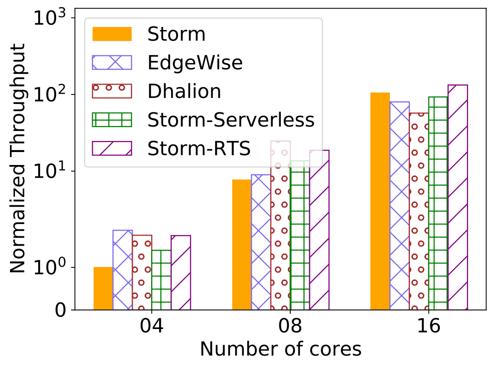
(a) AWS Nitro (Amazon EC2)



(b) Hyper-V (Azure)



(c) KVM (Google Cloud)

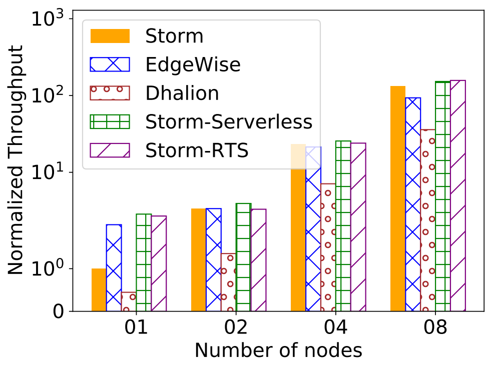


(d) Bare Metal

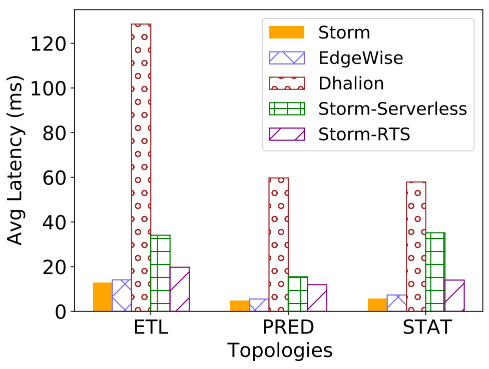
Figure 6.22: Maximum throughput of RIoTBench workflows on a single machine. The geometric mean of workflows' throughput, each is normalized by Storm throughput on a 4-core machine.

normalized maximum throughputs of three RIoTbench workflows on four different machine configurations in Figure 6.22. The performance of Storm, Edgewise, and Dhalion scales poorly, falling slightly behind Storm-Serverless and Storm-RTS at eight cores and badly behind at 16 cores. Both Storm-RTS and Storm-Serverless scale well with the system capacity, with workflow maximum throughput increasing almost linearly with the number of cores. These results are consistent across all cloud VMs and the bare metal configuration, confirming that FaaS-based SPEs can achieve equal or superior resource efficiency.

Multiple Machines We deploy workflow separately over multiple 4-core VMs and report the geometric mean normalized throughput for each SPE on Azure in Figure 6.23a. The



(a) Throughput



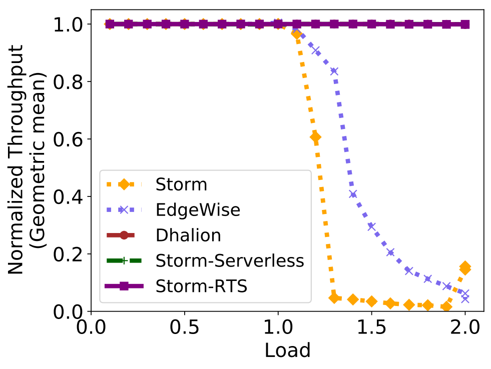
(b) Average Latency

Figure 6.23: Storm-RTS achieves comparable throughput and latency versus worker-based SPEs.

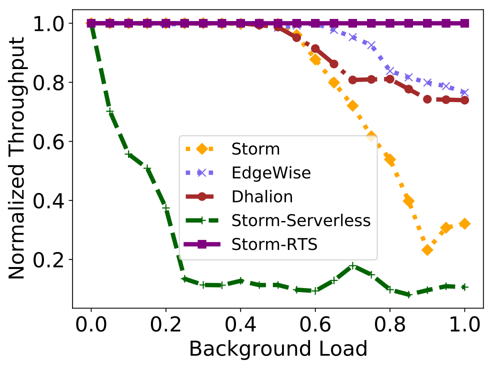
other resource configurations are omitted because their results are the same as we have presented for Azure. All SPEs have comparable performance. Both Storm-Serverless and Storm-RTS scale well, increasing throughput with more machines. This result confirms their resource efficiency compared to worker-based SPEs in a distributed computing setting.

Processing Latency Figure 6.23b 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). Compared to Storm and EdgeWise, Storm-RTS and Storm-Serverless experience higher latency due to FaaS invocation overhead. However, by batching tuples into a single invocation request, the overhead is amortized. Storm-RTS keeps the latency below 20ms, slightly above Storm and EdgeWise while significantly better than Dhalion. The results demonstrate that Storm-RTS is efficiently equivalent to other worker-based SPEs in terms of processing speed.

Takeaway. (Resource Efficiency) Storm-RTS achieves comparable performance with state-of-the-art worker-based stream processing engines.



(a) Dynamic Scaling



(b) Performance Isolation

Figure 6.24: Storm-RTS flexibly reconfigures for various workloads and protects workflow performance from collocated applications while other SPEs fail to do so. (Results are from Azure VMs only; other configurations are omitted due to similarity).

6.2.3.3 Performance Stability

Scalable Workflow Performance We run each RIoTBench workflow separately in a system with ample resources at varying input rates but keeping their parallelism and rate configuration fixed. The results are presented in Figure 6.24a. The x-axis represents the input rate normalized by Storm's saturation rate (maximum throughput). The y-axis represents 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 this point, 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 throughputs are below 20% of the input rate, and at 2.0, their throughput drops further, approaching 0%

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

Performance Isolation We consider the case of multiple workflows competing for shared resources. This is common in production settings and can lead to performance interference. To evaluate how well SPEs protect workflow from interference, we run each RIoTBench workflow with SCAN. This single-bolt workflow performs expensive arithmetic operators on input tuples, competing for CPU cycles with the foreground RIotBench workflows.

In Figure 6.24b, 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 fail to provide performance isolation, showing a throughput decrease after the background load exceeds 50%. Due to relying on best-effort invocation allocation, Storm-Serverless sees its throughput drop from the introduction of very small levels of resource competition. The decrease is severed, and nearly 100% loss of throughput with about 30% competitive load. In contrast, the RBAM allocations enforce rate guarantees with strong resource isolation, allowing Storm-RTS to provide good performance isolation all the way up to 100% competitive load. This demonstrates the ability to deliver predictable performance of RBAM SPEs as discussed in Section 6.2.2.1.

Supporting Bursty Workloads We consider a common load pattern in practice: bursty workflows whose input rate varies over time. Workflow developers can configure Storm-RTS to handle bursty loads by setting the desired input rate equal to the peak input rate when the load bursts. We deploy a PRED workflow at around 35 thousand tuples/sec on Azure VMs. However, after the 10th second, the input rate doubles and lasts for around 30 seconds (see the first graph of Figure 6.25). We execute this load with different SPEs. The workflow's throughput and latency are shown in the second and third graphs of Figure 6.25, respectively.

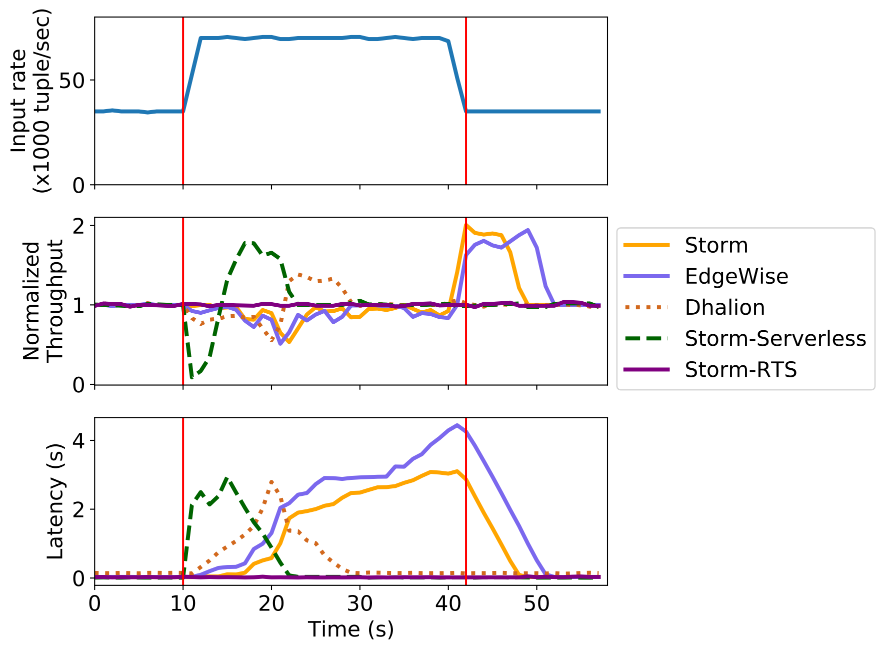


Figure 6.25: Storm-RTS guarantees the performance of bursty workloads while other SPEs fail to do so. (The throughput is normalized to the input rate)

Storm and EdgeWise have their resource allocated statically. When the burst arrives, they cannot process the excessive tuples in time, causing significant high processing latency and a noticeable throughput drop. Dhalion and Storm-Serverless support dynamic allocation to scale up during the burst. However, it takes time for both to detect the burst and scale resource allocation accordingly. Thus, both see significant performance degradation for 10-20s (35 to 65% of the burst period). Storm-RTS, on the other hand, has the desired rate set to the burst peak (70 thousand tuples/sec), helping it maintain the desired throughput and latency throughout the burst period. This demonstrates the robustness of performance stability provided by Storm-RTS.

Takeaway. (Performance Stability) Storm-RTS provides stable performance with minimal tuning efforts.

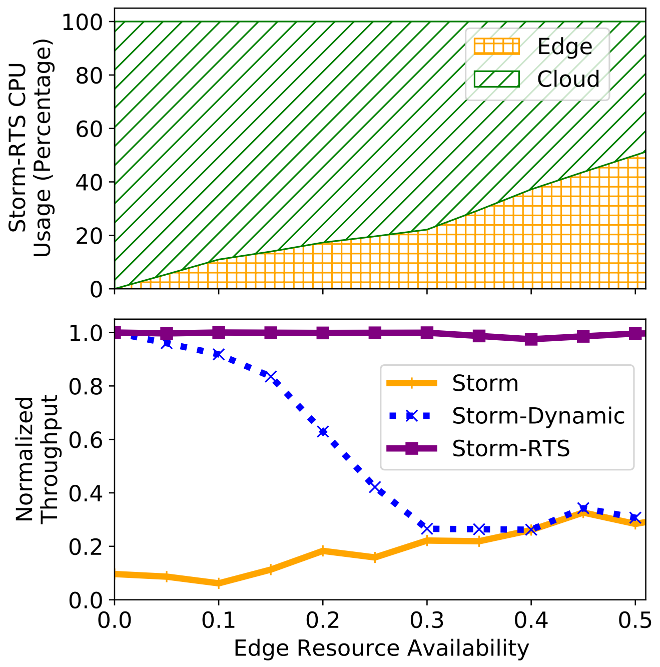


Figure 6.26: Storm-RTS, Storm-Dynamic, and Storm across cloud and edge. Storm-RTS has well-defined performance guarantees, enabling it to respond to resource availability changes and maintain desired throughput.

