THE UNIVERSITY OF CHICAGO

CLOUD RESOURCE MANAGEMENT FOR BURSTY, REAL-TIME WORKLOADS

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCE
IN CANDIDACY FOR THE DEGREE OF
MASTER

DEPARTMENT OF COMPUTER SCIENCE

BY
HAI DUC NGUYEN

CHICAGO, ILLINOIS
JULY 2019
# TABLE OF CONTENTS

LIST OF FIGURES ................................................................. v
LIST OF TABLES ................................................................. vii
ACKNOWLEDGMENTS ............................................................... viii
ABSTRACT ................................................................. ix

1 INTRODUCTION ................................................................. 1
  1.1 The Rise of Bursty, Real-time Applications ................................ 1
  1.2 Limitations of Current Cloud Resource Offerings .......................... 2
  1.3 Approach: Real-time Serverless ............................................ 4
  1.4 Contributions ............................................................... 6
  1.5 Thesis Outline ............................................................. 7

2 BACKGROUND ................................................................. 8
  2.1 Cloud Computing .......................................................... 8
  2.2 IT Server Workloads and Virtual Machines ................................. 10
  2.3 Bursty Workload and Finite Duration Resources ........................... 11
    2.3.1 Serverless ........................................................... 11
    2.3.2 Volatile Resources .................................................... 12
    2.3.3 Amazon Burstable Instances ........................................... 13

3 BURSTY, REAL-TIME WORKLOADS AND REAL-TIME SERVERLESS ......... 15
  3.1 Real-time, Bursty Workload ............................................... 15
  3.2 Key Characteristics and Challenges ...................................... 16
  3.3 Limitations of Today’s Cloud Resources .................................. 18
  3.4 Approach: Real-time Serverless .......................................... 20
  3.5 Support Bursty, Real-time Workloads with Real-time Serverless ...... 21
  3.6 Research Questions ....................................................... 23

4 ANALYTICAL MODELING AND EVALUATION .................................. 25
  4.1 Analytical Model .......................................................... 25
  4.2 Real-time Serverless Capability .......................................... 30
    4.2.1 Guaranteeing Application Quality .................................... 30
    4.2.2 Robustness against Burst Shape ..................................... 31
    4.2.3 Optimizing Resource Usage ........................................... 33
  4.3 Economic Attractiveness of Real-time Serverless ......................... 36

5 EMPIRICAL EVALUATION ....................................................... 40
  5.1 Simulation with Synthetic Workloads ..................................... 40
    5.1.1 Simulation and Workload Generation .................................. 40
    5.1.2 Validating Analytical Model .......................................... 41
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1.3 Robustness against Burst Variability</td>
<td>42</td>
</tr>
<tr>
<td>5.1.4 Robustness against Burst Interference</td>
<td>44</td>
</tr>
<tr>
<td>5.1.5 Concurrent Bursty Real-time Applications</td>
<td>46</td>
</tr>
<tr>
<td>5.2 Case Study: Traffic Intersection Monitoring</td>
<td>48</td>
</tr>
<tr>
<td>5.2.1 Real Traffic Video Statistic and Video Analysis Pipeline</td>
<td>49</td>
</tr>
<tr>
<td>5.2.2 Computation Quality Guarantee</td>
<td>50</td>
</tr>
<tr>
<td>5.2.3 Cost Attractiveness</td>
<td>51</td>
</tr>
<tr>
<td>6 A REAL-TIME SERVERLESS IMPLEMENTATION</td>
<td>53</td>
</tr>
<tr>
<td>6.1 Feasibility of Real-time Serverless Implementation</td>
<td>53</td>
</tr>
<tr>
<td>6.2 Architecture</td>
<td>54</td>
</tr>
<tr>
<td>6.3 Demonstration</td>
<td>56</td>
</tr>
<tr>
<td>7 RELATED WORK</td>
<td>60</td>
</tr>
<tr>
<td>7.1 Serverless and FaaS</td>
<td>60</td>
</tr>
<tr>
<td>7.2 Datacenter Resource Management</td>
<td>60</td>
</tr>
<tr>
<td>7.3 Job Scheduling for Heterogeneous Workloads</td>
<td>61</td>
</tr>
<tr>
<td>7.4 Coping with Bursty Workloads</td>
<td>62</td>
</tr>
<tr>
<td>7.5 Coping with Real-time Workloads</td>
<td>63</td>
</tr>
<tr>
<td>8 SUMMARY AND FUTURE WORK</td>
<td>64</td>
</tr>
<tr>
<td>8.1 Summary</td>
<td>64</td>
</tr>
<tr>
<td>8.2 Future Work</td>
<td>65</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

1.1 Evolving computing infrastructure promises to enable exciting classes of applications ........................................... 2
1.2 Existing cloud resources do not address real-time requirements. Real-time serverless tackles the problem by extending FaaS with guaranteed allocation rate .......................... 3
1.3 Limitation of current cloud resource offerings and real-time serverless solution for real-time, bursty workloads .......................................................... 4

3.1 Real-world events such as traffic incidents give rise to burst computational requirements in video analytic applications. ............................................................. 16
3.2 Bursty, real-time workloads in traditional cloud resources give either poor application quality (low allocation) or high cost (high allocation) .................................................. 18
3.3 Serverless’s best effort allocation vs. real-time serverless’s guaranteed allocation rate ......................................................... 21
3.4 Resource cost of real-time serverless vs. VM .......................................................... 23

4.1 Burst parameters, computation requirements, allocation of UI and RTS instances, and the delay of each frame (e.g. \( l_2 \) and \( l_7 \)) .................................................. 27
4.2 Achievable burst frame value for UI and RTS with various guaranteed allocation rate. \((H=140, D=3,600, \tau=2,607 \ (1/2 \text{ per minute}) \)) ................................................ 29
4.3 Impact of burst shape on burst value, for varied guaranteed allocation rates, \( A_{RTS} \). Solid lines are achieved burst value; dashed lines are maximum potential burst value. ......................................................... 32
4.4 Burst value vs. Computation Cost .......................................................... 35
4.5 Comparing resource cost for achieving 100% of the maximum value (UI only, RTS only) with various cost ratios \((k)\). \((\lambda = 3/\text{hour}, D = 360 - 9,000, A_{RTS} = 1, P = H)\) ................................................ 37
4.6 Cumulative density functions of burst interference \((P(n))\) ........................................ 39

5.1 Validating the analytical model for uniform bursts \((H=140, D=3,600, \tau=2,607 \ (1/2 \text{ per minute}), DF = 0.01, \lambda = 0.3/\text{hour})\) ................................................ 41
5.2 Burst value vs. Guaranteed Allocation Rate \((A_{RTS})\) for varied burst duration standard deviations. \(D = 3,600\), duty factor is adjusted by varying \(\lambda\). \((ft = \text{frame-time})\) ................................................ 42
5.3 Burst value vs. Guaranteed Allocation Rate \((A_{RTS})\) for varied burst durations. Durations vary statistically with standard deviation \(=0.1 \cdot D\). \((ft = \text{frame-time})\) 43
5.4 Burst value vs. \(A_{RTS}\) at different duty factors (varying \(\lambda\)) ......................... 44
5.5 Cost for maximizing value using UI and RTS at various cost ratio \((k)\) and Cost of UI at different duty factors, \(P = \text{maximum observed computation height (several bursts)}\). .......................................................... 45
5.6 Allocation rate needed to achieve burst value fraction (at varied duty factors) .... 46
5.7 RTS resource consumption scales with application demand and much lower than UI resource consumption .......................................................... 47
5.8 Glimpse System Architecture (from [17]). ........................................... 49
5.9 Value vs. \(A_{RTS}\) .......................................................... 50
5.10 RTS is efficient in terms of both application cost and cloud provider resource cost

6.1 A real-time serverless resource pool can be implemented with a fixed number of instances.

6.2 System architecture.

6.3 Real-time serverless application interface

6.4 Demonstration diagram

6.5 RTS and serverless performance when background load is absence

6.6 RTS and serverless performance with the presence of background load
# LIST OF TABLES

4.1 Terms and Notation ......................................................... 26

5.1 Burst Statistics for Traffic Video ....................................... 49
ACKNOWLEDGMENTS

I would like to express my deepest appreciation to all those who provided me the possibility to complete this thesis. A sincere gratitude I give to my advisor, Prof. Andrew A. Chien for his patient, motivation, great guidance for my study and research. Prof. Chien is always there to support me and steer me to the right direction in every possible way.

Furthermore, I would like to thank my teammates at LSSG, especially Chaojie (Sam) Zhang, Yuanwei (Kevin) Fang and Chen Zou for their valuable advice, experience, and support. Working as a member of the team is my pleasure and an important part of my growth as a researcher.

Finally, this accomplishment would not have been possible without the unfailing support and love from my family and my best friends Tri Huynh, Son Truong Hy, Adam Dziedzic, and Giang (Zana) Tran. Your continuous encouragement was an endless source of energy for my years of study and through the process of writing this thesis.
ABSTRACT

Today’s cloud resource offerings provide no guarantees for resource allocation, so bursty, real-time applications must reserve, and pay for resources they do not use - to achieve real-time guarantees. We propose Real-time Serverless, a new type of cloud resource with a new service-level objective – guaranteed allocation rate for transient cloud function invocations. This guarantee enables timely resource allocation, enabling applications to achieve real-time performance efficiently with good resource utilization and cost. With a simple burst model, we study real-time serverless analytically, exploring its effect on application quality, guarantees, and cost. Next, we use simulation to explore a statistical variety of bursts and higher loads (multi-application), to study the benefits of Real-time Serverless for applications.

In both analytic and simulation studies, the availability of real-time serverless service enables bursty, real-time applications to achieve guaranteed high quality at reasonable cost. Specifically, for a desired application quality, the required allocation rate can be determined. Real-time serverless also enables applications to match resources to demands and thereby minimize the cost. For example, for duty factors from 0.025 to 0.25, the value per unit resource of Real-time Serverless instances to a bursty application can be 4x higher than the traditional. Further results show the robustness of real-time serverless with bursty, real-time applications over a wide range of properties. Multiple applications can share a real-time serverless pool efficiently, supporting duty factor increases of 25x with only a 1.6x increase in allocation rate (provider resource cost), revealing high resource management flexibility and economic potential to cloud providers.

We present a case study of a video-based traffic monitoring application. Despite more complex burst statistics, real-time serverless delivers major benefits for application quality and cost. For this application, real-time serverless instances are worth nearly 16x per unit resource, when compared to virtual machine resources. Finally, we introduce a simple implementation of real-time serverless to demonstrate its feasibility and capability to support bursty, real-time workloads efficiently.
CHAPTER 1
INTRODUCTION

1.1 The Rise of Bursty, Real-time Applications

Technology advances such as infrastructure improvement, the increasing popularity of mobile devices as well as the growth of cloud and edge computing enable more and more IoT applications to emerge. New IoT applications are not only diverse but also experience massive scale deployment. Examples include Amazon’s 100 million intelligent assistants [11], large scale urban and rural monitoring systems [15, 43], and even expensive large equipment [21, 45]. Many of these lead to more than 20 billion devices created by 2019 and the figure is projected to continue increasing in the years to come [22]. Therefore, designing an appropriate resource offering models to efficiently support the IoT workload is essentially important.

However, observations over IoT workloads reveal many difficulties (Figure 1.1). Applications such as voice assistant, texting, and traffic monitoring exhibit bursty computational requirements, requiring many small invocations, and rapid scaleup and scaledown of total computational resources to achieve both responsive performance and acceptable cost. Besides, there is a large number of emerging applications working under strict deadlines of computation such as health care, real-time streaming, online games, etc. These applications demand highly responsiveness from the underlying system for making critical decisions in time and delivering real-time experience.

We are interested in real-time, bursty applications – a new class of IoT-enabled applications for distributed intelligence that are characterized by bursty load and real-time requirements. Their bursty load typically arises from external real-world or unpredictable cyberworld events. Their response is subject to real-time constraints with hard deadlines. For example, distributed video analytics such as traffic or crowd control combines continuous sensing, with bursts of deeper computational analysis and timely decisions when interesting
Figure 1.1: Evolving computing infrastructure promises to enable exciting classes of applications events, such as detecting a “wanted” car or car crash, occur. Failing to be timely (missing a critical event) might allow a fugitive to escape, delay car accident victim to get emergency treatment, etc. Another example is network monitoring. In-depth analysis can be triggered when the system detects abnormal traffic or unusual packets and responses are cybersecurity actions such as closing vulnerable ports, rerouting traffic, etc. Making such decisions is subjected to hard deadlines because failing to act quickly may put the system under dangerous attacks.