6.2.3.4 Flexible Cloud-Edge Reconfiguration

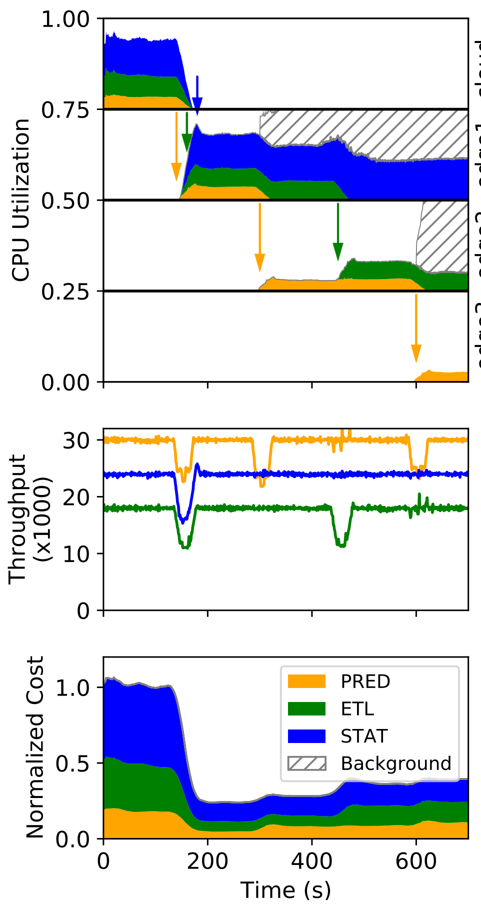
Most SPEs are designed for cloud deployment, but increasingly there are opportunities for stream-processing at the edge in combination with the cloud. However, edge resources are limited, and when there is competition for resources, they may be unavailable. To maintain stable performance for stream processing 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 (ETL, PRED, or 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 6.26. In the first graph, when the availability of edge resources decreases, Storm-RTS shifts operators from the edge to the cloud. This confirms Storm-RTS' ability to reconfigure workflow deployment in response to resource availability change.

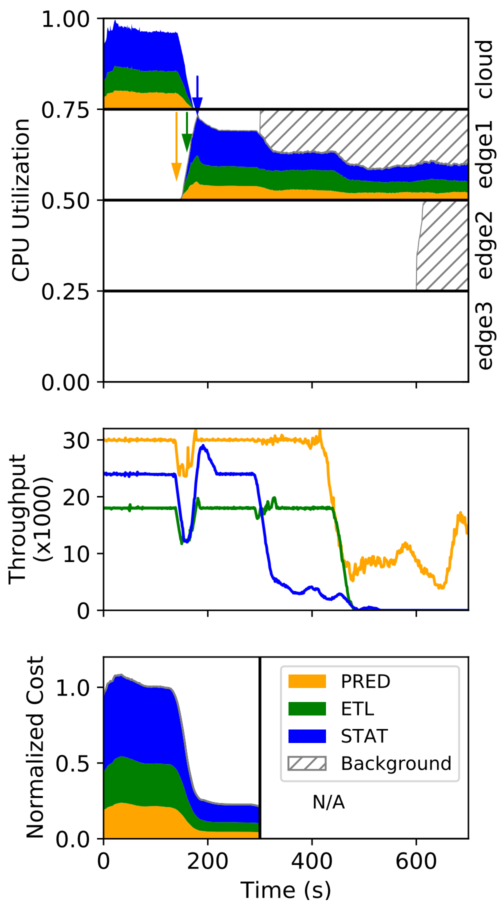
Next, we evaluate Storm-RTS's ability to maintain workflow performance under reconfiguration. We use Storm as a baseline. However, Storm does not respond to resource availability, statically spreading workers across available nodes in a round-robin manner. This produces a deployment with half of the 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 edge see performance degradation of 25%, Storm-Dynamic will rebalance workers across cloud and edge as shown in the second graph in Figure 6.26. 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 6.19). 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 are available. Workers in the cloud perform differently from those in the edge. Thus, shifting a worker causes operator performance changes to 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`, precisely reconfiguring the workflow to restore the guarantee.

Performance stability allows Storm-RTS to simplify application management for other objectives. Consider a simple declarative policy MinCost: minimize resource cost of stream processing workflows at any point in time. Storm-RTS (Figure 6.19) reduces the policy to



(a) Storm-RTS



(b) Storm-Serverless

Figure 6.27: Storm-RTS shifting workflows across edge data centers while maintaining stable performance. The flexibility enabled by Storm-RTS enables simple optimization of cost.

placing operators in the data center with the lowest cost. If this data center is full, operators will be placed in the data center with the next lowest cost, and so on. Consider a resource environment shown in Figure 6.21, where the cost of *edge1*, *edge2*, and *edge3* are equal to 25%, 50%, and 75% respectively relative to the *cloud*'s 100%. On this testbed, we conduct an experiment showing how Storm-RTS operates workflows stably at optimal cost.

The first graph of Figure 6.27a shows events during the experiment and decisions made by Storm-RTS in response. At $t = 0$, Storm-RTS deploys three RIoTBench workflows in *cloud*. 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*. However, at $t = 300$, a SCAN workflow starts at *edge1*, consuming CPU resources. *edge1* becomes oversubscribed, and the local RTS reports this situation to the `application coordinator`. The `application coordinator` has the `operator distributor` move PRED, the most minor workflow, to maintain adequate performance. To minimize resource cost, *edge2* is selected. At $t = 450$, the SCAN load increases. *edge1*'s RTS system notifies the `application coordinator` again, leading to a move of ETL 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. In response, Storm-RTS 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 (the second graph of Figure 6.27a). 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 6.27a).

To understand the importance of Storm-RTS in implementing such declarative policy, consider the same scenario with Storm-Serverless (Figure 6.27b). Since Storm-Serverless allocates resources in a best-effort manner, it 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.

Takeaway. (Flexible Reconfiguration) Storm-RTS simplifies application management, allowing applications to maintain performance stability while reconfiguring workflows for cost minimization.

6.3 Summary

In this chapter, we use RBAM to implement two classes of applications: distributed real-time video analytics and a stream processing engine called Storm-RTS. We evaluate their real-time performance under different conditions, including varying workload burstiness, real-time requirements, and deployment settings to demonstrate the RBAM applicability.

Our RBAM-based video analytics implementation demonstrates that RBAM maximizes application value while minimizing resource costs. Additionally, RBAM can serve as a quality of service parameter, allowing the application to adjust its deployment for optimal performance and cost efficiency. The guaranteed invocation rate provided by RBAM is robust against workload burstiness, requiring only sublinear cost increases in response to changes in burstiness.

Storm-RTS showcases RBAM's broad applicability by using it as a resource model within the stream processing engine (SPE). This integration allows the SPE to configure FaaS deployment and rate guarantees for performance transparency and predictability. Furthermore, Storm-RTS provides stable performance with minimal tuning effort, simplifying application management and maintaining desired performance levels while remaining flexible enough to be reconfigured for other objectives.

In conclusion, let us revisit the applicability research questions raised in Section 3.3 and answer them based on our evaluation results as follows.

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?

- **Answer: Yes**. As shown in the distributed video analytics evaluation (Section 6.1.3), RBAM can support a wide range of real-time requirements, including statistical and absolute guarantees, as well as soft and hard real-time requirements, without incurring any additional cost for the application. The real-time guarantee capability is also robust against various workload burstiness properties.

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

- **Answer: Yes**. As shown in the real-time stream processing evaluation (Section 6.2.3), RBAM's rate guarantee enables performance stability and opens new capabilities, including performance isolation, modularity, and flexible deployment across multiple data centers, that broaden the scope of RBAM usage to deliver more computation value.

With these answers, we complete the demonstration of the RBAM's high applicability.

CHAPTER 7
RELATED WORK

7.1 Supporting Bursty, Real-time Applications

7.1.1 Handling Workload Burstiness

Scalable internet services deal with bursty loads, managing yield and latency by dropping requests [78]. For example, the WeChat microservice system has an elaborate system for load shedding that orders drops to minimize wasted work [350]. 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.

Many state-of-the-art solutions for bursty workloads rely on dynamic allocation to minimize resource waste. 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. This is typically an iterative control process with two main steps:

- *Burst Detection*: the application detects bursts through various methods, including workload monitoring [199, 196] and real-time violation detection (soft real-time) [154, 152, 227]. Some applications avoid violations by trying to predict potential burst arrivals and then proactively allocate resources in advance [70, 220].

- *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) [152, 196]. There are also approaches combining different cloud services and using highly flexible services, such as FaaS, to absorb the burst [199, 154].

Dynamic allocation approaches, while offering potentially efficient solutions, are not without their limitations. They are built upon heuristic methods, which means that any application with burstiness properties or real-time constraints not covered by the heuristic solution may experience misallocation. This can result in either real-time violation or resource waste, highlighting the need for careful consideration and evaluation of these approaches. Unfortunately, no magic formula exists to capture the application burstiness and their real-time requirement. These factors and their combinations vary over a broad spectrum, often requiring expertise to discover the burstiness structure and find a proper dynamic allocation strategy to meet their real-time constraints. As a result, bursty, real-time applications have to undergo a complicated refinement process to achieve high-performance, low-cost outcomes, reducing their applicability. RBAM, on the other hand, rules out the need for refinement by providing a human-friendly, high-level rate-guarantee configuration, allowing applications to specify and enforce their performance guarantee without explicitly implementing it, significantly improving application productivity and making the model highly applicable for different classes of applications.

7.1.2 Satisfying Real-time Requirements

The most straightforward yet common approach for guaranteeing real-time performance requirements is pre-allocation or resource reservation [130, 188, 347, 205, 65]. That is, the application developer estimates the maximum resources required by the applications at their burst and pre-allocates the corresponding amount of resources (e.g., reserved VMs) in the cloud in advance. The approach is simple and can be applied to almost every application, yet it becomes expensive if the workload is bursty. When bursts appear, these reserved VMs offer sufficient computation requirements and meet real-time deadlines. When bursts are absent, however, the VMs remain idle, and their resources are wasted. Because bursts have a low duty factor and high variance, the amount of waste is enormous in terms of time and space. This makes the approach expensive. In contrast, applications using RBAM can scale resources up to the rate specified in their guaranteed invocation configuration. Thus, no pre-allocation is needed, significantly reducing their resource cost.

Many research efforts have been spent on task runtime prediction for efficient scheduling and resource allocation. Some techniques are developed for repeating jobs [102, 110, 163] while others use the job structure and characterize input to construct predicting models [129, 253]. JamaisVu [308] and 3Sigma [246] extract tasks' features from execution history, use them for constructing runtime distributions, and apply a tournament prediction to estimate task runtime. From runtime prediction, many scheduling such as backfilling [306, 345] or packing [312, 309] can be applied to minimize real-time constraint violation while still maximizing resource utilization.

There are also other approaches in the literature. For example, [90] uses the earliest-deadline-first scheduling policy and explicitly handles the transient of dynamic real-time workload to reduce deadline misses. [76] 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 [95, 351].

However, all of the work mentioned above deals 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 a guaranteed allocation rate, RBAM enables applications to plan to achieve real-time guarantee and guarantee computation quality.

Summary. There are limited efforts tackling the problem of cost-effectively handling bursty demands in a timely fashion. Many existing studies either consider real-time constraints or bursty demand but not both, as RBAM did. Further, studies explicitly addressing bursty, real-time applications find it hard to handle the resource control - usability trade-off of cloud service and, thus, unable to meet all solution requirements as RBAM does.

7.2 Improving FaaS-based Application Performance

7.2.1 Improving Best-effort FaaS Performance

Most FaaS systems, including commercial [23, 25, 68] and open-source [180, 240] follow the same FaaS serving pattern in Section 2.2 where efficiency lie in the implementation of FaaS sandboxes, schedulers, and resource management (i.e., autoscaler).

Sandbox implementation. FaaS sandbox implementation is an active research area with three main concerns: isolation, flexibility, and low startup latency. Resolving these concerns often comes with conflicting ideas that require careful balancing. Traditional sandboxing mechanisms adopt Virtual Machine (VM) Manager [32, 16, 24] to provide strong isolation but typically suffer from high startup latency. There are efforts to strip heavy-weight virtualization features for startup acceleration, including gVisor [27], FireCracker [44], and more [176, 187, 315]. Another workaround is to use containers and their customization for high flexibility and performance [48, 238, 116, 303, 165, 293]. Unikernel [206], which provides enhanced security and performance [211, 276, 239], is also an increasingly attractive alternative.

Autoscaling. FaaS autoscaling seeks to provision "just enough" sandboxes for performance. This is typically done through an iterative process of monitoring, predicting resource demand, and adjusting sandbox allocations accordingly. The process is heuristic-based [193] and relies on dynamic (de)allocations driven by historical knowledge [96, 279], Machine Learning training [349, 318, 336, 263, 262], or optimizers constructed from specialized workload and system modeling [52, 195, 159].

Scheduling. FaaS schedulers carefully dispatch invocations across shared resources for high performance, utilization, and short end-to-end latency. Meanwhile, mainstream solutions for the FaaS scheduler try to balance requests across allocated sandboxes through hash-based [298, 194] or objective-based methods leveraging resource monitoring information [93, 173, 294, 149] and quality of service constraints [213, 178, 208] for one or multiple scheduling objectives.

Efforts from Application Space. Since FaaS systems are best-effort, current FaaS applications have to implement their own performance solutions. A general approach is to proactively reserve sandboxes for high-traffic loads beforehand [66, 197] but this is expensive. Recent efforts tend to develop ad-hoc solutions that exploit domain-specific information to create dedicated strategies for a narrow set of application classes such as scientific [94, 209] real-time [297, 296], Machine Learning training [243, 242, 148] and serving [341, 342, 143, 186], etc. Different from RBAM, these approaches attempt to modify the application configuration for performance instead of the FaaS system. This is typically unproductive and challenging due to the lack of performance support from the current FaaS abstraction.

Multiple FaaS Invocations. While RBAM only guarantees performance for a single function, there is a rich body of recent work that addresses the performance issue of FaaS workflow, which consists of multiple FaaS functions to handle complicated tasks. This is challenging because the application not only has to solve the problem of provisioning resources for multiple FaaS invocation along the workflow [106, 290] but also has to find efficient data exchange between serverless functions. The general approach includes leveraging share storage [88, 132, 133, 179, 254, 258], integrating function code into storage [344, 275] with various locality optimizations based on workflow structure and data exchange patterns [225, 89, 165, 183, 207, 284].

Summary. All of the mentioned approaches focus only on either improving conventional FaaS implementation efficiency or workaround them by adding work for the application or a third-party orchestration system. Different from RBAM, they do not add any new capability to the existing FaaS system, leaving them behaving in a best-effort, heuristic-based manner, resulting in limited applicability and productivity.

7.2.2 Performance Abstraction

Serverless Abstraction there are various performance abstractions based on FaaS have been proposed for better applicability and productivity. Commercial clouds offer a *concurrency guarantee* abstraction that allows FaaS functions to declare a target concurrency C. The FaaS system will guarantee the function can execute up to C concurrent invocation at the same time with zero latency [65]. The abstraction ensures fast ramp-up time, enabling applications to effectively handle burst requests. However, figuring out the optimal concurrency configuration can be tricky because the function concurrency depends on its request arrival rate and execution time and both are highly dynamic. In contrast, RBAM's guaranteed invocation rate only requires the developer to know the peak request arrival rate to use the abstraction effectively.

There is a large body of FaaS execution guarantees that try to *statistically limit FaaS execution time* up to a certain percentage (e.g., "99% of invocation complete within 10s"). These guarantees, however, are often tied to specific classes of applications with well-known computation patterns and stable execution times (e.g., ML serving [341], Web service [152]) while RBAM can be applied for various different class of application with a wide range of burstiness and real-time properties.

INFaaS [270] utilize FaaS to abstract away model management and resource configuration, providing a *model-less* abstraction for efficient ML inferences. The abstraction only required the developer to provide the desired ML inference latency, accuracy, cost, and a base model.

The underlying implementation will automatically adjust the model configuration to generate FaaS deployment that meets the desired requirements. Similar to the above abstraction, INFaaS is applicable for a narrow class of applications (ML inference), opposite to RBAM, which is applicable for various classes of applications.

Real-time Abstractions Besides RBAM, there are many efforts on abstraction for real-time performance. *Real-time ABS* [168, 49, 167] provides an abstract behavioral specification language to formally model and analyze cloud resource management for real-time performance. The interface lets applications declare the desire *soft deadline* of real-time tasks and use this parameter to drive the dynamic resource management to provide sufficient resources to enforce the deadline, even under bursty demand. Besides the real-time deadlines, however, the abstraction also requires many other configurations, such as task execution flow, making it complicated to use the abstraction effectively and productively. In contrast, RBAM only requires a guaranteed invocation rate configuration for performance, which is much simpler to use.

Szalay et al. [296, 297] propose Real-time FaaS, abstracting FaaS resource management and back-end services to enable real-time performance management. The abstraction allows the application developer to realize real-time tasks as a FaaS function with predefined deadlines. The underlying implementation will enforce that every invocation that performs the real-time task will have end-to-end latency bounded by the provided deadline. This abstraction, however, only focuses on real-time applications, making it lack the versatility to support other classes of applications, such as RBAM.

7.2.3 Performance Abstraction Implementation

Commercial cloud implements the *concurrency guarantee* by pre-allocating resources needed by the concurrency but letting the application developers pay for them, compromising the

FaaS pay-as-you-go and scale-to-zero objective. This implementation, however, can be used to implement other performance abstractions, such as RBAM, but incur very high resource costs.

To enforce *statistical latency guarantee*, most of the current approaches still rely on best-effort heuristics-based dynamic (de)allocation. The process is typically driven by priority-based policies [272, 317, 152], historical knowledge [218, 302, 341] or optimizers constructed from specialized workload and system modeling [121, 330, 335, 51]. Recent efforts leverage ML achievement, such as Reinforcement Learning [262, 261, 150, 343], GNN [244, 245], and more [200, 164, 252, 45], to improve efficiency. Unfortunately, the performance guarantees implemented by these approaches are statistically different from RBAM's rate guarantee, which is a configurable SLA. Thus, most of them are not applicable to RBAM implementation. Also, these approaches are built on heuristic foundations. they mostly try to align with the uncertainty of workload and cloud environment, so their efficiency is unpredictable and sensitive to the changes in workload dynamics and deployment environment. In contrast, RBAM implementation ideas are proposed to *eliminate* the impact of uncertainty in workload dynamics and deployment environment, making them robust against various workload and deployment scenarios.

Real-time abstraction implementation is even more complicated since the end-to-end latency is highly uncertain, especially in a complicated environment like the cloud. Szalay et al. [296, 297] propose real-time abstraction implementation that adds many constraints on backend databases and networks in the implementation. Some of the approaches are just about the idea, not yet sufficiently supported by current infrastructure (e.g., ultra latency offered by 6G). The implementation approaches are overkill to RBAM as the abstraction mostly focuses on resource management and scheduling yet is flexible enough to enforce real-time performance and other guarantees.

7.3 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.

7.3.1 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 [202, 79, 99, 214, 85] and/or workflow characterization, including operator dependencies [99, 216], queue size [136] and query context [328, 311]. In distribution settings, SPEs place workers among computing nodes in traffic-aware [224, 327, 126] or topology-aware [224, 321, 223] 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 [201, 229, 177, 320, 127] and task placement [83, 166, 177, 107, 228, 84, 103, 53] 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 [189, 79] and workflow throughput [128]. Some approaches even use the monitor data for predicting potential resource contention [128, 157, 158], proactively preventing interference beforehand. Detected interferences are resolved with heuristic algorithms, which either dynamically readjust resource sharing among competitive workflows [221, 128, 157, 158] or migrate them to another set of computing nodes [79].

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 will 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 provides strong isolation from resource configurations.

7.3.2 Stream Processing and FaaS

Leveraging FaaS for dynamic scalability has been proposed in many SPEs [100, 277, 230, 17, 72, 28]. However, these SPEs only outsource the processing logic to FaaS. Other parts of operators, such as transmission and synchronization, are implemented through 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 [240, 20, 180, 26, 23, 25]. When these systems fail to acquire needed resources, the performance of SPE relies on them degrades. Recent years witnessed many attempts to minimize the chance of these failures, including optimizing invocation resource consumption [119, 226], proactive pre-allocation, and reusing terminated invocations [280, 137, 141]. There are also active studies on intelligent resource sharing and function placement to improve resource efficiency and avoid interference [295, 118, 111, 210, 346]. 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 ensures 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.

7.3.3 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., [77, 307, 82, 47]). 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 much effort to address. These include reliability [348, 147, 161, 322, 352], communication latency and overhead [171, 332], and managing limited, heterogeneous resource pools [108, 174], balancing task placement and parallelism [109, 104, 283]. 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 resource 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.