1.2 Limitations of Current Cloud Resource Offerings

Despite the emergence of real-time bursty workloads and their increasing use of cloud, current cloud resource offerings do not focus on satisfying real-time requirements. The primary driving force, however, is the convenience of resource use. Starting from virtual machines, in which users must manage the whole stack of development, now cloud resources are more about simplicity. For example, the emerging serverless (a.k.a Function-as-a-Service) totally relieves users from server management so they can focus on application design (Figure 1.2). This improves application development productivity and cost efficiency. Unfortunately, be-
Figure 1.2: Existing cloud resources do not address real-time requirements. Real-time serverless tackles the problem by extending FaaS with guaranteed allocation rate.

cause commercial serverless resource offerings lack real-time support, real-time, bursty applications cannot exploit such convenience but have to employ their own solutions to satisfy real-time requirements. They must use cloud resource offerings that allow more control over resources (and hence, least convenient) for freely implementing real-time solutions.

Virtual machine is a primary cloud resource with the highest control over raw resources on which real-time, bursty applications can rely. However, the lack of response in a timely fashion to bursty loads makes them not a good fit for the bursty and real-time demand as requests of “on-demand” virtual machines can be delayed arbitrarily, or even rejected. The only solution is to reserve resources in-advance via reserved or on-demand allocation at the cost of resource payments during idle time. However, in order to ensure no real-time deadline is missed, the application has to reserve resources to the peak of the burst. And as can be seen from Figure 1.3a, this leads to significant resource waste, preventing applications from cost-effectively guaranteeing computation quality.

To reduce the cost, applications must have the capability of scaling allocation up and down to bursty demand. Serverless with autoscaling capability is a good match for such requirement. However, current serverless offerings cannot respond in a guaranteed timely fashion to bursty loads, as startup latencies and throughput rates for serverless invocations are strictly best-effort. And despite efforts at deficiency [34, 41], invocations can be delayed
arbitrarily. As a result, applications have no guarantee of when their computation will run and finish, and thereby are unable to make any guarantee of quality. In bad cases, slow resource allocation may significantly prolong burst processing, as depicted in Figure 1.3b, possibly leading to real-time constraint violation. Thus, despite high scalability and excellent resource pricing, serverless cannot enable any cost-effective and quality guaranteed solution for real-time bursty applications.

In short, the lack of capability for timely responding to bursty demands of cloud resources makes applications miss real-time deadline and SLO’s and prevent their realization at reasonable cost. As can be seen from Figure 1.3, guaranteeing real-time objectives requires static allocation, leading to a significant cost of unused resources. On the other hand, efforts to scale down the cost by dynamic allocation leading to possibilities of violating real-time constraints.

1.3 Approach: Real-time Serverless

To meet the needs of bursty, real-time applications, we propose a new cloud resource type, Real-time Serverless, with the capability of timely respond to bursty demand. In particular, real-time serverless is serverless with guaranteed allocation rate – the minimum rate of allocating new instances (Figure 1.2). The guaranteed allocation rate shapes allocation latency so applications know the resources they can utilize at present and also in the future.
For example, using real-time serverless of guaranteed allocation rate equal to 10 instances/sec allows the application to know that it will have at least 50 instances in the next 5 seconds and 100 instances in the next ten seconds. This capability enables real-time bursty applications to guarantee real-time performance with small or even no preallocation. Furthermore, real-time serverless inherits serverless’s dynamic property, allowing applications to scale computation and cost to actual resource use.

Figure 1.3c depicts resource consumption of an application utilizing real-time serverless of guaranteed allocation rate equal to burst rate. The application has no static reservation for the burst and allocates new real-time serverless instances once demand increases. Because the real-time serverless guarantees allocation of a new instance at a given rate, it can be designed to match the slope of the burst, ensuring that the application always has sufficient resources for absorbing bursty demand, and thereby satisfying real-time requirements. Further, the application only allocates new instances when needed, so the computation scales with resource need and the cost scales up and down with actual resource use. Therefore, it is possible to both satisfy real-time requirements and scale down the cost to bursty demand with real-time serverless.

Start from this idea, we systematically construct analytical models for capturing key characteristics of real-time, bursty workloads, current cloud resource offerings and real-time serverless. We conduct analysis over different combinations of these characteristics to evaluate the benefits of real-time serverless for application quality\textsuperscript{1} guarantees and cost-efficiencies. Further, we carry out simulation using synthetic data generated by varying burstiness characteristics to examine the robustness of real-time serverless. To validate the results as well as to show the applicability of real-time serverless, we present a case study of a traffic monitoring application. Finally, to prove feasibility, we implement real-time serverless on the top of Mesos [33] and run it on a local computing cluster [53].

Analytical and empirical results of our study show robust advantages of real-time server-

\textsuperscript{1} Throughout the thesis, we use the terms application quality and value synonymously.
less on a wide range of bursty properties, enabling optimal solutions for cost-efficient high application value. Furthermore, real-time serverless has a higher value per unit resources than traditional offering and resource efficiency scales well for multiple applications, suggesting this structure is attractive for cloud providers. Even with complex burst statistics exhibited by the real trace, real-time serverless still brings major benefits for cost and application quality. Our implementation demonstrates the capability of guaranteeing resource allocation under different application loads, even at the presence of an unstable background load.

1.4 Contributions

The primary contribution of the thesis is real-time serverless, a new type of cloud resource offering with guaranteed allocation rate for supporting bursty applications with real-time constraints. The thesis also presents an analytical model and simulation for systematically evaluating real-time serverless’s benefits and characterizing its capability. Further, we propose a simple implementation for real-time serverless to prove its feasibility.

Specific contributions of the thesis include:

1. Design of a new cloud resource type, Real-time Serverless, that enables applications to achieve real-time guarantees for bursty, real-time workloads, and do so cost-effectively.

2. Analytical studies that compare traditional cloud offerings and real-time serverless. Analysis for varying burst shape, rate and computation quality target reveals the promising potential of real-time serverless in supporting real-time bursty applications. Even low guaranteed allocation still allows applications to rationally design their resource consumption for guaranteeing responsive performance.

3. Analysis of resource pricing: real-time serverless are valuable for bursty, real-time applications – for duty factors from 0.025 to 0.25 the value per unit resources of real-time serverless resources is 4x to nearly 16x greater than traditional cloud resource types.
4. Study of resource usage: real-time serverless enables optimal resource usage strategies for delivering cost-efficiently responsive application performance. With a greedy+drop resource allocation algorithm, applications can effectively exploit real-time serverless to minimize cost as well as maximize computation quality.

5. In-depth study with simulation and synthetic workloads confirms the robustness of real-time serverless against burst shape, variability and interference. The results suggest real-time serverless can work well with a wide range of bursty, real-time applications.

6. Evaluation for multiple bursty applications reveals the scalability of real-time serverless. Supporting duty factor increases of 25x with only a 1.6x increase in allocation rate (provider resource cost), suggesting the approach can be effectively implemented by cloud resource providers.

7. A case study, using a Glimpse-like pipeline, on traffic monitoring video traces that shows the benefit of a guaranteed allocation rate, demonstrating 16x value per unit real-time serverless resource compared to traditional.

8. An implementation of Real-time Serverless on the top of Mesos shows the feasibility of implementation and demonstrates its robustness against background interference.

1.5 Thesis Outline

The remainder of the thesis is organized as follows. Background of cloud services and their workloads are covered in Chapter 2. Chapter 3 introduces the emerging real-time, bursty workload, challenges of serving them efficiently and describes our approach of Real-time Serverless. Chapter 4 explores Real-time Serverless’s benefits for bursty, real-time applications. Chapter 5 broadens and deepens this study, via simulation of more complex scenarios and over a specific video analysis application. Implementation of Real-time Serverless is discussed in Chapter 6. Finally, Chapter 7 and 8 discuss related work and then summarize the thesis with suggestions for future work.
CHAPTER 2

BACKGROUND

2.1 Cloud Computing

Cloud Computing is a delivering of computing services such as servers, storages, networks, etc. across the Internet [4, 46, 30]. Cloud services are offered by cloud providers with the idea of replacing the cost for building, maintaining data center and software by elastic, utility-like resources with pay-as-you-go pricing. Most of their services can be classified into one of four categories.

1. Infrastructure as a Service (IaaS): services offered as basic resources like servers, virtual machines, etc. Users have full control, even at very low levels, over the resources they own.

2. Platform as a Service (PaaS): services offered as on-demand environments for quick deployment. Resources often come with operating systems, databases, programming-language execution environments, etc. so that users can easily deploy and run their application fast at the extra cost of buying and managing the underlying platform.

3. Function as a Service (FaaS) and Serverless Computing: resources are offered as slate-less instances triggered by pre-defined events and last for a few minutes. The model allows users to quickly run short tasks without no infrastructure configurations.

4. Software as a Service (SaaS): services offered as software and databases so that users just come and use existing solutions offered by cloud providers. No software/environment deployment is required and users only pay for the service they actually use.

In general, the above classification groups cloud resources based on the ease of use and how much control users have over the resource. Goes from IaaS to SaaS, the convenience increases. IaaS resources are barely raw resources so users have to build everything from
scratch by themselves. With SaaS resources, in contrast, everything is set. Users just simply call the functions offered by the services, no implementation or deployment is needed. Ease of use, however, comes with the cost of losing control over the resources. Users cannot modify the SaaS database for their purpose while they have full control over IaaS virtual machines. The trade-off between convenience and resource control creates more opportunities to deliver cloud resources to users with different backgrounds, budgets, and goals.

Cloud resources are shaped with virtualization technologies. Resources provided to users are abstracted from their physical counterparts (virtual machine, container). Virtualization allows cloud providers to hide the complexity of underlying physical resources and improve resource utilization. Further, virtualization provides performance isolation so that cloud providers can guarantee resource quality to customers and make it harder for hackers to attack the system. Due to virtualization, cloud workload can be represented as instances (virtual machines or containers) to be executed. Thus, one of the main jobs for the cloud is to map the instances to resource availability to provide a minimum amount of resources for application to maintain a desirable quality.

From the user point of view, Cloud Computing relieves them from costly data center operations, shorten production processes, as well as allow them to rapidly deliver their product with high reliability and scalability. From the cloud provider point of view, Cloud Computing creates opportunities for profit as the cost of employing a large-scale data center is lower than that of multiple medium-sized ones. They can also earn more profit from the capability of multiplexing load across user groups [9] where the same quantity of resources can be “sold” multiple times. Therefore, after having emerged for more than 10 years since Amazon introduced Elastic Compute Cloud (EC2) and AWS in 2007, Cloud Computing nowadays captures a large portion of both Internet and enterprise IT computing services [25].
### 2.2 IT Server Workloads and Virtual Machines

IT server workloads are the most common cloud workload and represent major applications in data centers. They can be simple applications, which can be packed into a single machine such as workload monitoring or logging. Yet they can be a complicated distributed applications, e.g., databases, storages, Map Reduce application, etc. In spite of size, IT server applications typically have steady resources demand and last for a very long time. Thus, the IT server workload needs permanent resource provision for efficient execution.

Due to its popularity, cloud providers put substantial effort to serve IT server well with their Virtual Machines (VMs). By offering VMs, cloud providers let users have full control over the granted resources. This not only enables cloud resources to cover a wide range of applications but also lets users have high flexibility to deploy multiple services for their needs. And to meet the availability requirements, cloud providers even let users keep VMs indefinitely.