CHAPTER 8
SUMMARY

8.1 Conclusions

We propose Rate-based Abstract Machine (RBAM), a new performance abstraction built on the foundation of Function-as-a-Service. RBAM allows application developers to deploy FaaS functions with guaranteed invocation rates, enabling applications to meet real-time requirements through timely invocations.

To demonstrate RBAM's ability to guarantee real-time performance, we have developed an analytical framework based on rate-monotonic real-time workloads. This framework verifies RBAM's capability to effectively bound FaaS invocation latency and ensure applications meet their real-time deadlines. Additionally, we demonstrate that RBAM can guarantee all real-time application deadlines with a minimum rate guarantee equal to the total task release rate while keeping the overhead below 100% of the actual use.

Furthermore, we propose new rate-based invocation scheduling and resource management algorithms integrated into a new FaaS architecture to implement RBAM efficiently. These algorithms share resources among functions and leverage allocation statistics to find the appropriate time and amount of resource allocation. Consequently, the rate guarantees can be implemented with 10× resource cost reduction. RBAM implementation is also robust and scalable across various application dynamics and deployment environments. We can deploy thousands of RBAM functions over resources at 99.9999% guarantee availability with only 38% resource overhead.

To demonstrate the applicability of RBAM, we use the model to implement two real-time, demanding applications: distributed real-time video analytics and Storm-RTS. Systematic evaluations of these applications confirm that RBAM's rate guarantee makes application performance highly robust against workload burstiness, offering stable performance with solid transparency, predictability, and optimized deployment flexibility.

8.2 Future Work

This work opens up many promising research directions.

8.2.1 Configurable Service Abstraction

RBAM demonstrates that by abstracting cloud resource control with a configurable performance parameter (i.e., the guaranteed invocation rate), we can significantly improve cloud resource usability utilization while allowing applications to achieve their performance needs at high-level configuration with minimal effort, thereby improving their productivity. Given this success, can we apply the same idea to other non-functional application requirements and get similar outcomes?

Hardware Abstraction Application developers have to select specific hardware for their services and resources. Since the cloud is highly heterogeneous. There are many options, yet mostly low-level, including processor types (e.g., GPU vs. CPU), architecture (e.g., ARM, x86), instance configuration (compute instances, storage instances), etc. This is typically unclear to the application developer if one option is better than the other. Worse, even if the application developers want to configure the resource hardware to meet certain requirements, it is challenging to translate these requirements into specific hardware configurations. Thus, *Can we abstract out the resource hardware configuration, replacing them with higher level SLA specifications that let the cloud choose the right hardware configuration for the application developer?* There are a number of interesting SLAs the cloud can offer to developers.

- *Carbon Emission Budget*: the cloud can let the application declare a target Carbon emission budget, limiting the operational Carbon emission generated by the application hardware from their data center. The cloud provider can enforce the SLA by deploying the application on high energy efficiency hardware in data centers fueled with low carbon intensity power grids.
- *Performance Abstraction*: Hardware configuration can be abstracted under performance-related SLAs. For example, ML inference services can simply declare their required inference rate, the cloud will automatically compute the required GPU and related configuration to meet the SLA. FaaS applications can declare their expected execution time and let the cloud use this parameter to find the optimal per-invocation CPU and memory configuration for them.
- *Cost Budget*: Similar to Carbon emission, the application developers can declare the cost budget to limit the spending on cloud resources. The cloud can enforce the SLA by dynamically deploying its applications on services (e.g., on-demand VM, volatile VM, serverless, etc.) with appropriate hardware configuration and cost.

Multi-SLA Abstraction *Can we integrate multiple SLAs into a single abstraction?* There are scenarios where having multiple SLAs is beneficial. For example, we can add the Carbon emission budget to RBAM performance abstraction to enforce cloud provider execute invocation at low Carbon intensity regions, meeting both carbon and performance objectives. This greatly improves the application deployment productivity while enabling the cloud to exploit resource control more effectively. However, doing so leads to a multi-objective problem that complicates the abstraction design and implementation. The more SLAs are added to an abstraction, the more constraints the cloud providers have to satisfy, limiting their decision space and making the implementation problem more challenging. These make designing and implementing Multi-SLA abstraction an appealing problem with many motivating research questions

- *How to define multiple SLAs in a single abstraction?* A good abstraction design should give a set of SLAs that is (i) application-friendly so that developers can easily figure out the right configuration for their needs and (ii) consists of well-co-existing SLAs that have no or little impact on the implementation of the other. While the former requirement ties to the application-specific constraints, the latter requires careful evaluation of SLA combinations to ensure their enforcement is practically feasible.
- *How to implement multiple SLAs efficiently?* Finding ways to realize the abstraction SLAs is tricky, strongly depending on the abstraction design. One can try to implement each SLA separately and then combine them. Another approach is to divide their joint configuration space into sub-regions and develop appropriate solutions for each sub-region separately. For example, we can realize RBAM with Carbon emission budget SLA together by splitting the RBAM rate guarantee and Carbon budget spectrum into two parts, high and low, then develop four different solutions for each combination: (high rate, high budget), (high rate, low budget), (low rate, high budget), and (low rate, low budget).

8.2.2 RBAM Implementation Improvement

The RBAM implementation solution presented in Chapter 5 currently utilizes only the underlying sandbox allocation statistics to reduce overhead. However, there are many additional factors related to invocation execution that can be leveraged to further improve RBAM implementation.

Workload Dynamic *Can we extract workload dynamic characteristics, such as invocation arrival patterns, and use them together with the guaranteed invocation rate to drive resource* *management decisions for better efficiency?* This is a promising direction as analysis on FaaS production [11] reveals a noticeable faction of FaaS functions are periodic (16%), and many are highly predictable [271]. This suggests precisely estimating the invocation request arrival is highly feasible. Further, our preliminary studies in Section 5.2 show that the average arrival rate of FaaS functions is typically 100× lower than their peak rate. If we can precisely predict invocation request arrival and prepare sandboxes accordingly, RBAM implementation cost can be reduced by a further 100×, close to the scale-to-actual-use objective of the FaaS abstraction.

However, there are challenges to make the approach practical. First, workload dynamic is an outcome of the application behavior, which is uncontrollable by the FaaS system. The application behavior may change over time, making it difficult to maintain the prediction accuracy and ensure the guarantee availability target. Second, the FaaS workload is highly diverse. Besides periodic and predictable functions, there is also a considerable amount of them that are highly unpredictable [278]. Thus, creating extremely high precision predictions aiming for high guarantee availability is challenging.

There are several approaches the FaaS system can apply to address these issues. For example, the system can group independent or anti-correlated functions together to stabilize the aggregate workload, thereby mitigating variability and improving prediction accuracy. Additionally, it can allocate extra resources to compensate for inaccurate predictions, thereby maintaining high guarantee availability even with poor-quality workload predictors or difficult-to-predict workloads.

Distributed Resources *Can we exploit distributed resources for better RBAM implementation efficiency and robustness?* Cloud resources are distributed across multiple data centers over a wide geographical area. This opens an opportunity the apply the idea of multiple tries across independent data centers in combination with temporal independence for higher efficiency. Distributed data centers also offer a richer set of underlying sandbox allocation statistics. We can, for example, prioritize allocating sandboxes from data centers with short allocation latency and/or low correlation probability to further reduce the overhead. Expanding the FaaS deployment into multiple data centers also increases the complexity of resource management and scheduling, which requires a lot of effort in their design and implementation. The RBAM system has to be upgraded for multiple data center deployments. The scheduler and resource manager need to be extended in this process to handle concurrent invocation requests and scaling decisions across multiple places. Additionally, capacity limitation, hardware heterogeneity, and different resource usage policies can become a problem, especially when the deployment expands across the cloud-edge continuum.

8.2.3 RBAM Performance Abstraction Extension

Reliable Invocation Execution Time Application responsiveness highly depends on invocation latency and execution time. RBAM's guaranteed invocation rate only bounds the invocation latency. *Can we extend the RBAM to provide a guarantee on execution time, improving RBAM's performance guarantee capability?* This is a crucial yet challenging research problem as invocation execution is determined by many complicated factors, such as per-invocation resource allocation, sandbox isolation capability, and communication overhead. Enforcing an invocation execution guarantee requires a thorough understanding of their impact and effective approaches to handle them.

- *Resource allocation*: Configuring appropriate resource allocation for invocation is important. However, the current RBAM abstraction leaves resource configuration to the *application developer*, whose have very little knowledge of the underlying cloud resources. Thus, they can either end up with over-provisioning, wasting resources, or under-provisioning, degrading invocation execution performance. To workaround, the execution guarantee needs to abstract out resource configuration, outsourcing this part to the FaaS system space and proposing a new resource control algorithm to use the execution guarantee abstraction to guide the configuration.

- *Sandbox Isolation*: sandbox implementation selection is typically a trade-off between invocation latency and isolation effectiveness. Light-weight sandboxes have short invocation latency yet provide weak isolation that could leave invocations suffering from security threats or performance interference with colocated applications. This problem still persists in the context of RBAM. Selecting light-weighted sandboxes lets the RBAM system be implemented with low overhead and can provide a high rate guarantee but could increase the execution time variability, making it more challenging to enforce the execution guarantee. A straightforward solution is to use heavy-weight sandboxes. Another workaround is to use lightweight sandboxes but carefully select colocated applications with FaaS functions, minimizing the risk of interference.

- *Communication overhead*: Many FaaS functions are I/O intensive, such as uploading a file, modifying a database record, etc. Their invocation involves many communications with backend storage systems and other functions. Long, highly variable communication latency negatively impacts execution time predictability, making it hard to enforce execution guarantees. Existing work attempted to resolve the challenges in many ways, including proposing new communication control, opening new chances for communication optimization, or colocating FaaS sandboxes with storage engines to minimize communication overhead. However, these methods are best-effort, focusing on minimizing the communication overhead, ignoring other important factors, such as communication overhead variability and predictability with respect to data transmission size, frequency, and so on. Thus, enforcing the execution guarantee with the appearance of data communication is challenging and requires much attention to resolve.

Rate-guarantee for FaaS Workflow As FaaS is gaining more popularity, its applications become more diverse and complicated. Many applications usually employ workflows of multiple FaaS invocations to accomplish their tasks rather than a single one. Thus, *can we generalize the RBAM's guaranteed invocation rate to the case of a workflow of multiple functions?* Answering this question opens many research opportunities

- *How to define the rate-guarantee for a FaaS workflow?* FaaS invocations within a workflow are connected in many ways: invocation chain (one function calls another), through shared storage (one function updates data in a shared storage, which triggers the execution of other functions), through an orchestration system, etc. These connection patterns trigger invocation requests differently, thus requiring different rate-guarantee abstractions. For example, in the function chain pattern, the execution is continuous. Once an invocation completes, another one will start. Thus, we can consider a whole workflow as a single big FaaS function, extending the guaranteed invocation rate as workflow guaranteed execution rate. Meanwhile, if invocations are connected through shared storage, workflow execution is no longer continuous but implicitly synchronized by the storage. The RBAM's rate-guarantee generalization should be tied to the storage performance.