Nowadays, cloud providers offer VMs either through reservations or on-demand fashion. In reservation, resource capacity, configuration, as well as price, are mostly set up through out-of-band (i.e., human-to-human) communications between customers and providers. Resources are often delivered via annual contracts so users have full control over the reserved VMs but they have to pay for them for the whole contract length even if the reservations are inactive. This limitation deters users with high variability in their workload so cloud providers offer an alternative: on-demand instances. With this offering, users no longer have to plan or organize their resource usage, in return for a higher price per VM-hour. They can request new VM instances anytime the computation load increases, keep them as long as they want and release them when the load decreases. This elasticity plus pay-as-you-go pricing makes the cloud become a very efficient environment and economically attractive for IT server applications.
2.3 Bursty Workload and Finite Duration Resources

In the cloud, there are many applications do not have a steady load. Their demand, however, changes frequently over time. For example, WeChat, one of the largest online message systems in China, often experiences significant growth in user demands from 5 pm to 11 pm everyday [68]. Workload changes can even faster, probably in minutes, such as a website recording the score of ongoing soccer matches would receive much more access when a goal is scored. In such situations, applications may need to add more computation resources to ensure request processing goes smoothly. We call workload generated from such a situation bursty workload.

2.3.1 Serverless

While only introduced a few years ago, Serverless has already taken a huge growth in usage [51] and received much attention from the research community [24, 8]. The core technology behind serverless computation is *cloud functions* written to perform specific tasks when some conditions are met. With serverless, users only have to implement functions, which typically have a few lines of code and find events that associate with them. No server deployment and management are needed. This totally relieves users from resource provision and management.

Serverless supports many concurrent invocations at best effort within strict concurrency limits [24, 37, 5]. Resources are allocated and released per-invocation, with execution as short as a few seconds. In contrast to a virtual machine service, there is no notion of a continuous resource and no indefinite resource commitment; invocation execution is strictly limited (5 minutes, 15 minutes). By associating function resources with requests, serverless executions increase as demand increases and when demand decrease, due to short lifetime, serverless executions also quickly decrease accordingly. Thus, utilizing serverless enables applications to autoscale automatically.

Serverless has fine-grain pricing [1, 2], typically in less than a second so cost scales both
down and up based on total use independent of its temporal. Consequently, despite a much higher cost per unit resource (gigabyte-seconds) compared to virtual machine offerings, the serverless model is cost-effective and popular among new applications, and is reputed to be the “fastest-growing element of cloud workload”.

Serverless functions are stateless and there is no direct way to let functions communicate with each other. Thus, exploiting serverless often requires an extra state holder such as shared storage or an operator server. [38] defines this combination (Serverless computing + supporting frameworks) as Function as a Service (FaaS). Another limitation of serverless is the lack of any guarantee of allocation rate. Thus, the bursty application has no idea of exactly how fast they can respond to workload changes. These are consistent with the “best-effort” resource allocation of the cloud system. We address these aspects in our work.

2.3.2 Volatile Resources

As the cloud market expands, cloud providers continuously increase their capacity to meet unexpected load spikes and growing demands. However, as users come and go together with high variability of workloads, foreground load varies over time leaving a fluctuating excess resource. These resources are idle but still consume energy, occupy space and cost of maintenance, etc. so it is important to extract value from them. In fact, cloud providers sell excess resources as volatile resources with less reliability and availability in return for a cheaper price.

In general, the volatile instances are VMs but users may fail to ask for new volatile instances and cannot keep them indefinitely. Cloud provider holds the right to reclaim them whenever they need. Many commercial volatile resource offerings have been established. Examples include Amazon Spot Instances [6] and Google Preemptible VM instances [31]. Spot instances are offered in an auction-like mechanism in which users bid for the highest price they willing to pay and Amazon allocates spot instances to those who place the highest bids. The lowest bid receiving spot instance defines the market clearing price. Allocated
instances are charged at this price regardless of the bid price. Each user keeps his/her spot instance until the market price exceeds his/her bid. Google’s preemptible instances are offered in a simpler manner. These instances have fixed prices of 20% of the on-demand price. Google gives no information about how revocation decisions are made but does notify the VM in 30 seconds before termination.

In spite of different pricing policies, volatile instances are much cheaper than traditional VM in most of the case. In fact, volatile VM is usually 90% cheaper than the reliable, indefinite duration VM with similar compute-memory-storage configuration. Due to low prices, volatile instances are very attractive to batch jobs and fault-tolerance workloads [19]. There are also active studies trying to mitigate the unreliable of volatile instances for delivering cost-effective solutions [32, 64].

2.3.3 Amazon Burstable Instances

Burstable instances (or T2 instances) [7] are intentionally created for bursty applications that have low CPU utilization for most of the time but may have high CPU demand for some short durations. Burstable instance performance is defined by a percentage of CPU utilization called CPU baseline. If instance actual CPU utilization is small than CPU baseline, it will receive an award called CPU credit. CPU credits are allocated continuously and can be accumulated until they reach the credit limit, which is the number of credits accumulated in 24 hours. A burstable instance can handle bursts by burning accumulated credits to push its utilization to go beyond the CPU baseline. If CPU credits get exhausted, burst instance performance is limited by CPU baseline until the load goes down and the instance starts building credits again. Credits are timeout after 24 hours.

From November 2007, Amazon launched Unlimited Burstable Instance which is similar to the “standard” burst instance except that instance can “borrow” the credits of the entire day’s worth of future credit to continue performing above the CPU baseline. Even if the day’s credits are exhausted, the instance can still deliver burstable performance by paying more
for the extra CPU utilization. With CPU credit accumulation, burstable instance enables users to estimate resource costs based on average use instead of maximum demand. Thereby, serving bursty applications using burstable instances would generally be more cost-effective than traditional VMs.

However, the burstable instance is not a fit solution for all bursty applications. For example, application with bursty use of other resources such as memory or network would get no benefit. Furthermore, the credit is only applied within an instance so the burstable performance is hard limited by instance capacity and distributed applications get no or limited benefits from them.

Even if instance capacity is sufficient for absorbing burst demand, using burstable instance alone is not sufficient for satisfying real-time requirements. Because applications utilize CPU beyond the baseline through accumulating credits. If bursts occur as a cluster (i.e., 2, 3 or even more bursts appear in a row) then credit would be quickly burnt out, leaving some bursts to fail to meet real-time deadline. So, in the worst case, applications still have to statically allocate a large set of burstable resources to make a guarantee.
CHAPTER 3
BURSTY, REAL-TIME WORKLOADS AND REAL-TIME SERVERLESS

In this chapter, we characterize and formally define real-time bursty applications. Based on the definition, we discuss about key challenges that make it difficult to serve applications possessing bursty, real-time characteristics with existing cloud resources. After that, we propose a new resource type called real-time serverless with a guaranteed resource allocation rate to overcome these challenges. At the end of the chapter, we state research questions needed to be answered to show real-time serverless is a good match for bursty, real-time workloads.

3.1 Real-time, Bursty Workload

Bursty real-time workloads are characterized by unpredictable bursts of computing load with critical real-time deadlines. Bursts are often triggered by external events (e.g. a traffic incident, a flash crowd in the physical world, cyberattack, a trending meme in an app, etc.), and responses must trigger critical action (emergency response, shutdown) with hard, real-time deadlines. Bursts can increase application compute requirements by 10 or 100-fold, or even more and last for durations of seconds to minutes. These applications must deliver guaranteed high quality with real-time deadlines because application quality decreases rapidly with latency.

An example of bursty, real-time application is traffic monitoring which continuously analyzes video streams of traffic recorded by cameras installed at intersections (Figure 3.1). The application is interested in unusual events such as car crashes or a pedestrian falls down, etc. When such event happens, it triggers an in-depth analysis to understand the situation for making proper decisions (e.g., call police, broadcast warning signal, etc.). The in-depth analysis uses a sophisticated model on video streams of high resolution, and eventually creates
significant high resource demand. Further, because the event is crucial, the analysis needs to be processed as fast as possible. Prolonged computation latency decreases application value/quality (e.g. delay to call emergency medical services after an accident may lead to severe consequences).

Therefore, computation value or quality of bursty, real-time applications depends on both analysis accuracy and processing speed. Fail to either making a proper decision or deliver decisions in time results low value/quality. Unlike analysis accuracy, processing speed is not defined by the application itself but also depends on resource provider in the sense of how fast they deliver resources.

### 3.2 Key Characteristics and Challenges

We generalize the definition of the workload. In particular, we define bursty, real-time workload by two key characteristics:

- **Bursty**: The demand of the workload is not stable but often experience bursts which have the following attributes.
  
  - **Short duration**: The burst length is short, typically in seconds or a few minutes.
– **Low frequency**: Burst rarely happens, the interval between two consecutive bursts is typically long compared to their length (10x or 100x longer).

– **Random**: Because bursts are triggered by external causes, their occurrence is hard to predict and should be treated as random events.

– **High computation demand**: During the burst, computation demand increases by 10 to 100-fold or even more.

• **Real-time**: Computation demand of bursts must be served within a strict deadline. Thus, the value of computation or application quality decreases rapidly as the latency of resource allocations or computation completion increases.

The characteristics above reveal that in order to serve the burst efficiently, the application must have the capability of handling computation quickly. This is only possible if the underlying cloud provider is able to make a large quantity of resources available at burst time. Unfortunately, it is very tricky to make it happen. Because of bursts’ unpredictability, applications cannot prepare for the burst in advance. The problem is even harder when cloud providers serve multiple bursty, real-time applications at the same time. In this context, bursts may interfere with each other resulting much higher computation demand that difficult to serve in time without planning.

Even if the application could predict burst occurrence and duration well (possibly based on periodic behavior of external causes), it is still difficult to cooperate with the cloud provider to autoscale to actual use. The reason is that the load changes very fast, in minutes or even a few seconds. In such speed, it takes a lot of effort including identify idle resources, configure them to meet application’s specification, data movement may also require, etc. Furthermore, scale up within short durations possible creates unexpected spikes in resource commitment, destroys resource stability structure, and deteriorates other services performance. Therefore, cloud autoscale can only effectively respond to slow changes of load, in hours or even days, limiting the application’s capability of scale down to actual use of
In summary, we believe that bursty, real-time workload poses the following challenges:

- Enable high quantity of resources available for a short duration of time.
- Resource availability must be robust against burst uncertainty.
- Low resource management cost and reasonable pricing for applications.

Cloud services dedicated to bursty, real-time applications must be able to address all of these challenges. However, as we will see in the next section, none of today cloud offering satisfies these requirements.

### 3.3 Limitations of Today’s Cloud Resources

As mentioned in Chapter 2, today cloud offers different types of service at different extents of convenience and control over resources (Figure 1.1). None of those addresses the real-time constraints, so the best bursty, real-time applications can do, is to take VMs – resources in which they have the highest control – to freely deploy their own resource management to meet real-time deadlines. This means the application must pre-allocate VMs in advance. The quantity of pre-allocated VMs must sufficient to absorb burst demand because low allocation resources.

(a) Low allocation

(b) High allocation

Figure 3.2: Bursty, real-time workloads in traditional cloud resources give either poor application quality (low allocation) or high cost (high allocation)
would result poor application quality (Figure 3.2a). This approach is very expensive as according to bursty characteristic, the VMs are under-utilized most of the time so a major part of application payment would go to unused resources (Figure 3.2b, used resources are shaded in green). The application may consider dynamic allocations if it can predict burst occurrences. However, the hourly granularity of billing makes dynamic allocations for bursts of seconds to minutes unproductive in reducing costs.

The application can opt to other resource types enabling autoscaling such as serverless to scale down the cost. However, none of these resource types provide any guarantee of allocation rate or allocation latency. The problem is, that for all of these services, the allocation is a best-effort activity. So, a bursty application cannot guarantee any level of quality with these resource types. In fact, serverless invocations can be delayed for more than 20 seconds [63, 3], a significant overhead for applications such as face detection or gaming. Further, cloud autoscale needs time to take effect. This is too long compared to burst durations, measured in seconds or minutes, so autoscale resources are not a good match to bursty, real-time workloads.

To make the cases clearer, let us revisit the video analytic for traffic monitoring example. Assume that interesting events occur at an average rate of 2 per hour, and each initiates a burst of computation with a duration of 2 minutes. This is a duty factor of 6.6%. While monitoring for interesting events, low-quality video (224x224 pixel resolution, 30 frames/second) is analyzed with a neural net of 50 layers. When burst happens, full frames (resolution of 1024x1080), and a much wider and deeper neural net (150 layers) is used, producing burst computation height of 60x higher per frame compared to normal monitoring for 2 minutes. Suppose we need 1 CPU instance\(^1\) for normal monitoring, then processing a burst frame requires 60 CPU instance-frame times of work.

With a traditional VM offering, reserving 60 CPU instances in advance would ensure each burst frame can begin processing right away at the cost of wasting 98.3\% of CPUs for

---

\(^1\) We use instances as a unit of computing power throughout the thesis.
93.3% of the time. Such waste increases the cost of real-time video analytic and may render many applications infeasible.

With dynamic allocation (spot, on-demand, serverless), due to no allocation guarantee, the computation quality cannot be guaranteed. Applications may fail to allocate new instances when the cloud is “busy” or experience extremely low allocation latency. Both situations lead to unacceptable burst response, preventing the video analytic from making any promise of timeliness in a decision for each event.

3.4 Approach: Real-time Serverless

We propose a new type of resource called Real-time Serverless. The new resources fundamentally inherit all characteristics of serverless (e.g., AWS Lambda Function, Google Cloud Function, etc.) plus the capability of offering computation reliably or in a timely fashion (Figure 1.2). More precisely, real-time serverless are computation instances organized as pools. Each pool is characterized by 2 key parameters

- **Guaranteed Allocation Rate**: The minimum allocation rate guaranteed by the pool to a single task.

- **Fixed Duration**: The maximum duration that a Real-time Serverless instance can be used. After reaching this limit, the allocated instance is reclaimed by the cloud provider. The duration is limited to a few up to 10 or 15 minutes.

For example, a Real-time Serverless pool of (10 instances/second, 5 minutes) allows any single application to allocate at least 10 new instances for every second. Each of those instances lasts for at most 5 minutes. Current commercial serverless offerings can be thought of a supporting a guaranteed allocation rate of zero.

For accounting, real-time serverless instances are charged proportionally to their runtime. And because of short duration, fine-grained resource pricing (i.e., 100ms pricing or less) is
applied. This is attractive for bursty, real-time workloads because of their low duty factor (ranging from a few to probably 20 percent).

### 3.5 Support Bursty, Real-time Workloads with Real-time Serverless

Guaranteed allocation rate and fixed duration offered by real-time serverless significantly broaden the capability and operation space of applications. First, the guaranteed allocation rate enables them to know how much resources they could utilize at present and even in the future. Second, the fixed duration lets them know the time and the computation capability they can exploit from the cloud. Such information is invaluable for scheduling computation demand flexibly to meet deadlines, adjust quality as well as utilize resources much more efficiently.

For bursty, real-time workload, knowing resource availability in advance allows the application to easily *scale up* when bursts happen. And when burst finishes, fixed duration let resource consumption *automatically* and *quickly* scale down. Furthermore, real-time serverless’s fine-grain pay-as-you-go pricing policy allows cost to scale up and down proportionally to resource consumption and thus, let real-time bursty application service bursts efficiently
at a reasonable cost.

We visualize these benefits in Figure 3.3 by comparing real-time serverless with the traditional serverless. In Figure 3.3a, the serverless allocation is a best-effort activity so the application may receive new instance at high latency, causing a negative impact on burst processing, possibly make application fail to meet the real-time deadline. On the other hand, guaranteed allocation rate bounds allocation latency so the application can bound their burst processing time as well. Thus, if the application uses the real-time serverless pool of sufficient allocation rate, it can scale down resource utilization to actual use while still able to meet the real-time deadline (Figure 3.3b).

For quantitative comparison, let us revisit the video analytic example. Assume that the application process a 2-minute, 30 fps video during the burst. If it uses one real-time serverless instance per frame, then the allocation rate of 30 instances per second enabling processing of each burst frame to begin without delay. If the application uses the real-time serverless pool with a lower allocation rate, says 10 instances per second, then it would take a longer time to allocate needed instances. Consequently, each burst frame (after the first), would be delayed by an increasing amount. The last frame of the burst (30 fps * 120 seconds = 3600), would start processing with a full 2 minutes of delay. However, even this case is better than serverless alone, at least the application can provide a hard guaranteed bound on this latency of 2 minutes. Such a bound allows a quality (response time) for the application to be defined.

Furthermore, since real-time serverless instances are allocated dynamically when needed, resource cost is only for the actual resource used for burst processing (Figure 3.4a, resource cost is shaded in orange). This quantity is lower than VM cost where pre-allocated resources are determined by the burst peak and they are charged all the time, even when they are idle (Figure 3.4b).

Goes back to the VA example, a burst lasts for 2 minutes so at a frame rate of 30 fps, 3,600 burst frames would be generated per burst. Assume that processing these frames requires 10
CPU instance*hour then to meet the real-time requirement (i.e. process all frames within 2 minutes), application with VM offering must reserve \( \frac{600}{2/60} = 300 \) CPU instances. On the other hand, if the application uses 1 CPU real-time serverless instance per frame, then 3,600 real-time serverless invocations are triggered per burst, each lasts for 10 seconds resulting a cost of \( 3,600 \times 10 = 36k \) CPU instance*second or 10 instance*hour per burst.

Assume that burst occurs twice within an hour, then utilizing real-time serverless would incur a cost of 20 CPU instance*hour, while the cost of VM is 300 CPU instance*hour. If a real-time serverless instance is charged at 6x higher than a VM with similar computation power (according to today’s pricing) then using VM is 2.5x more expensive than real-time serverless. The only way to reduce VMs cost is to decrease VM reservation but this makes VM fail to meet the real-time requirement.

### 3.6 Research Questions

The example above demonstrates the performance and economic advantages of real-time serverless. However, it is not enough to show that the real-time serverless solves problems posed by real-time bursty workload. In order to show that real-time serverless is a sufficient solution for real-time bursty workloads, we believe the following question should be answered...
properly

- Does real-time serverless enable bursty, real-time applications to guarantee high burst processing quality?

- Does real-time serverless robust against a wide range of bursty, real-time properties?

- Is real-time serverless economically attractive to application and cloud providers?

- Can real-time serverless be provided efficiently?

We will answer these questions in the coming chapters using a mixed of analytical and empirical methods and some real world examples.
CHAPTER 4

ANALYTICAL MODELING AND EVALUATION

In this chapter, we consider the video analytic (VA) application as a representative for bursty, real-time workload and use it for modeling key characteristics of the workload. We also construct models for traditional cloud resource types and real-time serverless. We relate the resource types with computation quality and cost, use these as metrics to show the advantages of real-time serverless.

4.1 Analytical Model

In Table 4.1 we define notation for the key attributes of bursts ($H$, $D$, and $\lambda$). Here we assume that burst frames are created at a rate of 30 frames per second. Each burst frame has fixed computation demand $H$, measured by the number of instances needed for processing the frame before the next comes. We call the interval between the arrivals of two consecutive frames frame-time ($ft$) and use it as a unit of time for the rest of the thesis. Note that with this time unit, burst frames arrive at rate of 1 frame per frame-time so burst duration ($D$) is equal to the number of frames in a burst.

We use duty factor $DF$ which is the fraction of burst frames over all video frames

$$DF = \lambda \cdot D$$

(4.1)

to measure the burst frame’s frequency of appearance. Increasing burst duty factor would create more burst frames, thereby increases the difficulty of serving the workload. However, if the duty factor is high then the workload becomes more stable (always has high demand), and less bursty. Therefore, we focus on a single bursty application with a low duty factor, below 25%. We use $\tau$ to represent the application’s need for fast burst processing. Application with high $\tau$ has a stricter deadline and requires a higher speed of resource delivery for frame processing. Figure 4.1a visualizes a single burst and its associated parameters.
Table 4.1: Terms and Notation

Next, we model VA’s bursty and real-time characteristics. To capture the uncertainty of burst occurrence, we model burst arrivals as a Poisson process with arrival rate $\lambda$. To capture the VA real-time requirement, we use value or quality to represent the satisfaction of VA on resource availability in helping it dealing 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 value that decays with delay. Specifically, for VA, we use an exponential decay with frame processing latency with $\tau$ as the critical time constant. That is, the value of a single burst frame $i$, delayed by $l(i)$ frame times:

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

(4.2)

Figure 4.1b shows how value decreases as latency increases. Here, $\tau$ is chosen to make value drop by half per minute. Note that the latency $l(i)$ does not include frame processing time because we are only interested in the capability of cloud resource management to
quickly deliver resources to VA during the burst. Furthermore, given that VA knows the computation demand, this is their responsibility to choose instances that are good enough to process frames within its desired time.

Finally, we model how cloud providers deliver and grant permissions over their resources to users. We consider cloud resource types as pools of identical instances. Each pool is characterized by two parameters:

- **Instance Runtime** $R$ (frame-time) represents the time limitation for the task that application put into the instance as well as how fast cloud providers can reclaim resources for reuse and/or rebalance resource distribution.

- **Guaranteed allocation Rate** $A$ (instance per frame-time) represents the capability of the pool to guarantee resource availability as well as the information about resources application can utilize at present and in the future.

We group traditional cloud resource offerings as Unreserved, Indefinite runtime instances (UI) as most of them share those characteristics. That is, they are unreserved, so when a
resource allocation request arrives, the system may choose whether or not to grant it. UI resources are also granted to application indefinitely. In other words, UI instances have no promise on resource allocation rate and allow VA to keep allocated instances as long as they want, so \( A_{UI} = 0 \) and \( R_{UI} = \infty \). On the other hand, real-time serverless (RTS) considered the rate as its instance property and limit instance runtime for scalability (see Section 6.1) so \( A_{RTS} \) and \( R_{RTS} \) are constants. For simplicity, we set \( R_{RTS} = H \) so all real-time serverless instances live just long enough to serve burst frames. For cost comparison, we relate UI and RTS instances cost \( (C_{UI} \) and \( C_{RTS} \)) via a constant \( k > 0 \).

Given the resource management model, we now consider how the VA utilizes each resource type for serving the bursts. For UI, because \( A_{UI} = 0 \), VA cannot allocate instances at the time of burst occurrence. It has to pre-allocate a fixed quantity of \( P \) instances (see Figure 4.1c). If \( P \geq H \) then VA has sufficient resource for processing all burst frames intermediating or \( V_{frame}(i) = V_{max} \) for all frame \( i \). If, however, \( P < H \), then processing latency appears. Assume that VA processes frames in FIFO fashion and uses all \( P \) instances for each frame processing then the latency for frame \( i \) can be expressed as

\[
l_{UI}(i) = \frac{i \cdot H}{P} - i
\]

where \( i \cdot H/P \) reflects the processing time of preceding burst frames and \( i \) is arrival time for the \( i \)-th frame. If no instances are pre-allocated \( (P = 0) \), the latency is infinite. The effect of insufficient allocation can be see in Figure 4.1c, \( P < H \) make the second frame waits for \( l_2 = 2 \) frame-times before being processed. Worse, because of FIFO scheduling, such latency is accumulated for later frames. The 7-th frame, for example, experiences a much higher latency of 7 frame-times. Also note that because burst occurrence is uncertain, VA has to keep the pre-allocation all the time to guarantee the desired computation value. This, however, incurs resource waste as we mentioned in previous sections.

In contrast, Real-time Serverless can provide instances at the maximum guaranteed rate of \( A_{RTS} \). If we assume that the application allocates instances at that maximum rate until
(a) Distribution of burst frame value with various $A_{RTS}$

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

Figure 4.2: Achievable burst frame value for UI and RTS with various guaranteed allocation rate. ($H=140$, $D=3,600$, $\tau=2,607$ (1/2 per minute))

all pending frames are being processed. For $A_{RTS} \leq 1$, a burst frame that arrives at time $i$, will be serviced starting at $i/A_{RTS}$. Consequently,

$$l_{RTS}(i) = i\left(\frac{1}{A_{RTS}} - 1\right)$$  \hspace{1cm} (4.4)

where each frame is processed on a single instance with processing time $H$ per frame. Figure 4.1d shows how VA uses RTS instances for dealing with bursts. Since RTS instances are allocated right after burst frames are detected and freed right after frame processing finishes, resource waste no longer exists.