- *How to implement the extended RBAM efficiently to meet the guarantee?* Enforcing workflow performance instead of each individual function actually opens more space for efficiency optimization. The FaaS system can track the workflow topology to pre-warm downstream function sandboxes, monitor the invocation resource usage to intelligently partition resources among functions, give more to highly resource-intensive functions to improve workflow efficiency, etc.

8.2.4 Broadening RBAM Applicability

ML Inference ML inference plays an important role in many modern cloud applications. Inference rate (i.e., the number of inferences processed per second) is one of the most important performance metrics for ML inference services. The metric well aligns with RBAM's guarantee invocation rate. Application developers can implement ML inference tasks as single FaaS functions and configure the guaranteed invocation rate to match their desired inference rate. The RBAM abstraction ensures these ML tasks will get sufficient resources at the same rate as the inference request, effectively meeting the desired inference rate. Implementing such service, however, is challenging

- *Big model size*: practical ML models are getting larger over time for better applicability and accuracy. Some models have even reached trillions of parameters (e.g., GPT-4 is estimated to have around 1.7 trillion parameters [314]). These models consume thousands or more GB of memory, impossible to fit within a single invocation.
- *Limited hardware supports*: ML model typically requires GPU access for efficiency. In contrast, current FaaS functions use the CPU for handling invocation computation since it is easy to program and share the CPU across applications, maximizing resource utilization. On the other hand, there is limited support to share GPU effectively, especially among fine-grain allocations such as FaaS invocations.

Thus, *can we propose an efficient implementation of RBAM to support ML inference service?* To answer the question, we must address the above issues effectively. One possible direction is combining resource sharing and batching. The RBAM system deploys the model directly over a specific set of GPUs and lets FaaS invocation share these GPUs. This resolves the large model problem and minimizes memory footprint. To increase GPU utilization, interference requests are batched by handling multiple inferences within an invocation and/or processing multiple invocations concurrently per model/GPU. However, the approach could prolong interference latency and be expensive for a bursty workload. Workarounds to be considered include dynamically adjusting the batch size to minimize the latency, sharing multiple models across GPUs and dynamically swapping them based on workload dynamics.

Portable Serverless RBAM deployments are highly portable. The FaaS abstraction allows application developers to easily move RBAM deployment across cloud data centers, cloud and edge, or even across vendors. Even live migration is feasibly straightforward as FaaS functions are stateless. Furthermore, the guaranteed invocation rate helps cloud providers enforce the FaaS performance guarantee. Thus, developers can spend the minimum effort to maintain the desired performance. These advantages make RBAM an excellent choice for applications that require frequent service migration or spatial shifting for their execution objectives, such as minimizing Carbon emission and resource costs or avoiding overloaded data centers for performance stability.

However, because FaaS invocations are stateless, FaaS-based applications must rely on a third-party stateful service to maintain their internal state and data (e.g., databases, cache, etc.). Applications have to migrate these stateful services together with the RBAM deployment as well. Maintaining RBAM guarantee during migration is challenging because doing so requires the migration to be transparent to the end-user. Thus, data/state must be moved under FaaS invocations, which poses many problems to be solved.

- *State consistency and data integrity.* The data migrated to the new location must reflect their state at the old location. Thus, data losses, corruption, or out-of-order delivery must be avoided/tolerated as much as possible.
- *Availability.* During migration, part of data/state can be inaccessible, hurting the application availability.
- *Migration Duration.* The migration is affected by many factors, such as network bandwidth, request rate, configuration differences between old and new locations, etc. All

of these, together, can make the duration of migration highly unpredictable. In a bad scenario, migration can be long and negatively affect the application performance and availability.

- *Security Risks* Data breaches or unauthorized access can happen during the migration process if there are insufficient proper security measures. Sensitive data might be exposed if intercepted during transfer.

Resolving these issues requires carefully designing a stateful service that maintains the application state and data. Multiple mechanisms can be employed to address the problem above. For example, applications can replicate data over potential locations to improve application availability and minimize data transmission during migration, reducing the risk of violating data consistency and integrity. They can also implement a checksum mechanism to recover from data losses or corruption. Furthermore, smart request redirection can be considered to help the application drive the demand according to computation and data availability, reducing the impact of migration on their performance.

REFERENCES

[1] AWS Lambda Pricing. https://aws.amazon.com/lambda/pricing/.

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

[3] 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.

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

[5] 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.

[6] 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.

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

[8] Google Cloud Function Pricing. https://cloud.google.com/functions/pricing.

[9] 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.

[10] 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.

[11] Lambda Serverless Benchmark. https://serverless-benchmark.com.

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

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

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

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

[16] KVM. https://www.linux-kvm.org/, Feb 2007.

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

[18] Prometheus. https://prometheus.io/, 2014.

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

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

[21] Apache Apex. https://kafka.apache.org/documentation/streams/, 2017.

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

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

[24] AWS Nitro System. https://aws.amazon.com/ec2/nitro/g, 2017.

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

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

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

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

[29] Google Cloud Function SLA. https://cloud.google.com/functions/sla, 2021. Accessed on 2024-04-10.

[30] Summary of the AWS Service Event in the Northern Virginia (US-EAST-1) Region. https://aws.amazon.com/message/12721/, 2021. Accessed on 2024-05-18.

[31] AWS Lambda SLA. https://aws.amazon.com/lambda/sla/, 2022. Accessed on 2024-04-10.

[32] Hyper-V. https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/about/, Apr 2022.

[33] Kafka Streams. https://apex.apache.org/docs.html, 2022.

[34] Mantis. https://netflix.github.io/mantis/, Jun 2022.

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

[36] Realtime serverless cell latency monitor. https://observablehq.com/@tomlarkworthy/serverless-cell-latency-monitor, 2022. Accessed: 2022-01-11.

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

[38] Market share analysis: Event stream processing platforms, worldwide, 2022. https://www.gartner.com/en/documents/4547999, 2023.

[39] Summary of the AWS Lambda Service Event in Northern Virginia (US-EAST-1) Region. `https://aws.amazon.com/message/061323/`, 2023. Accessed on 2024-05-18.

[40] AWS Post-Event Summaries. `https://aws.amazon.com/premiumsupport/technology/pes/`, 2024. Accessed on 2024-05-18.

[41] Kubernetes. `https://kubernetes.io/`, 2024. Accessed on 2024-04-11.

[42] Microsoft Azure SLA. `https://www.microsoft.com/licensing/docs/view/Service-Level-Agreements-SLA-for-Online-Services`, 2024. Accessed on 2024-04-10.

[43] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the borealis stream processing engine. In *Cidr*, volume 5, pages 277–289, 2005.

[44] 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.

[45] S. Agarwal, M. A. Rodriguez, and R. Buyya. A deep recurrent-reinforcement learning method for intelligent autoscaling of serverless functions. *IEEE Transactions on Services Computing*, 2024.

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

[47] 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.

[48] 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.

[49] E. Albert, F. S. de Boer, R. Hähnle, E. B. Johnsen, R. Schlatte, S. L. Tapia Tarifa, and P. Y. Wong. Formal modeling and analysis of resource management for cloud architectures: an industrial case study using real-time abs. *Service Oriented Computing and Applications*, 8:323–339, 2014.

[50] A. AlHammadi, A. AlZaabi, B. AlMarzooqi, S. AlNeyadi, Z. AlHashmi, and M. Shatnawi. Survey of iot-based smart home approaches. In *2019 Advances in Science and Engineering Technology International Conferences (ASET)*, pages 1–6, 2019.

[51] 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.

[52] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang. {CherryPick}: Adaptively unearthing the best cloud configurations for big data analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 469–482, 2017.

[53] 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.

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

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

[56] Amazon Burstable Instances. `https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/burstable-performance-instances.html`.

[57] Amazon Spot Instance. `https://aws.amazon.com/ec2/spot/`.

[58] Amazon T2 Instances. `https://aws.amazon.com/ec2/instance-types/t2/`.

[59] L. Aniello, R. Baldoni, and L. Querzoni. Adaptive online scheduling in storm. In *Proceedings of the 7th ACM international conference on Distributed event-based systems*, pages 207–218, 2013.

[60] L. Ao, L. Izhikevich, G. M. Voelker, and G. Porter. Sprocket: A Serverless Video Processing Framework. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, pages 263–274, New York, NY, USA, 2018. ACM.

[61] Apache Gearpump. `http://gearpump.github.io/overview.html`, 2022.

[62] Apache Nifi. `https://nifi.apache.org/`, 2018.

[63] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, et al. Above the Clouds: A Berkeley View of Cloud Computing. 2009.

[64] M. S. Aslanpour, A. N. Toosi, C. Cicconetti, B. Javadi, P. Sbarski, D. Taibi, M. Assuncao, S. S. Gill, R. Gaire, and S. Dustdar. Serverless edge computing: vision and challenges. In *Proceedings of the 2021 Australasian computer science week multiconference*, pages 1–10, 2021.

[65] AWS Lambda Provisioned Concurrency. https://docs.aws.amazon.com/lambda/latest/operatorguide/provisioned-scaling.html, 2024. Accessed on 2024-04-09.

[66] AWS Lambda Reserved Concurrency. https://docs.aws.amazon.com/lambda/latest/operatorguide/reserved-concurrency.html, 2024. Accessed on 2024-04-09.

[67] AWS Step Functions. https://aws.amazon.com/step-functions/, 2024. Accessed on 2024-04-02.

[68] Azure Function. https://azure.microsoft.com/en-us/services/functions/, 2016. Accessed on 2024-04-01.

[69] Azure Durable Functions. https://learn.microsoft.com/en-us/azure/azure-functions/durable/, 2024. Accessed on 2024-04-01.

[70] 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.

[71] E. Badidi, K. Moumane, and F. El Ghazi. Opportunities, applications, and challenges of edge-ai enabled video analytics in smart cities: a systematic review. *IEEE Access*, 2023.

[72] 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.

[73] 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.

[74] 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.

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

[76] 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.

[77] 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.

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

[79] 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.

[80] 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.

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

[82] 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.

[83] 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.

[84] 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.

[85] 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.

[86] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams: A new class of data management applications. In *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB '02, page 215–226. VLDB Endowment, 2002.

[87] M. Carrascosa and B. Bellalta. Cloud-gaming: Analysis of google stadia traffic. *Computer Communications*, 188:99–116, 2022.

[88] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, and R. Katz. Cirrus: A serverless framework for end-to-end ml workflows. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 13–24, 2019.

[89] B. Carver, J. Zhang, A. Wang, A. Anwar, P. Wu, and Y. Cheng. Wukong: A scalable and locality-enhanced framework for serverless parallel computing. In *Proceedings of the 11th ACM symposium on cloud computing*, pages 1–15, 2020.

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

[91] 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.

[92] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. Telegraphcq: Continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, page 668, New York, NY, USA, 2003. Association for Computing Machinery.

[93] C.-C. Chang, S.-R. Yang, E.-H. Yeh, P. Lin, and J.-Y. Jeng. A kubernetes-based monitoring platform for dynamic cloud resource provisioning. In *GLOBECOM 2017-2017 IEEE Global Communications Conference*, pages 1–6. IEEE, 2017.

[94] R. Chard, Y. Babuji, Z. Li, T. Skluzacek, A. Woodard, B. Blaiszik, I. Foster, and K. Chard. Funcx: A federated function serving fabric for science. In *Proceedings of the 29th International symposium on high-performance parallel and distributed computing*, pages 65–76, 2020.

[95] 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.

[96] L. Chen and H. Shen. Considering resource demand misalignments to reduce resource over-provisioning in cloud datacenters. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*, pages 1–9. IEEE, 2017.

[97] 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.

[98] Z. Chen, J. Xu, J. Tang, K. A. Kwiat, C. A. Kamhoua, and C. Wang. Gpu-accelerated high-throughput online stream data processing. *IEEE Transactions on Big Data*, 4(2):191–202, 2016.

[99] 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.

[100] 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.

[101] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. B. Zdonik. Scalable distributed stream processing. In *CIDR*, volume 3, pages 257–268, 2003.

[102] 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.

[103] 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.

[104] 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.

[105] 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.

[106] N. Daw, U. Bellur, and P. Kulkarni. Speedo: Fast dispatch and orchestration of serverless workflows. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 585–599, New York, NY, USA, 2021. Association for Computing Machinery.

[107] 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.

[108] 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.

[109] 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.

[110] 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.

[111] 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.

[112] M. Dias de Assunção, 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.

[113] C. Dietrich, S. Naumann, R. Thrift, and D. Lohmann. Rt. js: Practical real-time scheduling for web applications. In *2019 IEEE Real-Time Systems Symposium (RTSS)*, pages 69–79. IEEE, 2019.

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

[115] Docker Containers, 2010. Access on 2024-04-01.

[116] 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.

[117] H. Duc Nguyen and A. A. Chien. A foundation for real-time applications onfunction-as-a-service. *ACM SIGMETRICS Performance Evaluation Review*, 51(4):54–65, 2024.

[118] 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.

[119] 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.

[120] 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.

[121] T. Elgamal, A. Sandur, K. Nahrstedt, and G. Agha. Costless: Optimizing cost of serverless computing through function fusion and placement. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 300–312. IEEE, 2018.

[122] M. Elsakhawy and M. Bauer. Faas2f: A framework for defining execution-sla in serverless computing. In *2020 IEEE Cloud Summit*, pages 58–65. IEEE, 2020.

[123] M. Elsakhawy and M. Bauer. Performance analysis of serverless execution environments. In *2021 International Conference on Electrical, Communication, and Computer Engineering (ICECCE)*, pages 1–6. IEEE, 2021.

[124] M. Elsakhawy and M. Bauer. Sla for sequential serverless chains: A machine learning approach. In *Proceedings of the Seventh International Workshop on Serverless Computing (WoSC7) 2021*, pages 36–41, 2021.

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

[126] 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.

[127] 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.

[128] 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.

[129] 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.

[130] S. Fiori, L. Abeni, and T. Cucinotta. Rt-kubernetes: Containerized real-time cloud computing. In *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, pages 36–39, 2022.

[131] 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.

[132] S. Fouladi, F. Romero, D. Iter, Q. Li, S. Chatterjee, C. Kozyrakis, M. Zaharia, and K. Winstein. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *2019 USENIX annual technical conference (USENIX ATC 19)*, pages 475–488, 2019.

[133] S. Fouladi, R. S. Wahby, B. Shacklett, K. V. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 363–376, Boston, MA, Mar. 2017. USENIX Association.

[134] S. Fouladi, R. S. Wahby, B. Shacklett, K. V. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein. Encoding, Fast and Slow: Low-latency Video Processing Using Thousands of Tiny Threads. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, pages 363–376, Berkeley, CA, USA, 2017. USENIX Association.

[135] M. Fragkoulis, P. Carbone, V. Kalavri, and A. Katsifodimos. A survey on the evolution of stream processing systems. *The VLDB Journal*, pages 1–35, 2023.

[136] 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.

[137] 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.

[138] Gartner. Gartner Forecasts Worldwide Public Cloud Revenue to Grow 17.3 Percent in 2019. `https://www.gartner.com/en/newsroom/press-releases/2018-09-12-gartner-forecasts-worldwide-public-cloud-revenue-to-grow-17-percent-in-2019`, September 2018.

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

[140] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *Nsdi*, volume 11, pages 24–24, 2011.

[141] 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.

[142] I. Gog, M. Schwarzkopf, A. Gleave, R. N. Watson, and S. Hand. Firmament: Fast, Centralized Cluster Scheduling at Scale. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 99–115, 2016.

[143] M. Golec, R. Ozturac, Z. Pooranian, S. S. Gill, and R. Buyya. Ifaasbus: A security-and privacy-based lightweight framework for serverless computing using iot and machine learning. *IEEE Transactions on Industrial Informatics*, 18(5):3522–3529, 2021.

[144] Google Cloud Platform. `https://cloud.google.com/`, 2008.

[145] Google Cloud Functions. `https://cloud.google.com/functions/`, 2016. Accessed on 2024-04-01.

[146] Google Preemptible Virtual Machine. `https://cloud.google.com/preemptible-vms/`.

[147] 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.

[148] A. Grafberger, M. Chadha, A. Jindal, J. Gu, and M. Gerndt. Fedless: Secure and scalable federated learning using serverless computing. In *2021 IEEE International Conference on Big Data (Big Data)*, pages 164–173. IEEE, 2021.

[149] X. Guan, X. Wan, B.-Y. Choi, S. Song, and J. Zhu. Application oriented dynamic resource allocation for data centers using docker containers. *IEEE Communications Letters*, 21(3):504–507, 2016.

[150] A. Gujarati, R. Karimi, S. Alzayat, W. Hao, A. Kaufmann, Y. Vigfusson, and J. Mace. Serving {DNNs} like clockwork: Performance predictability from the bottom up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 443–462, 2020.

[151] 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.

[152] 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.

[153] H. Guo, J. Li, Y. Xu, Y. Long, J. Zhou, and W. Wu. Faascom: Mitigate cold start problem in faas via function community. In *2023 IEEE 29th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 1023–1030. IEEE, 2023.

[154] 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.

[155] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *NSDI*, volume 11, pages 22–22, 2011.

[156] J. Hojlo. Future of Industry Ecosystems: Shared Data and Insights. `https://blogs.idc.com/2021/01/06/future-of-industry-ecosystems-shared-data-and-insights/` Visited April 19, 2024.

[157] 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.

[158] 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.

[159] M. R. HoseinyFarahabady, A. Y. Zomaya, and Z. Tari. A model predictive controller for managing qos enforcements and microarchitecture-level interferences in a lambda platform. *IEEE Transactions on Parallel and Distributed Systems*, 29(7):1442–1455, 2017.

[160] 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.

[161] 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.

[162] IPC. The Growth in Connected IoT Devices Is Expected to Generate 79.4ZB of Data in 2025. `https://www.idc.com/getdoc.jsp?containerId=prUS45213219`, Jun 2019.

[163] 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.

[164] B. Jeong, J. Jeon, and Y.-S. Jeong. Proactive resource autoscaling scheme based on scinet for high-performance cloud computing. *IEEE Transactions on Cloud Computing*, 2023.

[165] 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.

[166] 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.

[167] E. B. Johnsen, K. I. Pun, M. Steffen, S. L. T. Tarifa, and I. C. Yu. Meeting deadlines, elastically., 2016.

[168] E. B. Johnsen, R. Schlatte, and S. L. Tapia Tarifa. Modeling resource-aware virtualized applications for the cloud in real-time abs. In *International Conference on Formal Engineering Methods*, pages 71–86. Springer, 2012.

[169] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht. Occupy the Cloud: Distributed Computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, pages 445–451, New York, NY, USA, 2017. ACM.

[170] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, et al. Cloud Programming Simplified: A Berkeley View on Serverless Computing. *arXiv preprint arXiv:1902.03383*, 2019.

[171] 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.

[172] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, Í. Goiri, S. Krishnan, J. Kulkarni, et al. Morpheus: Towards Automated SLOs for Enterprise Clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 117–134, 2016.

[173] K. Kaffes, N. J. Yadwadkar, and C. Kozyrakis. Centralized core-granular scheduling for serverless functions. In *Proceedings of the ACM symposium on cloud computing*, pages 158–164, 2019.

[174] 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.

[175] I. A. Kash, G. O'Shea, and S. Volos. DC-DRF: Adaptive Multi-Resource Sharing at Public Cloud Scale. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 374–385. ACM, 2018.

[176] Kata Containers, 2017. Access on 2024-04-01.

[177] 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.

[178] Y. K. Kim, M. R. HoseinyFarahabady, Y. C. Lee, and A. Y. Zomaya. Automated fine-grained cpu cap control in serverless computing platform. *IEEE Transactions on Parallel and Distributed Systems*, 31(10):2289–2301, 2020.

[179] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 427–444, 2018.

[180] Knative, 2024. Access on 2024-04-01.

[181] Knative Autoscaling. `https://knative.dev/docs/serving/autoscaling/`, 2024. Accessed on 2024-04-09.

[182] T. Kolajo, O. Daramola, and A. Adebiyi. Big data stream analysis: a systematic literature review. *Journal of Big Data*, 6(1):47, 2019.

[183] S. Kotni, A. Nayak, V. Ganapathy, and A. Basu. Faastlane: Accelerating {Function-as-a-Service} workflows. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 805–820, 2021.

[184] 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.

[185] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 239–250, New York, NY, USA, 2015. Association for Computing Machinery.

[186] M. S. Kurz. Distributed double machine learning with a serverless architecture. In *Companion of the ACM/SPEC International Conference on Performance Engineering*, pages 27–33, 2021.

[187] H. A. Lagar-Cavilla, J. A. Whitney, A. M. Scannell, P. Patchin, S. M. Rumble, E. de Lara, M. Brudno, and M. Satyanarayanan. Snowflock: rapid virtual machine cloning for cloud computing. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, page 1–12, New York, NY, USA, 2009. Association for Computing Machinery.

[188] Y. Lei and S. Jasin. Real-time dynamic pricing for revenue management with reusable resources, advance reservation, and deterministic service time requirements. *Operations Research*, 68(3):676–685, 2020.

[189] 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.

[190] Y. Li, A. Padmanabhan, P. Zhao, Y. Wang, G. H. Xu, and R. Netravali. Reducto: On-camera filtering for resource-efficient real-time video analytics. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 359–376, 2020.