In the rest of this chapter, we will use the model constructed above to show that real-time serverless is a useful and sufficient solution for bursty, real-time workloads. First, we will show that real-time serverless robustly enables applications to guarantee computation quality. Furthermore, the new resource type opens the opportunity of adjusting resource usage for applications to optimize computation quality and cost under restricted conditions. Second, we will analyze the economic attractiveness of real-time serverless to users and cloud providers. The solution not only reduces users’ resource cost but also can plausibly increase the cloud provider’s revenue simultaneously.
4.2 Real-time Serverless Capability

4.2.1 Guaranteeing Application Quality

We believe the key characteristic making real-time serverless a good match for bursty, real-time workloads is its guaranteed allocation rate. To see why, let us compare the application quality achieved using UI and RTS resources, exploring a variety of guaranteed allocation rates. For the baseline, we use UI resource pool alone with $P = 1$ instance, reflecting an acceptable cost, and $A_{RTS} = 0$ (in blue in Figure 4.2a). For UI, no additional resources can be allocated for each burst, so only a tiny fraction of maximum value is delivered (the frames mostly at left). For RTS, as the guaranteed allocation rate, $A_{RTS}$, is increased, the application can service a burst by allocating resources more and more rapidly. Even a small $A_{RTS}$ significantly increases the number of frames achieving close to the maximum value (orange), and further increases in $A_{RTS}$ (green, red) improve the situation dramatically. For example, with $A_{RTS} = 0.6$, the application can ensure that all frames exceed 40% of the maximum value. And as $A_{RTS}$ increases towards 1 instance/frame time, a growing frame value guarantee can be achieved, reaching 100%. This illustrates that using real-time serverless helps applications improve computation value/quality.

Another way to think about application quality is to ask what fraction of frames achieve a particular fraction of maximum quality. We plot this metric versus guaranteed allocation rate, $A_{RTS}$, in Figure 4.2b. To achieve 50% of the maximum value for even half of the frames, the video analytic application requires an $A_{RTS}$ of 0.5 instances/frame-time. To achieve 50% of the maximum value for 100% of the frames, an $A_{RTS}$ of 0.67 instances per frame time is needed. At the high end, to achieve 90% of the maximum value, 0.85 instances/frame-time is required for 50% of the frames, and 0.9 for 100% of the frames. At $A_{RTS} = 1$, the application can deliver 100% of the maximum value because an instance can be allocated for each arriving frame. This illustrates that RTS enables bursty applications to provide a guarantee of high quality. The results are essential because they suggest that with a proper
choice of \( A_{RTS} \), applications are able to meet any target quality. Such capability unlocks rational designs that open more space for applications to operate and exploit resources more efficiently.

In summary, our analytical model shows that adding real-time serverless resource pools can enable bursty, real-time applications to guarantee high computation value. In fact, the results show that the allocation rate is the critical enabler of high value. This is striking as no major cloud services provide any resource types with guaranteed allocation rate.

### 4.2.2 Robustness against Burst Shape

The degree of real-time serverless impact on quality depends on burst shape (height\*duration) and arrival rate. In this subsection, we consider various burst shapes, holding total computation demand (i.e. \( H \cdot D \)) constant. For simplicity, here we consider one burst at a time (burst interference will be covered in Chapter 5). For each burst, we compute the burst value, which is the sum of frame values for burst frames in the burst as below:

\[
BurstValue = \sum_{i=0}^{D-1} V_{frame}(i) = V_{max} \sum_{i=0}^{D-1} e^{-\frac{i}{\tau}}
\]  

(4.5)

If a RTS pool is available then

\[
BurstValue(A_{RTS}) = V_{max} \sum_{i=0}^{D-1} e^{-\frac{i}{\tau}(\frac{1}{A_{RTS}}-1)}
\]  

(4.6)

Critical factors are \( D \) (duration), the resources available \((A_{RTS})\), and the value decay \((\tau)\). Failing to keep up with burst demand (i.e. allocation delay) will directly lead to burst value deterioration.

In Equation 4.6, value deterioration due to limited resources can be seen in \( e^{-\frac{i}{\tau}(\frac{1}{A_{RTS}}-1)} \) which decreases as a function of \( i \), but at a rate determined by \( A_{RTS} \). For longer durations \((D)\) this produces more terms in the sum, but each is exponentially smaller, so the sum grows slowly with \( D \). The only way to mitigate this effect is to have a larger \( A_{RTS} \). Because
we only process frames on a single instance, burst height $H$ does not affect value.

![Figure 4.3: Impact of burst shape on burst value, for varied guaranteed allocation rates, $A_{RTS}$. Solid lines are achieved burst value; dashed lines are maximum potential burst value.](image)

We plot these results in Figure 4.3. First, increasing $A_{RTS}$ always improves application value regardless of the burst shape. And a finite value of $A_{RTS} = 1$ is sufficient to achieve 100% of the maximum application value. Further, our plot shows five different burst duration scenarios, varying from 900 frame-times (30 seconds) to 14,400 frame-times (8 minutes); holding total compute demand constant. At a guaranteed allocation rate, long bursts always achieve higher burst values than shorter bursts – because they have more frames. For example, at $A_{RTS} = 0.6$, value for $D = 1,800$ frame-times is about 1,400 smaller than 2,300 achievable by $D = 3,600$, 3,300 by $D = 7,200$ and so on. However, the longer bursts require a higher allocation rate in order to achieve the same fraction of maximum potential burst value. For example, $A_{RTS} = 0.3$ is needed to achieve 50% of maximum value at $D = 1,800$ frame-times but if $D$ increases to 7,200 frame-times, allocation rate of $A_{RTS} = 0.63$ achieves the same level of quality.

In summary, burst shape can affect the guaranteed allocation rate needed to achieve a high fraction of maximum value. In fact, longer bursts required a higher allocation rate. And at a finite allocation rate $A_{RTS} = 1$, real-time serverless can deliver the maximum potential value regardless of the burst shape. These studies confirm that real-time serverless can robustly deliver high burst value for a wide range of burst shapes.
4.2.3 Optimizing Resource Usage

Besides guaranteeing value, guaranteed allocation rate lets applications know the quantity of resources it can use at present and also in the future. Such information is essential since it brings application opportunity to look ahead at resource availability to proactively plan frame processing and resource allocation. The application can also map resources to computation demand. From these, applications can rationally design optimal resource usage strategies to meet different quality/cost requirements.

In this subsection, we consider several scenarios that require careful application resource usage strategies to get as much benefit from resources as possible. In particular, we consider two scenarios:

1. Maximize burst value, given that application has a fixed computation budget (cost) of $C$ (instance\*frame-time) per burst.

2. Minimize computation budget, given that application want to achieve target value $V_{\text{target}}$.

Now, we will prove that, in these situations, real-time serverless is sufficient to enable the best strategies for applications. First, let us consider Case 1. Processing one burst frame takes $H$ instance\*frame time, so with the cost $C$, the application can process at most $\frac{C}{H}$ frames. Assume that $V_{\text{max}} = 1$ then the maximum burst value is

$$BurstValue_{\text{max}}(C) = \frac{C}{H}$$  \hspace{1cm} (4.7)

With RTS, the application knows exactly when new instances will be available so it can selectively choose $\frac{C}{H}$ out of $D$ frames, which arrive right at the time a new instance can be allocated to maximize their value. This is done with our Greedy+Drop strategy: whenever a burst frame arrives, the application asks the RTS pool for a new instance. If the pool can allocate a new instance right at this time, the application will obtain the instance and use it
Algorithm 1 Greedy+Drop Strategy

Allocate instances from a pool of rate $A(\text{RTS}) = \frac{C}{H \cdot D}$

for each burst frame arrival do
    if there is a RTS instance can be allocated then
        Allocate 1 RTS instance and use it to service the frame
    else
        Drop the burst frame
    end if
end for

for the new burst frame (so the frame achieves maximum value). Otherwise, the application just drops the burst. Because burst frame arrives at every frame-time during burst time, this frame selection strategy ensures that application can serve $A_{\text{RTS}} \cdot D$ burst frames right after their arrival. If application set the allocation rate $A_{\text{RTS}} = \frac{C}{H \cdot D}$ then the total computation cost is

$$\text{ComputationCost}(A_{\text{RTS}}) = A_{\text{RTS}} \cdot D \cdot H = C \quad (4.8)$$

satisfying the cost constraint. Also, the burst value is

$$\text{BurstValue}(A_{\text{RTS}}) = D \cdot A_{\text{RTS}} = \frac{C}{H} = \text{BurstValue}_{\text{max}}(C) \quad (4.9)$$

Thus, the Greedy+Drop strategy allows the application to maximize value given a fixed cost. The strategy is formally described in Algorithm 1. Note that the Greedy+Drop is just one of many possible strategies that applications can use to maximize the value. However, it is sufficient to confirm the capability of RTS resources to enable applications to maximize value given a fixed cost.

We now consider Case 2. Since every burst frame takes the same processing cost of $H$ instance*frame-time, minimizing the cost per burst requires the application to process as few frames as possible. For the target value $V_{\text{target}}$, the lowest number of burst frames to be processed is $V_{\text{target}}$, achievable by selectively processing $V_{\text{target}}$ out of $D$ frames without delay. Therefore, the minimum cost, given $V_{\text{target}}$, is
\[ ComputationCost_{\text{min}}(V_{\text{target}}) = V_{\text{target}} \cdot H \] (4.10)

Recall that in the Greedy+Drop strategy, \( \text{BurstValue} = A_{RTS} \cdot D \). So with RTS resources, \( V_{\text{target}} \) is achievable if the application applies the Greedy+Drop strategy with \( A_{RTS} = V_{\text{target}} / D \) at the cost of

\[ ComputationCost(A_{RTS}) = A_{RTS} \cdot D \cdot H = V_{\text{target}} \cdot H = ComputationCost_{\text{min}}(V_{\text{target}}) \] (4.11)

Thus, the Greedy+Drop strategy also minimizes the computation cost. The results confirm that applications using RTS can minimize the cost for any target value. Therefore, Real-time Serverless is sufficient to enable the best strategies for bursty, real-time applications.

![Figure 4.4: Burst value vs. Computation Cost](image)

Also note that from the Greedy+Drop algorithm, we can relate burst value and computation cost as follows

\[ \text{BurstValue} = \frac{\text{ComputationCost}}{H} \] (4.12)

And because Greedy+Drop is an optimal algorithm, the maximum value is a linear function of computation cost. This function is also invertible:
\[ \text{ComputationCost} = \text{BurstValue} \cdot H \] (4.13)

That is, the optimal computation cost also increases linearly with target burst value. Burst value and computation cost relationship is visualized in Figure 4.4. This result is important for engineering as the application can infer the cost for a specific target value and vice versa, thereby enable rational designs to extract more value from cloud resources.

In summary, real-time serverless enables application flexibility in computation scheduling and resource consumption so that they can employ efficient strategies to get the maximum value out of cloud resources. Even in strict scenarios, real-time serverless is still sufficient to enable the best strategies.

### 4.3 Economic Attractiveness of Real-time Serverless

Now we develop an analytical model to assess whether real-time serverless is more economically attractive than traditional cloud resources. First, let us start with computing the resource cost that the application has to pay per burst. When using RTS, the application does not pre-allocate resources in advance so the cost scales with demand. In particular, processing one frame is charged for \( H \cdot C_{RTS} \) so the total cost per burst is

\[ \text{Cost}_{RTS} = H \cdot D \cdot C_{RTS} \] (4.14)

For the UI pool, we assume pre-allocation of \( H \) instances, that maximizes burst value (as in Figure 1.3). Pre-allocation allows resource charges for the idle periods between bursts \((1/\lambda)\), produces the cost for a single burst:

\[ \text{Cost}_{UI}(P) = \frac{1}{\lambda} H \cdot C_{UI} \] (4.15)

For comparing real-time serverless with traditional offerings, we relate the cost (price) of
RTS to UI resource with ratio $k$ in cost/instance-second as below:

$$C_{RTS} = k \cdot C_{UI} \quad (4.16)$$

Real-time serverless instances are only allocated when bursts happen, so we need to include the workload’s duty factor ($DF = D \cdot \lambda$). Figure 4.5 shows the resource cost normalized to $Cost_{UI}$ for UI and RTS for two duty factor ranges. We change the duty factor by keeping the burst arrival rate $\lambda$ fixed and varying the duration $D$. Fixing $D$ and varying $\lambda$ have the same effect. We set $P = H$ and $A_{RTS} = 1$ so both approaches achieve 100% of the maximum burst value. For RTS, we plot total cost at various cost ratios ($k$).