[191] Y. Li, D. Zeng, L. Gu, M. Ou, and Q. Chen. On efficient zygote container planning toward fast function startup in serverless edge cloud. In *IEEE INFOCOM 2023-IEEE Conference on Computer Communications*, pages 1–9. IEEE, 2023.

[192] Z. Li, L. Guo, Q. Chen, J. Cheng, C. Xu, D. Zeng, Z. Song, T. Ma, Y. Yang, C. Li, et al. Help rather than recycle: Alleviating cold startup in serverless computing through {Inter-Function} container sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 69–84, 2022.

[193] Z. Li, L. Guo, J. Cheng, Q. Chen, B. He, and M. Guo. The serverless computing survey: A technical primer for design architecture. *ACM Computing Surveys (CSUR)*, 54(10s):1–34, 2022.

[194] Z. Li, Y. Liu, L. Guo, Q. Chen, J. Cheng, W. Zheng, and M. Guo. Faasflow: Enable efficient workflow execution for function-as-a-service. In *Proceedings of the 27th acm international conference on architectural support for programming languages and operating systems*, pages 782–796, 2022.

[195] C. Lin and H. Khazaei. Modeling and optimization of performance and cost of serverless applications. *IEEE Transactions on Parallel and Distributed Systems*, 32(3):615–632, 2020.

[196] 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.

[197] W. Ling, L. Ma, C. Tian, and Z. Hu. Pigeon: A dynamic and efficient serverless and faas framework for private cloud. In *2019 International Conference on Computational Science and Computational Intelligence (CSCI)*, pages 1416–1421. IEEE, 2019.

[198] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, Jan. 1973.

[199] 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.

[200] M. Liu, A. Zeng, M. Chen, Z. Xu, Q. Lai, L. Ma, and Q. Xu. Scinet: Time series modeling and forecasting with sample convolution and interaction. *Advances in Neural Information Processing Systems*, 35:5816–5828, 2022.

[201] 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.

[202] 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.

[203] 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.

[204] 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.

[205] C. Lu, W. Chen, K. Ye, and C.-Z. Xu. Understanding the workload characteristics in alibaba: A view from directed acyclic graph analysis. In *2020 International Conference on High Performance Big Data and Intelligent Systems (HPBD&IS)*, pages 1–8. IEEE, 2020.

[206] 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.

[207] A. Mahgoub, L. Wang, K. Shankar, Y. Zhang, H. Tian, S. Mitra, Y. Peng, H. Wang, A. Klimovic, H. Yang, et al. {SONIC}: Application-aware data passing for chained serverless applications. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 285–301, 2021.

[208] N. Mahmoudi, C. Lin, H. Khazaei, and M. Litoiu. Optimizing serverless computing: Introducing an adaptive function placement algorithm. In *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*, pages 203–213, 2019.

[209] M. Majewski, M. Pawlik, and M. Malawski. Algorithms for scheduling scientific workflows on serverless architecture. In *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 782–789. IEEE, 2021.

[210] 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.

[211] F. Manco, C. Lupu, F. Schmidt, J. Mendes, S. Kuenzer, S. Sati, K. Yasukata, C. Raiciu, and F. Huici. My vm is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 218–233, 2017.

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

[213] S. McDaniel, S. Herbein, and M. Taufer. A two-tiered approach to i/o quality of service in docker containers. In *2015 IEEE International Conference on Cluster Computing*, pages 490–491. IEEE, 2015.

[214] 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.

[215] F. Metzger, S. Geißler, A. Grigorjew, F. Loh, C. Moldovan, M. Seufert, and T. Hoßfeld. An introduction to online video game qos and qoe influencing factors. *IEEE Communications Surveys & Tutorials*, 24(3):1894–1925, 2022.

[216] 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.

[217] Microsoft Azure Cloud. `https://azure.microsoft.com/`, 2010.

[218] V. Mittal, S. Qi, R. Bhattacharya, X. Lyu, J. Li, S. G. Kulkarni, D. Li, J. Hwang, K. Ramakrishnan, and T. Wood. Mu: An efficient, fair and responsive serverless framework for resource-constrained edge clouds. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 168–181, 2021.

[219] H. MohammadReza, C. L. Young, Y. Z. Albert, and T. Zahir. A QoS-Aware Resource Allocation Controller for Function as a Service (FaaS) Platform. In *Service-Oriented Computing - 15th International Conference, ICSOC 2017, Malaga, Spain, November 13-16, 2017, Proceedings*, pages 241–255, 2017.

[220] C. Morariu, O. Morariu, S. Răileanu, and T. Borangiu. Machine learning for predictive scheduling and resource allocation in large scale manufacturing systems. *Computers in Industry*, 120:103244, 2020.

[221] 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.

[222] Most actions per minute in a videogame. `https://www.guinnessworldrecords.com/world-records/88069-most-actions-per-minute-in-a-videogame`.

[223] 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.

[224] 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.

[225] I. Müller, R. Marroquín, and G. Alonso. Lambada: Interactive data analytics on cold data using serverless cloud infrastructure. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 115–130, 2020.

[226] 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.

[227] R. K. Naha, S. Garg, A. Chan, and S. K. Battula. Deadline-based dynamic resource allocation and provisioning algorithms in fog-cloud environment. *Future Generation Computer Systems*, 104:131–141, 2020.

[228] 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.

[229] 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.

[230] 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.

[231] Nest. Nest Learning Thermostat. `https://nest.com/thermostats/nest-learning-thermostat/overview/`, 2011.

[232] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *2010 IEEE International Conference on Data Mining Workshops*, pages 170–177. IEEE, 2010.

[233] H. D. Nguyen and A. A. Chien. Storm-rts: Stream processing with stable performance for multi-cloud and cloud-edge. In *2023 IEEE 16th International Conference on Cloud Computing (CLOUD)*, pages 45–57. IEEE, 2023.

[234] 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.

[235] 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.

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

[237] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R. H. Campbell. Samza: stateful scalable stream processing at linkedin. *Proceedings of the VLDB Endowment*, 10(12):1634–1645, 2017.

[238] 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.

[239] P. Olivier, D. Chiba, S. Lankes, C. Min, and B. Ravindran. A binary-compatible unikernel. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 59–73, 2019.

[240] OpenFaaS, 2024. Accessed on 2024-04-01.

[241] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: Distributed, Low Latency Scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 69–84. ACM, 2013.

[242] T. Oyar and A. Deri. Faastest-machine learning based cost and performance faas optimization. In *Economics of Grids, Clouds, Systems, and Services: 15th International Conference, GECON 2018, Pisa, Italy, September 18–20, 2018, Proceedings*, volume 11113, page 171. Springer, 2019.

[243] E. Paraskevoulakou and D. Kyriazis. Ml-faas: Towards exploiting the serverless paradigm to facilitate machine learning functions as a service. *IEEE Transactions on Network and Service Management*, 2023.

[244] J. Park, B. Choi, C. Lee, and D. Han. Graf: A graph neural network based proactive resource allocation framework for slo-oriented microservices. In *Proceedings of the 17th International Conference on emerging Networking EXperiments and Technologies*, pages 154–167, 2021.

[245] J. Park, B. Choi, C. Lee, and D. Han. Graph neural network-based slo-aware proactive resource autoscaling framework for microservices. *IEEE/ACM Transactions on Networking*, 2024.

[246] 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.

[247] D. C. Parkes, A. D. Procaccia, and N. Shah. Beyond Dominant Resource Fairness: Extensions, Limitations, and Indivisibilities. *ACM Transactions on Economics and Computation (TEAC)*, 3(1):3, 2015.

[248] N. Parmiggiani, A. Bulgarelli, D. Beneventano, V. Fioretti, L. Baroncelli, A. Addis, and M. Tavani. Rtapipe, a framework to develop astronomical pipelines for the real-time analysis of scientific data. In *Astronomical Society of the Pacific Conference Series*, volume 532, page 139, 2022.

[249] N. Parmiggiani, A. Bulgarelli, A. Ursi, V. Fioretti, L. Baroncelli, A. Addis, A. Di Piano, C. Pittori, F. Verrecchia, F. Lucarelli, et al. The agile real-time analysis pipelines in the multi-messenger era. In *37th International Cosmic Ray Conference*, page 933, 2022.

[250] A. Passwater. 2018 Serverless Community Survey: Huge Growth in Serverless Usage. `https://serverless.com/blog/2018-serverless-community-survey-huge-growth-usage/` Visited May 15, 2019.

[251] D. Pellerin, D. Ballantyne, and A. Boeglin. An Introduction to High Performance Computing on AWS. `https://d0.awsstatic.com/whitepapers/Intro_to_HPC_on_AWS.pdf`, 2015. Amazon White Paper.

[252] Y. Peng and H. Peng. Inferfair: Towards qos-aware scheduling for performance isolation guarantee in heterogeneous model serving systems. *Future Generation Computer Systems*, 150:10–20, 2024.

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

[254] M. Perron, R. Castro Fernandez, D. DeWitt, and S. Madden. Starling: A scalable query engine on cloud functions. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 131–141, 2020.

[255] Philips. Philips Hue Light Bulbs. `https://www2.meethue.com/en-us/bulbs`, 2012.

[256] A. Poniszewska-Maranda, R. Matusiak, N. Kryvinska, and A.-U.-H. Yasar. A real-time service system in the cloud. *Journal of Ambient Intelligence and Humanized Computing*, 11(3):961–977, 2020.

[257] PTC. Howden Creates Mixed Reality Solutions to Enhance Customer Experience. `https://www.ptc.com/en/case-studies/howden-mixed-reality`, Feb 2019.

[258] Q. Pu, S. Venkataraman, and I. Stoica. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *16th USENIX symposium on networked systems design and implementation (NSDI 19)*, pages 193–206, 2019.

[259] A. M. Qasim, N. Abbas, A. Ali, and B. A. A.-R. Al-Ghamdi. Abandoned object detection and classification using deep embedded vision. *IEEE Access*, 2024.

[260] B. Qian, Z. Wen, J. Tang, Y. Yuan, A. Y. Zomaya, and R. Ranjan. Osmoticgate: Adaptive edge-based real-time video analytics for the internet of things. *IEEE Transactions on Computers*, 72(4):1178–1193, 2022.

[261] H. Qiu, W. Mao, A. Patke, C. Wang, H. Franke, Z. T. Kalbarczyk, T. Başar, and R. K. Iyer. Reinforcement learning for resource management in multi-tenant serverless platforms. In *Proceedings of the 2nd European Workshop on Machine Learning and Systems*, pages 20–28, 2022.

[262] H. Qiu, W. Mao, A. Patke, C. Wang, H. Franke, Z. T. Kalbarczyk, T. Başar, and R. K. Iyer. Simppo: A scalable and incremental online learning framework for serverless resource management. In *Proceedings of the 13th Symposium on Cloud Computing*, pages 306–322, 2022.

[263] H. Qiu, W. Mao, C. Wang, H. Franke, A. Youssef, Z. T. Kalbarczyk, T. Başar, and R. K. Iyer. {AWARE}: Automate workload autoscaling with reinforcement learning in production cloud systems. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 387–402, 2023.