Starting at Figure 4.5a, at low duty factor such as $DF < 0.025$, RTS resource premiums of 10-20x can still be cost-effective for bursty applications. For moderate duty factors, RTS is more cost-effective for $k$ as large as 8. For high duty factors, approaching 0.25 (25%), RTS is used more often, and can eventually approach and exceed that of UI. For instance, when $DF > 0.125$, RTS is less cost-effective for $k > 6$. This suggests that premiums of 6-8x may still be attractive for bursty applications – much as 6x premiums for serverless resources are still attractive [1].

These results are essential. From users’ point of view, serving real-time bursty appli-
cations with RTS is much cheaper than UI, enabling bursty applications to deliver desired computation value at an attractive cost. From cloud provider point of view, real-time serverless instances have a higher value per unit resource than UI so provide such resource plausibly increases their revenue.

Now let us consider the case of multiple real-time, bursty applications. Assuming that application bursts arrive independently according to a Poisson process with arrival rate $\lambda$. For the UI approach, each application pre-allocates $P$ instances. For RTS, because each frame processing take $H$ frame-times, the cloud provider can use $P$ instances to allocate resource at a per-application rate of

$$A_{RTS} = \frac{P}{H} \quad (4.17)$$

Consequently,

$$L_{RTS}(i) = \frac{i}{A_{RTS}} - i = \frac{i \cdot H}{P} - i \quad (4.18)$$

which is equal to UI latency (see Equation 4.3). Therefore, with a fixed quantity of $P$ instances, RTS and UI approach process burst frames at the same rate so application achieves the same computation quality regardless of the approach it chooses.

However, when cloud providers serve $N > 1$ applications, the UI approach would require $N \cdot P$ instances or the cost goes linear with the number of concurrent applications. For RTS, however, application scales allocated instances with demand so a single RTS instance can serve multiple bursts unless they happen at the same time (i.e., burst interference). Let $P(n)$ be the probability of $n$ bursts overlap with each other. Because each burst last for $D$ frame-time, $P(n)$ is equivalent to the probability of $n$ burst arrive within an interval of length $D$. This probability follows a Poisson distribution with the arrival rate of $N \cdot \lambda$

$$P(n) = e^{-N \lambda \cdot D} \frac{(N \lambda \cdot D)^n}{n!} \quad (4.19)$$
Clearly, the probability is non-zero for all $n$ so supporting multiple applications with the RTS approach is equivalent to UI in the worst case. However, because bursty applications have very low duty factor, the probability will approach zero very fast as $n$ increases. Therefore, in most of the cases, the cloud just needs a small number of resources to guarantee application quality.

Figure 4.6 shows the cumulative distribution function of $P(n)$ for different $N$ and each application has duty factor $DF = \lambda \cdot D = 0.01$. For $N = 10$, at least $99\%$ of burst interferences consist of less than 5 bursts out of 10 applications. This means for $99\%$ of the time, RTS resource consumption is $50\%$ of UI. And this number goes higher as $N$ increases: $20\%$ for $N = 100$ and $14\%$ for $N = 1000$. In other words, the cloud provider can sell an RTS instance two times for $N = 10$, five times for $N = 100$ and seven times for $N = 1000$. Recall that RTS instances are charged at 4-6x higher than UI, the value per RTS unit instance is about 8-42x higher than UI instances. Therefore, real-time serverless resources have very high potential to increase the cloud provider’s revenue.

In summary, our analytical model shows that real-time serverless is very cost attractive to not only users but also cloud providers. For users, utilizing real-time serverless enables them to avoid resource waste, scale cost to actual demand, and thus potentially reduce overall resource cost. For cloud providers, real-time serverless allows them to charge more on resource consumption, sell a single instance to multiple applications, thereby potentially increase their revenue.
Our analytical evaluation suggests that real-time serverless enables bursty, real-time applications to not only achieve higher quality/value but also may have some economic advantages. However, the analytic models simplified a number of aspects of resource management – in this chapter, we conduct empirical studies of a wider range of workloads and cloud resource settings. Our goal is to understand how robust the real-time serverless benefits are across a range of different types of bursty workloads. First, we study the basic scenario used for the analytical model, using our simulator, and validate the analytical model results. Second, we study a variety of bursty workloads, varying burstiness (arrival rate), burst duration, and systems with multiple sources of bursts. These studies characterize how broadly beneficial the real-time serverless resource pools can be. Finally, we consider a case study, which is consisted of several realistic workloads with video analytic pipeline, to show that the guaranteed allocation rate enables higher application quality on video analytic applications while showing economic benefits.

5.1 Simulation with Synthetic Workloads

5.1.1 Simulation and Workload Generation

We construct a simulator as an extension of the analytical model. The terms and notation described in Chapter 4 are reused. We add variability to burstiness properties to capture the uncertainty of bursty workloads. In particular,

- **Burst arrival rate** ($\lambda$) bursts arrive in a memoryless Poisson process. Burst interference is possible.

- **Burst duration** ($D$) varies according to a Gaussian distribution.
Adding variability to burstiness properties allows the simulation to cover a wide range of bursty workloads. For example, we use low variance, low arrival rate to simulate a uniform single bursty source. For a collection of uncorrelated, varied duration sources, we use high variance and high arrival rate.

In resource management, real-time serverless resource pools guarantee allocation rate by splitting the time into fixed $1/A_{RTS}$ frame-time slots. Within each time slot, the pool makes one instance available to the application. The application can consume this instance without delay. If the application asks for more, it has to wait until the next time slot. We capture the scenarios we analyze in Chapter 4 by letting applications allocate resources immediately for every incoming frame. If frame arrival rate larger than allocation rate, delay accumulation occurs (frame experiences the delay of previous frames) harming application quality severely.

**5.1.2 Validating Analytical Model**

We first simulate a scenario similar to that studied with the analytical model (Chapter 4), a single fixed duration bursty video analytic (VA) workload. Parameters include burst height of 140x normal load, and burst duration of 3,600 frame-times. We use 0.01 duty factor to avoid burst interference assumed by the analytical model.
We vary the guaranteed allocation rate and try to reproduce the analytical results from Figure 4.2 via simulation. The results of one-day simulation are shown in Figure 5.1. The simulation results are close to that of analytical modeling with some noisiness due to the discrete method of simulation. For example, Figure 5.1a exhibits similar results but small variation due to the uncertainty in arrivals of bursts and lead to a slight reduction in achievable value comparing to the analytical model. Similarly, Figure 5.1b shows the increase in application value while the guaranteed allocation rate is increasing. Although results show less achievable value comparing to the analytical model, real-time serverless pool shows comparatively significant growth in slope. Therefore, real-time serverless pool presents promising benefits in maximizing value under bursty real-time workload simulations.

5.1.3 Robustness against Burst Variability

In analytical modeling, we assume that bursts have fixed length and thus omitted the effect of bursts variability. Here, we generate burst duration using Gaussian distribution to study the robustness of real-time serverless against burst length uncertainty. To understand how duration variability affects burst value, we observe the burst value achieved by VA at various allocation rate from simulations over different workloads having the same burst characteristics but burst duration standard deviation ($\sigma$). Simulation results are depict in Figure 5.2 with three duty factors: high ($DF = 0.25$), medium $DF = 0.1$ and low ($DF = 0.01$).
At low duty factor, changing $\sigma$ does not cause much effect on application value (Figure 5.2a). However, as the duty factor increases, value deterioration becomes more severe and high variability bursts will experience worse impact. At duty factor $DF = 0.1$, burst with $\sigma = 1x$ duration suffers 15% burst value loss at $A_{RTS} = 1$ (Figure 5.2b), and if duty factor jumps to $DF = 0.25$, the loss increases to 19%. If $\sigma$ is doubled to $2x$ duration then the loss is 2.2x to around 42% (Figure 5.2c). However, value reduction can be solved by simply increasing the allocation rate. For example, $A_{RTS} = 2$ will let application achieve 100% burst value from bursts with $\sigma = 1x$ duration. Even for highly variable bursts of $\sigma = 2x$ duration, an allocation rate of 3 instance/frame-time is sufficient. Also note that burst variability only takes effect at high duty factor, but in response, only a small allocation rate increase is needed to saturate the impact. Even at $\sigma = 2x$ duration, rising duty factor from 0.01 to 0.25 (25x demand increase) only require allocation rate to rise from 1 to 3 instance/frame-time (3x resource commitment increase). This confirms the robustness of real-time serverless against burst variability.

The reason burst variability reduces application value is that at high variability, many long bursts appear and they have a high chance to overlap with others. During the interval of burst overlap, the application has to service multiple bursts at a time so demand increase significantly, thereby resource allocation is not fast enough to guarantee value for all of the bursts. In the next subsection, we will address this phenomenon in more detail.
5.1.4 *Robustness against Burst Interference*

We vary burst duration from 900 to 14,000 frame-times with standard deviation of $0.1 \cdot D$, and resulting increased duty factor to induce increased burst interference (low ($DF < 0.01$), medium ($DF < 0.1$), and high ($DF < 0.25$)). For low burst interference, (see Figure 5.3a), the simulation results match the analytical model results in Figure 4.3, validating its accuracy for a single burst. As we increase duty factor (see Figure 5.3b and 5.3c), we see degradation of burst value, particularly for the higher duty factors (longer duration). For example, at $A_{RTS} = 1$, $D = 14,400$ for $DF = 0.1$ and $DF = 0.25$, burst value is reduced to 88% and 77% respectively.

Interference can also cause degradation at low $A_{RTS}$, as the RTS pool has a lesser ability to respond to concurrent bursts. The value of long bursts is hit harder due to their high duty factor. For example, burst value for $D = 14,400$ with $A_{RTS} < 0.5$ drops below $D = 7,200$ in high duty factor scenarios (see Figure 5.3c).

![Figure 5.4: Burst value vs. $A_{RTS}$ at different duty factors (varying $\lambda$).](image)

For better understanding, we test the sensitivity of burst value to $A_{RTS}$ at various duty factors. The results are shown in Figure 5.4. Note that at burst interference, demand is doubled, tripled or more depending on the number of bursts involves. This means to saturate double burst interference, the application needs to allocate resource 2x faster, for triple burst interference, 3x allocation rate is required and so on. Thus, in the figure, the breaks of curves
Figure 5.5: Cost for maximizing value using UI and RTS at various cost ratio \((k)\) and Cost of UI at different duty factors, \(P = \) maximum observed computation height (several bursts).

at \(A_{RTS} = 1\) and \(A_{RTS} = 2\) indicate value reduction effect of burst interference. However, due to low interference probability (see Section 4.3), 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 \(A_{RTS}\). Therefore, real-time serverless is robust against burst interference.

Burst interference affects the UI approach as well. During burst interference, burst frames are “stacked” so the demand is doubled or even tripled. Since applications pre-allocate instances for worst cases, many more instances are needed to overcome the situation. As a result, resource cost skyrockets when burst interference happens more frequently. Figure 5.5a confirms our observation as UI resource cost grows with duty factor, which we choose as a proxy for burst interference. Increasing duty factor also creates more chance of having more bursts involve in an interference. For example, at duty factor of 0.13, the worst case of interference starts to include 3 bursts, instead of 2 in lower duty factors, making the application to triple UI pre-allocation, creates a large jump in UI cost.

The high extra cost for dealing with burst interference makes the UI approach economically inefficient compared to real-time serverless. In Figure 5.5, we reproduce the results from Figure 4.5 using simulation. At low duty factor, Figure 5.5b shows that RTS is more cost-effective than UI, even with RTS at a 10-20x premium. This result is similar to Figure 4.5a confirming the correctness of the analytical model.
At higher duty factor (Figure 5.5c), RTS cost grows as burst frequency grows, and UI is eventually more cost-effective for $k = 24$. However, these results are much better than for the analytical modeling (Figure 4.5b) where $k = 6$ made RTS more expensive than UI for $DF > 0.125$. Here, RTS is still cheaper than UI even at $k = 12$ and $DF = 0.25$ since UI must reserve enough resources for the highest burst peak. We believe the jumps of UI cost according to the increase of burst peak are the cause of the spikiness in Figure 5.5c.

5.1.5 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 real-time serverless pool. Thinking of the latter, we explore how higher guaranteed allocation rates can increase burst value towards the potential maximum.

![Figure 5.6: Allocation rate needed to achieve burst value fraction (at varied duty factors).](image)

We examine the potential for multiple applications to share a single real-time serverless resource pool 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 5.6. For low burst value ($< 0.5$) there is little difference in the required $A_{RTS}$. For moderate values, the difference grows but at a deeply sublinear rate. For example, for the value of 80% potential maximum value, an increase from 1 to 25
applications require only a 2x increase in $A_{RTS}$, and based on our model implementation (see Section 6.1), only a 2x in cloud provider resource commitment.

The curves cover the allocation needed for a wide range of duty factors from 0.01 to 0.25 but they are very close to each other indicates that only a small increment of allocation rate is sufficient to deal with a significant increment of burst demand. At 90% max value, the multiple is even smaller. requiring a 1.6x $A_{RTS}$ increase for a 25x increase in the number of applications. These results suggest that real-time serverless scales well – supporting a growing number of bursty, real-time applications at high quality with a slowly growing number of resources. Our results show this growth is deeply sublinear, suggesting that real-time serverless may be best implemented as a shared cloud service (not privately by a single application) and that doing so may be quite cost attractive for cloud providers.

To get a better understanding, we run another simulation varying number of concurrent applications. Applications are identical and independent with duty factor = 0.1 and resources are provided to ensure each of them achieves 100% of their maximum value. We record their demand, the number of allocated UI and RTS instances throughout 24 hours and depict the results in Figure 5.7.

Within burst durations, a single application consumes the same maximum quantity of UI and RTS resources (Figure 5.7a). This validates the analysis results in Section 4.3. In the normal situation, however, applications with UI approach still hold pre-allocated UI
instances so their resource commitment is a constant. In contrast, resource commitment of applications with RTS approach scales with actual demand as they release resources right after the bursts ended. The released resources, in turn, can be used for other applications experiencing bursts. As a result, the total resource commitment of multiple concurrent applications using RTS is much lower than UI: for 10 applications, RTS maximum resource commitment is 60% of UI, and for 100 applications, it is 20%.

Also note that due to quality guarantee, UI actual resource consumption cannot adjust to the foreground load. Thus, even if the cloud provider applies resource multiplexing techniques, the best that they can do is to pull UI resource consumption down to the peak of demand. In contrast, the guaranteed allocation of real-time serverless enables the cloud provider to know the resource they have to prepare for the foreground changes, thereby pulls RTS actual resource consumption to close to application actual demand. In average, this (dashed cyan lines) is 2-9x lower than the best of UI (dashed purple lines).

In summary, real-time serverless can efficiently support a growing number of real-time bursty applications. The resource type also requires much fewer resources to guarantee application quality than traditional resource types.

5.2 Case Study: Traffic Intersection Monitoring

In the previous section, we used the simulation for a range of bursty workload parameters to show the benefits of real-time serverless for application quality and cost. Here, we use a traffic monitoring case study, derived from a real traffic video, deploying a deep neural network that does object detection, and the induced bursty workload trace to evaluate the benefits of guaranteed allocation for real-time, online video analytics.
Figure 5.8: Glimpse System Architecture (from [17]).

<table>
<thead>
<tr>
<th>Burst Duration (frames)</th>
<th>Burst Height</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
</tr>
<tr>
<td>Night</td>
<td>116</td>
</tr>
<tr>
<td>Day</td>
<td>120</td>
</tr>
<tr>
<td>Rush hours</td>
<td>917</td>
</tr>
<tr>
<td>Overall</td>
<td>197</td>
</tr>
</tbody>
</table>

Table 5.1: Burst Statistics for Traffic Video

5.2.1 Real Traffic Video Statistic and Video Analysis Pipeline

We model a Glimpse-like pipeline with a client and server (the cloud) that processes frames considered interesting by the shallow processing at the client [17] (see Figure 5.8). Glimpse uploads frames to the server for object detection. Our empirical measurements characterized the server-side frame processing cost at 20x for object detection, but for richer analytics, this ratio could be much higher. We use a rush-hour traffic video captured from a traffic cam in Southampton, NY at the intersection of County Rd. 39A and North Sea Rd. and available from [60]. The video is 30 frames per second at a resolution of 1920 × 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 [35] with ResNet trained on KITTI dataset[26]) 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 5.1). To scale up to a full 24-hour from our short, rush-hour clip, we replicate it to create two 60-minute segments (morning and afternoon rush hour). We scale the time base by 20x while holding object interval duration constant, creating an 8-hour segment of lower traffic (daytime). Finally, we scale the time base by 40x while holding object interval duration constant, creating a 14-hour segment of the lowest traffic (nighttime). The total number of bursts is 2,311 and the duty factor is 0.175 for the 24-hour period. The number of objects present in each frame multiplying the server-side frame computation cost ratio (20x) defines the burst height.

5.2.2 Computation Quality Guarantee

We first explore the basic characteristics of the traces, as shown in Table 5.1. The burst durations are much shorter than those explored in Chapter 4 and Synthetic workloads with an average burst duration of 7 seconds (210 frame-times), as shown in Table 5.1. Moreover, both the burst duration and height are highly variable within each part of the day.

Figure 5.9a shows the value distribution of the video trace; while similar qualitatively to Figures 4.2a and 5.1a, the real burst trace is much noisier. The traffic analysis quality benefits from increasing \( A_{RTS} \), are shown in Figure 5.9b and 5.9c. Three curved sections are clearly visible and correspond to the three different operating points – rush-hour, daytime,
Figure 5.10: RTS is efficient in terms of both application cost and cloud provider resource cost and nighttime. At $A_{RTS}$ as little as 0.25 instances/frame-time, all of the nighttime value is captured. At $A_{RTS}$ of 0.9, the daytime value is captured. In Figure 5.9c, we see flat curves and very little separation by fraction of burst-frame value. This reflects a difficult workload for increases in $A_{RTS}$ to improve application quality. To achieve full quality on the intense activity during rush hour (10 objects in frame) requires $A_{RTS} = 1$.

Results from Figure 5.9 confirm that using real-time serverless guarantee the traffic monitoring application value. And by increasing $A_{RTS}$, the application can improve the guaranteed quality, although comparing to the synthetic data, quality increase much slower due to extreme high duty factor during rush hours. Furthermore, at $A_{RTS} = 1$ real-time serverless enables the application to achieve the maximum target value. This confirms the robustness of RTS against realistic workloads.

5.2.3 Cost Attractiveness

Figure 5.10a 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 ($k$) and a UI implementation achieving the same (100%) quality. 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 $k = 8$. In short, the RTS resources are 16x more valuable than traditional UI resources.

As discussed in Section 6.1, guaranteed allocation rate can be achieved at cloud provider resource cost linear in $A_{RTS} \cdot R_{RTS}$. However, we find that for our case study, the effective cloud provider resource requirement is much lower as shown in Figure 5.10b. This is because the typical burst height is much less than $R_{RTS}$, so many instances are returned long before the full $R_{RTS}$ time. In this case, resource cost is reduced 70-fold to only 123 instances. We expect savings such as this to occur in many cases but will vary in magnitude.

In summary, real-time serverless allows applications to allocate resources on burst occurrence and thus, autoscale the cost to computation demand. Furthermore, short actual instance runtime per invocation allows the cloud to quickly reuse RTS instances for multiple burst frames to scale actual allocation to resource commitment, thereby improve cloud resource management flexibility. These capabilities make real-time serverless cost attractive to not only applications but also cloud providers.
CHAPTER 6
A REAL-TIME SERVERLESS IMPLEMENTATION

In this chapter, we describe a simple implementation of real-time serverless to show its feasibility and conduct a simple experiment to show its advantages in supporting real-time bursty applications over the current best-effort serverless.

6.1 Feasibility of Real-time Serverless Implementation

As covered in Section 3.4, a real-time serverless pool is characterized by its allocation rate $A_{RTS}$ and fixed duration $R_{RTS}$. The pool ensures that every application can allocate instances at the minimum guaranteed allocation rate of $A_{RTS}$. To do so, it needs to reserve $C$ instances for each application so that once the application creates a burst and starts allocating new instances, the pool can have enough resource to quickly meet the unusual demand. But what is the right choice for $C$?

![Figure 6.1: A real-time serverless resource pool can be implemented with a fixed number of instances.](image)

Figure 6.1 shows how we determine $C$. Given the fixed duration of $R_{RTS}$, an instance allocated at time $t$ can be reclaimed by the resource manager at the time no later than $t + R_{RTS}$. After reclamation, the instance can be reallocated, bounding the instances required to implement the guarantee. Given the allocation rate, $A_{RTS}$, the pool should have enough instances to give to application $A_{RTS}$ instance per frame-time from the beginning to $R_{RTS}$
(when they can start reclaiming resource and reuse them for next allocations). Thus, for each application, the number of instances that the pool has to pre-allocate is

\[ C(A_{RTS}, R_{RTS}) = A_{RTS} \cdot R_{RTS} \]  \hspace{1cm} (6.1)

So to support \( N \) applications, the size of the pool is

\[ \text{Size}(A_{RTS}, R_{RTS}) = N \cdot C(A_{RTS}, R_{RTS}) \]  \hspace{1cm} (6.2)

Thus, the resource cost grows linearly with both the guaranteed allocation rate and the finite duration.

### 6.2 Architecture

In Figure 6.2, we depict the overall architecture of our real-time serverless implementation. In particular, the system is considered as a Mesos cluster and real-time serverless pools are implemented as a Mesos framework. Each pool has three key components: interface, scheduler, and executor.

![System architecture](image)

*Figure 6.2: System architecture.*

The real-time serverless interface stands between the resources and applications. It pro-
vides data types for application to define real-time serverless tasks and a set of function calls for allocating, monitoring, and getting results from real-time serverless instances. We use Apache Thrift to describe the interface and organize the communication between applications and resource pools. Figure 6.3 provides the definition of the interface in detail.

**deploy**(name, body, allocation_rate, max_runtime) → id

Deploy a new function given its name and body. The function specifies its desired guaranteed allocation rate and maximum runtime. The real-time serverless framework prepares resource accordingly and then return the identifier for the function.

**invoke**(id) → id

Execute a function given its id. Returns identifier for the invocation if it initiated successfully. Otherwise, returns a null identifier.

**getInfo**(id) → info

Get information (execution status, elapsed time, etc.) of an invocation given its identifier.

**delete**(id)

Remove a function given its identifier from the real-time serverless framework.

Figure 6.3: Real-time serverless application interface

Here, an application is a set of functions. Each function $f$ has a runtime limit $R_f$ and guaranteed allocation rate of $A_f$. $A_f$ ensures a minimum invocation processing rate delivered by that function. Formally, for any 1-second interval, the number of invocation initiated by the system $I_f$ must satisfy

$$I_f \geq \min[A_f, Req_f] \quad (6.3)$$

where $Req_f$ is the number of invocation requests of $f$ in this interval. The constraint means that, within any 1-second interval, if the number of invocation requests is fewer than $A_f$, invocations will be initiated for all requests. If, the number of invocation requests is greater than $A_f$, at least $A_f$ invocations will be initiated. For example, if an IoT service for a sensor with data reporting interval of 100ms wants to ensure no data lost and, all data is processed in a timely fashion, it made deploy data reporting function with guaranteed allocation rate of 10 for instance per second.
Behind the interface, the real-time serverless scheduler is the place where all of the resource management logics are implemented. The module cooperates with Mesos to pre-allocates resources for each function. In particular, once a new function $f$ is deployed, the real-time serverless framework tries to create a new pool for it by asking Mesos to reserve at least $A_f \cdot R_f$ (see Section 6.1) instances. It is also responsible for controlling function invocation. Because function can be invoked at any rate, if their invocation requests rate exceeds $A_f$ but the scheduler does not apply any request filters (try to accept all of them), then the pool would be exhausted before the allocated instances can be reclaimed and reused. To avoid such situations, the scheduler maintains a list of invocations initiated in the last 1-second interval for each function. Every time a new invocation request arrives, it sums the number of invocations in the list. If the sum is less than $A_f$, the scheduler will randomly select one free instance in the resource pool and use it to initiate the invocation. If the sum, however, is equal or greater the allocation rate, then the minimum allocation rate has been already achieved, the scheduler may reject the request.