[264] D. Quaresma, D. Fireman, and T. E. Pereira. Controlling garbage collection and request admission to improve performance of faas applications. In *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 175–182. IEEE, 2020.

[265] A. Radovanovic, R. Koningstein, I. Schneider, B. Chen, A. Duarte, B. Roy, D. Xiao, M. Haridasan, P. Hung, N. Care, et al. Carbon-aware computing for datacenters. *arXiv preprint arXiv:2106.11750*, 2021.

[266] E. G. Renart, J. Diaz-Montes, and M. Parashar. Data-driven stream processing at the edge. In *2017 IEEE 1st International Conference on Fog and Edge Computing (ICFEC)*, pages 31–40. IEEE, 2017.

[267] Ring. Ring Video Doorbell. `https://ring.com/`, 2012.

[268] RIVER. River. `http://river.cs.uchicago.edu`.

[269] H. Röger and R. Mayer. A comprehensive survey on parallelization and elasticity in stream processing. *ACM Comput. Surv.*, 52(2), Apr. 2019.

[270] F. Romero, Q. Li, N. J. Yadwadkar, and C. Kozyrakis. {INFaaS}: Automated model-less inference serving. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 397–411, 2021.

[271] 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.

[272] G. R. Russo, A. Milani, S. Iannucci, and V. Cardellini. Towards qos-aware function composition scheduling in apache openwhisk. In *2022 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*, pages 693–698. IEEE, 2022.

[273] M. Rychlỳ, P. Škoda, and P. Smrž. Heterogeneity-aware scheduler for stream processing frameworks. *International Journal of Big Data Intelligence*, 2(2):70–80, 2015.

[274] 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.

[275] J. Sampé, M. Sánchez-Artigas, P. García-López, and G. París. Data-driven serverless functions for object storage. In *Proceedings of the 18th ACM/IFIP/USENIX middleware conference*, pages 121–133, 2017.

[276] F. Schmidt. Uniprof: A unikernel stack profiler. In *Proceedings of the SIGCOMM Posters and Demos*, pages 31–33. 2017.

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

[278] 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.

[279] 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.

[280] 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.

[281] D. Shan, F. Ren, P. Cheng, R. Shu, and C. Guo. Micro-burst in data centers: Observations, analysis, and mitigations. In *2018 IEEE 26th International Conference on Network Protocols (ICNP)*, pages 88–98. IEEE, 2018.

[282] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, 1948.

[283] 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.

[284] S. Shillaker and P. Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 419–433, 2020.

[285] E. Shmueli and D. G. Feitelson. Backfilling with Lookahead to Optimize the Performance of Parallel Job Scheduling. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 228–251. Springer, 2003.

[286] 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.

[287] A. Shukla and Y. Simmhan. Model-driven scheduling for distributed stream processing systems. *Journal of Parallel and Distributed Computing*, 117:98–114, 2018.

[288] 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.

[289] 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.

[290] V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. E. Gonzalez, J. M. Hellerstein, and A. Tumanov. Cloudburst: stateful functions-as-a-service. *Proc. VLDB Endow.*, 13(12):2438–2452, jul 2020.

[291] G. Staples. Torque Resource Manager. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 8. ACM, 2006.

[292] Statista. Number of active users of pokémon go worldwide from 2016 to 2020, by region. `https://www.statista.com/statistics/665640/pokemon-go-global-android-apple-users/`.

[293] J. Stojkovic, T. Xu, H. Franke, and J. Torrellas. Mxfaas: Resource sharing in serverless environments for parallelism and efficiency. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pages 1–15, 2023.

[294] A. Suresh and A. Gandhi. Fnsched: An efficient scheduler for serverless functions. In *Proceedings of the 5th international workshop on serverless computing*, pages 19–24, 2019.

[295] 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.

[296] M. Szalay, P. Mátray, and L. Toka. Real-time task scheduling in a faas cloud. In *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, pages 497–507. IEEE, 2021.

[297] M. Szalay, P. Matray, and L. Toka. Real-time faas: Towards a latency bounded serverless cloud. *IEEE Transactions on Cloud Computing*, 2022.

[298] S. Talluri, N. Herbst, C. Abad, A. Trivedi, and A. Iosup. A trace-driven performance evaluation of hash-based task placement algorithms for cache-enabled serverless computing. In *Proceedings of the 20th ACM International Conference on Computing Frontiers*, pages 164–175, 2023.

[299] 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.

[300] W. Tang, D. Ren, Z. Lan, and N. Desai. Adaptive Metric-aware Job Scheduling for Production Supercomputers. In *2012 41st International Conference on Parallel Processing Workshops*, pages 107–115. IEEE, 2012.

[301] N. Tantalaki, S. Souravlas, and M. Roumeliotis. A review on big data real-time stream processing and its scheduling techniques. *International Journal of Parallel, Emergent and Distributed Systems*, 35(5):571–601, 2020.

[302] 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.

[303] J. Thalheim, P. Bhatotia, P. Fonseca, and B. Kasikci. Cntr: Lightweight OS containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 199–212, Boston, MA, July 2018. USENIX Association.

[304] M. Tirmazi, A. Barker, N. Deng, M. E. Haque, Z. G. Qin, S. Hand, M. Harchol-Balter, and J. Wilkes. Borg: the next generation. In *EuroSys'20*, Heraklion, Crete, 2020.

[305] R. Tönjes, P. Barnaghi, M. Ali, A. Mileo, M. Hauswirth, F. Ganz, S. Ganea, B. Kjærgaard, D. Kuemper, S. Nechifor, A. Sheth, V. Tsiatsis, and L. Vestergaard. Real time iot stream processing and large-scale data analytics for smart city applications. In *poster session, European Conference on Networks and Communications*, page 10. sn, 2014.

[306] 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.

[307] 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.

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

[309] 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.

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

[311] 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.

[312] 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.

[313] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale Cluster Management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems*, page 18. ACM, 2015.

[314] M. Vitiello. GPT-4 Parameters. `https://textcortex.com/post/gpt-4-parameters`, 2023.

[315] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. C. Snoeren, G. M. Voelker, and S. Savage. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 148–162, 2005.

[316] 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.

[317] 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, 2021.

[318] H. Wang, D. Niu, and B. Li. Distributed machine learning with a serverless architecture. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pages 1288–1296. IEEE, 2019.

[319] 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.

[320] 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.

[321] 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.

[322] 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.

[323] J. Wilkes. Jsspp 2018 keynote. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1045–1046. IEEE, 2018.

[324] W. Wingerath, F. Gessert, S. Friedrich, and N. Ritter. Real-time stream processing for big data. *it-Information Technology*, 58(4):186–194, 2016.

[325] Z. Wu, Y. Deng, Y. Zhou, J. Li, and S. Pang. Faasbatch: Enhancing the efficiency of serverless computing by batching and expanding functions. In *2023 IEEE 43rd International Conference on Distributed Computing Systems (ICDCS)*, pages 372–382. IEEE, 2023.

[326] L. Xiao, Y. Cao, Y. Gai, J. Liu, P. Zhong, and M. M. Moghimi. Review on the application of cloud computing in the sports industry. *Journal of Cloud Computing*, 12(1):152, 2023.

[327] 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.

[328] 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.

[329] G. Yan, K. Liu, C. Liu, and J. Zhang. Edge intelligence for internet of vehicles: A survey. *IEEE Transactions on Consumer Electronics*, 2024.

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

[331] S. Yi, A. Andrzejak, and D. Kondo. Monetary Cost-aware Checkpointing and Migration on Amazon Cloud Spot Instances. *IEEE Transactions on Services Computing*, 5(4):512–524, 2011.

[332] 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.

[333] A. B. Yoo, M. A. Jette, and M. Grondona. Slurm: Simple Linux Utility for Resource Management. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 44–60. Springer, 2003.

[334] K. K. Young, H. MohammadReza, C. L. Young, Y. Z. Albert, and J. Raja. Dynamic Control of CPU Usage in a Lambda Platform. In *IEEE International Conference*

on Cluster Computing, CLUSTER 2018, Belfast, UK, September 10-13, 2018, pages 234–244, 2018.

[335] G. Yu, P. Chen, Z. Zheng, J. Zhang, X. Li, and Z. He. Faasdeliver: Cost-efficient and qos-aware function delivery in computing continuum. *IEEE Transactions on Services Computing*, 2023.

[336] H. Yu, A. A. Irissappane, H. Wang, and W. J. Lloyd. Faasrank: Learning to schedule functions in serverless platforms. In *2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, pages 31–40. IEEE, 2021.

[337] M. Yu, T. Cao, W. Wang, and R. Chen. Following the data, not the function: Rethinking function orchestration in serverless computing. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1489–1504, Boston, MA, Apr. 2023. USENIX Association.

[338] 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.

[339] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the twenty-fourth ACM symposium on operating systems principles*, pages 423–438, 2013.

[340] C. Zhang, V. Gupta, and A. A. Chien. Information Models: Creating and Preserving Value in Volatile Cloud Resources. In *IEEE International Conference on Cloud Engineering.* IC2E, 2019.

[341] C. Zhang, M. Yu, W. Wang, and F. Yan. MArk: Exploiting cloud services for Cost-Effective, SLO-Aware machine learning inference serving. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1049–1062, Renton, WA, July 2019. USENIX Association.

[342] C. Zhang, M. Yu, W. Wang, and F. Yan. Enabling cost-effective, slo-aware machine learning inference serving on public cloud. *IEEE Transactions on Cloud Computing*, 10(3):1765–1779, 2020.

[343] H. Zhang, W. Huang, L. Zhao, and K. Li. Maxwell's demon in tail-tolerant, resource-efficient serverless computing. In *2022 IEEE 28th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 762–769. IEEE, 2023.

[344] T. Zhang, D. Xie, F. Li, and R. Stutsman. Narrowing the gap between serverless and its state with storage functions. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–12, 2019.

[345] 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.

[346] 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.

[347] Y. Zhang, Y. Yu, W. Wang, Q. Chen, J. Wu, Z. Zhang, J. Zhong, T. Ding, Q. Weng, L. Yang, et al. Workload consolidation in alibaba clusters: the good, the bad, and the ugly. In *Proceedings of the 13th Symposium on Cloud Computing*, pages 210–225, 2022.

[348] 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.

[349] W. Zheng, M. Tynes, H. Gorelick, Y. Mao, L. Cheng, and Y. Hou. Flowcon: Elastic flow configuration for containerized deep learning applications. In *Proceedings of the 48th International Conference on Parallel Processing*, pages 1–10, 2019.

[350] H. Zhou, M. Chen, Q. Lin, Y. Wang, X. She, S. Liu, R. Gu, B. C. Ooi, and J. Yang. Overload Control for Scaling WeChat Microservices. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 149–161. ACM, 2018.

[351] 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.

[352] 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.