Note that the implementation we present in this chapter is naive and uses too many resources. But it is one of many possible implementations for real-time serverless. One alternative is to share the pre-allocated instances among multiple functions. To optimize resource utilization, dynamic resource control is required: the pool must expand when burst interference occurs and shrink when applications stop generating bursts. This is feasible only if the underlying resource manager promises to give the framework new resources every time it needs (so that no allocation rate guarantee is violated during burst overlap). However, it is not supported by the current resource manager, Mesos. Thus, we leave this problem to future research.

### 6.3 Demonstration

In this section, we describe a demonstration of the real-time serverless’s capability of supporting real-time bursty applications. We consider a simple streaming application creating
animation by generating a single frame for every second. The application has two parts: a
driver for frame generation and a frame viewer for display animation in front of its user as
shown in Figure 6.4. The driver launches one RTS/serverless function per frame generated.
After being invoked, RTS/serverless instance generate a new frame and put it in the buffer
of the frame viewer. The viewer loads new frames at the same speed as frame generation
(one per second). The viewer picks the oldest frame in the buffer for loading. If the buffer is
empty, then the viewer stalls and users experience interruption. The application’s ultimate
goal is to show the animation smoothly by trying to load frames at a constant rate. Thus,
the driver must not let the buffer empty when the viewer loads a new frame.

![Diagram](image)

Figure 6.4: Demonstration diagram

We deploy two systems of equal capacity and let the application run on the top of them:

- **RTS**: Consist of two resource pool: a real-time serverless with $A_{RTS} = 1$ instance
  per second and a serverless pool whose instances are allocated in a best-effort manner.
  Applications use the real-time serverless framework for frame generation.

- **Serverless** only has a single best-effort serverless pool. Applications use this pool for
  frame generation.

Each system is a single machine in the RIVER cluster [53] equipped with an Intel Xeon(R)
Gold 6138 CPU and 512 GB of memory. Applications components are run on another
machine. For the RTS framework, we decide to not allocate more instances to functions when the minimum allocation rate is met.

To demonstrate the robustness of real-time serverless against system load, we use the statistics of the traffic video from the use case study (Section 5.2) to create another workload and feed it to the serverless pool of both systems as a presence of background load. The workload consists of multiple applications. Each application requests for one new instance per second. The runtime of an allocated instance is randomly drawn from a Gaussian distribution with mean, standard deviation got from the video statistics. The sample is truncated by statistic min and max. Experiment diagram is shown in detail in Figure 6.4. We compare resource usage, allocation rate and frame arrival time of the two systems.

We first explore the best performance of each system with no background load. The serverless pool can use its full capacity to serve frame generation so its performance should
be equivalent to RTS. This is confirmed by the results shown in Figure 6.5. The serverless pool not only enables application to allocate resource at the rate equal to RTS (Figure 6.5b) but also consumes similar resources with RTS (Figure 6.5a) over time. As a result, the two systems deliver frames with latencies very close to the ideal latency expected by the application (Figure 6.5c).

When we turn on the background load, however, serverless performance decreases significantly (Figure 6.6). The streaming application now has to share serverless instances with others so there are situations that it cannot allocate resources in time. In Figure 6.6b, serverless’s allocation rate fluctuate widely in the range between 0 and 1 instance/second. This reflects the behavior of best-effort allocation: the system always gives free instances to applications without any caution of resource exhaustion. Thus, when multiple applications continuously ask for resources, the system quickly gets exhausted. Then, because instances run for a very short duration, the pool is filled again, and then due to pending requests, it quickly becomes empty again, and so on.

In contrast, RTS pool reserves instances in advance so even at the presence of background load, RTS still has enough resources to ensure the guaranteed allocation rate. As a result, the application working with RTS can allocate resources at the application’s desired rate. Therefore, RTS’s resource usage is also better than serverless (Figure 6.6a) and finishes frame generation about 20 seconds sooner than serverless. Also note that because of streaming, frames are processed in a FIFO manner. Frame latency is accumulated across all preceding frames. As can be seen in Figure 6.6c, latter frames experience higher latency than earlier ones. This reflects the model we constructed in Chapter 4.

In summary, our implementation proved that real-time serverless is feasible. Also, thanks to its guaranteed allocation rate, applications using the pool is unaffected from background load thereby, be able to guarantee the service quality delivered to their customers.
To our best knowledge, no commercial cloud resource management guaranteed allocation rate. A wide variety of research studies explore resource management in clusters and data centers. We discuss these classes of resource managers and the resource models they support.

7.1 Serverless and FaaS

The serverless (also function-as-a-service or FaaS) is cloud event-driven resource model which creates instances when some pre-defined conditions are met to process events within short duration. Representative of industrial serverless implementation are AWS’s Lambda [5], Google Cloud Function [29] and Azure Function [10]. Besides, there are many open source options available, such as Knative [42] and OpenFaaS [47].

Serverless limits execution duration to a few minutes and maximum concurrent invocations to a few hundreds within a region. Although there are studies to improve invocation latency [34, 41], all of those serverless offerings provide no guarantees of resource allocation rate or latency, thereby cannot be used by applications to guarantee real-time computation. In contrast, our real-time serverless provides a guaranteed allocation rate while supporting a similar fine-grained “burst” computing model.

7.2 Datacenter Resource Management

Datacenter resource management formulates a more complex problem, mixing long-running processes with more typical jobs with dependent sets of short-running tasks. Mesos [33] uses a distributed two-level scheduling model, offering resources to computing frameworks, who in turn schedules tasks. Borg [62] schedules a mixed workload of end-user-facing service jobs (high priority) and batch jobs (low priority). Firmament [28] is a centralized scheduler desired to be scalable at low placement latency if only partial workload consists of short
tasks. Sparrow [48] is a decentralized scheduler designed for data centers that support jobs composed of very short, sub-second tasks. It achieves very high throughput rates. Chaojie et al. [66] propose different information disclosure models to enable cloud users to get more value from volatile resources. The study also develops algorithms for reserving statistical guarantees and preserving resource management flexibility. None of these systems provide guaranteed allocation rates. Morpheus [39] cleverly infers periodic resource requirements and Service Level Objective (SLO) based on history. However, this approach does not work for bursty workloads which may have little or no periodicity or repeated structure. And, Morpheus provides no guaranteed allocation rates – all rates are subject to the current load and job interactions competing for resources. In contrast, our work’s signature difference is a guaranteed allocation rate.

7.3 Job Scheduling for Heterogeneous Workloads

Mainstream resource management schedulers such as TORQUE [55], SLURM [65] support batch jobs that run for hours. Extensive research schedules jobs with heterogeneous resource requirements (including parallelism) and run times, optimizing for resource utilization, job wait time, and more. For example, a number of studies explore FCFS and backfilling algorithms [54, 56]. Further, Tang et al. [56] propose metric-aware scheduling that optimizes performance based on such as fairness and system utilization. In a shared compute systems, Dominant Resource Fairness (DRF) [27] is a well-known resource allocation algorithm designed to ensure fairness among applications with different resource demands. The algorithm also has many variants adjusted for different shared environments and scales [50, 40]. However, these approaches are not responsive to burst demands, do not provide guaranteed allocation rates and do not support fine-grained (seconds) resource allocation.
7.4 Coping with Bursty Workloads

Scalable internet services deal with bursty loads managing yield and latency by dropping requests [13]. For example, the WeChat microservice system has an elaborate system for load shedding that orders drops to minimize wasted work [68]. However, all of these approaches drop requests in order to maintain service quality for accepted requests. In contrast, because we assume the cloud has sufficient resources to service our bursty, real-time applications, we take the approach of guaranteeing allocation rate to maintain quality for all of the received requests.

Closest to our study, one recent effort attempts to accommodate real-time advanced photon source (APS) experiment data analysis (a coarse-grained task), sharing resources on a batch-job scheduled system [44]. This system dynamically shifts nodes from the batch schedulers resources into an on-demand allocation pool. This shifting is based on prediction of the bursty, real-time APS workload. Our work differs in application granularity (coarse-grained vs. fine-grained), and the predictability of their domain. Because of their dependence on prediction, their resulting systems cannot provide real-time or application quality guarantees. Our approach creates a dependable, guaranteed allocation rate, upon which application guarantees can be built.

Tributary [32] tries to exploit volatile cloud resources (Spot) to create an affordable solution for bursty workloads. The system creates models of preemption likelihood and partial independence between resources, dynamically select and adjust resources to meet job SLOs. Evaluation results show Tributary can achieve comparable performance with a smaller cost than AWS AutoScale service. However, we believe real-time serverless with the capability of autoscaling and fine-grain pricing could enable applications to achieve the same performance at comparable cost and much less development/deployment effort.
7.5 Coping with Real-time Workloads

Many research efforts have been spent on task runtime prediction for efficient scheduling and resource allocation. Some techniques are developed for repeating jobs [18, 20, 36] while others use the job structure and characterize input to construct predicting models [23, 52]. JamaisVu [58] and 3Sigma [49] extracts tasks’ features from execution history, uses them for constructing runtime distributions and applies a tournament prediction to estimate task runtime. From runtime prediction, many scheduling such as backfilling [57, 67] or packing [61, 59] can be applied to minimize real-time constraint violation while still maximizing resource utilization.

There are also other approaches in the literature. For example, [14] uses the earliest-deadline-first scheduling policy and explicitly handling the transient of dynamic real-time workload to reduce deadline misses. [12] 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 [16, 69].

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

In summary, to the best of our knowledge, none of the recent studies tackles the problem of cost-effectively handling bursty demands in a timely fashion. They either consider real-time constraints or bursty demand but not both as real-time serverless did.
CHAPTER 8
SUMMARY AND FUTURE WORK

8.1 Summary

Today’s cloud resource offerings provide no guarantees for resource allocation preventing real-time, bursty applications from sufficiently exploiting cloud capability to guarantee high application quality. To address this issue, we have proposed and evaluated a new type of cloud resource with a distinct service-level objective – real-time serverless.

Using analytical modeling, simulation, and a case study, we have shown that the guaranteed allocation rate promised by real-time serverless expands application design space significantly. With real-time serverless, applications know the availability of resources at current time and future so they can proactively schedule computation flows, adjust resource allocation and consumption to guarantee burst processing quality and optimize resource usage/cost.

Studies of burstiness characteristics (shape, variability, duty factor, and interference) prove the robustness of real-time serverless against a wide range of real-time, bursty workloads. Even an extreme noisy realistic workload generated by video at rush hour is still handled efficiently with real-time serverless.

Furthermore, real-time serverless’s on-demand allocation and fine-grain pricing enable computation cost to autoscale up and down to burst demand. In fact, even at the cost ratio of 10-15x, the real-time serverless approach can be still cheaper than the traditional cloud offering. Such high value per unit resource opens a great opportunity for cloud providers to improve their revenue. In addition, our study on resource allocation shows that cloud providers can even earn more from real-time serverless as their short duration per invocation allows the cloud to quickly reclaim resources for other uses so that they can serve more applications and have more resource management flexibility.

In the thesis, we also introduce a simple implementation of real-time serverless. The
implementation not only proves the feasibility of the new resource type but also demonstrates its robustness against background noise, plausibly efficiently support bursty applications.

In summary, the results of our study successfully answered all questions in Section 3.6 and consequently proved that real-time serverless meets all requirements of a solution for bursty, real-time workloads. That evidence strongly suggests that real-time serverless would be a valuable resource offering on the cloud. Successfully implementing them could be beneficial for both cloud providers and customers and let us put a step toward getting more value from the promising cloud computing.

8.2 Future Work

Our results open a number of interesting research questions. In the thesis, we only study on a video analytic application but there is a vast diversity of real-time bursty applications in practice. Each of them has different burst rates and quality requirements. If real-time serverless is deployed on the cloud, then how should we combine guaranteed allocation rates, runtime limits, and pricing policies to create real-time serverless pools that wide enough to create sufficient resource usage flexibility for applications to meet their needs?

From resource management point of view, fast resource reclamation/reuse and high potential of application multiplexing open a broad space of efficient implementations for real-time serverless leaving many interesting questions to be answered: For a given quantity of resources, how should a cloud provider divide across traditional cloud resources and real-time serverless to best serve applications and/or maximize revenue? Can it do so dynamically? How do resource partition response to the change of workloads? How to share real-time serverless resource efficiently among multiple applications? etc. We look forward to future research by the community to address these questions.
REFERENCES


